Improving Convolutional Neural Network-based Image Classification by Exploiting Network Layer Information

Inauguraldissertation

zur

Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

der

Mathematisch-Naturwissenschaftlichen Fakultät

der

Universität Greifswald

vorgelegt von

Daniel Lehmann

Greifswald, 13. Juli 2023

Dekan*in: Prof. Dr. Gerald Kerth

1. Gutachter*in: Prof. Dr. Marc Ebner
(Universität Greifswald)

2. Gutachter*in: Prof. Dr. Thomas Kirste
(Universität Rostock)

Tag der Promotion: 18. Januar 2024

Abstract

Convolutional Neural Network-based image classification models are the current state-of-the-art for solving image classification problems. However, obtaining and using such a model to solve a specific image classification problem presents several challenges in practice. To train the model, we need to find good hyperparameter values for training, such as initial model weights or learning rate. However, finding these values is usually a non-trivial process. Another problem is that the training data used for model training is often class-imbalanced in practice. This usually has a negative impact on model training. However, not only is it challenging to obtain a Convolutional Neural Network-based model, but also to use the model after model training. After training, the model might be applied to images that were drawn from a data distribution that is different from the data distribution the training data was drawn from. These images are typically referred to as out-of-distribution samples. Unfortunately, Convolutional Neural Network-based image classification models typically fail to predict the correct class for out-of-distribution samples without warning, which is problematic when such a model is used for safety-critical applications. In my work, I examined whether information from the layers of a Convolutional Neural Network-based image classification model (pixels and activations) can be used to address all of these issues. As a result, I suggest a method for initializing the model weights based on image patches, a method for balancing a class-imbalanced dataset based on layer activations, and a method for detecting out-of-distribution samples, which is also based on layer activations. To test the proposed methods, I conducted extensive experiments using different datasets. My experiments showed that layer information (pixels and activations) can indeed be used to address all of the aforementioned challenges when training and using Convolutional Neural Network-based image classification models.

Zusammenfassung

Modelle basierend auf Convolutional Neural Networks sind der aktuelle Stand der Technik zur Lösung von Bildklassifizierungsproblemen. Das Training und die Nutzung eines solchen Modells zur Lösung eines spezifischen Bildklassifizierungsproblems bringt in der Praxis jedoch einige Herausforderungen mit sich. Um das Modell trainieren zu können, müssen geeignete Werte für die Trainingshyperparameter identifiziert werden wie beispielsweise die initialen Gewichte des Modells oder die Lernrate. Die Ermittlung dieser Werte ist üblicherweise jedoch kein trivialer Prozess. Daneben ist zudem ein weiteres Problem, dass in der Praxis häufig die Trainingsdaten nicht balanciert sind bezüglich der Klassenverteilung. Dies wirkt sich in der Regel negativ auf das Modelltraining aus. Allerdings ist es nicht nur eine Herausforderung ein Modell basierend auf einem Convolutional Neural Network zu trainieren sondern dieses auch nach dem Training zu nutzen. Nach dem Training wird das Modell möglicherweise auf Bilder angewandt, die aus einer anderen statistischen Verteilung stammen als die Trainingsdaten. Diese Bilder werden üblicherweise als Out-of-Distribution Samples bezeichnet. Leider schlagen Modelle basierend auf Convolutional Neural Networks häufig auf diesen Out-of-Distribution Samples ohne Warnung fehl. Dies ist vor allem problematisch wenn ein solches Modell für sicherheitskritische Anwendungen verwendet wird. In meiner Arbeit habe ich untersucht, ob Informationen aus den Layern eines Modells basierend auf einem Convolutional Neural Network (Pixels und Activations) verwendet werden können, um die aufgeführten Probleme zu lösen. Ich schlage eine Methode zur Initialisierung der Modellgewichte basierend auf Bildpatches vor, eine Methode zur Balancierung von unbalancierten Trainingsdaten basierend auf Layer Activations und eine Methode zur Erkennung von Out-of-Distribution Samples, welche ebenfalls auf Layer Activations basiert. Um meine vorgeschlagenen Methoden zu testen habe ich umfassende Experimente unter Verwendung von verschiedenen Datensätzen durchgeführt. Meine Experimente haben gezeigt, dass Informationen von den Layern eines solchen Modells (Pixels und Activations) tatsächlich verwendet werden können um alle aufgeführten Herausforderungen beim Training und bei der Nutzung des Modells zu adressieren.

Contents

List of Figures

List of Tables

1 Introduction and Dissertation Outline

Convolutional Neural Network-based (CNN) image classification models are the current state-of-the-art for solving image classification problems [46, 71]. To solve a specific image classification problem, we need to train the weights of such a model using a set of training images until the model achieves a sufficient classification performance on a given test dataset. For instance, suppose we have a certain type of plant that has been grown for agriculture. Some of the plants, however, suffer from a particular plant disease. Therefore, we need to identify which plants have the disease before the disease spreads to even more plants and eventually destroys the entire crop. Unfortunately, an expert cannot manually check all of the plants for the disease as there are too many plants. However, we can identify plants suffering from the disease by automatically taking images of all of the plants and aiming to classify them into the two classes *healthy plant* and *plant with disease* using a Convolutional Neural Network-based image classification model. To obtain the model, we first need to choose a model architecture consisting of a certain amount of model layers and initialize the weights of each model layer in some way. Then, an expert manually labels a few of our plant images with their respective classes. These labeled images are shown to the model for training, i.e., the labeled images are our training images for the model. During model training, we adjust the model weights over multiple iterations using the training images with the goal to improve the classification performance of the model. We train the model until the model achieves a sufficient classification performance on a test dataset. This test dataset consists of a few other labeled images that the model did not see during training. Chapter 2 gives more details about how to train a Convolutional Neural Network-based model. After training the model, we expect it to have learned an image feature representation of each class of the respective image classification problem. Then, we typically deploy the model to a production system (e.g., a mobile app). This production system can be used to classify all plant images in order to identify the plant disease.

However, obtaining and using a Convolutional Neural Network-based image classification model to solve a specific image classification problem presents several challenges. To train the model, we need to find good hyperparameter values for training, such as the model architecture, the initial values for the model weights, or specific hyperparameters for adjusting the weights during model training (e.g., the learning rate). The hyperparameters are critical to a successful training process. Therefore, it is important to find good values for them. However, finding good values for the hyperparameters is a non-trivial process. Moreover, good hyperparameter values for one classification problem are not necessarily good hyperparameter values for another classification problem. Thus, every time we aim to solve a specific image classification problem, it is important that we find good hyperparameter values for model training. A further issue with respect to model training is that the training data is often class-imbalanced in practice. A class-imbalanced dataset contains images that belong to a specific set of classes, but the images are not uniformly distributed among the classes. For instance, in our plant disease example, initially, only a few plants are affected by the disease. We aim to identify the plants affected by the disease as early as possible in order to prevent the disease from spreading to other plants. However, this also means that when we collect our training images, we only find a small number of images showing a plant affected by the disease. As a result, our collected training dataset may contain a large number of images showing a healthy plant but only a small number of images showing a plant with the disease. This class imbalance usually has a negative impact on model training. During training, the model might only be able to learn an adequate image feature representation of the class *healthy plant* but not the class *plant with disease* because the model sees only a small number of images of the *plant with disease* class during model training. However, not only model training can be challenging but also using the model in a production system after training. As part of the production system, the model might be applied to an image that was drawn from a data distribution that is different from the data distribution the training data was drawn from. The model did not see this kind of image during model training. We typically refer to such an image as an out-of-distribution sample. However, as the model did not see such an out-of-distribution sample during training, the model has not been able to learn an adequate image feature representation of that sample. Therefore, the model will most likely fail to predict the correct class without warning in this case, which is especially problematic for safety-critical applications such as driving assistance or medical diagnosis systems. This problem may also arise in our plant disease example. For instance, suppose the disease suddenly mutates and changes its visual appearance. However, our model has not yet learned anything about this new appearance. This results in the model no longer being able to recognize the disease.

In my work, I examined whether information from the model layers of a Convolutional Neural Network-based image classification model (pixels and activations, Figure 1.1) can be used to address all of the aforementioned challenges. I suggest a method for initializing the model weights based on image patches, a method for balancing a class-imbalanced dataset based on layer activations, and a method for detecting out-of-distribution samples, which is also based on layer activations. All of these methods search for clusters within the information obtained from the model layers (pixels and activations). The identified clusters are then exploited for the proposed methods in order to improve the classification process. More details on finding clusters within the layer information are presented in Chapter 3. To test the proposed methods, I conducted extensive experiments using different datasets. The experiments showed that layer information (pixels and activations) can indeed be used to address all of the aforementioned challenges when training and using Convolutional Neural Network-based image classification models. In Chapter 4, it is shown how image patches obtained from the training images can be used for initializing the model weights. To find suitable image patches, the proposed method uses clustering in image space of a set of candidate patches extracted from the training images (results published in Lehmann and Ebner [80]). However, not only the information obtained from the input layer of the model (i.e., the image pixels) are exploited but

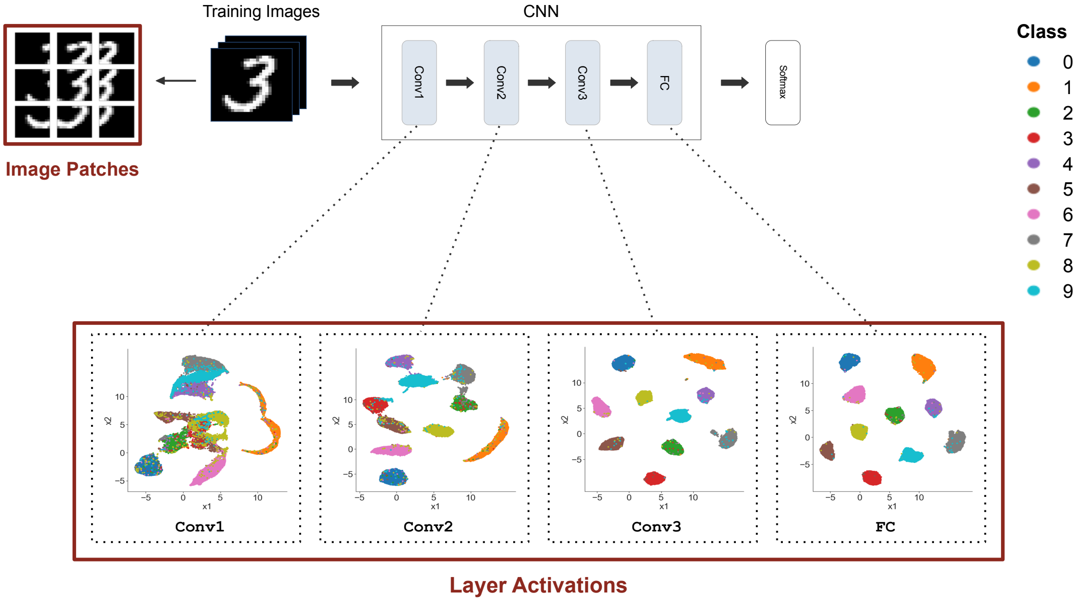


Figure 1.1: Considered types of exploiting layer information: The proposed approach uses the pixels from image patches extracted from the training images (e.g., from the MNIST [76] dataset) and the activations of the training images with respect to a Convolutional Neural Network-based image classification model (CNN) to improve the classification process.

also the information from other model layers. These other model layers contain the intermediate representations of an input image in image feature space towards classifying this image. The values of such an intermediate representation of the input image with respect to a specific model layer are the activations of the input image created at that layer. In Chapter 5, it is shown that clusters identified within the layer activations of a specific higher model layer can be used to address the class imbalance problem by balancing the training dataset before model training. Training a model with the balanced dataset should result in a model with a significantly better classification performance than training a model with the class-imbalanced training dataset (results published in Lehmann and Ebner [83]). Furthermore, it is shown in Chapter 6 that clusters identified within the activations of multiple model layers also help to identify out-of-distribution samples. Moreover, out-of-distribution samples can not only be detected using clusters obtained from the layer activations but can also be detected quickly. This is important for safety-critical applications running in real-time, such as a driving assistance system, which usually does not have much time to prevent the car from having an accident (results presented in three publications [81, 82, 84]). The following contributions have been made: (1) It was shown that information obtained from the layers of a Convolutional Neural Network-based image classification model (pixels and activations) can be used to improve the overall classification process, (2) a method was suggested that initializes the model weights using clusters identified in image space (i.e., the pixels) of image patches extracted from the training images, (3) a method was proposed that addresses the class imbalance problem by balancing the training dataset using clusters identified within the activations from a higher model layer, (4) a method was introduced that is based on identifying clusters within the activations of multiple model layers in order to detect out-of-distribution samples, and (5) the suggested methods were evaluated through extensive experiments on multiple datasets. The code of my work can be found on GitHub[1].

---

[1] `https://github.com/bam098/dissertation`

2 Fundamentals of Convolutional Neural Network-based Image Classification

Before explaining how the information from the layers of a Convolutional Neural Network-based (CNN) image classification model (pixels and activations) can be used to improve the classification process, this chapter provides an overview of the fundamentals of these models. Chapter 2.1 describes how Convolutional Neural Network-based image classification models are constructed using a certain set of model layers. After constructing such a model, Chapter 2.2 explains how to train the model to solve a specific image classification problem. Finally, Chapter 2.3 describes what the model learned during model training in order to classify images. More detailed information about the fundamentals of Convolutional Neural Network-based image classification models can be found in Goodfellow et. al. [40], Aggarwal [2], or Howard and Gugger [53].

2.1 Model Architecture

A Convolutional Neural Network-based (CNN) image classification model consists of an input layer, multiple hidden layers, and an output layer (as illustrated in Figure 2.1). The input layer is the image that is fed into the model as input, while the output layer contains the classification result with respect to that input image. The hidden layers contain the intermediate representations of the input image that are required by the model in order to obtain the classification result at the output layer. The values of the intermediate image representations with respect to the hidden layers are the activations of these layers. A Convolutional Neural Network-based image classification model typically requires multiple intermediate image representations in order to obtain the classification result. Each intermediate image representation is created by a different hidden layer. The number and type of the required hidden layers, however, depend on the image classification problem that needs to be solved. Nevertheless, the set of hidden layers usually has a certain structure. The model typically contains multiple consecutive convolutional hidden layers followed by one or more optional linear hidden layers. Each convolutional layer contains a specific intermediate image representation in the form of a three-dimensional activation tensor, while each linear layer contains a specific intermediate image representation in the form of an activation vector. Furthermore, each of the hidden layers (convolutional and linear) contains weights. The values of these weights are learned during model training with respect to the classification objective of the model [2], as described in Chapter 2.2.

The weights of a convolutional layer are structured into multiple filters (also referred to as kernels). A filter is a three-dimensional tensor containing a subset of the weights of the respective convolutional layer. The first two dimensions of this tensor reflect the size of the filter. Typical filter sizes are 3×3, 5×5, or 7×7. The third dimension depends on the depth of the activation tensor of the previous convolutional layer, or the depth of the input image if the previous layer is the input layer (depth of grayscale images: 1, depth of color images: 3) [53]. After model training, each filter of a convolutional layer detects a specific image feature of the input image. Zeiler and Fergus [148] showed that the filters of the lower convolutional layers (i.e., the layers closer to the input layer) detect low-level image features (e.g., edges, corners, simple textures), while the filters of the higher convolutional layers (i.e., the layers closer to the output layer) detect high-level image features (e.g., object parts, objects in various poses). The filter size and the number of filters of each convolutional layer depend on the classification problem that needs to be solved. Therefore, size and number need to be set as hyperparameters for each convolutional layer for model training. The weights of all filters of a convolutional layer result in the total amount of weights of that layer [53].

When an image is fed into the model, each filter of the first convolutional layer is applied to each pixel of that image using the convolution operation [30]. By applying the filters to each pixel of the image, we obtain a matrix of values for each of the filters. The values of this matrix are the pre-activation values [2] of the layer with respect to the respective filter. Each obtained matrix of pre-activations has the same height and width as the input image if the convolution operation was applied to every pixel of the input image. All pre-activation matrices together form the three-dimensional pre-activation tensor of the layer. Each pre-activation matrix is also referred to as a channel of this tensor. The total number of channels of the tensor reflects its depth. After obtaining the tensor of pre-activations, we then apply an activation function, such as the ReLU activation function (Rectified Linear Unit) [39], to the pre-activations. The ReLU activation function simply sets all negative pre-activation values to 0. The resulting values are the post-activation values [2] of the layer. Below, the post-activation values of a layer are simply referred to as the activations of that layer. After obtaining the activation tensor of the first convolutional layer, we compute the activation tensor of the second convolutional layer. To obtain the activation tensor of the second convolutional layer, the filters of that layer are applied to each activation obtained from the first convolutional layer using the convolution operation in the same way as the filters of the first convolutional layer were applied to the pixels of the input image. As a result, we obtain the tensor of activations of the second convolutional layer. The activations of all following convolutional layers are computed in the same way [40, 53].

However, the filters of a convolutional layer do not need to be applied to every pixel or activation of the previous layer using the convolution operation. Certain pixels or activations can be skipped. If we apply the filters only to every second pixel or activation using the convolution operation, we reduce the resolution of the pre-activation matrices by half and thus reduce the final activation tensor of the layer by half compared to the activation tensor of the previous layer. For instance, the width and height of the activation tensor of the first convolutional layer would be only half the size of the input image in this case. When going from the first to the last convolutional hidden layer, the resolution of the image representations (i.e., the activation tensors) is usually decreased to be able to detect increasingly complex image features. Due to the resolution decrease, an activation of a specific higher layer (i.e., a layer closer to the output layer) corresponds to a region of the input image in image feature space that is larger than an image pixel. This region is referred to as the receptive field of the layer with respect to the input image. The receptive field of a higher layer could encompass a high-level image feature present in the image, such as a characteristic object part. An activation of a lower layer (i.e., a layer closer to the input layer), however, corresponds to a receptive field with respect to the input image that is larger than an image pixel but smaller than the receptive field of a higher layer. Thus, a lower layer is usually able to detect low-level image features in the image, such as simple textures. To obtain the different receptive fields, we must reduce the resolution of the activation tensors among the different convolutional layers when going from the first to the last convolutional hidden layer. One way to reduce the resolution is the aforementioned skipping of pixels or activations when applying the convolution operation. The number of skipped activations or pixels is called stride. The stride depends on the image classification problem we aim to solve and is therefore a hyperparameter that needs to be set for model training. However, there is also a second option to reduce the resolution of the activation tensors. We can apply the pooling operation [110] to them. The pooling operation moves over each position of each channel of an activation tensor using a sliding window (e.g., of size 2×2 for reducing the resolution by half). In order to reduce the resolution, either the average or the maximum from the activations within the sliding window is taken at each position. However, the stride is usually preferred over pooling nowadays as it works better for modern Convolutional Neural Network-based image classification models [40, 53].

After the convolutional layers, we can have one or more optional (fully-connected) linear layers. A linear layer contains its activations in the form of an activation vector. Each activation of this activation vector is computed by a linear combination of all activations of the previous linear layer, followed by applying an activation function (e.g., ReLU) to the result of this linear combination. The weights of the linear combination used to compute the respective activation are the model weights associated with that activation [2]. If the previous layer is the last convolutional layer, however, we first need to convert the three-dimensional activation tensor of the convolutional layer into an activation vector. There are several ways to convert the tensor into a vector. We could simply flatten the activation tensor. However, this is not a good approach as the model then only accepts input images with a fixed size because input images of different sizes result in different-sized activation tensors, which lead to different-sized activation vectors. Another approach to converting the three-dimensional activation tensor into an activation vector is using a global average pooling layer [88] between the last convolutional layer and the first linear layer. A global average pooling layer averages the activations from the last convolutional layer along the width and height of the activation tensor but not along its depth. As a result, we obtain a vector of activations of size $1 \times 1 \times depth$ from the last convolutional layer. This approach is independent of the size of the input images. The third option is to incrementally reduce

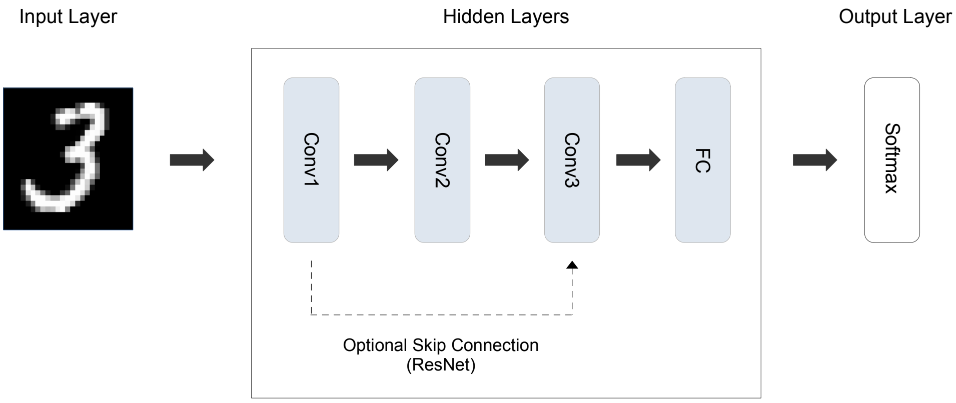


Figure 2.1: The general model architecture of a Convolutional Neural Network-based (CNN) image classification model consisting of the input layer (e.g., an MNIST [76] image), multiple consecutive convolutional hidden layers (Conv) followed by one or more optional linear fully-connected hidden layers (FC), and the output layer.

the resolution of the last convolutional layer to a size of $1 \times 1 \times depth$ by stride or pooling, which provides the last convolutional layer in its vector form. This approach is also independent of the size of the input image. We finally apply the weights of the following first linear layer in a linear combination to the resulting activation vector followed by applying the activation function in order to obtain the activations of that linear layer. After obtaining the activations from the first linear layer, we compute the activations of the following linear layers in the same way. Finally, we apply the softmax function (instead of ReLU) to the result of the linear combination of the last linear layer to obtain the classification result. The resulting values reflect the classification scores of the output layer. The highest resulting classification score corresponds to the predicted class with respect to the image that was fed into the model [40, 53].

Above, the architecture of a standard Convolutional Neural Network-based model is described. Standard models such as these, with few hidden layers, are usually well suited to solving simple image classification problems. However, in order to be able to solve complex problems, we typically need a large number of hidden layers. Unfortunately, training a standard Convolutional Neural Network-based image classification model does not work well if the model contains such a large number of hidden layers. Thus, He et. al. [46] suggested the ResNet (Residual Network) model architecture. The ResNet model architecture contains additional skip connections (Figure 2.1). At some locations in the model architecture, the current layer output is combined with the output of a layer that is further back in the model architecture, i.e., a few layers are skipped. Hereinafter, the skipped layers will be referred to as a ResNet block. We usually add skip connections at multiple locations in the model architecture. Adding the skip connections has a positive effect on model training [53]. Finally, after defining the model architecture, we can train the model using our training dataset (Chapter 2.2).

2.2 Model Training

After specifying the architecture of the Convolutional Neural Network-based (CNN) image classification model (Chapter 2.1), we need to train the weights of the model using a set of training images until the model achieves a sufficient classification performance on a given test dataset. Once the model is trained, we can use the trained model for solving the respective image classification problem. Such an image classification problem requires classifying images into a fixed set of classes. However, instead of using a Convolutional Neural Network-based model to solve the image classification problem, it would be easier to simply separate the images linearly in pixel space according to the respective classes. Unfortunately, this is not possible because images are high-dimensional spaces. Consequently, they cannot be separated linearly in pixel space according to their classes. A Convolutional Neural Network-based model, therefore, attempts to incrementally change the representation of the images in multiple steps to eventually find a representation of the images that is linearly separable. The model finds this linearly separable image representation through multiple intermediate image representations. Each intermediate image representation is stored in a certain hidden layer of the model in the form of the activations of that layer, while the final model layer should contain the desired linearly separable image representation. In order to find the linearly separable image representation in the final model layer, we need to train the model using an iterative optimization algorithm (e.g., Adam[1] [65]) for a sufficient number of training epochs. The optimization algorithm searches for the optimal weights of the model in an iterative process. Optimal (or near-optimal) weights are required to find the intermediate and final linearly separable image representation [40, 53].

To train the model, we need to set several hyperparameters for model training. First of all, we must specify the model architecture (for more information, see Chapter 2.1). After specifying the architecture, we then need to set the initial values of the model weights. Initializing the model weights is critical to the training time. State-of-the-art methods to set the initial weight values are based either on random values or on a pre-trained model (for more information, see Chapter 4.1). Additionally, I suggest an alternative weight initialization method in Chapter 4.2, which is based on image patches extracted from the training images. In addition to the model architecture and initial weights, we also need to set certain hyperparameters for model training that specify how the weights should be adjusted during the training process. The most important of these hyperparameters is the learning rate. The learning rate specifies how much the model weights should be adjusted at each training step, which is critical to the training process. If we set the learning rate too high, we might not even get close to the optimal weight values because we will reach a point where the optimal weight values are between the weight values of the current training epoch and the weight values of the previous training epoch. If we set the learning rate too small, it may take us a long time to reach the optimal weight values, and we may even get stuck in saddle points or even local minima during the training process. Thus, it is important to find a good value

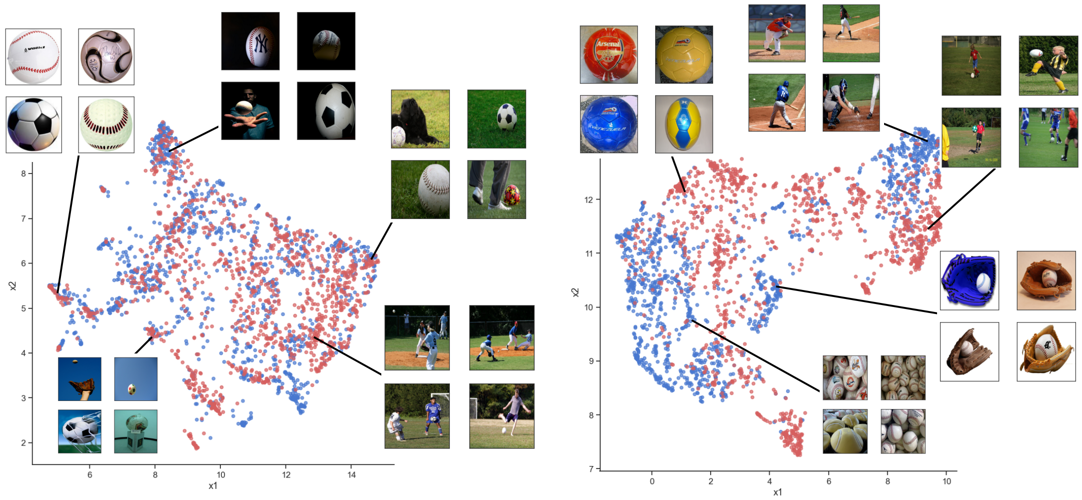
---

[1] `https://pytorch.org/docs/stable/generated/torch.optim.Adam.html`

for the learning rate. Furthermore, it may also be beneficial to adjust the learning rate during model training using a learning rate schedule (e.g., a cosine-annealing learning rate schedule [91]) or to use different learning rates for different layers (discriminative learning rates) [54]. In addition to the learning rate, there are other hyperparameters that are used to adjust the model weights, such as the momentum and the weight decay. However, the learning rate usually has the strongest impact on training [40, 53].

Along with setting the training hyperparameters, we also need to ensure that we use a sufficient training dataset for model training. In practice, it can be difficult to collect and label a sufficient number of training images. Therefore, to add more variation to our training dataset, we can extend it by using data augmentation [128]. Data augmentation uses image processing techniques (e.g., cropping, flipping, rotation, adjusting brightness) to create additional training images by adjusting the original training images. These additional training images differ slightly from the original training images. Thus, by using data augmentation, our training dataset obtains a wider range of images with respect to each class of the image classification problem. This is important because our model needs to see each class in different forms in order to learn sufficient image feature representations of the classes [53]. However, if our training dataset is class-imbalanced, our model will not be able to learn such a sufficient image feature representation for at least some of the classes. A class-imbalanced training dataset does not contain an approximately equal number of images from all classes. This has a negative impact on model training [104]. Unfortunately, training datasets are often class-imbalanced in practice. As a result, we need to address the class imbalance problem before model training. In Chapter 5.2, I suggest a method to address this problem, which is based on exploiting the layer activations of a higher hidden layer of the model.

After obtaining a sufficient training dataset, we group the training images of that dataset into mini-batches and show the model all obtained mini-batches for several training epochs in order to train the model. During training, the model adjusts its weights each time it sees a mini-batch with the aim of incrementally separating the training images from the first to the final model layer according to their classes. Training images of the same class are increasingly pushed together in activation space of the first to the last hidden layer, while training images of different classes are increasingly pushed apart. We train the model until the model achieves a sufficient classification performance on a given test dataset. After model training, the obtained model weights should transform the input images of the model into the intermediate image representations in activation space of the hidden layers and into the final linearly separable image representation in activation space of the last model layer [40, 53]. Zeiler and Fergus [148] showed that the final weights after model training result in an image representation in activation space of each hidden layer corresponding to the image features that the model learned to recognize from the training images. The weights of the lower model layers (i.e., the layers closer to the input layer) detect low-level image features (e.g., edges, corners, simple textures), while the weights of the higher model layers (i.e., the layers closer to the output layer) detect high-level image features (e.g., object parts, objects in various poses). Finally, the trained model can then be applied to novel images for inference (Chapter 2.3).



Lower Layer

Higher Layer

Figure 2.2: The images of the ImageNet [25] classes *soccer ball* (red) and *baseball* (blue) are arranged in image feature space of the lower layers (left) according to low-level image features such as color distributions (e.g., black background, white background, grass background), while they are arranged in image feature space of a higher layer (right) according to high-level image features such as characteristic object parts (e.g., baseball glove, baseball player, soccer player, soccer ball).

2.3 Model Inference

After model training (Chapter 2.2), we usually deploy the obtained model to a production system (e.g., a mobile app, a web server) in order to classify novel images for inference. After feeding such a novel image into the model, each layer of the model tries to find specific image features in the image in order to predict the correct class of the image. Zeiler and Fergus [148] showed that the lower model layers (i.e., the layers closer to the input layer) detect low-level image features (e.g., edges, corners, simple textures), while the higher model layers (i.e., the layers closer to the output layer) detect high-level image features (e.g., object parts, objects in various poses). The images fed into the model are therefore arranged in image feature space of a specific hidden layer (in the form of its activations) according to the type of image features the layer detects. As shown in Figure 2.2, a lower model layer arranges the images according to low-level image features, such as the overall color distributions of the images, while a higher model layer arranges the images according to high-level image features, such as characteristic object parts. From layer to layer, the model tries to find image features that become increasingly class-specific. At the final layer, the model then tries to detect the image object for predicting the class of the image. As a result, a novel image fed into the model should be located in image feature space of each model layer according to the image features the image contains. As a result, the image should be close to other images of the same class, which should have similar image features [109].

However, if a novel image of a certain class was drawn from a data distribution that is different from the data distribution the training data of the model was drawn from, the novel image may be located near training data samples of a different class in image feature space of the lower as well as the higher model layers and is therefore misclassified without warning. This type of image is commonly referred to as an out-of-distribution sample. Out-of-distribution samples can occur naturally [51, 108], or they can be created artificially [41, 59, 135] by an attacker (for more information, see Chapter 6). Out-of-distribution samples pose a serious threat, especially when using the model for safety-critical applications (e.g., driving assistance systems, medical diagnosis systems). Data augmentation [128] can help make the model more robust against out-of-distribution samples [48, 146]. However, this does not work in all cases. As a result, out-of-distribution samples need to be detected as quickly as possible. In Chapter 6.2, I propose a method for detecting out-of-distribution samples that is based on the location of an image in image feature space of various model layers.

3 Fundamentals of Identifying Clusters in Model Layer Information

After giving an overview of the fundamentals of Convolutional Neural Network-based (CNN) image classification models in Chapter 2, I will briefly describe my general approach to identifying clusters within the information from the layers of such a model (pixels or activations) in this chapter. My suggested methods for improving the classification process in various ways require these clusters, as described in Chapter 4, Chapter 5, and Chapter 6. In order to obtain the clusters at a specific model layer, the proposed approach requires three steps. First, the information from the layer needs to be compressed by reducing its dimensionality, as described in Chapter 3.1. Without reducing the dimensionality, it would most likely not be possible to find meaningful clusters within the layer information. After compressing the layer information using dimensionality reduction, clusters need to be identified within the compressed information, as described in Chapter 3.2. However, not all of the obtained clusters are useful for my proposed approach. Therefore, the obtained clusters need to be evaluated in a final step in order to find the useful clusters, as described in Chapter 3.3. Finally, the obtained useful clusters are exploited in order to improve the classification process.

3.1 Dimensionality Reduction

The goal is to identify clusters within the information of a specific model layer l (pixels or activations). This information consists of a set of input images in the pixel or activation space of the layer. My suggested approach to finding the clusters first stores the information from layer l in a matrix A^l of size $N \times M$. The rows of A^l represent the N input images, while the columns represent the M pixels or activations of the images with respect to layer l. However, before searching for clusters within matrix A^l (i.e., the layer information), the dimensions of A^l need to be reduced. Reducing dimensionality is necessary as the layer information in A^l in the form of pixels (from the input layer) or activations (from the hidden layers) is usually high-dimensional, and identifying clusters in high-dimensional spaces does not work well, as pointed out by Chen et. al. [19]. Clustering algorithms use distance metrics to identify clusters, but distance metrics are not effective in high-dimensional spaces. However, as Domingos [28] highlights, the data samples of most applications are located within a low-dimensional subspace within such a high-dimensional space. Thus, Chen et. al. [19] suggested to use dimensionality reduction to project matrix A^l onto such a low-dimensional subspace before searching for clusters. As a result, A^l is projected to two dimensions. However, before projecting A^l, each of its values needs to be normalized as a preprocessing step for the dimensionality reduction. A normalized value is referred to as the z-score[1] of the value. Suppose $A^l_{i,j}$ is the value in the i^{th} row and the j^{th} column of matrix A^l. To normalize this value, the mean μ_j over all values of the j^{th} column of A^l need to be subtracted from $A^l_{i,j}$, and then $A^l_{i,j}$ needs to be divided by the standard deviation σ_j over all values of the j^{th} column of A^l. As a result, the normalized value $z(A^l_{i,j})$ is obtained (Equation 3.1). This normalization step is applied to all values of A^l. Hereinafter, it is assumed that matrix A^l contains the normalized values $z(A^l_{i,j})$ instead of the non-normalized values $A^l_{i,j}$.

$$\begin{aligned} z(A^l_{i,j}) &= \frac{A^l_{i,j} - \mu_j}{\sigma_j} \\ \mu_j &= \frac{1}{|A^l_{:,j}|} \sum_i A^l_{i,j}, \quad \sigma_j = \sqrt{\frac{\sum_i (A^l_{i,j} - \mu_j)^2}{|A^l_{:,j}| - 1}} \end{aligned} \tag{3.1}$$

Once each value of matrix A^l has been normalized, dimensionality reduction is used to reduce the dimensions of the matrix from $N \times M$ to $N \times 2$. In general, I achieved good projection results with the non-linear dimensionality reduction technique UMAP (Uniform Manifold Approximation and Projection) [98]. UMAP constructs a high-dimensional graph from the data in order to approximate its topology. Then, UMAP maps the resulting high-dimensional graph to a low-dimensional graph in order to project the data onto a low-dimensional space. However, if matrix A^l is huge, we may not receive a sufficient projection result. UMAP does not always work well when the dimensionality is too high. Moreover, UMAP usually requires a long computation time and high memory consumption when applied to a huge amount of data. Thus, in order to project larger matrices, I use a combination of UMAP and the linear dimensionality reduction technique PCA (Principal Component Analysis) [116]. PCA finds the principal components within the data. The principal components are variables that are constructed by linear combinations of the variables of the original data (e.g., pixels or activations). However, the principal components are constructed in such a way that the first principal components contain the most information, while the last principal components contain the least information. As a result, when only the first principal components are used (in this case the first 2 to obtain a two-dimensional space), we do not lose too much information because the first principal components contain the most information. The selected first principal components form the projected layer information. To combine PCA with UMAP in order to project A^l, I use a similar approach to the method suggested by Nguyen et. al. [109]. First, the layer information is reduced from $N \times M$ to $N \times 50$ using PCA, and then it is further reduced from $N \times 50$ to $N \times 2$ using UMAP. This combination of PCA and UMAP achieved the best results in my comparison of different dimensionality reduction approaches, as shown in Chapter 6.3.2. The resulting reduced matrix contains the compressed pixels or activations of each image.

---

[1] `https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html`

3.2 Cluster Analysis

After obtaining the compressed information from a specific model layer (compressed pixels or activations) through dimensionality reduction (Chapter 3.1), clusters are searched within this compressed layer information. The compressed layer information consists of the set of input images in the compressed two-dimensional pixel or activation space of the respective layer. To search for the clusters within this compressed layer information, I use the k-Means [95] clustering algorithm. I chose k-Means for identifying the clusters as I obtained the best clustering results using k-Means in my experiments (Chapter 6.3.2). k-Means requires us to set the hyperparameter k, which specifies the number of clusters that k-Means should search for in the data. First, k-Means places k cluster centers randomly within the compressed pixel or activation space of the respective layer. Then, each image data sample in that space is assigned to the nearest cluster center. Once all image data samples have been assigned to their nearest cluster center, each cluster center is moved to the center of its assigned image data samples by setting the cluster center to the average over all assigned image data samples in the compressed pixel or activation space of the respective layer. These last two steps are then repeated until the assignment of the image data samples to the cluster centers no longer changes. The final assignments are the identified clusters.

3.3 Cluster Evaluation

Chapter 3.2 describes how to identify clusters within the information of the model layers (pixels or activations) using the k-Means [95] clustering algorithm. However, k-Means requires us to set the hyperparameter k. Unfortunately, it is not obvious how to set k. Thus, I simply identify the best value for k by testing different values. The clustering result obtained from each tested k is evaluated using a cluster quality metric. The value for k that results in the clusters with the best cluster quality is chosen. The silhouette score [121] is used as the cluster quality metric in order to evaluate the obtained clusters. Chen et. al. [19] reported that the silhouette score is well suited to evaluating clusters identified in activation data of Convolutional Neural Network-based (CNN) models. The silhouette score is calculated using the intra-cluster distance d_{intra} and the nearest-cluster distance d_{near} of each image data sample of the dataset. To calculate the silhouette score $silscr(u_i)$ of an image data sample u_i, it is first necessary to measure the distance $d(u_i, u_j)$ from that image data sample u_i to other image data samples u_j that are located in the same cluster h_u as u_i ($u_i \in h_u, u_j \in h_u, u_i \neq u_j$). Then, the mean of these distance measurements is calculated to receive the intra-cluster distance $d_{intra}(u_i)$ of sample u_i (Equation 3.2).

$$d_{intra}(u_i) \;=\; \frac{1}{|h_u| - 1} \sum_{u_j \in h_u, u_i \neq u_j} d(u_i, u_j), \quad u_i \in h_u \tag{3.2}$$

After determining the intra-cluster distance $d_{intra}(u_i)$ of image data sample u_i, the distance $d(u_i, v_i)$ from that image data sample u_i to image data samples v_i of the nearest cluster h_v to cluster h_u ($u_i \in h_u, v_i \in h_v$) is measured. Then, the mean of these distance measurements is calculated to receive the nearest-cluster distance $d_{near}(u_i)$ of sample u_i (Equation 3.3).

$$d_{near}(u_i) \;=\; \min_{h_u \neq h_v} \frac{1}{|h_v|} \sum_{v_i \in h_v} d(u_i, v_i), \quad u_i \in h_u \tag{3.3}$$

After determining the intra-cluster distance $d_{intra}(u_i)$ and the nearest-cluster distance $d_{near}(u_i)$ of image data sample u_i, the silhouette score $silscr(u_i)$ of sample u_i can be calculated (Equation 3.4).

$$silscr(u_i) \;=\; \begin{cases} \frac{d_{near}(u_i) - d_{intra}(u_i)}{max\{d_{intra}(u_i), d_{near}(u_i)\}} & \text{if } |h_u| > 1 \\ 0 & \text{if } |h_u| = 1 \end{cases}, \quad u_i \in h_u \tag{3.4}$$

The silhouette score is calculated for all image data samples of the dataset using the procedure described above. After determining the silhouette score for each image data sample, the mean over all obtained silhouette scores is calculated to receive the total silhouette score for the obtained clusters. The value of the resulting total silhouette score ranges from -1 to 1. A silhouette score close to 1 means that the identified clusters are well-separated because in this case we mainly have $d_{intra}(u_i) \ll d_{near}(u_i)$. A silhouette score close to -1, however, means that the majority of the image data samples should be located in the nearby cluster rather than in their current cluster. Thus, a higher silhouette score reflects a better cluster quality and is therefore favorable. This results in selecting the k value for k-Means that leads to the clusters achieving the highest silhouette score.

4 Exploiting Image Patches to Initialize the Model Weights

As already shown in Chapter 2.2, training a Convolutional Neural Network-based (CNN) image classification model is a non-trivial process. To train such a model, we need to find a good training setup, which involves finding good initial values for the weights of the model and finding good values for the hyperparameters of the training process itself. These values are not trivial to find. However, information extracted from the training images of the model can be exploited to improve the process of finding good values for the model weights and the training hyperparameters. Before going into the details about how this information can be exploited, however, I need to recap the training process itself. During model training, we aim to find optimal values for the model weights with respect to the classification objective of the model. Unfortunately, these optimal weight values are unknown. To obtain the optimal weight values or weight values that are at least close to the optimal weight values, we need to use an iterative optimization algorithm [120, 122]. We first set each model weight to an initial value. After setting the initial values for the weights, we measure the current loss of the model with respect to the training data samples. The loss describes how bad the model predictions are for the labels of the training data samples, when using the model with the current weight values. Then, we adjust the weight values in multiple training epochs to minimize the loss. In each epoch, we update the values of the weights towards their optimal values using our optimization algorithm. The optimization objective is to obtain the model that reaches zero loss. The weight values of this model are the optimal weight values. In practice, however, we usually do not reach zero loss. Nevertheless, we aim to minimize the loss as much as possible by adjusting the weight values of the model without overfitting the model on the training dataset. To control how much the weight values get adjusted in each training epoch, we need to set different training hyperparameters beforehand. The most important of these training hyperparameters is the learning rate. The learning rate specifies how much we adjust the weight values. Finding a good value for the learning rate is important. If we set the learning rate too high, we will not even get close to the optimal weight values because we will reach a point where the optimal weight values are between the current weight values and the weight values of the previous training epoch. If we set the learning rate too small, we may need a long time to reach the optimal weight values, and we may even get stuck in saddle points or even local minima during the optimization process. Unfortunately, similar to the initial weight values, it is unknown how to set the training hyperparameters, such as the learning rate, in advance. Furthermore, the training hyperparameters and the initial weight values usually differ between different image classification problems [53].

However, we aim to set the initial weight values at least as close as we can to the optimal weight values in order to keep the number of required training epochs as low as possible. If we set the initial weight values close enough to the optimal weight values, we can even use a lower learning rate because we will not need a high number of training epochs anymore. Furthermore, the risk of getting stuck in saddle points or even local minima will be reduced as well. To set the initial weight values close to the optimal weight values, we usually use the weight values from a pre-trained model [154]. Normally, such a pre-trained model was trained on the ImageNet [25] dataset. The ImageNet dataset contains images showing a natural image object (e.g., an orange, a baseball) in front of a natural image background. If the training images of our current image classification problem are similar to the ImageNet training images, then the weight values of this pre-trained model are usually good initial weight values for the model that we aim to train in order to solve our image classification problem. We expect the weight values of the pre-trained model to be close to the optimal weight values of our current image classification problem as our image classification problem is similar to the ImageNet classification problem. If the training images of our current image classification problem are significantly different from the ImageNet training images (e.g., medical images), however, using the weight values of the pre-trained model is most likely not beneficial for training our model. In this case, the pre-trained model weights are probably too far from the optimal model weights of our current classification problem. Therefore, we could alternatively pre-train our model manually using self-supervised learning [32]. Self-supervised learning uses an auxiliary optimization objective for the pre-training to obtain the initial weight values. These initial weight values are then fine-tuned using our classification objective. More information about model pre-training is given in Chapter 4.1. Nevertheless, we still need to find initial weight values at least for the pre-training. To find these initial weight values, we cannot use any model training, i.e., we need to train the model from scratch. State-of-the-art methods to find these initial weight values are based on random values in combination with information about the model architecture [38, 45]. More details about these state-of-the-art methods are given in Chapter 4.1 as well. However, the state-of-the-art methods only use a general approach to finding good initial values for the model weights. They do not take into account information about the classification problem, such as the training data samples. Thus, we may still need a long training time because the obtained initial weight values might still be far from the optimal weight values of our current classification problem.

Moreover, it is not clear what the model learns during model training, i.e., how we get from the initial weight values to the final weight values throughout model training. Therefore, not only for reasons of efficiency but also for reasons of explainability, it would be better to set the initial weight values close to the optimal weight values. If we set the initial weight values for at least some model layers close to the optimal weight values, the weight values of those layers only need little or no adjustment at all. In this case, we obtain a simplified training process, as the weights of these layers do not need much training anymore. Furthermore, the model becomes more explainable, at least with respect to the layers that require little or no training at all, since we no longer need to find out what those layers learned during model training when their weights were adjusted in a certain way. We only need to interpret the initial setting of the weight values. As a result, avoiding the need to adjust the weights of at least some model layers, would be a first step towards making Convolutional Neural Network-based models more explainable. The explainability of Convolutional Neural Network-based models is still an active area of research [7, 85, 144]. By explaining how the model classifies a data sample, it would be easier to detect when the model fails to classify that sample. This is especially important when using Convolutional Neural Network-based image classification models for safety-critical applications such as medical diagnosis systems.

I proposed a method that initializes the model weights based on image patches extracted the from the images of the training dataset of the model (published in Lehmann and Ebner [80]). The details of the proposed method are presented in Chapter 4.2. By using information about the current classification problem in the form of image patches extracted from the training images, I expected the method to set the initial values for the model weights in such a way that fewer training epochs are needed to train the model. My first research goal, however, was to examine if image patches can be used to initialize the model weights at all. My second research goal was then to compare the proposed weight initialization method with different state-of-the-art methods for initializing the model weights. In my experiments (Chapter 4.3), I showed that image patches extracted from the training images can be used to set the initial weights of the model. A model, whose weights were initialized by my proposed method, even reached a similar classification performance in my experiments as a model, whose weights were initialized by a state-of-the-art method, when using an optimal learning rate value. Furthermore, initializing the model weights using my proposed method makes the choice of the learning rate for model training more robust. If we chose a smaller learning rate for model training (i.e., a suboptimal value), the model initialized by my proposed method needed fewer training epochs to achieve a certain classification performance compared to a model initialized by a state-of-the-art method. The following contributions have been made: (1) A method was suggested for initializing the weights of a Convolutional Neural Network-based model that is based on information extracted from the training images of the model, and (2) it was shown that using the proposed method makes the choice of the learning rate more robust with respect to the training process.

4.1 Related Work on Model Weight Initialization

To initialize the weights of a Convolutional Neural Network-based (CNN) image classification model, we typically use the weight values of a pre-trained model. This pre-trained model needs to have the same model architecture as our current model. Only the output layer of our current model may differ from the output layer of the pre-trained model as the pre-trained model was usually trained with respect to a different classification problem with a different number of classes compared to our current classification problem. Such a pre-trained model is typically trained on a huge training dataset such as ImageNet [25]. After initializing our current model with the weight values of the pre-trained model, we train our model with respect to its respective classification objective. Training a model whose weights have been initialized with the weight values of a pre-trained model is referred to as transfer learning [154]. However, using the weight values of a pre-trained model to initialize the weights of our current model usually works well only if our current classification problem is similar to the classification problem of the pre-trained model. If our training images are significantly different from the training images of the pre-trained model, however, then using the weight values from the pre-trained model may not be too beneficial, as pointed out by Raghu et. al. [118] (e.g., medical images in comparison to ImageNet images). In this case, we could alternatively pre-train our current model manually using only the training images of our current classification problem. To pre-train our current model, we use an auxiliary optimization objective instead of our actual classification objective. One potential auxiliary optimization objective is learning to reconstruct the input images. Therefore, we must replace the classification objective of our current model with a reconstruction objective. After pre-training, the obtained model weights are used as the initial weights to train our model using its actual classification objective. As a result, the training setup of the model must be adjusted again to change the reconstruction objective back to the classification objective. Then, we train our model using its actual classification objective. This pre-training technique was initially referred to as unsupervised pre-training [10, 31]. It was later summarized under the term self-supervised learning [9, 32], along with other pre-training techniques (e.g., colorization [75, 152], context prediction of image patches [27, 111], inpainting [115], contrastive learning [63]). However, in order to obtain a pre-trained model, we still need to find the initial weight values at least for pre-training. Therefore, it is still important to find good initial weight values without any model training, i.e., the model needs to be trained from scratch. This is the focus of my work.

State-of-the-art methods for initializing the model weights (without any model training) are techniques that are based on random values and information about the model architecture. Glorot and Bengio [38], for instance, proposed the Xavier[1] initialization method. To initialize a specific weight at a particular model layer, Xavier initialization selects a value from a uniform distribution and initializes the weight with the selected value. They consider a uniform distribution that is located within an interval around 0. The bounds of the interval are determined by the number of activations of the current model layer, to which the weight that should be initialized belongs, and by the number of activations of the previous model layer. The number of activations of the previous model layer is equal to the number of inputs to each activation of the current model layer. Xavier initialization is usually well suited for Convolutional Neural Network-based image classification models that use the tanh (hyperbolic tangent) or the softsign activation function. However, modern Convolutional Neural Network-based image classification models rather use the ReLU activation function (Rectified Linear Unit) [39]. Therefore, He et. al. [45] proposed the Kaiming initialization method, which works better for models with the ReLU activation function. The Kaiming method initializes each model weight by selecting a value from a normal distribution whose standard deviation is determined either by the number of activations of the previous model layer or by

---

[1] `https://pytorch.org/docs/stable/nn.init.html#torch.nn.init.xavier_uniform_`

the number of activations of the current model layer. An alternative to the Kaiming method that uses a normal distribution is a variation of Kaiming that uses a uniform distribution that is located within an interval around zero in order to select the values for the model weights. The bounds of that interval are calculated using either the number of activations of the previous model layer or the number of activations of the current model layer. Hereinafter, I refer to the Kaiming method that uses a normal distribution as Kaiming Normal[2] and to the Kaiming method that uses a uniform distribution as Kaiming Uniform[3]. Furthermore, for models that are based on a ResNet model architecture (Residual Network) [46], Zhang et. al. [150] suggested an extension of Xavier and Kaiming named Fixup. Fixup uses either Xavier or Kaiming initialization along with a special scaling of the weight values in the residual branches of the model (i.e., the skipped layers, Figure 2.1). This scaling speeds up model training and is an alternative to using batch normalization layers [61] (patent held by Google LLC [60]) within the model architecture, which have also been introduced to increase the speed of model training. However, Fixup, Kaiming and Xavier rely only on random values along with information about the model architecture. None of these techniques uses information about the current classification problem. My proposed patch-based initialization method, on the other hand, uses information about the classification problem in the form of patches extracted from the training images. I claim that this information is beneficial for the training process.

Furthermore, there have also been other studies suggesting alternative methods to initialize the model weights. Gray et. al. [42], for instance, proposed a method to find block-sparse weight matrices for the model layers. Saxe et. al. [126], on the other hand, suggested initializing the weight matrices with orthonormal matrices. This technique was later extended by Mishkin and Matas [102]. Castillo and Wang [17] introduced an initialization technique for image forensics that initializes the weights of the first convolutional model layer as a high-pass filter. Dauphin and Schoenholz [24] proposed a method that finds weight values through a meta-model. Seuret et. al. [127] suggested using PCA (Principal Component Analysis) [116] to initialize the model weights. Ozbulak and Ekenel [112] introduced an initialization method that uses predetermined Gabour filters for the first convolutional model layer. However, the weight initialization methods that are most similar to my method are the method suggested by Krähenbühl et. al. [69] and the method suggested by Koturwar and Merchant [66]. Krähenbühl et. al. [69] feed a few training images into the model and calculate the mean and variance of each channel of the created activations at each model layer. Then, they draw values from a normal distribution and scale those values using the obtained means and variances. The resulting scaled values are finally used to set the model weights. Koturwar and Merchant [66], on the other hand, initialize the model weights by selecting the weight values from a normal distribution. The mean and variance of the normal distribution are obtained from the mean and covariance matrix that are computed from a few random crops of a subset of the training images. However, none of these methods uses the image data directly as my method does.

---

[2] `https://pytorch.org/docs/stable/nn.init.html#torch.nn.init.kaiming_normal_`

[3] `https://pytorch.org/docs/stable/nn.init.html#torch.nn.init.kaiming_uniform_`

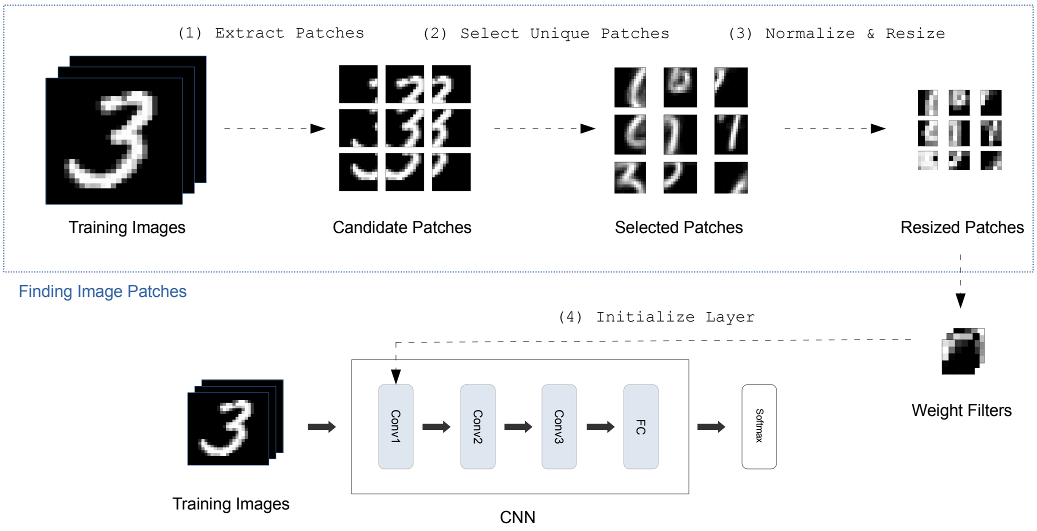


Figure 4.1: The proposed method: (1) Extract candidate patches from the training images, (2) select unique patches from the candidate patches, (3) normalize and resize selected patches, and (4) use resized patches as weight filters to initialize convolutional layer.

4.2 Image Patch-based Model Weight Initialization

A Convolutional Neural Network-based (CNN) image classification model consists of multiple consecutive convolutional hidden layers followed by one or more optional linear hidden layers, as described in Chapter 2.1. Each of these hidden layers (convolutional and linear) contains weights. The values of these weights are learned during model training with respect to the classification objective of the model, as described in Chapter 2.2. However, before we can train the model, we need to assign initial values to the weights of all hidden layers of the model. The choice of these initial values is crucial as they affect the training time of the model [2, 53].

My proposed method finds initial values for the weights of a convolutional hidden layer of the model. Before describing my method in more detail, however, I will recap how the weights of such a convolutional layer are organized. The weights of a convolutional layer are structured into multiple filters (also referred to as kernels). A filter is a three-dimensional tensor containing a subset of the weights of the respective convolutional layer. The first two dimensions of this tensor reflect the size of the filter. Typical filter sizes are 3 × 3, 5 × 5, or 7 × 7. The third dimension depends on the depth of the activation tensor of the previous convolutional layer, or on the depth of the input image if the previous layer is the input layer (depth of grayscale images: 1, depth of color images: 3). After model training, each filter of a convolutional layer detects a specific image feature of the input image of the model. As shown by Zeiler and Fergus [148], the filters of the lower convolutional layers (i.e., the layers closer to the input layer) detect low-level image features (e.g., edges, corners, simple textures), while the filters of the higher convolutional layers (i.e., the layers closer to the output layer) detect high-level image features (e.g., object parts, objects in various poses). The filter size and the number of filters of each convolutional layer depend on the classification problem that needs to be solved. Therefore, size and number need to be set as hyperparameters for each convolutional layer for model training. The weights of all filters of a convolutional layer result in the total amount of weights of that layer [53].

My proposed method initializes the weights of each filter of a convolutional layer of the model using image information (i.e., pixels) from the training data of the model. This image information is obtained from a specific set of image patches extracted from the training images. Each of these patches shows a different characteristic part of an image object. To identify the patches, my method first extracts a set of candidate patches from the training images and then selects those candidate patches that are as unique as possible. The selected candidate patches are the desired characteristic patches. Finally, the selected patches are used to initialize the filters of the convolutional layer. An overview of my method is shown in Figure 4.1. Hereinafter, I describe in more detail how to obtain the candidate patches (Chapter 4.2.1), how to select the characteristic patches from the candidate patches (Chapter 4.2.2), and how I use the selected patches for initializing the filters of the respective convolutional layer (Chapter 4.2.3).

4.2.1 Finding Candidate Patches

My proposed method obtains the image patches pa, required for the weight initialization, from a set of candidate patches pa_{ca}. These candidate patches pa_{ca} need to be extracted from the training images of the model, as shown in step (1) in Figure 4.1. To explain how to extract the candidate patches pa_{ca} from the training images, I use the training images of the MNIST [76] dataset as an example in the following. The MNIST dataset contains grayscale images of size $28 \times 28 \times 1$ showing a bright handwritten digit in the center of the image on black background. First, all of the MNIST training images are cut into rectangular patches. It is necessary that all of these patches are of the same size, as I will explain in Chapter 4.2.2. If a patch size of $14 \times 14 \times 1$ is chosen, for instance, then 4 of those patches are obtained from each of the MNIST training images, as shown in (a) in Figure 4.2. However, none of the resulting 4 patches contains information from the region of the image along the central vertical and central horizontal axis, i.e., the border region of the patches in (a). Thus, 5 additional patches of size $14 \times 14 \times 1$ are extracted along the central vertical and central horizontal axis from each of the MNIST training images (including the central patch), as shown in (b) in Figure 4.2. These additional patches overlap the initial patches in (a) by 7 pixels. As a result, 9 patches of size $14 \times 14 \times 1$ are obtained in total from each of the MNIST training images, as shown in Figure 4.2. However, not all resulting patches contain a sufficient amount of useful information. Some patches located at the image edges may only show tiny parts of the digit (i.e., the image object) as the digits always appear in the center of the MNIST images. These patches have mainly black pixels from the image background. Thus, if a patch contains less than 20 non-black pixels, I do not consider this patch for the weight

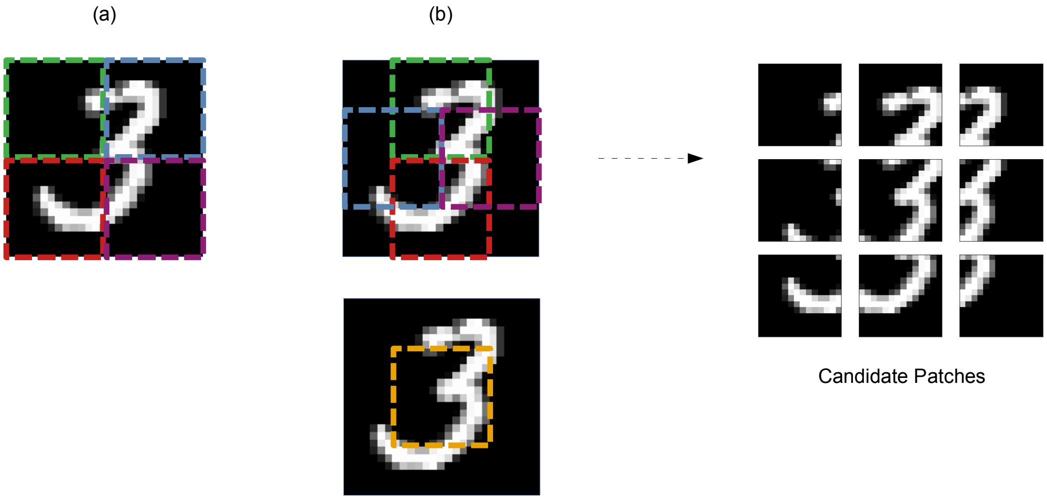


Figure 4.2: Extracting candidate patches from the MNIST [76] training images: (a) Cutting images in 4 patches, and (b) extracting the central patch and 4 more patches around the central vertical and the central horizontal axis.

initialization as it does not contain a sufficient amount of information about the image object (i.e., the digit). The remaining patches form the set of candidate patches pa_{ca}.

For MNIST, I considered candidate patches pa_{ca} from all parts of the training images as long as these patches contain a sufficient amount of information about the image object. As the MNIST images are small, I obtained a reasonable number of small-sized candidate patches. However, for datasets containing large color images showing a natural scene (e.g., a horse in a meadow), this approach would result, depending on the chosen patch size, either in a reasonable number of large-sized candidate patches or a high number of small-sized candidate patches. Neither a high number of candidate patches nor large-sized patches are desirable for my proposed weight initialization method. A large patch size makes it more difficult to use the patches for the weight initialization, as discussed in Chapter 4.2.3. A high number of candidate patches, on the other hand, can lead to memory problems when selecting characteristic patches pa from pa_{ca}, as shown in Chapter 4.2.2. Moreover, as the images show a natural scene, it is not possible to reduce the number of candidate patches by excluding patches that mainly show the image background, as done for MNIST. The MNIST images show a bright grayscale handwritten digit in front of a black background. Hence, if a patch, extracted from one of the MNIST training images, contains almost only black pixels, we know that this patch shows mainly the background. Images of natural scenes, on the other hand, show a natural object (e.g., a horse) in front of a natural background (e.g., a meadow). Normally, natural objects and natural backgrounds each have multiple colors. It is not known whether a pixel belongs to the image object or the image background. Therefore, it is not possible to exclude the candidate patches that mainly show the background as these patches cannot be identified.

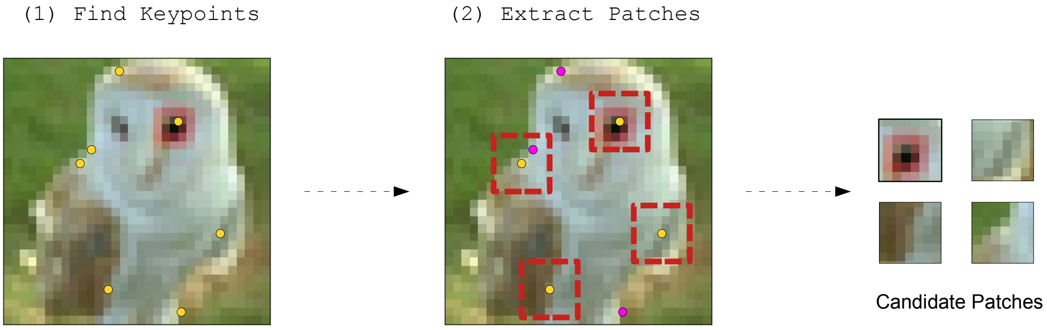


Figure 4.3: Extracting candidate patches from the CIFAR-10 [70] training images: (1) Searching for keypoints using SIFT [92], and (2) extracting a candidate patch around each identified adequate keypoint (yellow). Keypoints that are too close to the image edge or too close to other keypoints (pink) are not considered.

Due to these issues, I use a different approach for datasets containing large color images showing a natural scene to be able to obtain a reasonable amount of small-sized candidate patches pa_{ca} from these images. Below, I use the CIFAR-10 [70] dataset as an example. The CIFAR-10 dataset contains color images of size 32 × 32 × 3 that show a natural image object (e.g., horse, deer, ship) in front of a natural image background (e.g., meadow, forest, sea). This dataset is still rather small-sized, but it serves as an example below to illustrate the proposed approach to obtaining candidate patches pa_{ca} from large color images showing a natural image object. A specified number of candidate patches pa_{ca} of a certain size are extracted from each of the CIFAR-10 training images. However, to obtain the candidate patches pa_{ca}, the entire image is not cut into patches as done for MNIST. Instead, patches are extracted around keypoints of interest in the image, as shown in Figure 4.3. For CIFAR-10, patches of size 15 × 15 × 3 may be considered for instance. I assume that patches around keypoints of interest contain a sufficient amount of information about the image object.

To find keypoints of interest in an image, I use the OpenCV[4] implementation of the SIFT keypoint detection method introduced by Lowe [92]. First, I search for keypoints in each of the CIFAR-10 training images using SIFT, as shown in step (1) in Figure 4.3. For each identified keypoint kp in a CIFAR-10 image, SIFT gives us the keypoint coordinate (i_{kp}, j_{kp}) and its keypoint strength $kpStr(kp)$. The keypoint coordinate specifies the location of the keypoint in the image, i.e., the keypoint is the pixel at the intersection of the i^{th} row and the j^{th} column of the image. The keypoint strength $kpStr(kp)$, on the other hand, expresses how strong[5] keypoint kp is according to the SIFT method. Then, for each image, I select a specified number of identified keypoints n_{kp} at which I aim to extract candidate patches from that image. However, it might be not possible to use all of the keypoints identified by SIFT for extracting candidate patches. If a

---

[4] `https://opencv.org`

[5] `https://docs.opencv.org/3.4.2/d2/d29/classcv_1_1KeyPoint.html`

keypoint kp is located near an image edge, it is not possible to extract a full patch around that keypoint, as shown in Figure 4.3. Thus, I only consider identified keypoints that are located ($patch\ size/2$) pixels away from an image edge. For patches of size $15 \times 15 \times 3$, for instance, I only consider keypoints that are 7 pixels away from an image edge. Furthermore, I also do not consider keypoints that are too close to each other. Extracting patches from two keypoints that are close to each other would result in two very similar patches. Two keypoints, kp_1 and kp_2, are considered to be too close to each other, if these two keypoints have a distance of $d(kp_1, kp_2)$ pixels from each other, and this distance $d(kp_1, kp_2)$ is less than a specified threshold distance d_{thresh} (e.g., 5 pixels), as shown in Figure 4.3. Thus, among the two keypoints, I only consider the keypoint with the higher keypoint strength (Equation 4.1). As a result, I only keep the set of remaining keypoints KP_{re} from the image. Each keypoint of this set is neither too close to an image edge nor too close to other keypoints with a higher keypoint strength.

$$\underset{kp \in \{kp_1, kp_2\}}{\arg\max}\ kpStr(kp); \quad d(kp_1, kp_2) < d_{thresh} \tag{4.1}$$

From the set of remaining keypoints KP_{re} found in an image, I select a subset KP_{se} ($KP_{se} \subseteq KP_{re}$) of n_{kp} keypoints with the highest keypoint strengths $kpStr$. In general, I assume that the number of remaining keypoints in KP_{re} from each image is significantly larger than n_{kp}. However, a few images may have less than n_{kp} remaining keypoints. In this case, I simply select all of the remaining keypoints of that image.

After selecting approximately n_{kp} keypoints from each of the CIFAR-10 training images, I extract a patch pa_{ca} around each selected keypoint $kp_{se} \in KP_{se}$, as shown in step (2) in Figure 4.3. As a result, I obtain patch pa_{ca} with keypoint kp_{se} in its center. However, it is possible that some of the resulting patches pa_{ca} mainly show the image background. It is not possible to filter out those patches because it is not known which pixels of the image show the image object and which pixels show the image background. However, I do not consider the occurrence of background patches among the candidate patches pa_{ca} as an issue. I assume that the most important keypoints found by SIFT are located at the image objects rather than at the image backgrounds. Thus, I assume that background patches rarely occur in the set of candidate patches pa_{ca}. Furthermore, even if I obtain a patch that mainly shows the image background, the patch may still contain useful information related to the image object. A ship, for instance, is usually found on water and not in a meadow. The context of the image object in the form of the image background is important as well in this case. Thus, those patches should be kept. As a result, I obtain the set of candidate patches pa_{ca}.

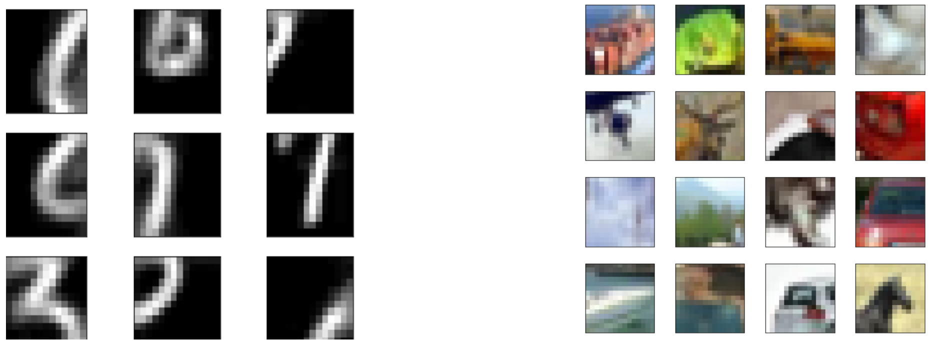
4.2.2 Selecting Required Patches

After obtaining the candidate patches pa_{ca} from the training images of the model using the method described in Chapter 4.2.1, a few patches pa need to be selected from these candidate patches pa_{ca} that are as unique as possible, as shown in step (2) in Figure

4.1. This selection is necessary because, in general, the number of obtained candidate patches pa_{ca} clearly exceeds the number of filters of the convolutional layer that should be initialized using image patches. Although I already decreased the number of possible candidate patches in Chapter 4.2.1 by excluding patches mainly showing the image background, or by only extracting patches around keypoints of interest, the remaining number of candidate patches is still significantly higher than the number of filters of the respective convolutional layer. If the convolutional layer contains 20 filters, for instance, then 20 patches pa need to be selected from the candidate patches pa_{ca} for initializing these filters. The 20 selected patches should be as unique as possible in order to avoid initializing each filter with a similar patch.

In order to select patches that are as unique as possible, I use an approach based on the k-Means [95] clustering algorithm. To describe the proposed selection approach, I use two different sets of candidate patches as an example below. From the MNIST training images, I extracted a set of candidate patches of size $14 \times 14 \times 1$. From the CIFAR-10 training images, on the other hand, I extracted a set of candidate patches of size $15 \times 15 \times 3$. Suppose n_{ca} candidate patches were extracted. First, each candidate patch pa_{ca} is flattened into a vector $a(pa_{ca})$. The size M of this vector depends on the number of pixels each candidate patch contains. As a result, a vector of size 196×1 is obtained for each of the MNIST candidate patches ($M = 196$), while a vector of size 675×1 is obtained for each of the CIFAR-10 candidate patches ($M = 675$). After receiving the n_{ca} patch vectors $a(pa_{ca})$, the vectors need to be concatenated into a matrix $A^{pa_{ca}}$. All vectors need to be of the same size to be able to concatenate them. However, the vectors are of the same size already. When I extracted the candidate patches from the training images, I made sure that all resulting candidate patches are of the same patch size, as described in Chapter 4.2.1. As a consequence, after flattening the candidate patches into vectors, all resulting vectors are also of the same size. Thus, all n_{ca} vectors $a(pa_{ca})$ are concatenated into a matrix $A^{pa_{ca}}$. The resulting matrix is of size $n_{ca} \times M$. In this matrix, I search for clusters using k-Means. Each identified cluster h contains a subset pa^{h}_{ca} of the candidate patches pa_{ca} ($pa^{h}_{ca} \subset pa_{ca}$) in the form of their vectors $a^{h}(pa_{ca})$ from the rows of $A^{pa_{ca}}$. The patches of one cluster are as different as possible from the patches of another cluster and therefore also as unique as possible. Hence, a characteristic patch can be selected from each cluster to obtain the patches pa ($pa \subset pa_{ca}$) that should be used for initializing the filters of the respective convolutional layer.

However, matrix $A^{pa_{ca}}$ is usually high-dimensional and identifying clusters in high-dimensional spaces does not work well, as pointed out by Chen et. al. [19]. Clustering algorithms, such as k-Means, use distance metrics for identifying clusters, but distance metrics are not effective in high-dimensional spaces. However, the data samples of most applications are located within a low-dimensional subspace of the high-dimensional space, according to Domingos [28]. Thus, as suggested by Chen et. al. [19], I use dimensionality reduction to project matrix $A^{pa_{ca}}$ onto such a low-dimensional subspace. The best results were achieved with the non-linear dimensionality reduction technique UMAP (Uniform Manifold Approximation and Projection) [98]. However, if matrix $A^{pa_{ca}}$ is huge, a sufficient projection result may not be received. UMAP does not always work well if the dimensionality is too high. Moreover, UMAP usually requires a long



MNIST

CIFAR-10

Figure 4.4: A few of the characteristic patches that were identified from MNIST [76] (left) and CIFAR-10 [70] (right) using the proposed method.

computation time and high memory consumption when applied to a huge matrix. Thus, I avoided extracting large-sized candidate patches or a high number of candidate patches from each image in Chapter 4.2.1 to keep the dimensionality of $A^{pa_{ca}}$ relatively low. Nevertheless, if the dimensionality of matrix $A^{pa_{ca}}$ is still too high for applying UMAP to the matrix, it is possible to alternatively use either the linear dimensionality reduction technique PCA (Principal Component Analysis) [116] or a combination of PCA and UMAP (as done in Chapter 5 and Chapter 6). PCA generally suffers less in very high dimensions and requires a lower computation time and lower memory consumption. However, PCA most likely leads to a worse projection result compared to UMAP when the dimensionality is not too high. Therefore, I only use PCA for large matrices. As a result, I chose UMAP to project the matrix of the MNIST candidate patches and PCA to project the matrix of the CIFAR-10 candidate patches. However, before projecting matrix $A^{pa_{ca}}$, each value of the matrix needs to be normalized as a preprocessing step for the dimensionality reduction (for more details, see Chapter 3.1). After normalizing each value of matrix $A^{pa_{ca}}$, the dimensionality of the matrix is reduced from $n_{ca} \times M$ down to $n_{ca} \times 2$. To reduce the dimensions, I use UMAP for the MNIST matrix, while I use PCA for the CIFAR-10 matrix. As a result, the projected matrix $r(A^{pa_{ca}})$ is obtained from the learned projection model r applied to matrix $A^{pa_{ca}}$. In this projected matrix $r(A^{pa_{ca}})$, I search for clusters using the k-Means algorithm.

The k-Means algorithm requires setting hyperparameter k. Hyperparameter k specifies how many clusters the k-Means algorithm should find in $r(A^{pa_{ca}})$. The k-Means algorithm should find as many clusters as there are filters to initialize. Then, a characteristic patch should be selected from each identified cluster in order to use it for initializing one of the filters of the respective convolutional layer. Thus, if the convolutional layer contains 20 filters, for instance, I set $k = 20$. After identifying the k clusters $h \in h_1, .., h_k$, a characteristic patch $a(pa)$ (in vector form) is picked from each of these clusters. The patch most characteristic of its cluster is most likely the patch closest to its cluster center. Thus, this patch is selected from each cluster h. However, for datasets whose images have a neutral image background, such as MNIST (black background), it is even possible to obtain artificial characteristic patches containing information from several real patches. In this case, not only the closest patch from its cluster center is selected but the 10 closest patches. The mean of these 10 patches is taken to obtain an average artificial patch $a(pa)_{avg}$ (in vector form). This process is repeated for the remaining clusters to receive such an artificial patch from each cluster h. However, this approach does not work well for the CIFAR-10 patches. In my experiments, averaging CIFAR-10 patches resulted in washed-out image patches. I assume that this effect is caused by the complexity of the CIFAR-10 images. They do not contain a neutral image background and the image objects are characterized by a higher variance of their visual appearance. As a consequence, the structure of the image object was not recognizable anymore in this case. Thus, only the closest patch from its cluster center is used for the CIFAR-10 patches. Finally, after obtaining a patch vector $a(pa)$ (real or artificial) from each cluster, each patch is reshaped from vector form $a(pa)$ back to image form pa to obtain the k characteristic image patches pa that should be used for initializing the filters of the respective convolutional layer. Figure 4.4 shows a few of these characteristic patches pa that were identified from MNIST and CIFAR-10.

4.2.3 Initializing the Weights of a Model Layer

The obtained k characteristic patches pa should be used to initialize the k filters of the convolutional layer of the model. Before using the patches pa for initializing the filters, however, two preprocessing steps need to be applied to the patches pa, as shown in step (3) in Figure 4.1. First, each patch pa needs to be normalized. To normalize a patch, I subtract from each pixel of the patch the mean over all pixels of that patch. After normalizing the patch, the mean over all pixels of the patch is 0. This normalization had a positive effect on the subsequent training of the model in my experiments. This also resembles the state-of-art weight initialization method for Convolutional Neural Network-based models, Kaiming weight initialization [45], which selects initial weight values from a distribution with a mean of 0. After normalizing the patches pa, I need to resize them in a second preprocessing step. The patches were extracted from the training images with a patch size larger than the size of the filters of the convolutional layer, as described in Chapter 4.2.1. I chose a patch size of $14 \times 14 \times 1$ for the MNIST training images and a patch size of $15 \times 15 \times 3$ for the CIFAR-10 training images as an example. In contrast, the typical size of a filter is 3×3, 5×5 or 7×7. Below, I assume that the filters have a size of 5×5. As a result, I need to adjust the size of the patches pa to the size of the filters of 5×5. Thus, the MNIST patches from $14 \times 14 \times 1$ are resized down to $5 \times 5 \times 1$, while the CIFAR-10 patches are resized from $15 \times 15 \times 3$ down to $5 \times 5 \times 3$. However, every time a patch is resized information is lost, and the larger the resize, the more information is lost. Thus, I avoided extracting huge patches from the training images, as pointed out in Chapter 4.2.1, to lose as little information as possible. Alternatively, I could have extracted patches from the training images of the appropriate size of 5×5 already to avoid the resizing step. However, extracting slightly larger patches followed by resizing them achieved better results in my experiments. I assume that this may be related to the receptive field of a convolutional layer. As explained in Chapter 2.1, the size of the receptive field increases with subsequent model layers to be able to detect image features of increasing complexity (e.g., from edges over complex shapes to object parts) [53]. Thus, extracting a patch of size 5×5 may encompass an image region that is too small for the layer.

After resizing the patches pa, I stack them together. As a result, tensor of size $k \times 5 \times 5 \times depth$ is obtained (depth of MNIST: 1, depth of CIFAR-10: 3). This tensor can be used to initialize the k filters of the first convolutional layer of the model, as shown in step (4) in Figure 4.1. Only the filters of this layer have the same depth as the input image, and thus also the same depth as the image patches (depth of MNIST: 1, depth of CIFAR-10: 3). Therefore, only the weights of the first convolutional layer are initialized using image patches in the standard setup. The weights of the other layers are initialized with a state-of-the-art weight initialization method. Nevertheless, it is still possible to initialize other convolutional layers besides the first one using image patches as well. The filter depth of these subsequent convolutional layers, however, usually increases significantly [2]. Therefore, for these layers, I select a higher number of patches pa than the number of filters of the respective convolutional layer, and use more than one patch for each filter of the layer. Then, I stack multiple patches together to initialize the filter. Finally, after initializing all of the weights of the model, I train the model.

4.3 Experiments

To find good initial values for the weights of a Convolutional Neural Network-based (CNN) image classification model, I proposed a weight initialization method that is based on image patches extracted from the training images of the model. The details of the proposed method are described in Chapter 4.2. To evaluate the method, I conducted several experiments. The general setup of these experiments is described in Chapter 4.3.1. In a first experiment, I tested whether a weight initialization method based on image patches allows model training at all. Furthermore, I also tested if a model whose weights were initialized by the proposed method is able to achieve a significantly higher classification performance than a model whose weights were initialized by a state-of-the-art weight initialization method. The results are presented in Chapter 4.3.2. For the experiment in Chapter 4.3.2, I used the optimal value for the learning rate hyperparameter for training the respective models. To evaluate the training process using a suboptimal value for the learning rate hyperparameter, I repeated the experiment using a smaller learning rate. The results are presented in Chapter 4.3.3. However, for the experiments in Chapter 4.3.2 and Chapter 4.3.3, only the weights of the first convolutional layer were initialized using the proposed method. The weights of the remaining model layers were initialized by a state-of-the-art weight initialization method (Kaiming Uniform [45] or Fixup [150]). Therefore, it was examined in a third experiment whether it is possible to apply the proposed image patch-based weight initialization method to multiple convolutional layers. The results are presented in Chapter 4.3.4.

4.3.1 Experimental Setup

To test my proposed image patch-based weight initialization method in comparison to different state-of-the-art methods (Kaiming Normal [45], Kaiming Uniform [45] and Fixup [150]), I conducted several experiments using the MNIST [76] dataset, the CIFAR-10 [70] dataset and the CIFAR-100 [70] dataset. For each experiment, the same experimental setup was used. I first initialized the weights of a Convolutional Neural Network-based (CNN) image classification model using either my method or a state-of-the-art method. However, when using my method in its standard setup to initialize the model weights, I only initialized the model weights of the first convolutional layer of the model using image patches (only in Chapter 4.3.4 I also initialized other convolutional layers using image patches). All remaining model layers (convolutional and linear) were initialized using a state-of-the-art method (Kaiming Uniform or Fixup). To initialize the weights of the first convolutional layer using image patches, I used the approach described in Chapter 4.2. I first extracted candidate patches from the training images of the respective dataset. Then, I selected a certain number of unique patches from these candidate patches. This number corresponds to the number of filters of the first convolutional layer. After identifying the unique patches, I first normalized and then resized the identified patches to the filter size of the layer. Finally, I initialized the filters of the layer using the resized image patches. As a result, I obtained two models. One model was initialized using my patch-based weight initialization method, i.e., the first convolutional layer was initialized using image patches and the remaining layers were initialized using a state-of-the-art weight initialization method (Kaiming Uniform or Fixup). The other model, in contrast, was initialized using a state-of-the-art weight initialization method (Kaiming Normal, Kaiming Uniform or Fixup), i.e., all model layers were initialized using this state-of-art method only. Then, I trained each model 5 times for a specified number of training epochs using the respective training dataset. For each of the 5 times, I trained the model with a different seed value from a set of 5 random seed values. The same 5 random seed values were used for both models. A seed value sets the random number generator used for model training[6]. By using different seed values, I ensured that the models do not achieve a certain classification performance only by chance. If I have only trained each model a single time using a particular seed value, the resulting model might have received a good classification performance. However, if I have trained the model again using a different seed value, the model might have received a classification performance that is significantly worse than the previously obtained classification performance using the first seed value. Thus, in order to make sure that the obtained model performances are not mainly dependent on the choice of a single seed value, I trained each model 5 times using a different seed value each time. After every training epoch, I tested each model on the respective test dataset to receive its classification performance in the form of the classification accuracy. As a result, I obtained for each model 5 accuracies (one for each seed) after each training epoch. From these 5 accuracies, I calculated the mean and the standard deviation in order to compare the classification performances of the two models after each training epoch.

---

[6] `https://pytorch.org/docs/stable/notes/randomness.html`

I conducted my initial experiments on the MNIST dataset in order to examine whether my proposed patch-based weight initialization method works at least for a simple dataset such as MNIST. The MNIST dataset contains grayscale images of size $28 \times 28 \times 1$ that are organized into 10 classes. Each MNIST class represents a different bright handwritten digit (digits: 0-9). The images of the classes show the respective digit in their center on black background. To obtain the candidate patches from the MNIST training images, I used the approach described in Chapter 4.2.1. Each MNIST training image was cut into 9 patches of size $14 \times 14 \times 1$ with an overlap of 7 pixels. After obtaining the set of candidate patches, I identified the unique patches from this set of candidate patches. These unique patches were used for the weight initialization of the first convolutional layer. To identify the unique patches, I used the approach described in Chapter 4.2.2. First, all obtained candidate patches were flattened to vectors and concatenated all received vectors to a matrix. Then, I needed to project the matrix to a lower dimensional subspace using UMAP (Uniform Manifold Approximation and Projection) [98] as a preprocessing step. Finally, I searched for clusters in the projected matrix using the k-Means [95] clustering algorithm. Hyperparameter k of k-Means was set to the number of filters of the first convolutional layer that I aimed to initialize. After identifying the k clusters using k-Means, I chose the 10 corresponding patches from each identified cluster that are closest to their respective cluster center. For each of the identified k clusters, I then averaged the 10 chosen patches in order to receive an artificial patch containing the information of the 10 patches. These k artificial patches were then used for the weight initialization of the convolutional layer. All remaining layers were initialized using the Kaiming Uniform initialization method. In order to train the MNIST model initialized using my method and the MNIST model initialized using a state-of-the-art method (Kaiming Uniform or Kaiming Normal), I chose a model architecture similar to the Caffe LeNet[7] architecture. The used architecture consisted of 2 consecutive convolutional layers (Conv) followed by 2 fully-connected linear layers (FC). Each convolutional layer is followed by a max pooling layer [110]. The ReLU function (Rectified Linear Unit) [39] was used as the activation function for the pre-activation output of the model layers. The complete model architecture of the MNIST models is as follows: Conv1 (number of filters: 20, kernel size: 5) - Pool - Conv2 (number of filters: 50, kernel size: 5) - Pool - FC1 (layer size: 500) - FC2 (layer size: 10). The images of the training dataset and the images of the test dataset were normalized before model training using the MNIST statistics (mean: 0.1307; std: 0.3081). Finally, I trained each model for 20 training epochs using the SGD[8] optimizer (Stochastic Gradient Descent) [122].

After my initial tests using the MNIST dataset, I also aimed to test my proposed patch-based weight initialization method using more complex datasets. Thus, I conducted additional tests using the CIFAR-10 dataset and the CIFAR-100 dataset for my experiments. The CIFAR-10 images are color images of size $32 \times 32 \times 3$, but they do not show a simple object such as a digit. Instead, the images of the CIFAR-10 dataset show a natural image object belonging to one of 10 classes (e.g., horse, deer, ship) in front of

---

[7] `https://github.com/BVLC/caffe/blob/master/examples/mnist/lenet.prototxt`

[8] `https://pytorch.org/docs/stable/generated/torch.optim.SGD.html`

a natural image background (e.g., meadow, forest, sea). The CIFAR-100 dataset, on the other hand, is similar to the CIFAR-10 dataset. However, instead of images belonging to 10 classes, each CIFAR-100 image belongs to one out of 100 classes. To obtain the candidate patches from the CIFAR-10 and CIFAR-100 images, I used the approach described in Chapter 4.2.1. I searched for keypoints on each training image using the SIFT keypoint detection method [92]. However, keypoints that are too close to the image edge or too close to other keypoints were not considered. Thus, each keypoint was required to be at least 7 pixels away from an image edge and 5 pixels away from other keypoints. Then, I chose the 9 best keypoints according to SIFT and extracted a patch from the location of each of those 9 keypoints. Two patch sizes were tested, $15 \times 15 \times 3$ and $3 \times 3 \times 3$. As the patches of size $15 \times 15 \times 3$ achieved a superior performance, I decided to use these patches as the candidate patches. These patches were the considered candidate patches. After obtaining the candidate patches from all training images, I identified the unique patches from these obtained candidate patches. These unique patches were used for the weight initialization of the first convolutional layer. To identify the unique patches, I used the approach described in Chapter 4.2.2 again. First, I flattened the obtained candidate patches to vectors and concatenated all received vectors to a matrix. Then, I needed to project the matrix to a lower dimensional subspace using PCA (Principal Component Analysis) [116] as a preprocessing step. PCA was used to project the matrix because the matrix was too huge for UMAP. Finally, I searched for clusters in the projected matrix using the k-Means clustering algorithm. Hyperparameter k of k-Means was set to the number of filters of the first convolutional layer that I aimed to initialize. After identifying the k clusters using k-Means, I chose the patch from each identified cluster that is closest to its respective cluster center. As a result, I obtained one patch from each of the k clusters. These k patches were then used for the weight initialization of the convolutional layer. All remaining layers were initialized using the Fixup initialization method. In order to train the CIFAR-10 and CIFAR-100 model initialized using my method and the CIFAR-10 and CIFAR-100 model initialized using the state-of-the-art method (Fixup), I chose the 20-layer ResNet (Residual Network) architecture [46] from Zhang et. al. [150]. However, as CIFAR-10 and CIFAR-100 are more complex than MNIST, I needed to adjust my training setup. To obtain a model achieving a sufficient classification performance, I used a training setup based on the training setup[9] used by Zhang et. al. [150]. To obtain more variations of the training data samples, I used different data augmentation techniques [128]. These techniques included randomly flipping training images horizontally, randomly cropping training images and applying the Mixup augmentation technique [149] to the training images. Moreover, the training and test images were normalized before model training using the CIFAR-10 statistics (mean: 0.4914, 0.4822, 0.4465; std: 0.2023, 0.1994, 0.2010). The model was trained for 200 training epochs using the SGD optimizer. Furthermore, I used a learning rate schedule instead of a fixed learning rate value. The best classification performance of the model was obtained by a cosine-annealing[10] learning rate schedule [91].

---

[9] `https://github.com/hongyi-zhang/Fixup`

[10] `https://pytorch.org/docs/stable/generated/torch.optim.lr_scheduler.CosineAnnealingLR`

4.3.2 Testing against State-of-the-Art Methods

In my first experiment, I aimed to examine whether my proposed method can be used to initialize the weights of a Convolutional Neural Network-based (CNN) image classification model at all. Furthermore, I also aimed to compare my method to several state-of-the-art weight initialization methods. The setup of my experiment is described in Chapter 4.3.1. First, the proposed method was tested using the MNIST [76] dataset. If my method can be used for initializing the model weights at all, it should at least work for a simple dataset such as MNIST. After initializing a model for MNIST using my method, I trained the model 5 times using a different seed value each time for 20 training epochs, as described in Chapter 4.3.1. To train the model, I used a learning rate of 0.01, a momentum of 0.9 and a weight decay of 5e−4. During model training, I tested the model on the MNIST test dataset and received the 5 accuracies of the model after each training epoch (one for each of the 5 seed values). It was observed that all 5 accuracies of the final training epoch were at least 98%. As a result, I concluded that my method can be used for initializing a Convolutional Neural Network-based model. Therefore, I then aimed to test my method in comparison to two state-of-the-art weight initialization methods for an MNIST model, Kaiming Normal[11] [45] and Kaiming Uniform[12] [45]. To compare my method to these two state-of-the-art methods, I initialized a second model using Kaiming Normal and a third model using Kaiming Uniform. Then, I trained each model 5 times in the same way as I trained the model initialized by my method. During model training, I tested the models on the MNIST test dataset and received the 5 accuracies for each model after each training epoch (one for each of the 5 seed values). As a result, I obtained 5 accuracies after each training epoch for the model initialized by my method, for the model initialized by Kaiming Normal, and for the model initialized by Kaiming Uniform. To better compare these models, I calculated the mean and the standard deviation from the 5 accuracies of each model. The results of my experiment with respect to the MNIST dataset are shown in Figure 4.5 (left). As shown in the figure, all models achieved a similar classification performance. The model initialized by my method and the model initialized by Kaiming Uniform even achieved a slightly better model performance than the model initialized by Kaiming Normal.

However, MNIST is a simple dataset. Therefore, I also aimed to test my proposed method in comparison to a state-of-the-art method for a more complex dataset. As a result, I conducted additional tests for CIFAR-10 [70]. After initializing a model for CIFAR-10 using my method, I trained the model 5 times using a different seed value each time for 200 training epochs, as described in Chapter 4.3.1. To train the model, I used an initial learning rate value of 0.1 (for the cosine-annealing learning rate schedule [91]), a momentum of 0.9 and a weight decay of 5e−4. During model training, I tested the model on the CIFAR-10 test dataset and received the 5 accuracies after each training epoch (one for each of the 5 seed values). It was observed that all 5 accuracies of the final training epoch were at least 91%. As a result, I concluded that my method can also be used for initializing a Convolutional Neural Network-based model for a more complex

---

[11] `https://pytorch.org/docs/stable/nn.init.html#torch.nn.init.kaiming_normal_`

[12] `https://pytorch.org/docs/stable/nn.init.html#torch.nn.init.kaiming_uniform_`

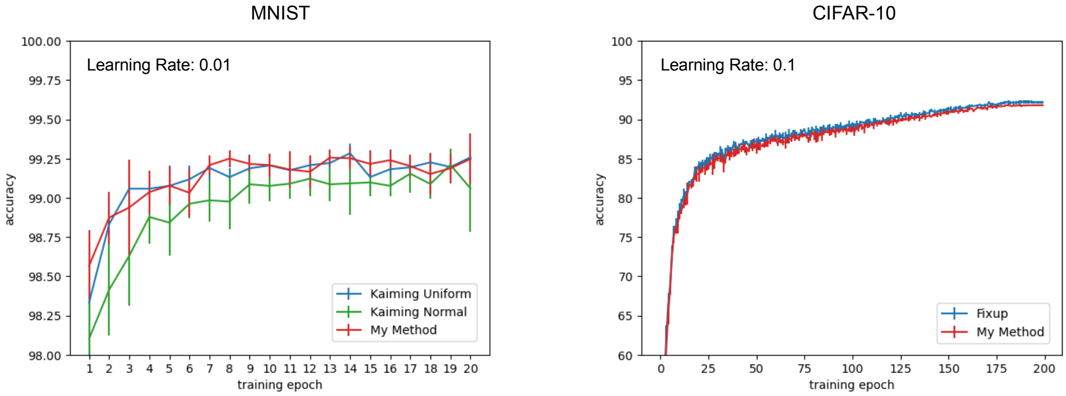


Figure 4.5: Comparison of my method with different state-of-the-art weight initialization methods with respect to the MNIST [76] dataset (left) and the CIFAR-10 [70] dataset (right). Each model was trained 5 times using a different random seed each time and an optimal learning rate value. The results from each model are given after each training epoch as mean and standard deviation over the 5 resulting accuracies.

dataset such as CIFAR-10. Therefore, I then aimed to test my method in comparison to a state-of-the-art weight initialization method for a CIFAR-10 model. Fixup [150] was chosen as the state-of-the-art method because Fixup performed slightly better than Kaiming Normal and Kaiming Uniform in my tests for CIFAR-10. To compare my method to Fixup, I initialized a second model using Fixup. Then, I trained the model 5 times in the same way as I trained the model initialized by my method. During model training, I tested the model on the CIFAR-10 test dataset and received the 5 classification accuracies after each training epoch (one for each of the 5 seed values). As a result, I obtained 5 accuracies after each training epoch for the model initialized by my method and for the model initialized by Fixup. To better compare these models, I calculated the mean and standard deviation from the 5 accuracies of each model. The results of my experiment with respect to CIFAR-10 are shown in Figure 4.5 (right). As shown in the figure, both models achieved a similar classification performance.

4.3.3 Weight Initialization with Suboptimal Hyperparameters

In my experiment in Chapter 4.3.2, I showed for MNIST [76] and CIFAR-10 [70] that a model initialized by my proposed method achieves a similar classification performance as a model initialized by a state-of-the-art method. To train each model, I used optimal values for the training hyperparameters, learning rate (MNIST: 0.01; CIFAR-10: 0.1), momentum (MNIST: 0.9; CIFAR-10: 0.9), and weight decay (MNIST: 5e−4; CIFAR-10: 5e−4). The same values for these hyperparameters were also used by the official Caffe LeNet training[13] to obtain a model for MNIST, and by Zhang et. al. [150] to train a model for CIFAR-10. However, for a novel dataset, it is not always easy to find

---

[13] `https://github.com/BVLC/caffe/blob/master/examples/mnist/lenet_solver.prototxt`

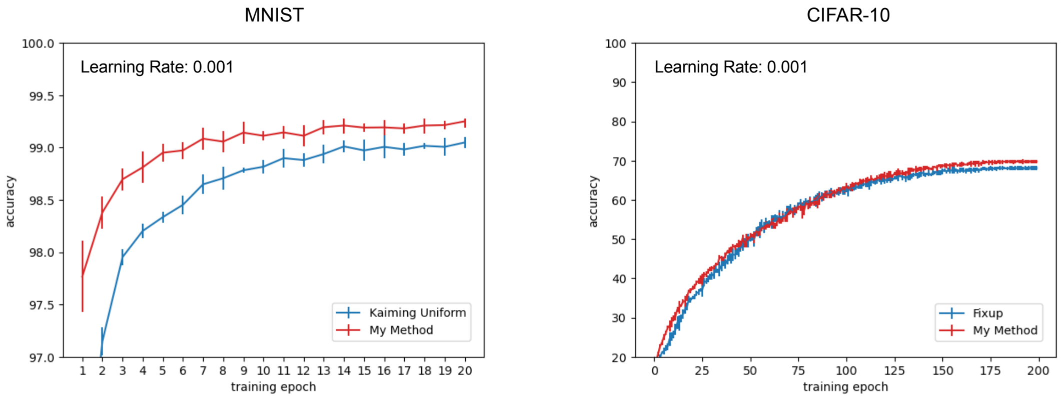


Figure 4.6: Comparison of my method with different state-of-the-art weight initialization methods with respect to the MNIST [76] dataset (left) and the CIFAR-10 [70] dataset (right). Each model was trained 5 times using a different random seed each time and a suboptimal learning rate value. The results from each model are given after each training epoch as mean and standard deviation over the 5 resulting accuracies.

these optimal values for the training hyperparameters. Thus, I repeated my experiment from Chapter 4.3.2 using suboptimal values for the learning rate (the most important training hyperparameter) to simulate a scenario in which the optimal values were not found. I wanted to find out how a model initialized by my method performs in this scenario in comparison to the state-of-the-art methods. To set the learning rate, I chose a value for each dataset that is smaller than the value used in my experiment in Chapter 4.3.2 (MNIST: 0.001, 0.0001, 0.00001; CIFAR-10: 0.001). A larger value (e.g., 0.1 for MNIST) could not be used as in this case model training was not possible anymore. Except for the learning rate value, I used the same experimental setup as for my experiment in Chapter 4.3.2. For MNIST, however, I only compared my method to the Kaiming Uniform [45] method as it performed better in my first experiment in Chapter 4.3.2 than Kaiming Normal [45]. The results of this second experiment with respect to the MNIST dataset are shown in Figure 4.6 (left), while the results with respect to the CIFAR-10 dataset are shown in Figure 4.6 (right). As shown in the figure, the classification performances of all models decreased in comparison to the classification performances of the models in my first experiment in Chapter 4.3.2. However, this was expected as I used suboptimal values for the learning rate hyperparameter. Nevertheless, the decline in the classification performance of the models initialized by my method is smaller than the decline in the classification performance of the models initialized by the state-of-the-art weight initialization methods with respect to MNIST and CIFAR-10. Therefore, I showed that a model initialized by my method is able to achieve a superior classification performance than a model initialized by a state-of-the-art method when using a suboptimal learning rate value. Similar effects could be observed with respect to the MNIST dataset when using a suboptimal value for the momentum (0.8, 0.7, 0.6, 0.5) and the weight decay hyperparameter (5e−5, 5e−3, 5e−2).

4.3.4 Initializing the Weights of Multiple Model Layers

In my first experiment in Chapter 4.3.2, I showed that a model initialized by my proposed method achieves a similar classification performance as the models initialized by a state-of-the-art weight initialization method when using an optimal value for the learning hyperparameter to train the models. Furthermore, I showed in my second experiment in Chapter 4.3.3 that the model initialized by my method achieves a superior classification performance than the models initialized by the state-of-the-art weight initialization methods when using a suboptimal value for the learning rate hyperparameter to train the models. However, the standard setup of my patch-based weight initialization method only initializes the weights of the first convolutional layer of the model using image patches. The weights of the other model layers (convolutional and linear) are initialized using a state-of-the-art weight initialization method (Kaiming Uniform or Fixup), as described in Chapter 4.2.3. However, modern Convolutional Neural Network-based (CNN) image classification models such as a ResNet-based model (Residual Network) [46] contain a high number of layers. For my tests using CIFAR-10 [70] in Chapter 4.3.2 and in Chapter 4.3.3, I also used a ResNet-based model using 20 layers. Therefore, the effect of initializing such a model using image patches might be rather small when initializing only the first convolutional layer of the model using the image patches. As a result, I conducted a third experiment in order to test whether initializing other convolutional layers besides the first convolutional layer of the model is beneficial or not. Therefore, I repeated my experiments for CIFAR-10 from Chapter 4.3.2 and Chapter 4.3.3 with a higher number of convolutional layers that were initialized using image patches. The following layers of the used ResNet-based CIFAR-10 model were chosen to be initialized with image patches: (a) Only the first convolutional layer (as in my standard setup) (1 layer), (b) the first convolutional layer and the output of the third ResNet block (2 layers), (c) the same layers as in (b) and the output of the sixth ResNet block (3 layers), and (d) the same layers as in (c) and the output of the ninth ResNet block (4 layers). Besides the number of convolutional layers that I initialized using image patches, I used the same experimental setup for this third experiment as in Chapter 4.3.2 and Chapter 4.3.3. A suboptimal learning rate value of 0.001 was used besides the optimal learning rate of 0.1. The results of my experiment with respect to the CIFAR-10 dataset are shown in Figure 4.7. As shown by the figure, the more layers I initialize using image patches the better gets the classification performance of the model when using a suboptimal learning rate value. When using an optimal learning rate value, on the other hand, the models initialized by my method achieved a classification performance that is only slightly worse than the model initialized by the state-of-the-art method (Fixup). However, the initialization approach (c) and (d) could not be trained with the optimal learning rate value of 0.1 anymore. In these two cases, the learning rate was already too high. Therefore, model training was not possible anymore.

To check whether these results are statistically significant, I conducted the Stuart Maxwell significance test [94] on the results. The Stuart Maxwell test is a variation of the McNemar test, which was recommended by Dietterich [26] to evaluate classification models. The McNemar test is a significance test to check whether two classification

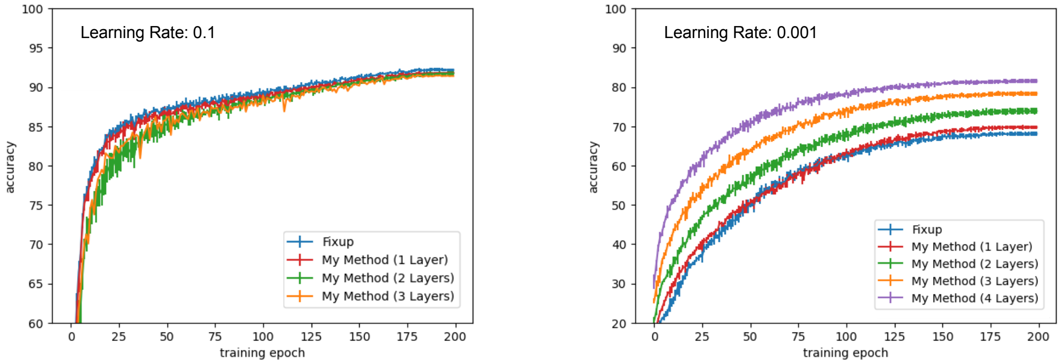


Figure 4.7: Comparison of my method applied to different convolutional layers with respect to the CIFAR-10 [70] dataset using an optimal learning rate value (left) and a suboptimal learning rate value (right) for the subsequent training process. Each model was trained 5 times using a different random seed each time. The results from each model are given after each training epoch as mean and standard deviation over the 5 resulting accuracies.

models are significantly different from each other or not. However, the McNemar test can only be used for classification problems with two classes. The Stuart Maxwell test, on the other hand, can also be used for classification problems with more than two classes. I conducted the Stuart Maxwell test on the results of the model initialized by my method with the initialization approach (b) and the results of the model initialized by the state-of-the-art method (Fixup) when using a suboptimal learning rate value (i.e., the results in the form of their classification accuracies after each training epoch). The Stuart Maxwell test showed that the model initialized by my method with the initialization approach (b) and the model initialized by the state-of-the-art method (Fixup) are significantly different with a probability of 99% after each training epoch when using a suboptimal learning rate value. As a result, I concluded that the model initialized by my method with initialization approach (b) is really superior to the model initialized by the state-of-the-art method (Fixup) in this case.

Furthermore, I also aimed to conduct the experiment for MNIST [76] and additionally CIFAR-100 [70] (as an even more complex dataset than CIFAR-10). For MNIST, I used the same experimental setup as in Chapter 4.3.2 and Chapter 4.3.3. As my MNIST model only contains two convolutional layers, I simply initialized both convolutional layers using image patches. Unfortunately, this made the classification performance slightly worse independently of the learning rate value. As a result, for models with a lower number of layers such as my MNIST model, initializing more than one layer does not seem to improve the classification performance when using a suboptimal learning rate value. For CIFAR-100, on the other hand, I used the same training setup as for the CIFAR-10 models including the 20-layer ResNet-based model architecture. Then, I compared my method for CIFAR-100 using the initialization approach (b) with the state-of-the-art weight initialization method (Fixup). The results of my tests with respect to

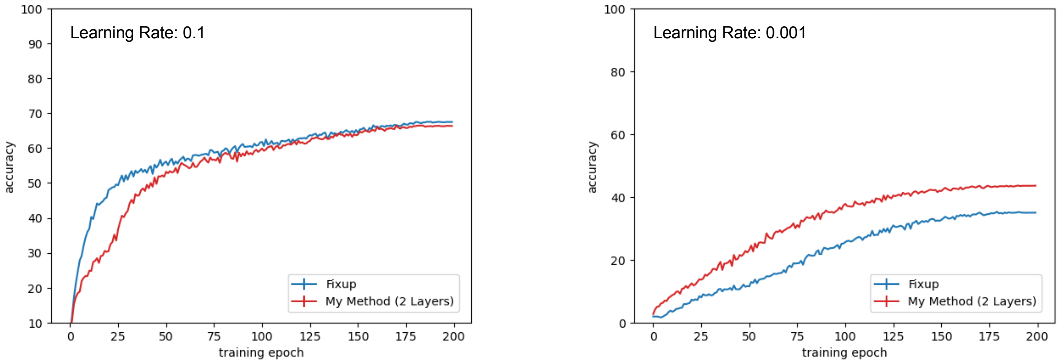


Figure 4.8: Comparison of my method applied to different convolutional layers with respect to the CIFAR-100 [70] dataset using an optimal learning rate value (left) and a suboptimal learning rate value (right) for the subsequent training process. The results from each model are given after each training epoch as accuracy.

CIFAR-100 are shown in Figure 4.8. Again, the figure shows that the model initialized by my method with multiple layers achieved a superior classification performance than the model initialized by the state-of-the-art weight initialization method (Fixup) when using a suboptimal learning rate value. As a result, I concluded that it seems to be a reasonable option to initialize multiple layers using image patches for models containing a high number of layers such as a ResNet-based model.

4.4 Discussion

To be able to train a Convolutional Neural Network-based (CNN) image classification model, we need to set the initial weight values of the model and the hyperparameters for the model training beforehand. Finding good initial weight values as well as good training hyperparameters is a non-trivial task if we aim to train the model from scratch. In Chapter 4.1, I gave an overview of different state-of-the-art methods to find good initial values for the model weights. These state-of-the-art methods, such as Kaiming [45] or Fixup [150], are based on random values and information about the model architecture. However, the state-of-the-art methods do not exploit any information about the classification problem in order to find the initial values for the model weights. Therefore, I investigated whether using information from the classification problem in the form of image patches extracted from the training images is beneficial for finding good initial weight values. I expected that my proposed weight initialization method (Chapter 4.2) would set the initial values for the model weights in such a way that I would need fewer training iterations to train the model compared to a model initialized by a state-of-the-art weight initialization method.

To evaluate my weight initialization method, I conducted several experiments using different datasets (Chapter 4.3). My first research goal, however, was to examine whether image patches extracted from the training images can be used to initialize the model weights at all. In Chapter 4.3.2, I showed that image patches can indeed be used for weight initialization. When initializing the model weights using my method, I was able to obtain a reasonable training process. My second research goal was then to examine how my proposed image patch-based weight initialization method performs in comparison to the state-of-the-art weight initialization methods. In Chapter 4.3.2, I also showed that my weight initialization method is able to achieve a similar classification performance as the state-of-the-art methods over the course of model training, when using an optimal value for the learning rate hyperparameter. When using a suboptimal value for the learning rate hyperparameter, however, my method even outperformed the state-of-the-art methods, as shown in Chapter 4.3.3. This indicates that the choice of the learning rate for model training becomes more robust when using my method. A suboptimal learning rate value does not reduce the classification performance as much as when using a state-of-the-art method. As a result, it would be necessary to train a model initialized by a state-of-the-art method for significantly more training epochs to receive the same classification performance as my method when using such a suboptimal learning rate value. Similar results were obtained for the momentum and the weight decay hyperparameter.

However, for my experiments in Chapter 4.3.2 and Chapter 4.3.3, I only initialized the model weights of the first convolutional layer using my method. The weights of the other layers were initialized using a state-of-the-art method (Kaiming Uniform[14] [45] or Fixup [150]). Therefore, I additionally examined whether it is possible to initialize multiple convolutional layers using my method, at least for models that contain a high number of layers. In Chapter 4.3.4, I showed that it is possible to initialize the weights of multiple convolutional layers of a model using my method. When using a suboptimal value for the learning rate, initializing the weights of multiple convolutional layers achieved even a higher classification performance than only initializing the weights of the first convolutional layer. However, in contrast to the weights of the first convolutional layer, the weights of the other convolutional layers are not applied to the input images but to the activations of their respective previous convolutional layer. The image patches, on the other hand, show parts of the input images but not parts of any layer activations. Why is it still beneficial then to use the image patches for initializing the weights of the convolutional layers after the first convolutional layer? I assume that the image patches may also match the activations of the convolutional layers after the first convolutional layer to some extent. These activations represent the image features detected by the model in the input images (e.g., simple textures, object parts). Some of these image features may be shown by the extracted image patches. Therefore, it is reasonable to initialize the weights of these other layers using image patches as well. As a result, I have shown that my image patch-based weight initialization method can be used to make the choice of the learning rate less critical.

---

[14] `https://pytorch.org/docs/stable/nn.init.html#torch.nn.init.kaiming_uniform_`

5 Exploiting Layer Activations to Balance the Training Dataset

I have shown in Chapter 4 that image information can be exploited to improve the training process of a Convolutional Neural Network-based (CNN) image classification model. In this chapter, I show that the layer activations of such a model can be exploited to improve model training as well. To explain how the layer activations are exploited, I get back to my example from Chapter 1. Suppose we have images of a certain type of plant that has been grown for agriculture. Each of these images shows such a plant either in a healthy state or suffering from a plant disease. We aim to classify the images into the two classes *healthy plant* and *plant with disease* in order to detect if a plant suffers from the disease or not. This detection is important for determining which plants may need treatment. Without treatment, the disease can spread quickly and destroy all plants, resulting in a poor harvest. To be able to classify the images into the two classes, we train a Convolutional Neural Network-based model on a training dataset containing the plant images until the model achieves a sufficient classification performance on a given test dataset. However, the model may never achieve the desired performance if the training dataset is class-imbalanced. A class-imbalanced dataset is a dataset that contains a significantly higher amount of images of one class compared to the other class of the dataset. The class with the higher amount of images is typically referred to as the majority class, while the class with the lower amount of images is typically referred to as the minority class. A class-imbalanced dataset usually occurs when it has not been possible to collect a sufficient amount of images of both classes for that dataset. For instance, we need to detect the disease before it spreads to the majority of the plants. However, this means that we are currently only able to collect a limited amount of images showing plants suffering from that disease. As a result, we are able to obtain a sufficient amount of images only from the healthy plants. Consequently, our training dataset is class-imbalanced. The class *healthy plant* is the majority class, while the class *plant with disease* is the minority class. However, this class imbalance has a negative impact on model training. During training, the model sees the training images in random mini-batches in order to learn an image feature representation of each class from these training images [53]. If the training dataset is class-imbalanced, however, the mini-batches will contain mainly majority class images as these images occur more frequently in the dataset. As a result, the model will see mainly majority class images during training in this case. Therefore, the model will only be able to learn an adequate image feature representation of the majority class but not an adequate image feature representation of the minority class. Nevertheless, it is still important to reliably detect the plants with the disease, as the disease poses a serious threat to these plants.

Various approaches have been suggested to address the problem of training a classification model using a class-imbalanced dataset. An overview of different methods is given in Chapter 5.1. A state-of-the-art method to approach model training using a class-imbalanced dataset is random undersampling [104]. Random undersampling balances the class-imbalanced dataset before model training. To balance the dataset, random undersampling keeps removing images of the majority class at random until the amount of majority class images is approximately equal to the amount of minority class images. When training the model with the balanced dataset, the model no longer sees mainly majority class images during model training but approximately an equal amount of images of both classes. As a result, for both classes, the model should now be able to learn an adequate image feature representation. Therefore, we expect the model to achieve a superior classification performance on the test dataset compared to the previous model that was trained using the original class-imbalanced dataset.

However, when removing images from the majority class at random, random undersampling does not pay attention to which of the majority class images it removes from the dataset. This is probably not an issue when the set of majority class images has a low intra-class variance, i.e., when all majority class images are similar to each other. If the set of majority class images has a high intra-class variance, however, removing majority class images at random may not be a good approach to balancing the dataset. A high intra-class variance indicates that the majority class images differ significantly from each other. Usually, the images may even belong to different subclasses of the majority class. The ImageNet [25] class *orange*, for instance, contains images showing a whole orange on an orange tree and images showing slices of an orange, as shown in Figure 5.1. Therefore, two potential subclasses of the class *orange* could be *whole orange* and *orange slices*. Both subclasses represent an orange, but each subclass represents an orange in a different form, and both forms have different image features. To be able to learn an adequate image feature representation of both subclasses, the model must see a sufficient amount of images of each subclass during model training. However, when removing majority class images at random, we cannot ensure that we retain a sufficient amount of images of both subclasses. Random undersampling may remove images mainly from only one of the subclasses. In this case, the model can only learn an adequate image feature representation of the subclass from which we still have a sufficient amount of images in the dataset. From the other subclass, however, the model is not able to learn an adequate image feature representation because random undersampling has removed most of the images of that subclass. It would be better to remove an equal amount of images from each subclass. However, we do not have the subclass information of the majority class to be able to identify which image is from which subclass.

To address this issue, I proposed a method named subclass-based undersampling (published in Lehmann and Ebner [83]). Subclass-based undersampling also balances a class-imbalanced dataset by undersampling the majority class. However, in order to balance the dataset, it does not remove arbitrary majority class images at random, as done by random undersampling. Instead, it randomly selects an equal amount of images from each subclass of the majority class. By doing this, I hoped to keep a sufficient amount of images of each of the subclasses. The selected majority class images are then merged

Figure 5.1: Potential subclasses of the class *orange* (left) and the class *baseball* (right) of the ImageNet [25] dataset.

with all of the minority class images in order to obtain the balanced dataset. Finally, the balanced dataset is used to train the model. However, to be able to select images from each subclass, the subclasses need to be identified beforehand. I suggested identifying the subclasses based on the activations of a higher layer of the model (i.e., a layer closer to the output layer). The activations of such a higher model layer represent specific high-level image features detected by the model [148]. Each subclass, on the other hand, is characterized by a certain set of such high-level image features. Therefore, I claim that clusters within the activations of a higher model layer correspond to the different subclasses of the majority class. Furthermore, for simplicity reasons, I have so far only considered a class-imbalanced dataset containing two classes, a majority class and a minority class. However, my method can also be used to balance a class-imbalanced dataset with multiple majority classes. In this case, my method must be applied to each majority class separately. The details of my method are presented in Chapter 5.2. To evaluate my method, I conducted several experiments. My first research goal was to examine whether the cluster information within the activations of a higher model layer can be used to undersample the majority class at all. This would be the case if a model trained using a training dataset balanced by my method achieves a superior classification performance compared to a model trained using the original class-imbalanced dataset. Furthermore, I expected that the model trained using the training dataset balanced by my method would achieve a superior classification performance compared to a model trained using a training dataset balanced by random undersampling. My second research goal was therefore to compare my method with random undersampling. Finally, my third research goal was to compare my method also to other state-of-the-art methods to address the class imbalance problem. In my experiments (Chapter 5.3), I showed that my method can be used to undersample a class-imbalanced dataset. Furthermore, I also showed that my method achieves superior results compared to random undersampling as well as other state-of-the-art methods to address the class imbalance problem with respect to two real-world datasets. The following contributions have been made: (1) a method was proposed that identifies the subclasses of a class, (2) an undersampling method was suggested that is based on the obtained subclass information, and (3) it was shown that the proposed undersampling method outperforms random undersampling as well as other state-of-the-art methods to address the class imbalance problem.

5.1 Related Work on Class-imbalanced Model Training

A state-of-the-art method to address the class imbalance problem is random undersampling [13, 77, 104]. Random undersampling balances the class-imbalanced training dataset before model training. To balance a class-imbalanced dataset, random undersampling removes image data samples at random from the majority class of the dataset until the amount of majority class samples and the amount of minority class samples of the dataset are approximately equal. In order to improve random undersampling, I suggested an alternative undersampling method. My suggested method does not remove majority class samples at random as random undersampling does, but it selects image data samples from each subclass of the majority class for the balanced dataset and discards all remaining majority class samples. By selecting image data samples from each subclass of the majority class, I expect the resulting undersampled set of majority class samples to still contain a sufficient number of image data samples from each of these subclasses. As a result, the undersampled set should have a similar intra-class variance as the original set of majority class samples. Random undersampling, on the other hand, may remove most of the image data samples from one of the subclasses and therefore reduces the intra-class variance, which may result in a model that was unable to learn an adequate representation of the majority class. However, besides my approach, there have been other studies that also suggested improved undersampling methods. Kubat and Matwin [72], for instance, proposed a method that undersamples the majority class by removing majority class samples that are noisy and majority class samples that are close to the decision boundary between the majority class and the minority class. The majority class samples that are close to the decision boundary are not favorable because even a tiny amount of noise could be enough to push them to the wrong side of the boundary. To identify majority class samples close to the decision boundary as well as noisy majority class samples, they use a nearest neighbor approach. Zhang and Mani [151], on the other hand, introduced a method that undersamples the majority class by removing those majority class samples that are closest to the minority class samples. Garcia and Herrera [37] proposed an undersampling method that is based on an evolutionary algorithm. They evaluate different subsets of the set of majority class samples by training a model from each of those subsets with the goal to find the model that achieves the highest classification accuracy. Koziarski [67] suggested a radial-based undersampling method that removes redundant majority class samples. The redundant majority class samples are identified by a Gaussian radial basis function. Liu et. al. [90] presented two undersampling techniques, EasyEnsemble and BalanceCascade. EasyEnsemble is based on an ensemble learning strategy. They use different subsets of the set of majority class samples to train different models. Finally, they combine the different models in order to obtain an ensemble model. BalanceCascade, on the other hand, is based on a sequential learning strategy. Again, they train multiple models using different subsets of the set of majority class samples. Then, they classify all majority class samples using these models. The majority class samples that are correctly classified by all models are removed from the dataset. They assume that these majority class samples are redundant because all models were able to classify them correctly. Finally, the models are retrained using the training dataset without the removed majority class samples. These steps are repeated in an iterative process. However, in contrast to my method, none of these approaches is based on clustering in order to undersample the majority class.

Nevertheless, there have also been studies that suggested undersampling methods based on clustering. Agrawal et. al. [3], for instance, proposed a method that searches for clusters within the majority class samples and then randomly removes a certain amount of majority class samples from each identified cluster. A similar method was also suggested by Sowah et. al. [133]. Majumder et. al. [97], on the other hand, introduced a method that also searches for clusters within the majority class samples, but then they do not randomly remove a certain amount of majority class samples from each identified cluster. Instead, they aim to identify redundant majority class samples within each cluster and then remove those redundant majority class samples. To identify redundant majority class samples within a cluster, they calculate the vector angle similarities between all majority class samples in the cluster. Tsai et. al. [137] also aimed to find an approach to selecting majority class samples from each cluster for the balanced dataset that is better than an approach that selects majority class samples at random. They compared different selection approaches (e.g., an approach based on a genetic algorithm). Yen and Lee [147], on the other hand, suggested an undersampling method based on clustering as well. However, they search for clusters within the entire dataset (i.e., within the majority and minority class samples). Then, they remove a specific amount of majority class samples from each identified cluster using a distance-based selection approach. They determine the amount of majority class samples to remove from a cluster based on the ratio between the number of majority class samples and the number of minority class samples in that cluster. Ng et. al. [107] also divide the dataset into different subsets. However, they do not use clustering to obtain the subsets but hashing. Finally, they select majority class samples from each obtained subset using a distance-based selection approach. However, all of these methods have been proposed for tabular data. They cannot be applied to image data directly. Image data is high-dimensional, and therefore it is difficult to identify meaningful clusters in image data space. Thus, I apply clustering to the high-level image features represented by the activations of a higher convolutional model layer instead. Koziarski [68] also suggests an undersampling method for image data. The proposed method is applied to high-level image features as well. However, Koziarski [68] simply applies random undersampling in image feature space. In contrast, my method searches for clusters within the majority class samples in image feature space, and then selects which majority class samples to keep from each identified cluster for the balanced dataset.

However, undersampling is not the only method to address the class imbalance problem. Other state-of-the-art methods to address this problem include oversampling and approaching the problem through the loss function during model training. Oversampling also balances the training dataset before model training. In order to balance a dataset, however, oversampling does not reduce the amount of majority class samples of the dataset. Instead, oversampling adds additional minority class samples to the dataset until the amount of minority class samples and the amount of majority class samples are approximately equal. A simple method to obtain additional minority class samples is to duplicate some of the existing minority class samples [104]. Alternatively, in order to avoid increasing the size of the dataset, we could also increase the frequency of selecting minority class samples for the mini-batches during model training. The minority class samples that should be duplicated (or selected more frequently for a mini-batch) are selected randomly. Thus, this naive oversampling approach is commonly referred to as random oversampling. In order to improve random oversampling, Singh et. al. [130] introduced an alternative oversampling method. Their method searches for clusters within the minority class samples and then selects the minority class samples for oversampling that are closest to their cluster centers. They assume that the minority class samples closest to the cluster centers are most representative of the minority class. Another alternative oversampling approach is to artificially obtain novel minority class samples, rather than simply duplicating already existing minority class samples. Several studies suggested methods to create synthetic minority class samples [5, 18, 44, 105]. Furthermore, Koziarski [68] suggested a method that uses a combination of oversampling and undersampling. First, the method applies random oversampling to the minority class samples in image space. Then, a model is trained using this oversampled training dataset. Finally, the method applies random undersampling to the majority class samples in image feature space and fine-tunes the higher model layers (i.e., the layers closer to the output layer) using the undersampled training dataset. Besides undersampling and oversampling, approaching the class imbalance problem through the loss function during model training is another state-of-the-art method to address this problem. In this case, information about the class imbalance of the dataset is added to the loss function. A simple approach is to add weights to the loss function that reflect the percentage of the images of each class of the training dataset (class-weighted loss) [104]. However, there have also been other studies that suggested more sophisticated loss functions to address the class imbalance problem [15, 29, 104, 124]. Furthermore, a vast amount of alternative methods to these state-of-the-art methods have been suggested to address this problem as well. Huang et. al. [55], for instance, proposed a method to obtain more discriminative image feature representations from the model in order to better separate minority and majority class samples in image feature space of the model output layer. Khan et. al. [62], on the other hand, introduced a method based on cost-sensitive learning. They assign a higher misclassification cost to minority class samples than to majority class samples. Zhang et. al. [153] suggested a novel classification method based on category centers in image feature space. Li et. al. [87] proposed a method based on a balanced group softmax. Wang et. al. [140] introduced a method based on dynamic curriculum learning, and Pouyanfar et. al. [117] suggested using dynamic sampling.

5.2 Subclass-based Undersampling

We aim to train a Convolutional Neural Network-based (CNN) image classification model using a training dataset (X^D_{imb}, Y^D_{imb}) containing images of two classes, c_{maj} and c_{min} (Equation 5.1).

$$(X^D_{imb}, Y^D_{imb}) \;=\; (X^D_{c_{maj}}, Y^D_{c_{maj}}) \cup (X^D_{c_{min}}, Y^D_{c_{min}}) \tag{5.1}$$

However, the training dataset is class-imbalanced, i.e., the amount of images of one class is significantly higher than the amount of images of the other class. The class with the higher amount of images is usually referred to as the majority class, while the class with the lower amount of images is referred to as the minority class. Hereinafter, I assume that class c_{maj} is the majority class and class c_{min} is the minority class of the dataset (Equation 5.2).

$$|(X^D_{c_{maj}}, Y^D_{c_{maj}})| \;\gg\; |(X^D_{c_{min}}, Y^D_{c_{min}})| \tag{5.2}$$

Unfortunately, this class imbalance of the training dataset negatively impacts model training. We can address this issue, for instance, by balancing the dataset before training the model, i.e., we adjust the amount of images of class c_{maj} or class c_{min} so that the final amounts of images of both classes become approximately equal. A model trained using the resulting balanced dataset (X^D_{bal}, Y^D_{bal}) should achieve a superior classification performance compared to a model trained using the initial class-imbalanced dataset.

My proposed method balances such a class-imbalanced dataset by undersampling the majority class of the dataset. To undersample majority class c_{maj}, it selects a subset $(X'^D_{c_{maj}}, Y'^D_{c_{maj}})$ of the training images of that majority class. The selected subset of majority class training images and all training images of the minority class form the balanced dataset (Equation 5.3).

$$(X^D_{bal}, Y^D_{bal}) = (X'^D_{c_{maj}}, Y'^D_{c_{maj}}) \cup (X^D_{c_{min}}, Y^D_{c_{min}}), \quad (X'^D_{c_{maj}}, Y'^D_{c_{maj}}) \subset (X^D_{c_{maj}}, Y^D_{c_{maj}}) \tag{5.3}$$

As a consequence, the balanced dataset exhibits a smaller difference between the amount of images of the majority class and the amount of images of the minority class compared to the initial class-imbalanced dataset. However, my method does not simply select the subset from the majority class images at random, as done by the state-of-the-art random undersampling method (for more details, see Chapter 5.1). Instead, it selects the subset from the majority class images based on information about potential subclasses sc_{maj} of c_{maj}. Subclasses of a class occur when the images of that class have a high intra-class variance, as shown by Nguyen at. al. [109] or Wei et. al. [141]. Multiple subclasses can be identified, for instance, for some of the classes[1] of the ImageNet [25] dataset used in the Large Scale Visual Recognition Challenge [123]. The ImageNet class *orange*, for instance, contains images showing a whole orange and images showing slices of an orange (Figure 5.1). Therefore, two potential subclasses of the class *orange* could be *whole orange* and *orange slices*. Each of the two subclasses shows an orange in a different physical form. As a result, the images of one subclass have different image features compared to the images of the other subclass, although the images of both subclasses belong to the same class *orange*. Hence, the model should be trained using a sufficient amount of images from both subclasses. Otherwise, the model will learn to recognize images of only one of the two subclasses. As a result, when the majority class has a high intra-class variance, it is not a good approach to selecting the subset from the majority class images of the initial class-imbalanced dataset at random without considering subclasses, as done by random undersampling. In this case, it cannot be ensured that a sufficient amount of images was selected from all subclasses of that class. We might have selected a sufficient amount of images from one subclass but only a tiny amount of images from another subclass. Hence, I aim to equally select images from all subclasses of the majority class when undersampling that class. However, I have only considered a dataset containing images of two classes so far. Nevertheless, my proposed method is also applicable to class-imbalanced image datasets containing more than two classes. If such a dataset contains multiple minority classes, my method is the same as described above because it only focuses on the majority class to balance the dataset. If the dataset contains multiple majority classes, on the other hand, my method needs to be applied to each majority class separately. However, the method itself is the same when applied to each majority class. Hence, to explain my method in detail, I will continue to consider a class-imbalanced dataset that contains only two classes, a majority class c_{maj} and a minority class c_{min}, without losing the generality of my method.

To undersample the majority class of the class-imbalanced training dataset using my proposed method, two steps need to be performed. First, potential subclasses sc_{maj} of the majority class c_{maj} need to be identified within the training images $X^D_{c_{maj}}$ of class c_{maj} of the initial class-imbalanced dataset. However, it does not only need to be identified out how many subclasses the majority class has. It is also necessary to know, for each image of $X^D_{c_{maj}}$ of the majority class, to which identified subclass this image belongs. Images of the same subclass should be rather similar, while images of different subclasses should be rather different. After identifying the subclasses sc_{maj} of the majority class c_{maj}, representative images are selected from each identified subclass in the second step of my method. By selecting images from each subclass, I ensure that as much information as possible is retained from the majority class, as images from different subclasses are rather different as well. In contrast, images from the same subclass tend to be similar. Therefore, it is not necessary to select all images from a subclass, but some of these images can be omitted to balance the class-imbalanced dataset. The selected majority class images make the required subset $(X'^D_{c_{maj}}, Y'^D_{c_{maj}})$.

---

[1] `https://www.image-net.org/challenges/LSVRC/2017/browse-synsets.php`

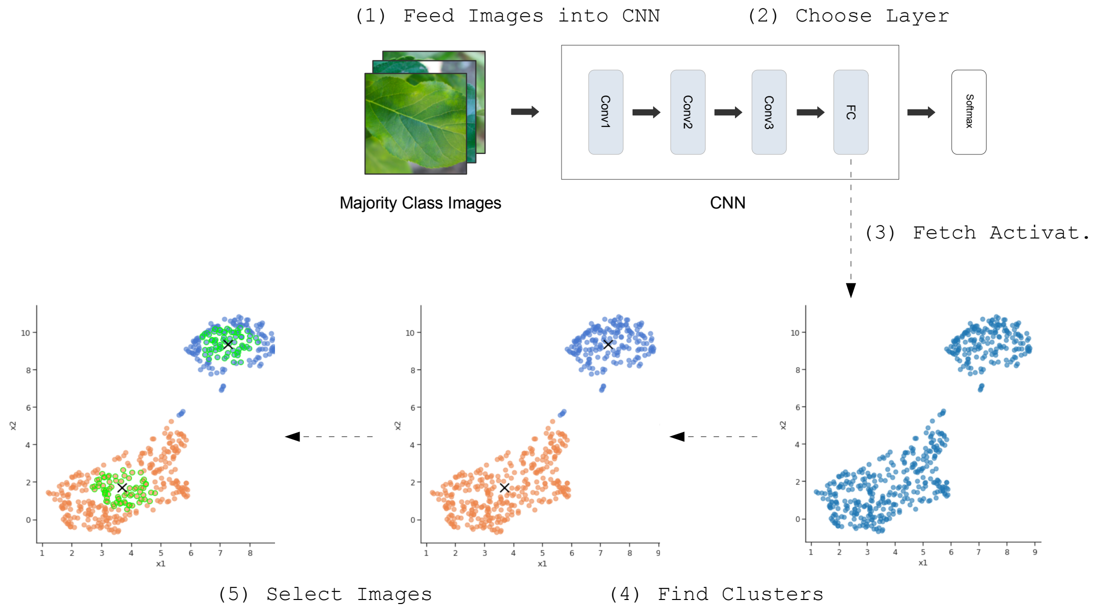


Figure 5.2: The proposed method: (1) Feed majority class images into the Convolutional Neural Network-based image classification model (CNN) that was trained on the class-imbalanced dataset, (2) choose a higher model layer, (3) fetch activations and project them to 2D, (4) find clusters within the activations, and (5) select images from each identified cluster to undersample the majority class.

This subset and all training images of the minority class form the balanced dataset. Finally, using this balanced dataset, a proper model can be trained. An overview of my proposed method is shown in Figure 5.2. Hereinafter, I describe in more detail how to identify potential subclasses sc_{maj} of the majority class c_{maj} within the training images $X^D_{c_{maj}}$ of class c_{maj} (Chapter 5.2.1) and how to select majority class images from each identified subclass sc_{maj} to undersample the majority class c_{maj} (Chapter 5.2.2).

5.2.1 Identifying Subclasses of the Majority Class

To identify potential subclasses sc_{maj} of the majority class c_{maj}, I use an approach based on the work of Ngyuen et. al. [109]. However, the method of Nguyen et. al. [109] was introduced to visualize multifaceted image features learned by a Convolutional Neural Network-based (CNN) model during model training. My proposed method, in contrast, identifies potential subclasses of a class, as shown in steps (1) to (4) in Figure 5.2, with the goal of using the subclass information to undersample that class (Chapter 5.2.2). Below, I will explain how such potential subclasses sc_{maj} of the majority class c_{maj} can be identified within the training images of class c_{maj}. First, the image features of all training images $X^D_{c_{maj}}$ of the majority class c_{maj} need to be obtained. To receive these image features, an initial model m_0 is trained using the entire class-imbalanced training dataset, i.e., all training images of the majority class c_{maj} and all training images of the minority class c_{min}. It is important for my method that all images of the dataset are of the same image size. If they are not, they need to be resized to a specific image size before model training. After training, the initial model m_0 is received. However, as the amount of minority class images is significantly lower than the amount of majority class images, the initial model did not see many minority class images during model training but mainly majority class images. Thus, I do not expect the initial model to have learned a proper representation of the minority class. Nevertheless, I expect the model to have learned at least a sufficient representation of the majority class. After training, the image features of the training images $X^D_{c_{maj}}$ of majority class c_{maj} can be obtained using the model. To receive the image features, I first freeze the model because I do not want to change the weights of the model anymore. Then, all of the training images of the majority class are fed into the model again. As a result, each of the training images of $X^D_{c_{maj}}$ gets classified by the model. However, I am not interested in the classification result but in the activations that are created at each layer of the model when feeding a training image $x^D_{c_{maj}} \in X^D_{c_{maj}}$ into the model again. These activations represent the image features that the initial model m_0 detected to classify this training image. To be able to identify potential subclasses sc_{maj} of the majority class c_{maj}, characteristic image features of these subclasses need to be found.

As shown by Zeiler and Fergus [148], each model layer of a Convolutional Neural Network-based model detects specific types of image features that are used to classify images fed into that model. The lower layers (i.e., the layers closer to the input layer) detect low-level image features (e.g., colors, corners, simple textures), while the higher layers (i.e., the layers closer to the output layer) detect high-level image features (e.g., object parts, objects in various poses). Which kind of image features, however, should be used to identify the subclasses of a class? It might be possible to find the subclasses using the image features detected by the lower layers of the initial model. However, these subclasses are not the subclasses I aim to find because they are subclasses with respect to low-level image features (e.g., colors, corners, simple textures). The ImageNet class *orange*, for instance, may contain subclasses with respect to different color distributions, such as a subclass containing images showing an orange on a black background, a subclass containing images showing an orange on a white background, or a subclass containing images showing an orange on a colored background (e.g., on an orange tree, in a kitchen). However, all low-level feature-based subclasses may contain images showing a *whole orange* and images showing *orange slices* (for more information, see Chapter 2.3). Instead, I aim to receive images showing a *whole orange* and images showing *orange slices*, each in a separate subclass that is characterized by its specific semantic concept regardless of the background. Therefore, low-level image features are not adequate for identifying subclasses of different semantic concepts, as these types of image features are most likely shared by several such subclasses of a class. It is necessary to identify subclasses with respect to different semantic concepts, such as a *whole orange* or *orange slices*. Semantic concepts are characterized by object parts or whole objects, i.e., high-level image features. Therefore, to identify subclasses with respect to different semantic concepts, high-level image features from the higher model layers need to be used (Figure 5.3). However, the image features from the output layer of the model

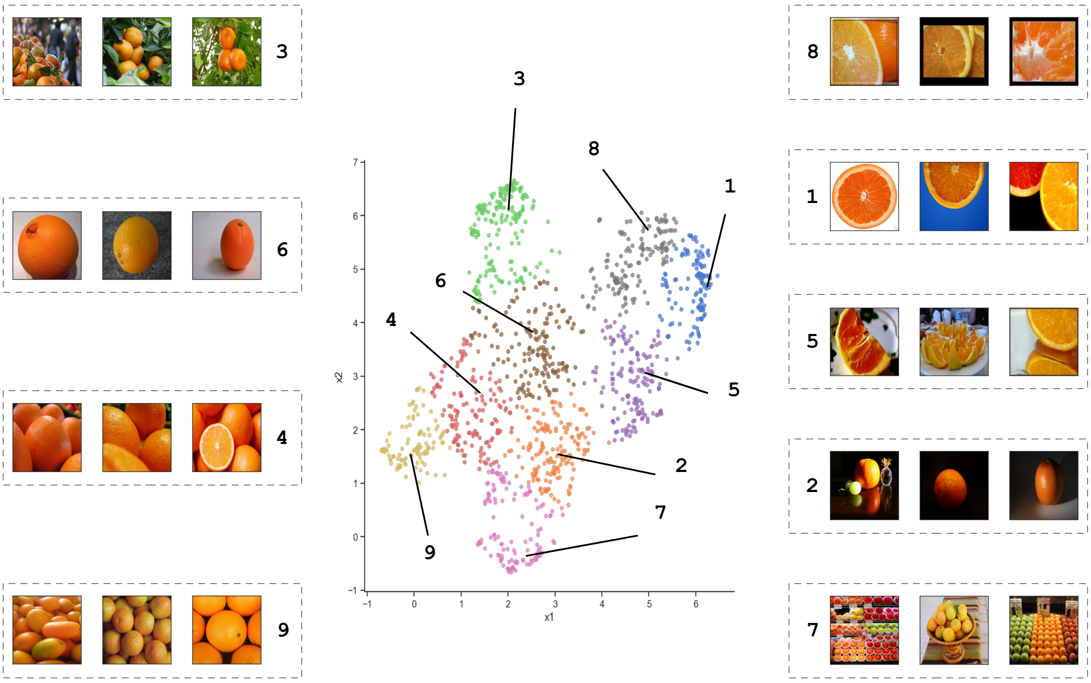


Figure 5.3: The Clusters that were found within the activations of the ImageNet [25] class *orange* with respect to a higher model layer represent different subclasses of that class regarding different semantic concepts (e.g., cluster 1: sliced orange, cluster 3: orange on a tree, cluster 9: multiple oranges).

cannot be used because the image feature representation of this layer does not contain any subclass information anymore. The optimization objective of model training is to find an image feature representation from the first to the final model layer that forces images of the same class to be located close to each other in image feature space and images of different classes to be located far from each other in image feature space. As a result, at the output layer, all images of a class will be close to each other, which also includes images of different subclasses of the class (for more information, see Chapter 2.3). Hence, it will be difficult to identify subclasses using the image features of the final layer. As a consequence, the image features detected by one of the higher layers before the output layer need to be used. However, which of these higher layers is the best for identifying the subclasses depends on the classification problem that should be solved. This can be a different higher model layer for different classification problems. To find the best layer for a specific classification problem, different higher model layers are evaluated with respect to the identifiable clusters within the activations of each layer. The layer with the best cluster quality is selected because clusters are important for identifying the subclasses using my proposed method.

To identify the subclasses sc_{maj} of the majority class c_{maj} based on image features of a particular higher model layer l, I first feed the N_{maj} majority class images $X^D_{c_{maj}}$ of the initial class-imbalanced training dataset into model m_0. Then, the activations of each image $x^D_{c_{maj}}$ are fetched from layer l. The activations represent the image features of these images detected by layer l. If layer l is a convolutional layer, the activations of an image are obtained in the form of a three-dimensional tensor. This three-dimensional activation tensor needs to be flattened to an activation vector. However, this step is only necessary if layer l is a convolutional layer. If layer l is a linear layer, on the other hand, this step can be omitted because the activations from linear layers are in vector form already. As a result, an activations vector $a^l(x^D_{c_{maj}})$ of a layer-specific length M^l is obtained for each of the N_{maj} training images $x^D_{c_{maj}}$. Finally, these activation vectors are concatenated into a matrix A^l_D of size $N_{maj} \times M^l$. Each row of this matrix represents the activations of a majority class image $x^D_{c_{maj}}$, generated at layer l.

After obtaining the activations A^l_D of the majority class images from the chosen higher layer l, I aim to search for clusters in these activations. I expect the clusters to represent the subclasses because each identified cluster h^l will contain images of the class c_{maj} with similar semantic image features in the form of their activations from higher layer l. Thus, the images of a particular semantic concept should be close to each other in image feature space, while they should be farther apart from the images of another semantic concept. This corresponds to the expectation of a subclass. As a result, the different subclasses of the majority class should be identifiable by finding clusters among the images of that class in image feature space of layer l. The clustering result with respect to the ImageNet class *orange*, for instance, may have one cluster containing images showing a *whole orange* and another cluster containing images showing *orange slices*. However, the matrix A^l_D, which contains the activations of the majority class images $X^D_{c_{maj}}$ of class c_{maj}, is usually high-dimensional because of the large amount of activations that are generally obtained from layer l. Unfortunately, identifying clusters in high-dimensional spaces does not work well, as pointed out by Chen et. al. [19]. Clustering algorithms use distance metrics to identify clusters, but distance metrics are not effective in high-dimensional spaces. However, as pointed out by Domingos [28], the data samples of most applications are located within a low-dimensional subspace within this high-dimensional space. Thus, I use dimensionality reduction to project matrix A^l_D onto such a low-dimensional subspace, as suggested by Chen et. al. [19].

Chapter 6.3.2 shows a comparison of suitable projection approaches with respect to finding well-separated clusters, which I evaluated with activation data from different datasets. The best projection result was obtained with a combination of the linear dimensionality reduction technique PCA (Principal Component Analysis) [116] and the non-linear dimensionality reduction technique UMAP (Uniform Manifold Approximation and Projection) [98] (for more information about PCA and UMAP, see Chapter 3.1). A similar approach was also used by Nguyen et. al. [109]. Thus, I use a combination of PCA and UMAP to project matrix A^l_D. However, before projecting the matrix A^l_D, each of its values need to be normalized as a preprocessing step for the dimensionality reduction (for more details, see Chapter 3.1). After normalizing each value of matrix

A^l_D, the dimensionality of A^l_D is reduced from $N_{maj} \times M^l$ down to $N_{maj} \times 50$ using PCA. Then, in a second reduction step, the dimensionality is further reduced from $N_{maj} \times 50$ down to $N_{maj} \times 2$ using UMAP. As a result, the projected matrix $r^l(A^l_D)$ is obtained from the learned projection model r^l applied to matrix A^l_D.

In this projected matrix $r^l(A^l_D)$, I search for clusters. I expect these clusters to represent the subclasses of the majority class c_{maj} (Figure 5.3). To determine the best approach to finding the clusters, I evaluated different clustering algorithms applied to activation data from different layers of a Convolutional Neural Network-based model. Furthermore, I also tested each clustering method for different image datasets. This evaluation is presented in Chapter 6.3.2. I found that the k-Means [95] clustering algorithm is generally best suited for activation data, which was also suggested by Chen et. al. [19]. Thus, I chose k-Means to search for clusters in the matrix $r^l(A^l_D)$. However, k-Means requires setting hyperparameter k. Hyperparameter k specifies how many clusters k-Means should search for in the matrix. Unfortunately, it is not known in advance how to set k. It cannot be predicted how many meaningful subclasses of c_{maj} can be identified in the matrix. Thus, it is also not known how many clusters k-Means should search for. Since I reduced the dimensions of the activations of the N_{maj} images of the majority class c_{maj} from M^l dimensions in A^l_D down to 2 dimensions in $r^l(A^l_D)$ (the rows of the matrices), the resulting compressed two-dimensional activations for each of the images can be visualized in a scatter plot. In this scatter plot, it might be possible to visually identify clusters. Thus, hyperparameter k of k-Means could be set according to the number of clusters that were identified in the scatter plot. However, it cannot be guaranteed that this method works well in every case. It may not always be easy to identify all clusters visually. Thus, I use a different approach to finding a good value for hyperparameter k. I simply test different values for k, e.g., values between 2 and 9. For each of these values, I apply k-Means to matrix $r^l(A^l_D)$ using this value for hyperparameter k. As a result, for each of these values, I obtain a set of clusters $h^l \in h^l_1, .., h^l_k$ found by k-Means and evaluate the received clusters with respect to their cluster quality. The value for k that results in the clusters achieving the best overall cluster quality is chosen.

I use the silhouette score [121] to evaluate the clusters. The silhouette score is a cluster quality metric, which is calculated using the intra-cluster distance and the nearest-cluster distance of each data sample of a set of clusters (for more details, see Chapter 3.3). In our case, a data sample is the compressed activation vector $r^l_D(A^l_D)_{i,:} = r^l(a^l(x^D_{c_{maj}}))$ of an image $x^D_{c_{maj}}$ in a row i of matrix $r^l(A^l_D)$. Chen et. al. [19] reported that the silhouette score is well suited for evaluating clusters identified in activation data of Convolutional Neural Network-based models. The value of the silhouette score of an obtained set of clusters ranges from -1 to 1. A silhouette score close to 1 means that we have well-separated clusters. A silhouette score close to -1, however, means that a majority of data samples should be located in the nearby cluster rather than in their current cluster. Thus, a higher silhouette score for a set of clusters reflects a better cluster quality and is therefore favorable. After computing the silhouette score for the resulting clusters from each of the considered values for k, I pick the clusters corresponding to the value for k that achieved the highest silhouette score.

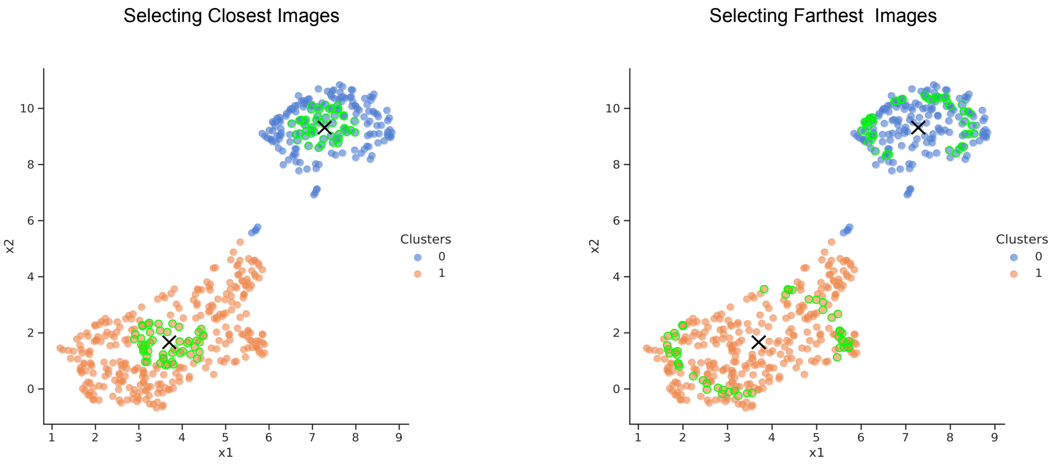


Figure 5.4: Selecting the images from each cluster to undersample the majority class: (a) Selecting the images closest to their respective cluster center, and (b) selecting the images farthest from their respective cluster center within an $80th$ percentile to avoid outliers.

5.2.2 Undersampling of the Majority Class

After the subclasses sc_{maj} of the majority class c_{maj} are identified in the form of the obtained clusters h^l in image feature space of the chosen higher layer l (Chapter 5.2.1), the majority class can be undersampled using these subclasses, as shown in step (5) in Figure 5.2. To undersample majority class c_{maj}, a subset of the training images of c_{maj} is selected. This reduces the difference between the amount of images of the majority class c_{maj} and the amount of images of the minority class c_{min}. For instance, if the number of majority class images is 300 and the number of minority class images is 100, then a subset of 100 images of the majority class may be selected to undersample this class. However, the images for this subset are not selected at random. I expect images of each subclass of the majority class to have different image features compared to the images of another subclass. Thus, to retain as much information as possible for model training, I uniformly select majority class images from each identified subclass through its corresponding cluster. If I found 2 clusters, and I aim to select 100 images in total, for instance, then I select 50 images from each of the 2 clusters. Unfortunately, the clusters do not contain the images themselves but the compressed activation vectors $r^l(A^l_D)_{i,:} = r^l(a^l(x^D_{c_{maj}}))$ of the images $x^D_{c_{maj}} \in X^D_{c_{maj}}$ in each row i of matrix $r^l(A^l_D)$. However, as these compressed activation vectors can be directly associated with their corresponding images $x^D_{c_{maj}}$, the clusters can be used for selecting the subset. Thus, I refer to the activation vectors contained in the clusters as images below.

To select n_{h^l} images from a cluster h^l, I consider different approaches. Similar to the state-of-the-art random undersampling method (for more details, see Chapter 5.1), the images from a cluster can be selected at random. However, I also consider two other more sophisticated approaches. First, I select from a cluster h^l the images $X_{near}^{h^l}$ that are closest to the center ct_{h^l} of that cluster h^l (Equation 5.4), as shown in Figure 5.4 (left). The images that are located closest to the center of the cluster are likely the images that are most representative of the cluster.

$$X_{near}^{h^l} = \underset{h'^l \subset h^l, |h'^l| = n_{h^l}}{\arg\min} \sum_{x_{c_{maj}}^{h^l} \in h'^l} d(x_{c_{maj}}^{h^l}, ct_{h^l}) \tag{5.4}$$

However, the images $X_{near}^{h^l}$ of a cluster h^l, which are closest to the cluster center of the cluster, could also be visually quite similar to each other because they are most likely close to each other in image feature space as well. Hence, I also consider a second approach. I select the images $X_{far}^{h^l}$ that are farthest from the cluster center ct_{h^l} of h^l, as shown in Figure 5.4 (right). These images should be still significantly different from images of other clusters, but they should also be at least slightly more different from each other. However, to avoid selecting outliers, such as noisy or mislabeled images, I only consider images within the $80th$ percentile $\eta_{.8}(h^l)$ of cluster h^l for the selection (Equation 5.5).

$$X_{far}^{h^l} = \underset{h'^l \subset \eta_{.8}(h^l), |h'^l| = n_{h^l}}{\arg\max} \sum_{x_{c_{maj}}^{h^l} \in h'^l} d(x_{c_{maj}}^{h^l}, ct_{h^l}) \tag{5.5}$$

After selecting the subset of images of the majority class c_{maj} from each cluster h^l using one of the three approaches described above (selecting randomly, selecting samples closest to the cluster center, or selecting samples farthest from the cluster center), this subset is merged with all images of the minority class c_{min} in order to obtain the balanced dataset. Finally, a final model m_1 is trained using this balanced dataset.

5.3 Experiments

To address the problem of training a Convolutional Neural Network-based (CNN) image classification model using a class-imbalanced training dataset, I proposed a method that balances the dataset beforehand by undersampling its majority class based on potential subclasses of the majority class. I describe the details of my method in Chapter 5.2. To evaluate my method, I conducted several experiments. The general setup of these experiments is described in Chapter 5.3.1. In a first experiment, I tested if a model trained using a training dataset balanced by my method is able to achieve a significantly higher classification performance than a model trained using the corresponding initial class-imbalanced training dataset. This way, I aimed to find out whether my method is beneficial for balancing a dataset at all. Furthermore, I also tested if a model trained using the training dataset balanced by my method is able to achieve a significantly higher classification performance than a model trained using the training dataset balanced by random undersampling [104]. The results are presented in Chapter 5.3.2. To ensure that the results of my experiment in Chapter 5.3.2 are not dependent on the model architecture that I used for the experiment, I repeated the experiment using two alternative model architectures. The results are presented in Chapter 5.3.3. Finally, I evaluated in a third experiment my proposed subclass-based undersampling method in comparison to two other state-of-the-art methods to address the class imbalance problem, random oversampling [104] and a class-weighted loss function [104]. The results are presented in Chapter 5.3.4.

5.3.1 Experimental Setup

To test my proposed subclass-based undersampling method in comparison to different state-of-the-art methods (e.g., random undersampling [104]), I conducted several experiments using two different class-imbalanced real-world datasets. For each experiment, I used the same experimental setup. I first addressed the class imbalance problem of the training dataset of the respective real-world dataset using either my proposed subclass-based undersampling method or a state-of-the-art method. To undersample the training dataset using my method, I used the approach described in Chapter 5.2. I first trained an initial Convolutional Neural Network-based (CNN) image classification model using the original class-imbalanced training dataset. After receiving this initial model, I fed all training images of the majority class (that should be undersampled) into the model again. The model then creates the activations of these images at each model layer. To obtain the activations required for undersampling the majority class using my subclass-based undersampling method, I picked a higher model layer (i.e., a layer closer to the output layer) and fetched the activations from this layer. Then, I searched for clusters within these activations. After identifying a set of clusters, I used them to undersample the respective majority class. I assumed that the identified clusters reflect the subclasses of the majority class. To undersample the majority class, I selected a specified number of majority class training images from each identified cluster. The selected images formed the undersampled set of training images of the majority class. Finally, I merged this undersampled set of training images of the majority class with the training images of the other classes of the dataset, which include all training images of the minority classes and optionally the undersampled sets of other majority classes if the dataset contains more than one majority class. The resulting dataset is the balanced dataset obtained by my method. As a result, I received two balanced datasets, the dataset balanced by my subclass-based undersampling method and a dataset balanced by a state-of-the-art method (e.g., random undersampling). Then, I trained a final model for each balanced dataset. Depending on the experiment, I trained a model for each dataset either a single time or 6 times for a specified number of training epochs. When I trained a model 6 times, I trained the respective model with a different seed value each time. I took each seed value from a set of 6 random seed values. The same 6 random seed values were used for both models. A seed value sets the random number generator used for model training[2]. By using different seed values, I should have obtained a more reliable classification performance of the respective models because different seed values for a model could lead to a certain variance in the corresponding classification performances. However, I only trained a model 6 times, if the model or the dataset were not too complex. Otherwise, I only trained each model a single time. After model training, I tested each model on the respective test dataset to receive its classification performance. As a result, depending on how many times I trained the models, I obtained for each model either one or 6 classification metrics (one for each seed). When I obtained only a single classification metric for each model, I directly compared the classification performances of the two models using their respective classification metrics. When I obtained 6 classification metrics for each model, on the other hand, I calculated the median and the standard deviation from these 6 classification metrics in order to compare the classification performances of the two models.

I conducted my experiments on two class-imbalanced real-world datasets taken from two image classification competitions that are hosted by the data science website Kaggle[3], the Plant Pathology 2020 competition and the Nature Conservancy Fisheries Monitoring competition. A few sample images of both datasets are shown in Figure 5.5. The images of the Plant Pathology dataset[4] [136] show apple trees that are either healthy or that suffer from a particular plant disease. These images are grouped into 4 different classes: The *rust disease* class (622 training data samples), the *scab disease* class (592 training data samples), the *healthy* class (516 training data samples), and the *multiple diseases* class (91 training data samples). The dataset is class-imbalanced because it contains a significantly lower amount of images of the *multiple diseases* class compared to the other three classes of the dataset. Therefore, I considered the *multiple diseases* class to be the minority class and the other three classes to be the majority classes. As a result, I aimed to undersample these three majority classes. The Fisheries Monitoring dataset[5], on the other hand, contains images taken from the surveillance camera on different fishing boats in order to detect the catch of illegal fish species. The images of the dataset are grouped into 8 different classes: The *Albacore tuna* class (*ALB*, 1719 training data samples), the *Yellowfin tuna* class (*YFT*, 734 training data samples), the *Bigeye tuna* class (*BET*, 200 training data samples), the *various sharks* class (*SHARK*, 176 training data samples), the *Dolphinfish* class (*DOL*, 117 training data samples), the *Moonfish* class (*LAG*, 67 training samples), the *another fish species* class (*OTHER*, 299 training data samples) and the *no fish in the image* class (*NoF*, 465 training data samples). The dataset is class-imbalanced because it contains a significantly higher amount of images of the *ALB* class compared to the other seven classes of the dataset. Therefore, I considered the *ALB* class to be the majority class and the other seven classes to be the minority classes. As a result, I aimed to undersample the *ALB* class as the majority class.

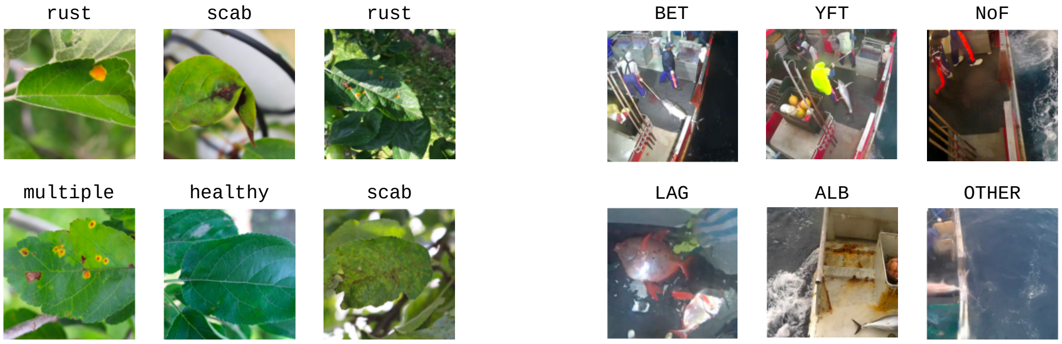
---

[2] `https://pytorch.org/docs/stable/notes/randomness.html`

[3] `https://www.kaggle.com`

[4] `https://www.kaggle.com/competitions/plant-pathology-2020-fgvc7/data`

[5] `https://www.kaggle.com/competitions/the-nature-conservancy-fisheries-monitoring/data`



Plant Pathology

Fisheries Monitoring

Figure 5.5: Sample images of the datasets used in my experiments, the Plant Pathology 2020 [136] dataset and the Nature Conservancy Fisheries Monitoring dataset.

In all of my experiments, I used the same training setup for the respective models. To train the models, I used a pre-trained model with a ResNet50 architecture (50-layer Residual Network) [46] that was trained on the ImageNet [25] dataset (only in Chapter 5.3.3 I also tested other model architectures). To fine-tune the pre-trained model using the respective dataset, I replaced the ImageNet-related output layer of the pre-trained model with the following model layers: A special pooling layer - batch normalization layer [61] - dropout ($p = 0.25$) [134] - fully-connected linear layer (layer size: 512) - batch normalization layer - dropout ($p = 0.5$) - fully-connected linear layer (layer size: $numberofclasses$). I used the ReLU activation function (Rectified Linear Unit) [39] for the pre-activations of the hidden linear layer. The special pooling layer consisted of a combination of an average pooling layer and a max pooling layer [88, 110] that were applied in parallel. Then, the results of both pooling layers were concatenated. Before model training, however, I first resized all training images of the respective dataset to 224 × 224 pixels. Furthermore, to obtain more variations of the training images, I used different data augmentation techniques [128]. For the Plant Pathology dataset, I used the data augmentation techniques that were used by the winner[6] of the original corresponding competition. These techniques included randomly flipping training images horizontally and vertically, randomly rotating and scaling training images, randomly shifting training images, adjusting their brightness and contrast, and applying blur. For the Fisheries Monitoring dataset, on the other hand, I chose almost the same data augmentation techniques. I only did not flip the training images vertically. Images showing the leaves of an apple tree could appear in a wide range of orientations. Images taken from the surveillance camera of the fishing boats, on the other hand, will most likely not appear to be upside down. Thus, I excluded flipping the training images of the fisheries monitoring dataset vertically. In order to train the model for both datasets, I used a mixed precision training setup [101]. Mixed precision training allowed me

---

[6] `https://github.com/ant-research/cvpr2020-plant-pathology`

to shrink the memory usage of the model in order to increase the batch size for model training to 25. Then, I trained the model in two stages using a discriminative fine-tuning strategy [54]. I first trained only the initial model weights of the newly added model layers (replacement for the ImageNet-related output layer) for 3 training epochs using the Adam[7] optimizer [65]. The model weights of all remaining layers (the ResNet50 layers) were not changed. Then, I fine-tuned the model weights of all layers of the model for 8 more training epochs using the Adam optimizer as well. To set the learning rate for model training, I used a cyclical learning rate schedule[8] [132] with an initial learning rate value of 0.01.

To evaluate the models for each dataset, I used the Kaggle submission system from each corresponding competition. The test images of each dataset are divided into two groups by Kaggle, a set of private test images and a set of public test images. While a competition is active, Kaggle only shows the results of a model on the public test images. When the competition ended, Kaggle also shows the results of the tested models on the private test images. The participant with the best result on these private test images wins the respective competition. However, the Plant Pathology 2020 competition and the Nature Conservancy Fisheries Monitoring competition were both no longer active. As a result, the public and the private test images from both corresponding datasets were already available. Thus, I chose to evaluate my models using the private test images as these are also important to win the competition. Furthermore, as I used the Kaggle submission system to evaluate the models, I used the classification performance metrics from each competition for evaluating my own models. Hence, I tested my Plant Pathology models using the ROC AUC[9] metric and the Fisheries Monitoring models using the multi-class logarithmic loss metric[10] (LogLoss).

5.3.2 Testing against Random Undersampling

In my first experiment, I aimed to examine whether my proposed method can be used to undersample a class-imbalanced training dataset at all. If my method can be used for undersampling, then a model trained using a training dataset balanced by my method should achieve a significantly higher classification performance than a model trained using the original class-imbalanced training dataset. Furthermore, I also aimed to test my method in comparison to random undersampling [104]. Therefore, I additionally investigated whether a model trained using a training dataset balanced by my method is able to achieve a superior classification performance than a model trained using a training dataset balanced by random undersampling. I conducted my experiment on the Plant Pathology dataset and the Fisheries Monitoring dataset. My setup for the experiment is described in Chapter 5.3.1.

---

[7] `https://pytorch.org/docs/stable/generated/torch.optim.Adam.html`

[8] `https://pytorch.org/docs/stable/generated/torch.optim.lr_scheduler.CyclicLR.html`

[9] `https://www.kaggle.com/competitions/plant-pathology-2020-fgvc7/overview/evaluation`

[10] `https://www.kaggle.com/competitions/the-nature-conservancy-fisheries-monitoring/overview/evaluation`

I first tested my proposed method on the Plant Pathology dataset. The Plant Pathology dataset contains images of 4 different classes: The *rust disease* class (622 training data samples), the *scab disease* class (592 training data samples), the *healthy* class (516 training data samples), and the *multiple diseases* class (91 training data samples). The dataset is class-imbalanced because it contains a significantly lower amount of images of the *multiple diseases* class compared to the other three classes. Therefore, I considered the *multiple diseases* class to be the minority class and the other three classes to be the majority classes. As a result, I aimed to undersample these three majority classes using 4 different undersampling strategies: (a) undersample only the *rust disease* class (from 622 down to 100 training data samples), (b) undersample only the *scab disease* class (from 592 down to 100 training data samples), (c) undersample only the *healthy* class (from 516 down to 100 training data samples), and (d) undersample the *rust disease* class, the *scab disease* class and the *healthy* class at the same time (down to 100 training data samples for each of the three classes). To undersample the training dataset using one of the strategies mentioned above, I needed to find clusters within the activations of the respective training images at a certain higher model layer. I considered fetching these activations from one of the newly added model layers (replacement for the ImageNet-related output layer) or from the last layer before the newly added model layers (i.e., the last ResNet50 layer) (for more details about the layers, see Chapter 5.3.1). To pick one of these layers, I used the best silhouette score [121] that I obtained from each layer when searching for the clusters within the activations of the layer (for more details, see Chapter 5.2.1). The silhouette score is a cluster quality metric that indicates how well the clusters are separated. I picked the layer with the highest silhouette score, i.e., the layer whose clusters are best separated. For the *rust disease* class and the *healthy* class, the layer with the highest silhouette score was the special pooling layer from the newly added layers. For the *scab disease* class, on the other hand, the layer with the highest silhouette score was the second last fully-connected linear layer from the newly added layers (i.e., the linear layer before the output layer). After obtaining the clusters from the chosen layer, I selected an equal amount of majority class training images from each of these clusters. For instance, if I aim to select 100 training images of the class that should be undersampled and I found 2 clusters within the activations of these images, I select 50 images from each of these clusters to obtain the 100 images in total for that class. In order to select the images, I tested my two selection approaches (picking the closest images from each cluster center and picking the farthest images from each cluster center). As a result, I obtained the following datasets: The original class-imbalanced dataset (baseline), a dataset balanced by my method for each of the 4 undersampling strategies (a) to (d), and my two selection approaches (closest images and farthest images from each cluster center), and 6 different datasets balanced by random undersampling for each of the undersampling strategies (a) to (d) (using a different random seed value for each of the 6 times to apply random undersampling). After creating the datasets, I trained a model 6 times using a different seed value each time for each of these datasets, as described in Chapter 5.3.1. Then, I tested the resulting models on the private test dataset from Kaggle. As a result, I obtained the classification performances in the form of their 6 ROC AUC metrics for each model (one for each seed). To be able to better compare the different models, I computed the median and standard deviation over the 6 ROC AUC metrics. The results of my experiment with respect to the Plant Pathology dataset are shown in Table 5.1. As shown in the table, the model trained using the dataset balanced by my method with strategy (b) (undersample only the *scab disease* class) and picking the closest images from each cluster center achieved the best classification performance. The model surpassed the performance of the baseline model and the models trained using a dataset balanced by random undersampling.

Table 5.1: Comparison of my method with the baseline (class-imbalanced dataset) and random undersampling with respect to the Plant Pathology 2020 dataset. All models were trained using 6 different random seed values. The results are given as median ROC AUC scores (a higher score is better).

Method	Undersampled Class	ROC AUC	
		median	*std*
Baseline	-	0.9387	0.0045
Random Undersampling	healthy	0.9375	0.0086
	rust	0.9187	0.0092
	scab	0.9377	0.0024
	healthy, rust, scab	0.9068	0.0233
My Method (closest)	healthy	0.9391	0.0058
	rust	0.9171	0.0069
	scab	**0.9425**	0.0088
	healthy, rust, scab	0.8909	0.0166
My Method (farthest)	healthy	0.9353	0.0059
	rust	0.9183	0.0098
	scab	0.9369	0.0080
	healthy, rust, scab	0.9004	0.0054

After the Plant Pathology dataset, I also tested my proposed method on the Fisheries Monitoring dataset. The Fisheries Monitoring dataset contains images of 8 different classes: The *ALB* class (1719 training data samples), the *YFT* class (734 training data samples), the *BET* class (200 training data samples), the *SHARK* class (176 training data samples), the *DOL* class (117 training data samples), the *LAG* class(67 training data samples), the *OTHER* class (299 training data samples) and the *NoF* class (465 training data samples). The dataset is class-imbalanced because it contains a significantly higher amount of images of the *ALB* class compared to the other seven classes. Therefore, I considered the *ALB* class to be the majority class and the other seven classes to be the minority classes. As a result, I aimed to undersample the *ALB* class as the majority class (from 1719 down to 734 data samples). To undersample the training images of the *ALB* class using my method, I needed to find clusters within the activations of these training images at a certain higher model layer (i.e., a layer closer to the output layer). I considered fetching these activations from one of the newly added model layers (replacement for the ImageNet-related output layer) or from the last layer before the newly added model layers (i.e., the last ResNet50 layer) (for more details about the layers, see Chapter 5.3.1). Again, to pick one of these layers, I used the best silhouette score [121] that I obtained from each layer when searching for the clusters within the activations of the layer (for more details, see Chapter 5.2.1). For the *ALB* class that I aimed to undersample, the layer with the highest silhouette score was the last layer before the newly added layers. After obtaining the clusters from the chosen layer, I selected an equal amount of training images from each cluster. In order to select the images, I tested my two selection approaches (picking the closest images from each cluster center and picking the farthest images from each cluster center). As a result, I received the following datasets: The original class-imbalanced dataset (baseline), a dataset balanced by my method with respect to my two selection approaches (closest images and farthest images from each cluster center), and 5 different datasets balanced by random undersampling (using a different seed value for each of the 5 times to apply random undersampling). After receiving the datasets, I trained a model for each of these datasets, as described in Chapter 5.3.1. As the Fisheries Monitoring dataset is larger than the Plant Pathology dataset, I only trained one model for each dataset. Then, I tested the resulting models on the private test dataset from Kaggle. As a result, I obtained the classification performances in the form of their LogLoss metric for each model. As I obtained 5 models trained using a dataset balanced by random undersampling (one for each of the 5 random seeds), I computed the median over their LogLoss metrics. The results of my experiment with respect to the Fisheries Monitoring dataset are shown in Table 5.2. As shown in the table, the model trained using the dataset balanced by my method with picking the closest images from each cluster center achieved the best classification performance. The model surpassed the performance of the baseline model and the models trained using random undersampling.

Table 5.2: Comparison of my method with the baseline (class-imbalanced dataset) and random undersampling (median over 5 random seed values) with respect to the Nature Conservancy Fisheries Monitoring dataset. The results are given as multi-class logarithmic loss scores (a lower score is better).

Method	LogLoss
Baseline	5.7333
Random Undersampling	4.0771
My Method (closest)	**3.8657**
My Method (farthest)	3.9302

Table 5.3: Comparison of my method with the baseline (class-imbalanced dataset) and random undersampling (median over 5 random seed values) with respect to the Plant Pathology 2020 dataset (undersampled class: *healthy*) using different model architectures. The results are given as ROC AUC scores (a higher score is better).

Method	CNN Architecture	ROC AUC
Baseline	VGG16	0.9435
Random Undersampling		0.9404
My Method (closest)		**0.9447**
My Method (farthest)		0.9322
Baseline	DenseNet121	0.9461
Random Undersampling		0.9384
My Method (closest)		0.9294
My Method (farthest)		**0.9465**

5.3.3 Comparing Different Model Architectures

In my experiment in Chapter 5.3.2, I examined whether my proposed method is able to address the class imbalance problem at all. Furthermore, I also tested my method in comparison to random undersampling [104]. I could show that a model trained using a training dataset balanced by my method achieves a better classification performance than a model trained using the original class-imbalanced training dataset and a model trained using a training dataset balanced by random undersampling. However, for all of my tests, I trained the respective models based on a pre-trained model with a ResNet50 architecture (50-layer Residual Network) [46], which was trained on the ImageNet [25] dataset. To make sure that my method does not only work for models with a ResNet50 architecture, I repeated my experiment from Chapter 5.3.2 with respect to undersampling of the *healthy* class of the Plant Pathology dataset with two alternative model architectures, a DenseNet121 [56] and a VGG16 [129] (with batch normalization layers [61]). However, except for the model architecture, I used the same experimental setup as in Chapter 5.3.2. For each model architecture, I received the following datasets: The original class-imbalanced dataset (baseline), a dataset balanced by my method with respect to my two selection approaches (closest images and farthest images from each cluster center), and 5 different datasets balanced by random undersampling (using a different seed value for each of the 5 times to apply random undersampling). After receiving the datasets, I trained a model for each of these datasets, as described in Chapter 5.3.1. Then, I tested the resulting models on the private test dataset from Kaggle. As a result, I obtained the classification performances in the form of their ROC AUC metric for each model. As I obtained 5 models trained using a dataset balanced by random undersampling (one for each of the 5 random seeds), I computed the median over their ROC AUC metrics. The results of my experiment are shown in Table 5.3. As shown in the table, the model trained using a dataset balanced by my method achieved the best classification performance for both model architectures. For each architecture, the model surpassed the performance of the baseline model and the models trained using random undersampling. However, for the DesNet121 architecture, my method was better when picking the farthest images from each cluster center instead of picking the closest images from each cluster center.

5.3.4 Testing against Different State-of-the-Art Methods

In my experiments in Chapter 5.3.2 and Chapter 5.3.3, I tested my proposed method only in comparison to random undersampling [104]. However, as pointed out in Chapter 5.1, there are also other state-of-the-art methods to address the class imbalance problem. Therefore, I conducted a third experiment with respect to the Plant Pathology dataset to test my method in comparison to two other state-of-the-art methods, random oversampling [104] of the minority class (the *multiple diseases* class, 91 training data samples) and using a class-weighted loss function [104] for model training. For random oversampling, I considered two approaches: Oversampling of the minority class to 182 training data samples (original size × 2) and oversampling of the minority class to 546 training data samples (original size × 6). For the class-weighted loss function, on the other hand, I used a loss function with special weights that were set by the fraction of the training data samples of each class in relation to the total size of the training dataset. Furthermore, I also tested additional undersampling approaches. For this experiment, I only considered undersampling the *healthy* class (516 training data samples). However, besides undersampling the *healthy* class to 100 training data samples using either my method or random undersampling, I also tested to undersample the *healthy* class to 250 training data samples. Moreover, for my method, I also tested a third selection approach to picking training images from each identified cluster. Besides picking the closest or the farthest training images from each cluster center, I also considered picking training images from each cluster at random. Except for these additional approaches, however, I used the same experimental setup as in Chapter 5.3.2. I received the following datasets: The original class-imbalanced training dataset (baseline), a training dataset balanced by my method with respect to my three selection approaches (closest training images and farthest training images from each cluster center as well as selecting training images at random from each cluster), and 5 different training datasets balanced by random undersampling (using a different seed value for each of the 5 times to apply random undersampling). After receiving the datasets, I trained a model for each of these datasets, as described in Chapter 5.3.1. Then, I tested the resulting models on the private test dataset from Kaggle. As a result, I obtained the classification performance in the form of their ROC AUC metric for each model. As I obtained 5 models trained using a training dataset balanced by random undersampling (one for each of the 5 random seeds), I computed the median over these 5 ROC AUC metrics. The results of my experiment are shown in Table 5.4. As shown in the table, the model trained using the dataset balanced by my method using the random selection approach of 250 training data samples achieved the best classification performance. The model surpassed the classification performance of all of the state-of-the-art methods.

Table 5.4: Comparison of my method with the baseline (class-imbalanced dataset), random undersampling, random oversampling and a class-weighted loss approach with respect to the Plant Pathology 2020 dataset. For my method and random undersampling, I undersampled the *healthy* class. The results are given as ROC AUC scores (a higher score is better).

Method	ROC AUC
Baseline	0.9343
Class-Weighted Loss	0.9466
Oversampling (182 samples)	0.9469
Oversampling (546 samples)	0.9268
Random Undersampling (100 samples)	0.9370
My Method (closest, 100 samples)	0.9451
My Method (farthest, 100 samples)	0.9301
My Method (random, 100 samples)	0.9377
Random Undersampling (250 samples)	0.9474
My Method (closest, 250 samples)	0.9488
My Method (farthest, 250 samples)	0.9453
My Method (random, 250 samples)	**0.9551**

5.4 Discussion

A state-of-the-art method to address the class imbalance problem is random undersampling [104]. Random undersampling balances the class-imbalanced training dataset before model training. To balance a dataset, random undersampling removes images of the majority class from the dataset at random until the amount of majority class images and the amount of minority class images are approximately equal. If the majority class of the training dataset contains different subclasses, however, random undersampling may remove a significant amount of majority class images from one of those subclasses. As a result, a model trained using this undersampled dataset may not have been able to learn an adequate representation of the majority class. Therefore, I suggested an improved undersampling method (Chapter 5.2). Rather than randomly removing majority class images to balance the dataset, my method identifies clusters within the majority class images in image feature space of a higher model layer (i.e., a layer closer to the output layer) and selects a specific amount of majority class images from each identified cluster for the balanced dataset. I assume that the identified clusters correspond to the subclasses of the majority class. Thus, a model trained using the dataset balanced by my method should learn an adequate representation of the majority class. As a result, if the majority class of a specific dataset contains subclasses, I expected that a model trained using the dataset balanced by my method would achieve a significantly higher classification performance than a model trained using the dataset balanced by random undersampling.

To evaluate my proposed undersampling method, I conducted several experiments (Chapter 5.3). My first research goal was to examine whether a subclass-based approach is beneficial for undersampling at all. In Chapter 5.3.2, I showed for two class-imbalanced real-world datasets that a model trained using a dataset balanced by my method achieves a higher classification performance than a model trained using the corresponding original class-imbalanced dataset. This indicates that a subclass-based approach is beneficial for undersampling a dataset. However, I not only aimed to show that a subclass-based approach can be used for balancing a dataset but also that my method achieves a higher classification performance than random undersampling. Therefore, my second research goal was to examine how my subclass-based undersampling method performs in comparison to random undersampling. In Chapter 5.3.2, I also showed for the two real-world datasets that a model trained using the dataset balanced by my method achieves a higher classification performance than a model trained using the dataset balanced by random undersampling. Moreover, I could show in Chapter 5.3.3 that my method outperforms random undersampling even for different model architectures. In my experiments, selecting the majority class images from each cluster that are closest to their respective cluster center turned out to work best in most cases. In one case, however, selecting the majority class images farthest from the cluster centers resulted in a better classification performance.

However, as shown in Chapter 5.1, random undersampling is not the only state-of-the-art method to address the problem of training a Convolutional Neural Network-based image classification model using a class-imbalanced training dataset. Other state-of-the-art methods to address this problem include oversampling and approaching the problem through the loss function during model training. Therefore, my third research goal was to examine how my subclass-based undersampling method performs in comparison to random oversampling [104] and an approach based on a class-weighted loss function [104]. In Chapter 5.3.4, I showed for the two class-imbalanced real-world datasets that my subclass-based undersampling method also outperforms these two alternative state-of-the-art methods. Selecting a slightly higher amount of majority class images from each identified cluster at random turned out to work best in this case. However, in contrast to oversampling and a loss function-based approach, an undersampling method, such as my method or random undersampling, removes information from the training dataset by excluding majority class images. As a result, I assume that the classification performance of undersampling methods might be dataset dependent. Nevertheless, I have shown that my subclass-based undersampling method is an additional approach to addressing the problem of training an image classification model using a class-imbalanced training dataset.

6 Exploiting Layer Activations to Detect Out-of-Distribution Samples

I have shown in Chapter 4 and Chapter 5 that the layer information (pixels and activations) of a Convolutional Neural Network-based (CNN) image classification model can be exploited to improve the training of such a model. In this chapter, I show that the layer activations of the model can be exploited to improve the usage of the model after model training as well. By using the activations of multiple model layers, it is possible to detect when the model fails to predict the correct class for an image at inference. Usually, we train such an image classification model on a training dataset until the model reaches a sufficient classification performance on a given test dataset. To be able to achieve a sufficient classification performance, the test dataset must have been drawn from the same data distribution as the training dataset. However, the test dataset does not share any images with the training dataset. After model training, we typically deploy the resulting model to a production system (e.g., a mobile app, a web server). This production system receives images at inference that should be classified by the model. If these incoming images have been drawn from the same data distribution as the training and test dataset, then our model usually achieves a similar classification performance on these incoming images as on the test dataset. We typically refer to this kind of incoming images as in-distribution samples. If the incoming images have been drawn from a data distribution that differs from the data distribution of the training and test dataset, however, then our model most likely achieves a classification performance on those incoming images that is significantly lower than the classification performance on the test dataset. We typically refer to this kind of incoming images as out-of-distribution samples [145]. The model achieves a lower classification performance on these out-of-distribution samples because it did not see this kind of images during model training. As a result, the model was not able to learn any image features of these images in order to classify them correctly. However, not only does our model fail to classify out-of-distribution samples correctly, it also fails without warning. When feeding an out-of-distribution sample into the model, we would like our model to signal that it is not sure how to classify this image. The model could signal that it is not sure by assigning each possible class an equal confidence value obtained by the softmax scores of the model, when predicting a class for the out-of-distribution sample. Unfortunately, this is not what typically happens in practice. In practice, such a model might predict a class for an out-of-distribution sample even with a high softmax score, as pointed out by Gal [35] as well as by Hendrycks and Gimpel [49]. This behavior of Convolutional Neural Network-based models makes it difficult to use them for safety-critical applications such as driving assistance systems [47] or medical diagnosis systems [103]. The problem could also arise in my plant disease example from Chapter 1. For instance, suppose after training the model to detect the plant disease, the disease suddenly mutates and changes its visual appearance. However, our model has not yet learned anything about this new appearance. As a result, the model is no longer able to recognize the disease because images of the mutated disease are out-of-distribution samples with respect to our model.

In general, two different types of out-of-distribution samples are considered, natural out-of-distribution samples and adversarial out-of-distribution samples. A natural out-of-distribution sample might belong to a class that is different from the classes that our model learned about during model training. For instance, suppose our model was trained to classify images of different kinds of apples. If the model then receives an image at inference that shows an orange, however, it is not able to correctly classify that image as an orange. Instead, the model incorrectly classifies that image as a certain kind of apple because the model only learned about apples during model training, not oranges. The model knows nothing about the concept of an orange. Nevertheless, the model might even misclassify the image with high confidence (i.e., a high softmax score). Therefore, the image showing an orange is a natural out-of-distribution sample with respect to our model that was trained to classify apples. However, not every natural out-of-distribution sample belongs to an unknown class. A natural out-of-distribution sample might also belong to one of the classes that the model learned about during model training. However, this out-of-distribution sample shows the class object in a form that is different from the form of the class object appearing on the training images, which the model saw during model training. For instance, suppose our model, which was trained to classify different kinds of apples, only saw images showing whole apples during model training. If the model then receives an image at inference that shows a certain type of apple in the form of a sliced apple, however, it is not able to correctly classify that image as the correct type of apple, even though it might have learned about this type of apple (but in the form of a whole apple) during model training. Images showing apple slices have different image features than images of whole apples. However, during model training, our model learned to recognize only the image features of whole apples, not the image features of apple slices. Again, it is possible that the model misclassifies the out-of-distribution sample even with high confidence (i.e., a high softmax score). Therefore, the image showing the sliced apple is a natural out-of-distribution sample with respect to our model, which only learned about whole apples during model training. These two kinds of out-of-distribution samples (unknown class, unknown form) are typically referred to as natural out-of-distribution samples [51, 108] because these images occur naturally. However, it is also possible to create out-of-distribution samples artificially. This kind of out-of-distribution samples are usually referred to as adversarial out-of-distribution samples. To create an adversarial out-of-distribution sample, an attacker usually alters an in-distribution image in a subtle way using the model by changing only a few specific pixels of that in-distribution image [41, 59, 135]. This alteration is usually imperceptible to the human eye but negatively impacts the model to predict the incorrect class for that image with high confidence (i.e., a high softmax score). Alternatively, it is also possible to obtain such an adversarial out-of-distribution sample by altering the physical image object before the image of that object was captured [12, 34, 73]. Adversarial out-of-distribution samples are typically used by attackers to attack our model [11]. The attacker can even create an adversarial out-of-distribution sample using his own model to attack our model because adversarial out-of-distribution samples are transferable between different models [58, 142]. It is debatable whether the model behavior caused by adversarial out-of-distribution samples is a bug or not. Ilyas et. al. [59] argue that the behavior is expected because the added noise from the attacker results in image features for the model as well. These noise-based image features just do not correspond to what humans recognize as image features. Nevertheless, out-of-distribution samples pose a serious threat, especially when using the model for safety-critical applications (e.g., driving assistance systems, medical diagnosis systems).

To be able to reliably use Convolutional Neural Network-based models in practice, it is important to detect whether an incoming image at inference is an out-of-distribution sample. Therefore, a large number of approaches to detecting out-of-distribution samples have been suggested [145]. An overview of different detection methods is given in Chapter 6.1. A promising method named DkNN (Deep k-Nearest Neighbors) was suggested by Papernot and McDaniel [113, 114]. DkNN calculates a credibility score for the incoming image at inference, which is a float value that ranges between 0 and 1. This credibility score indicates how closely this image resembles the training images of the model. A credibility score close to 1 indicates that the image highly resembles the training images. Therefore, DkNN concludes that the image is an in-distribution sample. A credibility score close to 0, on the other hand, indicates that the image does not resemble or only slightly resembles the training images. Therefore, DkNN concludes that the image is an out-of-distribution sample. To calculate the credibility score of an image, the DkNN method runs a k-nearest neighbor classification at each model layer in order to predict the class of the image in activation space (i.e., image feature space) of the layer. DkNN is based on the assumption that each in-distribution image is always close to other in-distribution images of the same class in activation space (i.e., image feature space) of each layer of the model. Therefore, DkNN checks the classes of the k-nearest neighbors of the image among the training images of the model in activation space (i.e., image feature space) of each model layer. The class that appears most frequently among the k-nearest neighbors at a certain layer, is assumed to be the class of the image with respect to that layer. If the predicted class is the same class at every layer, the assumption of DkNN is satisfied. The image is always close to other in-distribution samples (the training images) of the same class in activation space (i.e., image feature space) of each model layer. As a result, DkNN computes a credibility score close to 1, which indicates that the image is also an in-distribution sample. If the predicted class differs significantly across the different model layers, however, the assumption of DkNN is violated. The image is not always close to other in-distribution samples (the training images) of the same class in activation space (i.e., image feature space) of each model layer. As a result, DkNN computes a credibility score close to 0, which indicates that the image is an out-of-distribution sample. Papernot and McDaniel [113, 114] showed on different simple datasets that they were able to detect out-of-distribution samples using their DkNN method.

However, DkNN has two disadvantages. First, DkNN is slow at inference due to the use of the k-nearest neighbor classification. A naive k-nearest neighbor classification requires measuring the distance between the incoming image and every training image at inference. However, a training dataset typically contains a huge number of training images. Therefore, the k-nearest neighbor classification requires a huge number of distance measurements at inference resulting in a slow runtime at inference. In order to reduce the runtime, it is possible to use an approximate k-nearest neighbor classification. DkNN uses such an approximate k-nearest neighbor classification based on locality-sensitive hashing [6]. An approximate k-nearest neighbor classification requires measuring the distance of the image not to every training image at inference but only to a subset of the images of the training dataset. However, this subset of training images usually still contains a relatively high number of images. As a result, an approximate k-nearest neighbor classification has a lower runtime at inference compared to a naive k-nearest neighbor classification, but the approximate k-nearest neighbor classification is still relatively slow. Moreover, DkNN does not need to run the k-nearest neighbor classification only once but once for each model layer in order to calculate the credibility of an image. This causes DkNN to be slow at inference. However, we usually want our model to quickly detect at inference if an incoming image is an out-of-distribution sample. A driving assistance system, for instance, usually only has a short period of time to react to dangerous situations. In addition to slow runtime at inference, the high memory consumption of the k-nearest neighbor classification is a second disadvantage of DkNN. To be able to measure the distance between the incoming image and the training images, DkNN needs to store the entire training dataset for inference. However, if our training dataset is huge, we might not be able to store the entire training dataset on a production system with only small storage capabilities (e.g., a mobile phone, an embedded system).

To address the disadvantages of DkNN, I proposed a novel method named LACA (Layer-wise Activation Cluster Analysis) to detect out-of-distribution samples (published in Lehmann and Ebner [81]). LACA is based on the same assumption as the DkNN method that each in-distribution image is close to other in-distribution images of the same class in activation space (i.e., image feature space) of each model layer. However, LACA is based on clustering rather than a k-nearest neighbor classification. LACA does not compare an incoming image at inference to the training images but applies a clustering model to the image and compares the clustering result to a set of precomputed in-distribution statistics. The details of LACA are described in Chapter 6.2. I expected that a clustering-based method such as LACA would have a significantly lower runtime and memory consumption at inference than a method based on a k-nearest neighbor classification such as DkNN. However, my first research goal was to find out whether LACA is able to detect out-of-distribution samples at all. In my initial work (published in Lehmann and Ebner [81]), I showed on different simple datasets that LACA is indeed capable of detecting out-of-distribution samples. However, the focus of my initial work was only to examine whether a method based on clustering works at all. The goal was not to already present a ready-to-use detection method that is better than DkNN. Therefore, a sufficient detection performance was not yet achieved. Furthermore, I only calculated a binary detection score (Chapter 6.2.3), which was not comparable to DkNN because DkNN calculates a credibility score that is a float value between 0 and 1. To improve LACA, I proposed in a follow-up work to calculate a credibility score (Chapter 6.2.4) based on the information provided by LACA (published in Lehmann and Ebner[82]). This credibility score is a float value that ranges between 0 and 1 as well. This finally allowed me to compare LACA with DkNN. My second research goal was therefore to examine whether LACA has a significantly lower runtime and memory consumption than DkNN, while achieving a similar detection performance as DkNN. In this follow-up work, I showed on different simple datasets that LACA is indeed faster at inference than DkNN, while achieving a similar detection performance as DkNN (Chapter 6.3.3). Moreover, LACA only needs to store a clustering model (Chapter 6.2.1) and a set of pre-computed in-distribution statistics of the training dataset (Chapter 6.2.2) from each model layer for inference. LACA does not need to store the entire training dataset for inference as DkNN. Thus, it also has a lower memory consumption at inference than DkNN. Additionally, LACA has the same advantages as DkNN. LACA does not require retraining the model, nor does it require obtaining any out-of-distribution samples in advance. Obtaining out-of-distribution samples in advance would be difficult as it is not known which kind of out-of-distribution samples will the model see at inference. However, similar to Papernot and McDaniel [113, 114], initially, I only tested LACA in comparison to DkNN on simple datasets. Thus, I later conducted additional experiments in order to also compare LACA to DkNN on more complex datasets (published in Lehmann and Ebner [84]). Again, I showed that LACA is significantly faster at inference than DkNN. Moreover, LACA was still able to detect out-of-distribution samples on these more complex datasets, while DkNN failed to detect any out-of-distribution samples (Chapter 6.3.5). The following contributions have been made: (1) It was shown that a clustering-based method is able to detect out-of-distribution samples, (2) a clustering-based method was proposed to detect out-of-distribution samples that has a significantly lower runtime and memory consumption than the DkNN method, while achieving at least a similar detection performance, and (3) it was shown that the proposed method outperforms the DkNN method on more complex datasets in terms of both runtime at inference and detection performance.

6.1 Related Work on Out-of-Distribution Detection

A wide variety of methods using different kinds of approaches have been suggested for detecting out-of-distribution samples with respect to a Convolutional Neural Network-based (CNN) image classification model. Grosse et. al. [43], for instance, proposed to detect out-of-distribution samples with an additional model output that signals if an image is out-of-distribution or not. Gal et. al. [36], on the other hand, suggested using dropout [134] as an approximate Bayesian inference that provides the model prediction for an image along with the uncertainty of that prediction. A low uncertainty indicates that the image is an out-of-distribution sample. Meng and Chen [99] introduced a detection method that is based on modeling the in-distribution samples using a generative model. In order to check if an image is an out-of-distribution sample, they measure the distance of that image to the learned generative representation of the in-distribution samples. Lee et. al. [78] suggested a novel loss function for model training in order to calculate improved confidence estimates for the model predictions. A low confidence of a model prediction for an image indicates that the image is an out-of-distribution sample. Hendrycks et. al. [50] proposed a detection method based on self-supervised learning. However, none of these methods uses the activations of the model layers directly to detect out-of-distribution samples, as my method or the DkNN method (Deep k-Nearest Neighbors) from Papernot and McDaniel [113, 114] does.

Nevertheless, the activations of the model layers also have proven to be beneficial for detecting out-of-distribution samples in a vast amount of other studies. Chen et. al. [20], for instance, introduced a method that calculates the confidence of a model prediction for an image using a meta-model. This meta-model is applied to the layer activations of the image. A low confidence indicates that the image is an out-of-distribution sample. Lee et. al. [79], on the other hand, suggested a method that computes the confidence of a model prediction for an image using a class-conditional Gaussian distribution. This class-conditional Gaussian distribution is applied to the layer activations of the image as well. Carrara et. al. [16] proposed a method that detects out-of-distribution samples by running a k-nearest neighbor scoring on the layer activations. Cohen et. al. [22], on the other hand, use sample influence scores along with a k-nearest neighbor classification that they apply to the activations of the model layers in order to detect out-of-distribution samples. Li and Li [86] suggested a cascade-based out-of-distribution detector. This cascade-based out-of-distribution detector uses specific statistics obtained from the activations of the convolutional layers of the model. Caldelli et. al. [14] proposed a method that computes for each class of the classification problem the mean image in activation space of each model layer. To check if an image is out-of-distribution or not, they compare the image with the computed mean images in activation space of the model layers. Ma et. al. [93] calculate local intrinsic dimensionality estimates from the activations of the model for detecting out-of-distribution samples. Metzen et. al. [100] attach a subnetwork to a particular model layer. They use this subnetwork as a detector for out-of-distribution samples. Crecchi et. al. [23] attach a detector to multiple layers of the model. When feeding an image into the model, these detectors calculate a probability score for each class of the classification problem that indicates whether the image is an in-distribution sample with respect to that class. Finally, the probability scores from each detector are fed into a final multi-layer detector, which decides if the image is an out-of-distribution sample or not. Sastry and Oore [125] introduced a method that checks if the intermediate layer activations of an image contain anomalies with respect to the predicted class for that image. If anomalies can be detected, they conclude that the image is an out-of-distribution sample. Lin et. al. [89] suggested a multi-level detection method. They added an out-of-distribution detector after multiple layers of the model. When they feed an out-of-distribution sample into the model, one of those detectors will recognize that the sample is out-of-distribution. However, all of these methods use layer activations of the model in order to detect out-of-distribution samples, but none of them obtains cluster information from these layer activations in order to detect the out-of-distribution samples, as my method does.

Huang et. al. [57] and Sinhamahapatra et. al. [131] also used cluster information for their suggested detection methods. Huang et. al. [57] claimed that out-of-distribution samples cluster together in activation space. Therefore, their method checks if an image is an out-of-distribution sample by measuring the distance of the image to the center of this cluster. If the distance is below a specified threshold, they concluded that the image is an out-of-distribution sample. Sinhamahapatra et. al. [131], on the other hand, adjusted the training of a model to obtain an activation representation at the last model layer that separates in-distribution samples from out-of-distribution samples by pushing in-distribution samples together into one cluster and out-of-distribution samples together into another cluster. Therefore, if an image is far from the in-distribution cluster, they considered that image to be out-of-distribution. However, both methods use an approach based on measuring the distance between in-distribution and out-of-distribution samples. My proposed method, on the other hand, detects whether an image is out-of-distribution by examining which training images (i.e., in-distribution samples) are in the same cluster as the image in activation space of each model layer.

However, clusters obtained from the layer activations are not only used for detecting out-of-distribution samples. Nguyen et. al. [109] identify clusters within the activations of the last model layer in order to visualize multi-faceted image features that the model learned during model training. Chen et. al. [19], on the other hand, use cluster information within the activations of the last model layer in order to check if an attacker poisoned the training dataset of the model to be able to trigger a specific behavior of the model after model training (backdoor attack). Furthermore, I suggest a method in Chapter 5 for balancing a class-imbalanced training dataset that is based on finding clusters in the activations of a higher layer (i.e., a layer closer to the output layer).

6.2 Layer-wise Activation Cluster Analysis

A Convolutional Neural Network-based (CNN) model is trained using a training dataset (X^D, Y^D) in order to predict the correct class for incoming images at inference among a fixed set of classes C. The training dataset consists of the training images X^D and their respective labels Y^D. I assume that all training images are of the same image size. This is important for my method LACA. Therefore, if the training images are not all of the same size, they need to be resized before model training. The value of each label of Y^D, on the other hand, corresponds to a specific class in C. After model training, the model is tested on a given test dataset (X^T, Y^T). The test dataset consists of the test images X^T and their respective labels Y^T. Again, the value of each label of Y^T corresponds to a specific class in C. The images from both datasets, the training and the test dataset, have been drawn from the same data distribution. However, both datasets do not share any images in order to avoid obtaining an overly optimistic classification performance of the model on the test dataset. Nevertheless, suppose that the model achieves at least a sufficient classification performance on the test dataset. As a result, the model is embedded into a production system (e.g., a mobile app, a web server). In this production system, the model should predict the correct class c_{x^I} of each incoming image x^I at inference among the set of classes C. However, the classification performance of the model on those images x^I depends on the data distribution from which the images x^I have been drawn. If the images x^I have been drawn from the same data distribution as the training and test images, the model should achieve a similar classification performance on the images x^I as on the test images x^T. We refer to this type of images at inference as in-distribution samples. If the images x^I have been drawn from a different data distribution than the training and test images, however, the model might achieve a significantly lower classification performance on the images x^I than the classification performance of the model on the test images x^T. For most images x^I, the model usually fails to predict the correct class c_{x^I}. We refer to this type of images at inference as out-of-distribution samples. The failing of the model to predict the correct class for an image at inference can lead to severe human-impacting consequences. This is especially the case when the model is used in the production system of a safety-critical application, such as a medical application for diagnosing a specific disease or a driving assistance system of a vehicle. Therefore, it is important to detect whether an image x^I is an in-distribution sample or an out-of-distribution sample with respect to the model. This detection helps to prevent any severe consequences. If we are able to detect that an image x^I is an out-of-distribution sample with respect to the model, we can issue an alert and request human intervention to manually check if the predicted class for that image is correct.

My proposed method LACA detects whether an image x^I at inference is an out-of-distribution sample with respect to a model. To detect whether x^I is out-of-distribution, it examines how closely x^I resembles the training images X^D of the model. If x^I highly resembles the training images, I assume that x^I has probably been drawn from the same data distribution as the training images. Thus, I conclude that x^I must be an in-distribution sample. If image x^I does not resemble or only slightly resembles the training images, however, I assume that x^I has probably been drawn from a different data distribution than the training images. Thus, I conclude that x^I must be an out-of-distribution sample. To express how closely an image x^I resembles the training images, I proposed two different metrics, a binary similarity score and a credibility score. The binary similarity score $\mathit{simDet}(x^I) \in \{0, 1\}$ is a boolean value reflecting whether image x^I resembles the training images. A similarity score of 1 indicates that x^I does resemble the training images and therefore, I conclude that x^I is an in-distribution sample. A similarity score of 0, on the other hand, indicates that x^I does not resemble the training images and therefore, I conclude that x^I is an out-of-distribution sample. The similarity score simDet is easy to calculate. However, simDet may not always achieve a sufficient detection performance. Thus, I proposed a second metric, the credibility score. The credibility score $\mathit{credib}(x^I) \in [0, 1]$ is a value between 0 and 1 reflecting how closely x^I resembles the training images. A credibility score close to 1 indicates that x^I closely resembles the training images and therefore, I conclude that x^I is most likely an in-distribution sample. A credibility score close to 0, on the other hand, indicates that x^I does not resemble the training images and therefore, I conclude that x^I is most likely an out-of-distribution sample. The credibility score is more complex to calculate than the binary similarity score. However, the credibility score generally achieves a better detection performance than the binary similarity score.

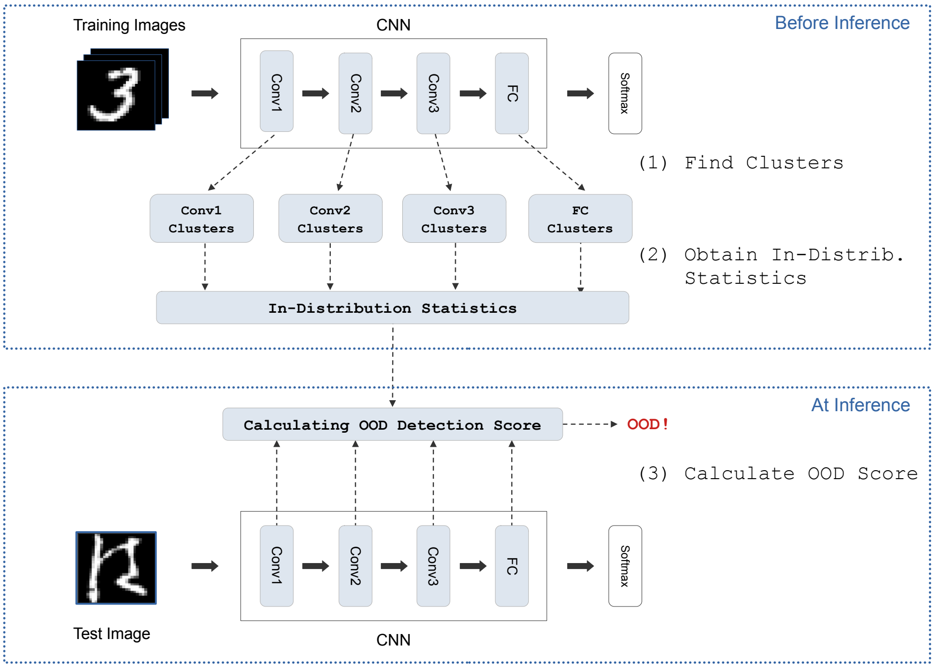


Figure 6.1: The proposed method: (1) Find clusters within the layer activations of the training images (e.g., MNIST [76]) before inference, (2) obtain several in-distribution statistics from the obtained clusters, and (3) calculate a detection score (e.g., credibility) for an image at inference reflecting whether the image is an out-of-distribution sample (OOD) (e.g., KMNIST [21]).

The calculation of these two metrics, *simDet* and *credib*, for an image x^I at inference is based on several in-distribution statistics. These in-distribution statistics are obtained from clusters found in the activations of the layers of the Convolutional Neural Network-based image classification model. The required in-distribution statistics include the class-distribution statistics obtained from each layer of the model, the layer weights computed from those class-distribution statistics, and the cluster-distribution statistics that are obtained from each model layer as well. These in-distribution statistics only need to be obtained once. They are typically computed after model training but before the model is embedded into the production system (i.e., before inference). An overview of my proposed method is shown in Figure 6.1. Hereinafter, I first describe in more detail how to identify clusters in the activations of the model (Chapter 6.2.1). To explain how to identify these clusters, I use some of the classes[1] of the ImageNet [25] dataset used for the Large Scale Visual Recognition Challenge [123] as an example. Then, I describe how to use the clusters to obtain the in-distribution statistics (Chapter 6.2.2), and how to calculate the binary similarity score (Chapter 6.2.3) as well as the credibility score (Chapter 6.2.4) based on these obtained in-distribution statistics.

---

[1] `https://www.image-net.org/challenges/LSVRC/2017/browse-synsets.php`

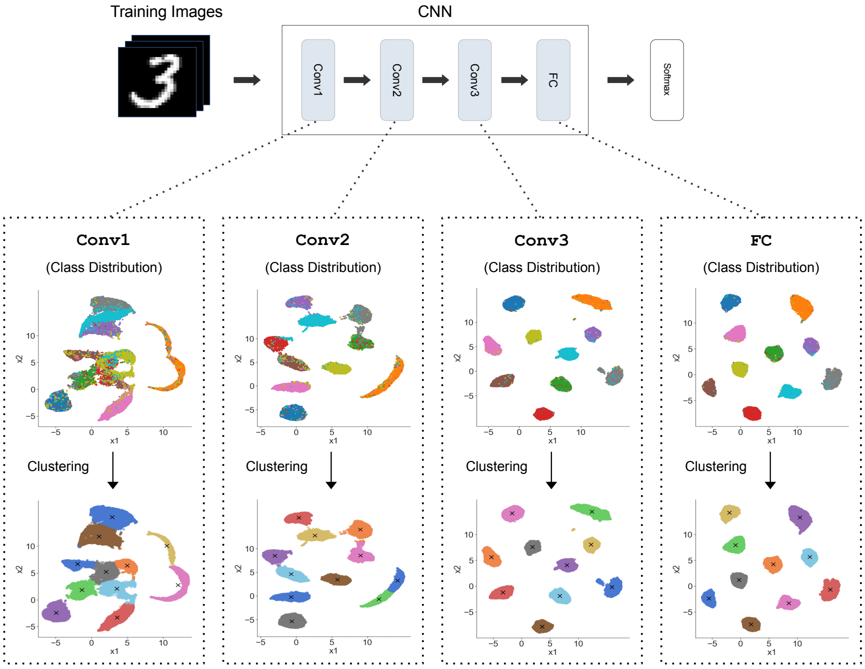


Figure 6.2: My method fetches the activations of the training images from each layer of the Convolutional Neural Network-based model (CNN) and then searches for clusters within the fetched activations.

6.2.1 Identifying Clusters in Layer Activations

My proposed method LACA for detecting out-of-distribution samples at inference with respect to a Convolutional Neural Network-based (CNN) model requires obtaining several in-distribution statistics (Chapter 6.2.2). These in-distribution statistics are based on clusters that need to be identified within the layer activations of the model, as shown in step (1) in Figure 6.1. To create activations suitable for obtaining the required in-distribution statistics, a set of in-distribution images is fed into the model. The activations created at the model layers represent the image features that the layers detected in those images. As shown by Zeiler and Fergus [148], the lower model layers (i.e., the layers closer to the input layer) detect low-level image features (e.g., colors, corners, simple textures), while the higher model layers (i.e., the layers closer to the output layer) detect high-level image features (e.g., object parts, objects in various poses). As a result, images with similar low-level image features are close to each other in image feature space (i.e., activation space) of the lower layers of the model. These images may even belong to a wide variety of different classes because low-level image features are typically not class-specific. Different classes usually share the same low-level image features (for more information, see Chapter 2.3). The ImageNet classes *baseball* and *soccer ball*, for instance, both contain images that share the same low-level image features, such as color-based or simple texture-based features of the playing field, the sportswear, or the faces of the players next to the respective ball. As a result, potential clusters within the activations of the lower model layers typically contain images (in image feature space) of a high number of different classes. Furthermore, these clusters are usually not well separated from each other, as shown in Figure 6.2. Images with similar high-level image features, on the other hand, are close together in image feature space (i.e., activation space) of the higher layers of the model. These images typically belong to only a few different classes because high-level image features tend to be class-specific (for more information, see Chapter 2.3). The ImageNet classes *baseball* and *soccer ball*, for instance, each contain distinct characteristic high-level image features, such as parts of a baseball or a soccer ball. As a result, potential clusters within the higher layer activations typically contain images (in image feature space) of only a few different classes. Clusters in feature space of the final layer generally even contain only a single class. In contrast to potential clusters in feature space of the lower layers, potential clusters in feature space of the higher layers tend to be well separated, as shown in Figure 6.2. This difference in the image feature representations between the lower and the higher model layers is caused by the classification objective of the model to find a linearly separable feature representation of the images at the final model layer that separates these images according to their classes C (for more information, see Chapter 2.2). To classify images, the model attempts to incrementally push images of the same class together and images of different classes apart from each other in image feature space when passing those images from the input to the output model layer, as shown in Figure 6.2.

My approach to identifying clusters within the model activations is based on the work of Nguyen et. al. [109]. However, the method of Nguyen et. al. [109] searches for clusters in the activations in order to visualize multifaceted image features learned by a Convolutional Neural Network-based model during model training. In contrast, I search for clusters within the activations to use the obtained cluster information for detecting out-of-distribution samples with respect to the model. However, before it is possible to search for any clusters, the activations need to be obtained from a set of in-distribution images using the model. I use the images X^D of the training dataset as the in-distribution images. As pointed out above, the training images must be in-distribution because they were used to train the model. To obtain the activations from the training images, I first freeze the model as I do not want to change the weights of the model anymore. Then, the training images are fed into the model again. As a result, each of the training images gets classified by the model. However, I am not interested in the classification result but in the activations that are created at each model layer, when a training image x^D is fed into the model. Thus, the activations of each image x^D are fetched from the model layers. In general, I fetch the activations from all model layers. However, it is also possible to omit some of the lower model layers to decrease the runtime of my method without significantly reducing its detection performance later, as my experiment in Chapter 6.3.4 shows. The higher model layers, however, should not be omitted as they are important to my method. In the following, I use a specific model layer l as an example in order to explain how to obtain the activations from a layer and how to identify clusters within the obtained activations. The same procedure must also be applied to the remaining layers.

After feeding the training images X^D into the model, the activations of these images are fetched from each model layer l. If layer l is a convolutional layer, the activations of a training image x^D are obtained in the form of a three-dimensional tensor. This three-dimensional activation tensor needs to be flattened to an activation vector. However, this step is only necessary if layer l is a convolutional layer. If layer l is a linear layer, on the other hand, this step can be omitted because the activations from linear layers are in vector form already. Suppose the training dataset contains N training images x^D. Then, an activations vector $a^l(x^D)$ of a layer-specific length M^l is obtained for each of the N training images x^D. To be able to obtain activations vectors that are all of the same length M^l, it is important that the training images X^D were all of the same image size, as pointed out above. Finally, the activation vectors are concatenated into a matrix A^l_D of size $N \times M^l$. Each row of this matrix represents a training image x^D in feature space of layer l in the form of its activations.

After obtaining the activations of the training images X^D from model layer l, I aim to search for clusters in these activations. However, matrix A^l_D, which contains the activations of the training images X^D, is usually high-dimensional because of the large amount of M^l activations that are generally obtained from model layer l. Unfortunately, identifying clusters in high-dimensional spaces does not work well, as pointed out by Chen et. al. [19]. Clustering algorithms use distance metrics to identify clusters, but distance metrics are not effective in high-dimensional spaces. However, as pointed out by Domingos [28], the data samples of most applications are located within a low-dimensional subspace within this high-dimensional space. Thus, I use dimensionality reduction to project matrix A^l_D onto such a low-dimensional subspace, as suggested by Chen et. al. [19]. However, my method for detecting out-of-distribution samples does not only require the projected matrix but also a projection model. Therefore, I needed to choose a projection technique that also provides a projection model in addition to the projected matrix. Chapter 6.3.2 shows a comparison of different suitable projection approaches with respect to finding well-separated clusters, which I evaluated with activation data received from different image datasets. The best projection result was obtained with a combination of the linear dimensionality reduction technique PCA (Principal Component Analysis) [116] and the non-linear dimensionality reduction technique UMAP (Uniform Manifold Approximation and Projection) [98] (for more information about PCA and UMAP, see Chapter 3.1). A similar approach was also used by Nguyen et. al. [109]. Therefore, I project matrix A^l_D using this approach. However, before projecting the matrix A^l_D, each of its values needs to be normalized as a preprocessing step for the dimensionality reduction (for more details, see Chapter 3.1). After normalizing each value of the matrix, the projection approach is used to learn a projection model r^l from this matrix. This projection model is actually a pipeline of two models, first the PCA and then the UMAP model. Finally, the learned projection model is applied to matrix A^l_D to project the matrix to a low-dimensional subspace. The projection model r^l first reduces the dimensionality of the matrix from $N \times M^l$ down to $N \times 50$ using the PCA model of r^l. Then, in a second reduction step, r^l further reduces the dimensionality from $N \times 50$ down to $N \times 2$ using the UMAP model of r^l. As a result, the projected matrix $r^l(A^l_D)$ and the projection model r^l are obtained.

In this projected matrix $r^l(A^l_D)$, I search for clusters. To determine the best approach to finding the clusters, I evaluated different clustering algorithms applied to activation data from different layers of a Convolutional Neural Network-based model. Furthermore, I also tested each clustering method for different image datasets. This evaluation is presented in Chapter 6.3.2. I found that the k-Means [95] clustering algorithm is generally best suited for activation data, which was also suggested by Chen et. al. [19]. Thus, I chose k-Means to search for clusters in the projected matrix $r^l(A^l_D)$. However, k-Means requires setting hyperparameter k. Hyperparameter k specifies how many clusters k-Means should search for in the projected matrix. The number of meaningful clusters within the activations of the final layer should correspond to the number of classes in the fixed set of classes C of the classification problem. This follows from the classification objective of the model to find a linearly separable feature representation of the images X^D at the final model layer that separates these images according to their classes (for more information, see Chapter 2.2). The other layers, however, may contain a number of meaningful clusters that is different from the number of classes in C. Unfortunately, it is not known how many meaningful clusters the activations of these layers contain. Furthermore, this number is most likely different for each of these layers. The activations of a higher layer before the final layer, for instance, may contain a number of meaningful clusters that is higher than the number of classes in C. If the images of a particular class can be grouped into different subclasses with respect to different semantic concepts, the images of each subclass are typically found in a different cluster in feature space of such a higher layer. Each cluster contains images of a particular semantic concept characterized by the high-level image features that were detected by this higher layer (for more information, see Chapter 2.3). The ImageNet class *baseball*, for instance, contains images showing only the ball and images showing a baseball player with the ball. Therefore, two potential subclasses of the class *baseball* could be *baseball ball* and *baseball player*. The images of the subclass *baseball ball* are located in a different cluster in feature space of the higher layer than the images of the subclass *baseball player*, as illustrated in Figure 6.3. As a result, the activations of this higher layer may contain $|C| + 1$ meaningful clusters. The activations of a lower layer, on the other hand, may contain a number of meaningful clusters that is smaller than the number of classes in C. As mentioned earlier, the lower layers detect low-level image features (e.g., color, corners, simple textures) (for more information, see Chapter 2.3). Therefore, in feature space of such a lower layer, a cluster containing images of different classes that share similar low-level features might be found. The ImageNet classes *baseball*, *soccer ball*, and *golf ball*, for instance, all contain images showing the respective ball and images showing the respective ball player. Thus, it might be possible to identify one cluster containing the images showing a ball and another cluster containing the images showing a ball player in feature space of the lower layer. Both clusters contain images with similar low-level image features, regardless of the classes of those images. As a result, the activations of these lower layers may contain $|C| - 1$ meaningful clusters. However, the number of meaningful clusters within the activations of a layer before the final layer is dataset-dependent. This number cannot be predicted in advance, and therefore, it is not known how to set hyperparameter k of the k-Means algorithm for these layers.

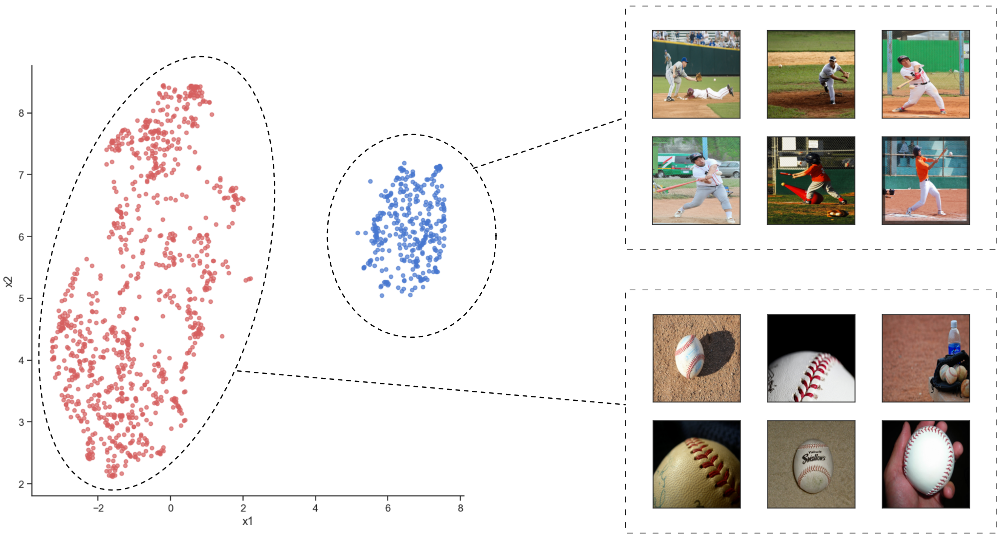


Figure 6.3: The representation of the images of the ImageNet [25] class *baseball* in image feature space (i.e., activation space) of a higher model layer. The images that show a baseball player are located in a different cluster than the images that show the ball.

To find a value for k, the number of meaningful clusters needs to be estimated in the activation data of the respective layer and hyperparameter k of k-Means needs to be set according to this estimated number. Since the dimensions of the activations of the N training images were reduced from M^l dimensions in A^l_D down to 2 dimensions in $r^l(A^l_D)$ (the rows of the matrices), the resulting compressed two-dimensional activations for each of the images can be visualized in a scatter plot. In this scatter plot, it might be possible to visually identify clusters. Thus, hyperparameter k of the k-Means algorithm could be set according to the number of clusters that were identified in the scatter plot. However, it cannot be guaranteed that this method works well in every case. It may not always be easy to identify all clusters visually. Therefore, I use a different approach to finding a good value for hyperparameter k. I simply test different values for k. We know that the activations of the final layer should contain as many meaningful clusters as we have classes. The activations of the remaining layers, on the other hand, may contain a smaller or a larger number of clusters compared to the number of classes in C. However, I expect this number to be only slightly different from the number of classes, which was also shown by my experiment in Chapter 6.3.2. As a result, I consider values between $|C| - 5$ and $|C| + 5$ for hyperparameter k. For each of these values for k, the k-Means algorithm is applied to matrix $r^l(A^l_D)$. As a result, for each of these values, I obtain a set of clusters $h^l \in h^l_1, .., h^l_k$ found by k-Means and evaluate each set of received clusters with respect to their cluster quality. The value for k that results in the clusters achieving the best overall cluster quality is chosen.

I use the silhouette score [121] to evaluate the clusters. The silhouette score is a cluster quality metric, which is calculated using the intra-cluster distance and the nearest-cluster distance of each data sample of a set of clusters (for more details, see Chapter 3.3). In our case, a data sample is the compressed activation vector $r^l(A^l_D)_{i,:} = r^l(a^l(x^D))$ of a training image x^D in a row i of matrix $r^l(A^l_D)$. Chen et. al. [19] reported that the silhouette score is well suited for evaluating clusters identified in activation data from Convolutional Neural Network-based models. The value of the silhouette score of an obtained set of clusters ranges from -1 to 1. A silhouette score close to 1 means that we have well-separated clusters. A silhouette score close to -1, however, means that a majority of data samples should be located in the nearby cluster rather than in their current cluster. Thus, a higher silhouette score for a set of clusters reflects a better cluster quality and is therefore favorable. In general, the silhouette scores of the lower layers are smaller than the silhouette scores of the higher layers because the potential clusters in feature space of the lower layers are typically not as well separated as the potential clusters in feature space of the higher layers, as shown in Figure 6.2. After computing the silhouette score for the resulting clusters from each of the considered values for k, I select the set of clusters corresponding to the value for k that achieved the highest silhouette score.

As a result, the $h^l_1, .., h^l_k$ clusters are obtained together with the clustering model g^l. The obtained clusters $h^l_1, .., h^l_k$ are required for calculating the in-distribution statistics for layer l in Chapter 6.2.2. Finally, the clustering model g^l is needed along with the projection model r^l and the obtained in-distribution statistics for checking whether later at inference an image x^I is an out-of-distribution sample (Chapter 6.2.3 and Chapter 6.2.4). After obtaining a set of clusters, a projection model, and a clustering model from layer l, this process is repeated with the remaining layers to receive clusters, a projection model, and a clustering model from these layers as well.

6.2.2 Obtaining In-Distribution Statistics

To detect whether an image x^I at inference is an out-of-distribution sample with respect to a Convolutional Neural Network-based (CNN) image classification model, I measure how closely image x^I resembles the training images X^D of the model. If image x^I does not resemble the training images, or resembles them only slightly, then x^I is most likely an out-of-distribution sample. To express whether image x^I resembles the training images, I propose two metrics, a binary similarity score $simDet(x^I)$ (Chapter 6.2.3) and a credibility score $credib(x^I)$ (Chapter 6.2.4) of image x^I. The calculation of these two metrics is based on several in-distribution statistics that I obtain from each layer l of the model (except for some of the lower layers that might have been omitted to further decrease the runtime of my proposed method, as shown in my experiments in Chapter 6.3.4). My approach to obtaining the in-distribution statistics is illustrated in step (2) in Figure 6.1. The in-distribution statistics of a layer l include the class-distribution statistic S^l_{cla}, the layer weight w^l of layer l computed from the class-distribution statistic, and the cluster-distribution statistic S^l_{clu}. However, to calculate the binary similarity score of an image at inference, only the class-distribution statistic from the model layers is needed. To calculate the credibility score of an image at inference, on the other hand, my method also needs the layer weight and the cluster-distribution statistic in addition to the class-distribution statistic from the model layers, as the calculation of the credibility score is more complex. In the following, I use a specific model layer l as an example in order to explain how to obtain the in-distribution statistics from a layer. The same procedure must also be applied to the remaining layers.

The class-distribution statistic S^l_{cla} of layer l is obtained from the k clusters $h^l_1, ..., h^l_k$ that were found within the layer activations of the training images (for more details, see Chapter 6.2.1). For each identified cluster h^l at a layer l, I check the classes of the training images $X^D_{h^l}$ that are located in that cluster (in feature space of layer l) through their respective labels $Y^D_{h^l}$. As a result, the set of classes C_{h^l} are obtained that cluster h^l contains. However, it is not only important to find out which classes occur in a cluster but also what is the percentage $p_{h^l}(c_{h^l})$ of each of those classes $c_{h^l} \in C_{h^l}$ in the cluster. Thus, for each class c_{h^l}, my method determines the percentage of training images $(X^D_{h^l}, Y^D_{h^l,y==c_{h^l}})$ in cluster h^l that belong to that class. However, as I assume that the training dataset contains outliers, classes that rarely occur in the cluster are not considered. Therefore, I only include those classes in the class-distribution statistic whose percentage is greater than a given threshold p_{thresh} (e.g., $p_{thresh} = 0.05$). The threshold p_{thresh} is a hyperparameter of my method and must therefore be specified beforehand. As a result, the distribution of the classes $S^l_{cla}(h^l)$ in cluster h^l is obtained (Equation 6.1).

$$\begin{aligned} S^l_{cla}(h^l) &= \left\{ \Big(c_{h^l}, p_{h^l}(c_{h^l}) \Big) \;\Big|\; c_{h^l} \in C_{h^l} \;,\; p_{h^l}(c_{h^l}) > p_{thresh} \right\} \\ p_{h^l}(c_{h^l}) &= \frac{|(X^D_{h^l}, Y^D_{h^l,y==c_{h^l}})|}{|(X^D_{h^l}, Y^D_{h^l})|} \end{aligned} \tag{6.1}$$

The class distribution $S^l_{cla}(h^l)$ is collected from each cluster h^l that was identified within the activations from layer l. The set of all class distributions forms the class-distribution statistic S^l_{cla} of that layer (Equation 6.2).

$$S^l_{cla} = \bigcup_{h^l} S^l_{cla}(h^l); \quad h^l \in \{h^l_1, ..., h^l_k\} \tag{6.2}$$

Finally, the same procedure is applied to the remaining layers. As a result, a class-distribution statistic S^l_{cla} is obtained from all model layers. This class-distribution statistic S^l_{cla} is used for calculating the binary similarity score $simDet(x^I)$ (Chapter 6.2.3) and the credibility score $credib(x^I)$ (Chapter 6.2.4) of image x^I at inference.

However, calculating the credibility score $credib(x^I)$ of image x^I at inference not only requires the class-distribution statistic S^l_{cla} from each layer l but also the type of the class distributions $S^l_{cla}(h^l)$ of the clusters h^l in statistic S^l_{cla}. This type typically depends on the layer l from which the clusters h^l are obtained. The class distributions of the clusters from the lower layers (i.e., the layers closer to the input layer) differ significantly from the class distributions of the clusters from the higher layers (i.e., the layers closer to the output layer). The class distributions of the clusters from the same layer, on the other hand, tend to be similar. To reflect the type of the class distributions of the clusters h^l obtained from a layer l, a score $w^l \in [0,1]$ is computed. As clusters from the same layer tend to have similar class distributions, my method computes only one score for all clusters from that layer. Therefore, hereinafter, I refer to the score w^l as the layer weight of layer l. As explained in Chapter 6.2.1, the clusters found within the activation of the lower layers usually contain images (in feature space) of a high number of classes. Moreover, these classes are usually uniformly distributed within those clusters. This type of class distribution is reflected in a low layer weight. The clusters found within the activation of the higher layers, on the other hand, usually contain images (in feature space) of a low number of classes. Moreover, the distribution of these classes within those clusters is usually highly imbalanced. The class distributions of the clusters from the final model layer typically even contain a majority class with a frequency of at least 90%. This type of class distribution is reflected in a high layer weight. After obtaining the layer weight w^l from each model layer l, I use them to calculate the credibility score $credib(x^I)$ of image x^I at inference (for more details, see Chapter 6.2.4).

In order to calculate the layer weight w^l of a layer l, I use a simple approach. From each cluster h^l in S^l_{cla}, I select the class with the highest frequency ${p'}_{h^l}$ and the class with the second-highest frequency ${p''}_{h^l}$. Then, I take the difference between their frequencies. As the result, I obtain the score w_{h^l} for the cluster h^l (Equation 6.3).

$$w_{h^l} = |{p'}_{h^l} - {p''}_{h^l}| \tag{6.3}$$

To obtain the layer weight w^l_U for the layer l, I take the average over the scores w_{h^l} of all clusters h^l. As a result, I receive the layer weight w^l_U for each layer l. These layer weights w^l_U are non-normalized, i.e., they do not sum up to 1. However, the calculation of the credibility score (Chapter 6.2.4) requires normalized layer weights. Therefore, the non-normalized layer weights w^l_U are normalized in order to obtain the normalized layer weights w^l (Equation 6.4).

$$w^l = \frac{w^l_U}{\sum_l w^l_U} \tag{6.4}$$

The obtained normalized layer weights $w^l \in [0,1]$ are each in the range between 0 and 1, and the sum of all layer weights w^l is equal to 1: $\sum_l w^l = 1$. These layer weights w^l are used for calculating the credibility score $credib(x^I)$ of image x^I at inference.

In addition to the layer weight w^l and the class-distribution statistic S^l_{cla}, my proposed method also needs a cluster-distribution statistic S^l_{clu} from each model layer l in order to calculate the credibility score $credib(x^I)$ of image x^I at inference. However, for calibration purposes, the cluster-distribution statistic is not obtained from the training dataset but from a held-out calibration dataset (X^{Ca}, Y^{Ca}). The calibration dataset consists of the calibration images X^{Ca} and their respective labels Y^{Ca}. The value of each label of Y^{Ca} corresponds to a specific class in the fixed set of classes C of the classification problem. Furthermore, the calibration dataset is also an in-distribution dataset, but it does not share any images with either the training dataset or the test dataset of the model. Papernot and McDaniel [113, 114] use a similar calibration approach in order to compute a credibility score based on their DkNN method (Deep k-Nearest Neighbors).

To be able to obtain the cluster-distribution statistic S^l_{clu} from a particular model layer l, the clusters are needed that were obtained within the activations A^l_D of the training images from that layer (for more details, see Chapter 6.2.1). The clusters from layer l are needed because it is necessary to find out into which of these clusters each calibration image x^{Ca} falls (in feature space of the layer). To be able to find the cluster $h^l_{x^{Ca}}$ into which an image x^{Ca} falls, it is necessary to first obtain the activation vector $a^l(x^{Ca})$ of the image from layer l. The activation vector $a^l(x^{Ca})$ is obtained in the same way as the activation vectors $a^l(x^D)$ of the training images X^D. First, image x^{Ca} is fed into the model. The image must be of the same image size as the training images X^D. If it is not, x^{Ca} must be resized to the size of the training images X^D before feeding x^{Ca} into the model. Then, the activation vector of image x^{Ca} is fetched from layer l (for more details, see Chapter 6.2.1). After receiving the activation vector, the cluster $h^l_{x^{Ca}}$ into which this activation vector falls must be identified. To identify the cluster, I use the projection model r^l and the clustering model g^l that was obtained from the activations A^l_D of the training images X^D from layer l (as described in Chapter 6.2.1). First, the projection model r^l is applied to the activation vector $a^l(x^{Ca})$ in order to obtain the compressed activation vector $r^l(a^l(x^{Ca}))$. Then, the clustering model g^l is applied to the compressed activation vector in order to obtain the cluster $h^l_{x^{Ca}}$ into which the compressed activation vector falls. As the activation vector reflects the image x^{Ca} in feature space of the layer, I assume for simplicity reasons hereinafter that image x^{Ca} falls into that cluster $h^l_{x^{Ca}}$ (although in reality it is the activation vector of the image).

After receiving the cluster $h^l_{x^{Ca}}$ for each image x^{Ca}, I check for each class c in which clusters h^l of layer l the class occurs with respect to the calibration images (in feature space of the layer). However, it is not only necessary to find out in which clusters a particular class c occurs but also what percentage $p_c(h^l)$ of all images of class c occurs in each cluster h^l. Thus, for each cluster h^l, the percentage of calibration images $(X^{Ca}_{h^l}, Y^{Ca}_{h^l,y==c})$ of class c that are located in that cluster is determined (in feature space of the layer). As a result, the cluster distribution $S^l_{clu}(c)$ of class c over the clusters h^l is obtained (Equation 6.5).

$$\begin{aligned} S^l_{clu}(c) &= \left\{ \left(h^l, p_c(h^l) \right) \middle| \, h^l \in \{h^l_1, ..., h^l_k\} \right\} \\ p_c(h^l) &= \frac{|(X^{Ca}_{h^l}, Y^{Ca}_{h^l,y==c})|}{|(X^{Ca}, Y^{Ca}_{y==c})|} \end{aligned} \tag{6.5}$$

The cluster distribution $S^l_{clu}(c)$ of each class c is collected from layer l. The set of all cluster distributions forms the cluster-distribution statistic S^l_{clu} of l (Equation 6.6).

$$S^l_{clu} = \bigcup_c S^l_{clu}(c); \quad c \in C \tag{6.6}$$

Finally, the same procedure is applied to the remaining layers. As a result, a cluster-distribution statistic S^l_{clu} is obtained from all model layers. These cluster-distribution statistics S^l_{clu} are used for calculating the credibility score $credib(x^I)$ of image x^I at inference.

6.2.3 Naive Out-of-Distribution Detection

To detect whether an image x^I at inference is an out-of-distribution sample with respect to a Convolutional Neural Network-based (CNN) image classification model, it needs to be checked if the image does or does not resemble the training images X^D of the model. If x^I does resemble the training images, I conclude that x^I is an in-distribution sample. If x^I does not resemble the training images, however, I conclude that x^I is an out-of-distribution sample. To check whether x^I does or does not resemble the training images of the model, I calculate a binary similarity score $simDet(x^I) \in \{0, 1\}$ of image x^I. A similarity score of 1 indicates that x^I does resemble the training images and is therefore an in-distribution sample. A similarity score of 0, on the other hand, indicates that x^I does not resemble the training images and is therefore an out-of-distribution sample. To calculate the similarity score of image x^I, x^I and the training images X^D first need to be fed into the model (the training images were actually fed into the model already, as described in Chapter 6.2.1). Image x^I must be of the same image size as the training images. If it is not, x^I must be resized to the image size of the training images before feeding x^I into the model. Then, at each model layer l, it is examined which of the training images are close to x^I in feature space of the layer. If the training images that are close to image x^I belong to the same class as x^I in feature space of each layer, I assume that x^I resembles the training images and is therefore an in-distribution sample. If the training images that are close to image x^I belong to a class that is different from the class of x^I in feature space of at least one layer, however, I assume that x^I does not resemble the training images and is therefore an out-of-distribution sample. Unfortunately, the class c_{x^I} of image x^I is not known at inference because its class label y^I ($y^I = c_{x^I}$) is not available. The class label y^I should be predicted by the model. Therefore, it is checked instead whether image x^I is always close to training images of the same class c_{com} in feature space of each model layer l. If image x^I is always close to training images of the same class c_{com} in feature space of each model layer, I conclude that x^I is most likely also of class c_{com} (i.e., $c_{x^I} = c_{com}$). As a result, I further conclude that x^I does resemble the training images and is therefore an in-distribution sample. This conclusion is based on my assumption: Each in-distribution sample is always close to other in-distribution samples of the same class in activation space (i.e., image feature space) of each model layer. In this case, the training images are the in-distribution samples. If the class of the training images that are close to x^I differs across the different model layers l, however, then my assumption is violated. As a result, I conclude that image x^I does not resemble the training images and is therefore an out-of-distribution sample.

My approach to calculating the similarity score $simDet(x^I)$ of image x^I requires the class-distribution statistic S^l_{cla} from each model layer l, which was obtained beforehand (for more details, see Chapter 6.2.2). The class-distribution statistic is needed in order to find out which of the training images are close to image x^I in feature space of each model layer l. In particular, the classes of these training images are required by my proposed method. I assume that a training image x^D is close to image x^I in feature space of a layer l, if both images are located in the same cluster among the clusters h^l that were identified beforehand within the activations of the training images from that layer (for more details, see Chapter 6.2.1). As a result, it is necessary to examine the classes of the training images that are located in the same cluster as image x^I. First of all, however, the cluster into which image x^I falls needs to be identified. To identify the cluster into which image x^I falls, it is first necessary to obtain the activation vector $a^l(x^I)$ of image x^I from layer l. The activation vector $a^l(x^I)$ is obtained in the same way as the activation vectors $a^l(x^D)$ of the training images X^D. First, the image x^I is fed into the model and then, the activation vector of image x^I is fetched from layer l (for more details, see Chapter 6.2.1). After receiving the activation vector $a^l(x^I)$, the cluster $h^l_{x^I}$ into which this activation vector falls must be identified. To identify the cluster, I use the projection model r^l and the clustering model g^l that were obtained from the activations A^l_D of the training images X^D from layer l (as described in Chapter 6.2.1). First, the projection model r^l is applied to the activation vector $a^l(x^I)$ in order to obtain the compressed activation vector $r^l(a^l(x^I))$. Then, the clustering model g^l is applied to the compressed activation vector in order to obtain the cluster $h^l_{x^I}$ into which the compressed activation vector falls. As the activation vector reflects the image x^I in feature space of the layer, I assume for simplicity reasons hereinafter that image x^I falls into that cluster $h^l_{x^I}$ (although in reality it is the activation vector of the image). After determining the cluster $h^l_{x^I}$ into which image x^I falls, it is examined which of the training images also occur in that cluster. However, it is not necessary to identify the exact training images but the classes of these training images. To obtain the classes of those training images, the class-distribution statistic S^l_{cla} can be used. Through $S^l_{cla}(h^l_{x^I})$, it is possible to obtain the set of classes $cset^l_{x^I}$ from the training images that are located in cluster $h^l_{x^I}$. I assume that $cset^l_{x^I}$ contains the potential classes of image

x^I. If image x^I is an in-distribution sample, then one of the classes in $cset^l_{x^I}$ must be the class c_{x^I} of x^I. However, $cset^l_{x^I}$ typically differs across the different model layers l. As shown in Chapter 6.2.1, the clusters of the lower model layers (i.e., the layers closer to the input layer) usually contain a high number of classes. Therefore, the class set $cset^l_{x^I}$ from a lower layer contains a high number of classes as well. The clusters of the higher layers (i.e., the layers closer to the output layer), on the other hand, usually contain a low number of classes. Therefore, the class set $cset^l_{x^I}$ from a higher layer contains a low number of classes as well. Nevertheless, if image x^I is an in-distribution sample, then there must be at least one common class c_{com} across the different model layers l. I assume that one of these common classes c_{com} is the class c_{x^I} of image x^I. This follows from my assumption that each in-distribution sample is always close to other in-distribution samples of the same class in activation space (i.e., image feature space) of each model layer. Therefore, I take the intersection of the class sets $cset^l_{x^I}$ that were obtained from each model layer l in order to receive the set $cset_{x^I}$ of common classes $c_{com} \in cset_{x^I}$ (Equation 6.7).

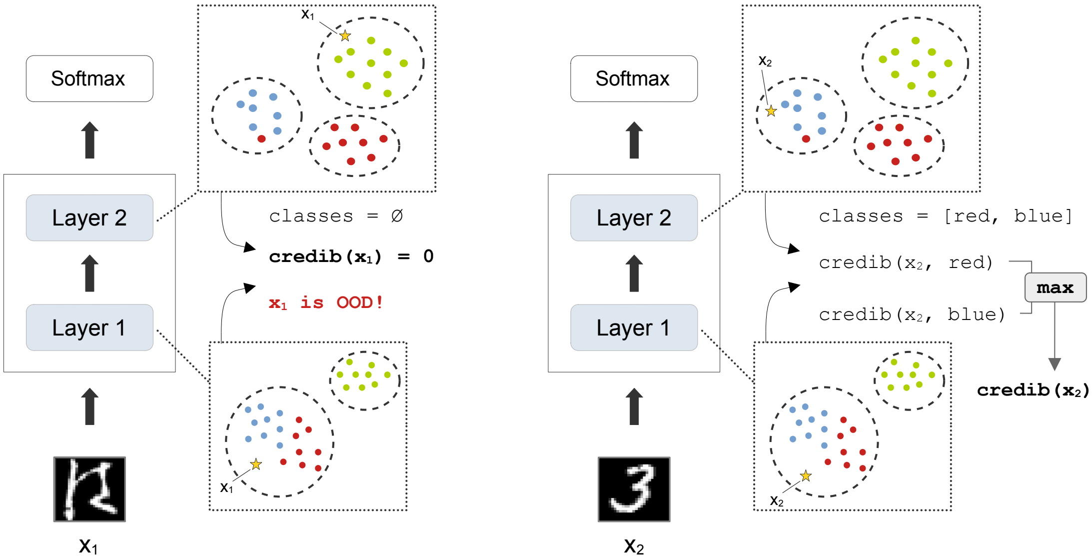
$$cset_{x^I} \ = \ \bigcap_{l} cset^l_{x^I} \tag{6.7}$$

If the resulting class set $cset_{x^I}$ is non-empty, I conclude that image x^I resembles the training images and is therefore an in-distribution sample. As a result, I set $simDet(x^I) = 1$. If the resulting class set $cset_{x^I}$ is empty, however, then my assumption is violated. Image x^I is not always close to in-distribution samples (i.e., training data samples) in feature space of each model layer. As a result, I conclude that image x^I does not resemble the training images and is therefore an out-of-distribution sample. As a result, I set $simDet(x^I) = 0$ (Equation 6.8).

$$simDet(x^I) \ = \ \begin{cases} 0\,, & if \ cset(x^I) = \emptyset \\ 1\,, & otherwise \end{cases} \tag{6.8}$$

6.2.4 Sample Credibility-based Out-of-Distribution Detection

To detect whether an image x^I at inference is an out-of-distribution sample with respect to a Convolutional Neural Network-based (CNN) image classification model, it needs to be checked if image x^I does or does not resemble the training images X^D of the model. If image x^I does resemble the training images, I conclude that x^I is an in-distribution sample. If image x^I does not resemble the training images, however, I conclude that x^I is an out-of-distribution sample. To check whether image x^I does or does not resemble the training images of the model, I have already proposed a binary similarity score $simDet(x^I)$ of image x^I in Chapter 6.2.3. The binary similarity score is easy to calculate. However, the similarity score may not always achieve a sufficient



(a) Out-of-Distribution Sample

(b) In-Distribution Sample

Figure 6.4: The calculation of the credibility score of an image with respect to a model trained on the MNIST [76] dataset to detect whether the image is (a) an out-of-distribution (OOD) sample (e.g., a KMNIST [21] image) or (b) an in-distribution sample.

detection performance. Thus, I propose a second detection metric, the credibility score $credib(x^I) \in [0, 1]$ of image x^I. The credibility score is a float value between 0 and 1 reflecting how closely image x^I resembles the training images. A credibility score close to 1 indicates that image x^I closely resembles the training images and therefore, I conclude that x^I is most likely an in-distribution sample. A credibility score close to 0, on the other hand, indicates that image x^I does not resemble the training images and therefore, I conclude that x^I is most likely an out-of-distribution sample. The credibility score is more complex to calculate than the binary similarity score. However, the credibility score generally achieves a better detection performance than the similarity score. An overview of the calculation of the credibility score is illustrated in Figure 6.4.

The calculation of the credibility score $credib(x^I)$ of image x^I is based on the same approach as the calculation of the binary similarity score $simDet(x^I)$ of image x^I (for more details, see Chapter 6.2.3). First, image x^I and the training images X^D need to be fed into the model (the training images were actually fed into the model already, as described in Chapter 6.2.1). Image x^I must be of the same image size as the training images. If it is not, x^I must be resized to the image size of the training images before feeding x^I into the model. Then, at each model layer l, it is examined which of the training images are close to x^I in feature space of the layer. If the training images that are close to x^I belong to the same class as x^I in feature space of each layer, I conclude that x^I resembles the training images and is therefore an in-distribution sample. If the training images that are close to x^I belong to a class that is different from the class of

x^I in feature space of at least one model layer, I conclude that x^I does not resemble the training images and is therefore an out-of-distribution sample. Unfortunately, the class c_{x^I} of image x^I is not known at inference because its class label y^I ($y^I = c_{x^I}$) is not available. The class label y^I should be predicted by the model. Therefore, it is checked instead whether image x^I is always close to training images of the same class c_{com} in feature space of each model layer l. If image x^I is always close to training images of the same class c_{com} in feature space of each model layer, I conclude that x^I is most likely also of class c_{com} (i.e., $c_{x^I} = c_{com}$). As a result, I further conclude that x^I does resemble the training images and is therefore an in-distribution sample. This conclusion is based on my assumption: Each in-distribution sample is always close to other in-distribution samples of the same class in activation space (i.e., image feature space) of each model layer. In this case, the training images are the in-distribution samples. If the class of the training images that are close to x^I differs across the different model layers l, however, then my assumption is violated. As a result, I conclude that image x^I does not resemble the training images and is therefore an out-of-distribution sample.

To find out which of the training images are close to image x^I in feature space of a model layer l, the cluster information is used that was obtained within the activations of the training images from that layer (for more details, see Chapter 6.2.1). I assume that a training image x^D is close to image x^I in feature space of layer l, if both images are located in the same cluster among the identified clusters h^l of this layer. As a result, it is necessary to examine the classes of the training images that are located in the same cluster as x^I at layer l. The set of these classes $cset^l_{x^I}$ of layer l is obtained in the same way as for calculating the binary similarity score $simDet(x^I)$ using the class-distribution statistic S^l_{cla} of layer l (for more details, see Chapter 6.2.3). First of all, the cluster into which x^I falls needs to be identified. To identify the cluster into which x^I falls, it is first necessary to obtain the activation vector $a^l(x^I)$ of x^I from layer l. The activation vector $a^l(x^I)$ is obtained in the same way as the activation vectors $a^l(x^D)$ of the training images X^D. First, x^I is fed into the model and then, the activation vector of x^I is fetched from layer l (for more details, see Chapter 6.2.1). After receiving the activation vector $a^l(x^I)$, the cluster $h^l_{x^I}$ into which this activation vector falls must be identified. To identify the cluster, I use the projection model r^l and the clustering model g^l that were obtained from the activations A^l_D of the training images X^D from layer l (as described in Chapter 6.2.1). First, the projection model r^l is applied to the activation vector $a^l(x^I)$ in order to obtain the compressed activation vector $r^l(a^l(x^I))$. Then, the clustering model g^l is applied to the compressed activation vector in order to obtain the cluster $h^l_{x^I}$ into which the compressed activation vector falls. As the activation vector reflects the image x^I in feature space of the layer, I assume for simplicity reasons hereinafter that x^I falls into that cluster $h^l_{x^I}$ (although in reality it is the activation vector of x^I). After determining the cluster $h^l_{x^I}$ into which x^I falls, it is examined which of the training images also occur in that cluster. However, it is not necessary to identify the exact training images but the classes of these training images. To obtain the classes of those training images, the class-distribution statistic S^l_{cla} can be used. Through $S^l_{cla}(h^l_{x^I})$, it is possible to obtain the set of classes $cset^l_{x^I}$ from the training images that are located in cluster $h^l_{x^I}$.

I assume that $cset^l_{x^I}$ contains the potential classes of image x^I. If image x^I is an in-distribution sample, then there must be at least one class c_{com} that is present in all class sets $cset^l_{x^I}$ across the different model layers l. I assume that one of these common classes c_{com} is the class c_{x^I} of image x^I. This follows from my assumption that each in-distribution sample is always close to other in-distribution samples of the same class in activation space (i.e., image feature space) of each model layer. In order to obtain the set $cset_{x^I}$ of common classes $c_{com} \in cset_{x^I}$, I use the same approach as for calculating the binary similarity score $simDet(x^I)$. I simply take the intersection of the class sets $cset^l_{x^I}$ that were obtained from each model layer l (for more details, see Chapter 6.2.3). However, if the resulting class set $cset_{x^I}$ is empty, then my assumption regarding in-distribution samples is violated. In this case, image x^I is not always close to in-distribution samples (i.e., training images) of the same class in feature space of each model layer. As a result, I conclude that image x^I does not resemble the training images X^D and is therefore an out-of-distribution sample. As a result, I set $credib(x^I) = 0$. If the resulting class set $cset_{x^I}$ is non-empty, on the other hand, I conclude that image x^I more or less resembles the training images. Thus, x^I might be an in-distribution sample.

However, it still cannot be ensured that image x^I is really an in-distribution sample. This is especially the case if the classes c_{com} of $cset_{x^I}$ occur in the cluster $h^l_{x^I}$ of at least some model layers with a probability $p_{h^l_{x^I}}(c_{com})$ ($c_{com} \in cset_{x^I}$) that is greater than the threshold p_{thresh} but that is still relatively low (for more information about the threshold p_{thresh}, see Chapter 6.2.2). Furthermore, each of these classes may also occur in another cluster with a significantly higher probability. This would suggest that the training images of the classes c_{com} in cluster $h^l_{x^I}$ might be outliers. As a result, image x^I might be an outlier as well or an out-of-distribution sample. However, it is also possible that the training images of class c_{com} are not outliers but simply have a low probability of occurring in cluster $h^l_{x^I}$ because k-Means was unable to find good clusters within the activations of the training images (for more information, see Chapter 3.2 and Chapter 6.2.1). Therefore, to find out if image x^I is really an in-distribution sample, a credibility score $credib(x^I, c_{com})$ of image x^I is calculated for each class c_{com} of $cset_{x^I}$. This credibility score expresses how closely image x^I resembles the training images, assuming that x^I is of class c_{com} (i.e., $c_{x^I} = c_{com}$). I expect that the class in $cset_{x^I}$ that achieves the highest credibility score is the true class c_{x^I} of image x^I. The highest credibility score is used as the final credibility $credib(x^I)$ of image x^I.

In order to calculate the credibility score $credib(x^I, c_{com})$, I use the probability $p_{c_{com}}(h^l_{x^I})$ that in-distribution samples of class c_{com} occur in cluster $h^l_{x^I}$ at each model layer l. The probability reflects how likely it is that an in-distribution sample of class c_{com} occurs in cluster $h^l_{x^I}$ at a layer l. A high probability indicates that in-distribution samples of class c_{com} normally occur in this cluster. Therefore, I conclude that image x^I closely resembles the training images (i.e., the in-distribution samples) in feature space of this layer, assuming that x^I is of class c_{com} because a high number of training images of the same class occur in this cluster as well. In this case, the credibility $credib(x^I, c_{com})$ should be also high. A low probability, on the other hand, indicates that in-distribution samples of class c_{com} do not normally occur in this cluster. Therefore, I conclude that image x^I does not resemble the training images (i.e., the in-distribution samples) in feature space of this layer, assuming that x^I is of class c_{com} because only a low number of training samples of the same class occur in this cluster as well. Image x^I might be an outlier or an out-of-distribution sample. In this case, the credibility $credib(x^I, c_{com})$ should be low as well. As a result, the probability $p_{c_{com}}(h^l_{x^I})$ is used as the credibility $credib^l(x^I, c_{com})$ of image x^I at layer l. The probability could also be obtained from the class-distribution statistic S^l_{cla} from layer l, which was obtained from the cluster information of the training dataset (for more details, see Chapter 6.2.2). However, for calibration purposes, the probability is not obtained from the cluster information of the training dataset but from the cluster information of the calibration dataset. The probability is received from the cluster-distribution static S^l_{clu} from layer l ($p_{c_{com}}(h^l_{x^I}) = S^l_{clu}(c_{com})(h^l_{x^I})$), which was obtained from the calibration dataset (for more details, see Chapter 6.2.2).

However, the probability $p_{c_{com}}(h^l_{x^I})$ should not be 0. A probability of 0 means that class c_{com} does not occur in cluster $h^l_{x^I}$ with respect to the calibration dataset. Image data samples from the training dataset occur in cluster $h^l_{x^I}$ but no image data samples from the calibration dataset. As a result, image x^I more or less resembles the training images X^D, but x^I does not resemble the calibration images X^{Ca}. Therefore, I assume that the training images of a class c_{com} in cluster $h^l_{x^I}$ might be outliers. I conclude that image x^I is either an outlier as well or an out-of-distribution sample. Hence, my assumption is violated because image x^I is not close to calibration images X^{Ca} (i.e., in-distribution samples) of the same class. As a result, I set $credib(x^I) = 0$.

If the probability $p_{c_{com}}(h^l_{x^I})$ is not 0, however, the probability is used as the credibility score $credib^l(x^I, c_{com})$ of image x^I at layer l, assuming that x^I is of class c_{com}. However, the credibility score only reflects the credibility of x^I with respect to layer l. To obtain the total credibility score $credib(x^I, c_{com})$, the credibility scores $credib^l(x^I, c_{com})$ are simply averaged over all model layers l. However, I do not use a standard average but a weighted average over all credibility values. This weighted average is necessary due to the characteristics of the class distributions in the clusters across the different layers, as pointed out in Chapter 6.2.1. The clusters found within the activations of the lower layers (i.e., the layers closer to the input layer) usually contain a high number of classes that tend to be uniformly distributed. The clusters within the activations of the higher layers (i.e., the layers closer to the output layer), on the other hand, usually contain a low number of classes that have a highly imbalanced class distribution. Therefore, the classes in the clusters of the lower layers normally have a significantly lower probability $p_{c_{com}}(h^l_{x^I})$ than the classes in the clusters of the higher layers. As a result, if a standard average was used, then a low overall credibility score $credib(x^I, c_{com})$ would always be obtained. Therefore, when the credibility scores $credib^l(x^I, c_{com})$ of all layers l are averaged, more weight is put on the credibility scores of the higher layers than on the credibility scores of the lower layers. This makes sense because a class with a low probability in a cluster of a lower layer might be normal, while a low probability in a cluster of a higher layer is suspicious. The layer weights w^l are used as the weights

```python
def calcCredib(x_i):
    # get potential classes of x_i
    classSet = []
    for l in range(1, numLayers):
        a = getActivations(x_i, l)
        h = getCluster(r(l), g(l), a)
        classSet_l = getClasses(s_d(l, h))
        if l == 1:
            classSet = classSet_l
        else:
            classSet = intersect(classSet, classSet_l)

    # check if class set is empty
    if classSet is empty:
        return 0  # x_i is out-of-distribution!

    # get credibility score
    credibList = []
    for c in classSet:
        for l in range(1, numLayers):
            a = getActivations(x_i, l)
            h = getCluster(r(l), g(l), a)
            prob = getProb(s_t(l, c)(h))
            if prob == 0:
                return 0  # x_i is out-of-distribution!
            credibList.append(prob * w(l))

    return max(credibList)
```

Figure 6.5: Python code for calculating the credibility score of an image.

for the weighted average. As a result, the credibility score $credib(x^I, c_{com})$ of image x^I is received for each class c_{com} (Equation 6.9).

$$\begin{aligned} credib(x^I, c_{com}) &= \sum_l w^l \cdot credib^l(x^I, c_{com}) \\ credib^l(x^I, c_{com}) &= p_{c_{com}}(h^l_{x^I}) = S^l_{clu}(c_{com})(h^l_{x^I}) \end{aligned} \tag{6.9}$$

The highest of the obtained credibility scores $credib(x^I, c_{com})$ is the final credibility score $credib(x^I)$ of image x^I (Equation 6.10).

$$credib(x^I) = \max_{c_{com}} credib(x^I, c_{com}) \tag{6.10}$$

The algorithm for calculating the credibility score $credib(x^I)$ of image x^I in Python is shown in Figure 6.5.

6.3 Experiments

To detect out-of-distribution samples with respect to a Convolutional Neural Network-based (CNN) image classification model, I proposed my method, LACA (Layer-wise Activation Cluster Analysis). I describe the details of LACA in Chapter 6.2. As pointed out in Chapter 6.2.1, identifying well-separated clusters within the layer activations of the model is crucial to my method. Therefore, I compared different clustering approaches on layer activations. However, I did not only compare these clustering approaches applied to the layer activations from one dataset but several datasets. The results of this comparison are presented in Chapter 6.3.2. After identifying the best clustering approach, I evaluated LACA with respect to various datasets in different experiments. The general setup of these experiments is described in Chapter 6.3.1. In a first experiment, I tested LACA on several simple datasets in order to find out if LACA is able to detect out-of-distribution samples at all. Furthermore, I also compared LACA to the DkNN method (Deep k-Nearest Neighbors) [113, 114] with respect to detection performance and runtime. The results are presented in Chapter 6.3.3. To further decrease the runtime of LACA, I tested in a second experiment whether the obtained cluster information from some of the lower model layers (i.e., the layers closer to the input layer) can be omitted without significantly decreasing the detection performance of LACA. The results are presented in Chapter 6.3.4. Finally, I evaluated LACA in a third experiment on more complex datasets in comparison to the DkNN method with respect to detection performance and runtime. The results are presented in Chapter 6.3.5.

6.3.1 Experimental Setup

To test my proposed method LACA (Layer-wise Activation Cluster Analysis), I conducted several experiments using different datasets. For each experiment, I used the same experimental setup. In order to describe this setup, I use the MNIST [76] dataset as an example below. I first trained a Convolutional Neural Network-based (CNN) image classification model. To train the model, I used the official MNIST training dataset as the training dataset (X^D, Y^D) for the model. After model training, I tested the model on a test dataset. However, I did not use the whole official MNIST test dataset to test the model. Instead, I split the official MNIST test dataset into a test dataset (X^T, Y^T) and a calibration dataset (X^{Ca}, Y^{Ca}). The calibration dataset is required by LACA (for more details, see Chapter 6.2.2). For each experiment, I used a set of 750 calibration images X^{Ca}, which I randomly selected once from the official MNIST test dataset and then used for all of my tests. The remaining images of the official MNIST test dataset were used as the test images X^T. I tested the model on these test images X^T in order to ensure that the model achieves a sufficient classification performance. Then, I created the in-distribution statistics using the training images X^D and the calibration images X^{Ca}, as described in Chapter 6.2.2. After obtaining the model and the in-distribution statistics, I evaluated the detection performance and runtime of LACA with respect to the obtained model.

In order to evaluate LACA, I used different test datasets ds_{test}. Each of these test datasets ds_{test} contained images of a specific type that potentially occur at inference. I distinguished between three different types of these test datasets ds_{test}: An in-distribution dataset, a natural out-of-distribution dataset, and different adversarial out-of-distribution datasets. As pointed out in Chapter 6.2, I assumed that the test images X^T had been drawn from the same data distribution as the training images X^D of the model, i.e., the test images X^T are in-distribution samples. As a result, I used the test dataset (X^T, Y^T) as the in-distribution dataset for evaluating LACA. To obtain the out-of-distribution datasets, however, I could not use any images from the official MNIST training dataset or the official MNIST test dataset because these are in-distribution samples. Therefore, I used the test dataset $(X^{T'}, Y^{T'})$ of a different dataset such as the KMNIST dataset [21] as the natural out-of-distribution dataset. The classification performance of the model on this natural out-of-distribution dataset should be low because the KMNIST images are significantly different from the MNIST images, which the model saw during model training. To obtain the adversarial out-of-distribution datasets, on the other hand, I applied different adversarial attacks to the test images X^T. The resulting adversarial out-of-distribution samples (X^T_{adv}, Y^T) from a certain attack were then used as an adversarial out-of-distribution dataset in order to test LACA. Again, the classification performance of the model on each adversarial out-of-distribution dataset should be low as this is the goal of the respective adversarial attack. Furthermore, I evaluated three different values for the hyperparameter p_{thresh} of LACA (for more information about hyperparameter p_{thresh}, see Chapter 6.2.2): 0.01, 0.05, and 0.1. After applying LACA to a test dataset ds_{test}, I received a credibility score $credib_{laca}$ for each image of ds_{test}. Alternatively, I could have also calculated the binary similarity score $simDet$ using LACA. However, in my experiments, I focused on the credibility score due to its better detection performance and the possibility to compare it to the credibility score $credib_{dknn}$ of the DkNN method (Deep k-Nearest Neighbors) [113, 114]. To obtain a credibility score for the whole test dataset ds_{test}, I simply took the mean credibility score over the obtained credibility scores from all images of ds_{test}. As a result, I received the mean credibility score for the in-distribution dataset as well as for each out-of-distribution dataset. The mean binary similarity score $simDet$ would need to be computed in the same way.

However, it is possible that LACA achieves a sufficient detection performance only on either the in-distribution dataset or an out-of-distribution dataset but not on both at the same time. For instance, LACA may achieve a high mean credibility score on the in-distribution dataset and also a high mean credibility score on the out-of-distribution dataset. A high mean credibility score on the in-distribution dataset is desired because a high mean credibility score also suggests that the dataset is an in-distribution dataset. A high mean credibility score on the out-of-distribution dataset, on the other hand, is not desired because a high mean credibility score still suggests that the dataset is an in-distribution dataset. However, this time the dataset is out-of-distribution and not in-distribution. Therefore, if LACA achieves a high mean credibility score on the in-distribution dataset and a high mean credibility score on the out-of-distribution dataset as well, both datasets cannot be distinguished by their credibility scores. We have a similar situation when LACA obtains a low mean credibility score on the out-of-distribution dataset and also a low mean credibility score on the in-distribution dataset. In both cases, it cannot be detected which of the datasets is the in-distribution dataset and which is the out-of-distribution dataset. To be able to detect whether a dataset is in-distribution or out-of-distribution, I aimed to achieve a difference between the mean credibility score on the in-distribution dataset and the mean credibility score on the out-of-distribution dataset that is as high as possible. This difference value is crucial to properly evaluate LACA. Therefore, I additionally calculated the difference value between the mean credibility score of the in-distribution dataset and the mean credibility score of each out-of-distribution dataset in order to better evaluate the detection performance of LACA.

Furthermore, I aimed to compare LACA to the DkNN method. Therefore, I evaluated the DkNN method on each test dataset ds_{test} as well. Similar to LACA, the DkNN method also requires a calibration dataset. I used the same calibration dataset of 750 images that I already used for LACA. As pointed out above, I obtained this calibration dataset from the official MNIST testing set. Papernot and McDaniel [113, 114] used the same approach to obtaining 750 calibration images for their experiments. Finally, the DkNN method also calculates a credibility score $credib_{dknn}$ for an image. This credibility score has the same properties as the credibility score obtained by LACA. Both credibility scores are float values ranging between 0 and 1. A value close to 0 indicates that the image is an out-of-distribution sample, while a value close to 1 indicates that the image is an in-distribution sample. As a result, it is possible to compare LACA with the DkNN method by their obtained credibility scores. The binary similarity score $simDet$ computed by my method, on the other hand, cannot be used for this comparison as it only provides a simple binary value. This binary value cannot easily be compared to the credibility score calculated by the DkNN method, which is a float value between 0 and 1. After applying the DkNN method to a test dataset ds_{test}, I received a credibility score from DkNN for each image data sample of ds_{test}. To obtain the credibility score for the whole test dataset ds_{test}, I used the same approach that I already used to obtain the credibility score for a whole test dataset ds_{test} using LACA. I simply took the mean over the DkNN credibility scores from all images of ds_{test} in order to obtain the DkNN credibility score for the whole test dataset ds_{test}. As a result, I received the mean DkNN credibility score for the in-distribution dataset as well as the mean DkNN credibility score for each out-of-distribution dataset. Furthermore, to better evaluate the detection performance of the DkNN method, I also calculated the difference between the mean DkNN credibility score of the in-distribution dataset and the mean DkNN credibility score of each out-of-distribution dataset as I did for evaluating LACA. Finally, to compare the detection performance of LACA and the DkNN method, I compared the difference values of the credibility scores obtained by DkNN to the difference values of the credibility scores obtained by LACA. However, I did not only compare LACA to the DkNN method with respect to their detection performances but also their runtimes at inference. I expected LACA to be significantly faster than the DkNN method.

6.3.2 Comparing Alternative Clustering Approaches

Identifying well-separated clusters within the layer activations of a Convolutional Neural Network-based (CNN) image classification model is crucial to LACA, as pointed out in Chapter 6.2.1. Therefore, I compared different clustering approaches on layer activations. I used the following three datasets for this comparison: The MNIST [76] dataset, the SVHN [106] dataset, and the CIFAR-10 [70] dataset. The MNIST dataset contains grayscale images of size $28 \times 28 \times 1$ that are organized into 10 classes. Each MNIST class represents a different bright handwritten digit in the center of the image on black background (digits: 0-9). The SVHN dataset contains images showing digits that are organized into 10 classes as well (digits: 0-9). However, these images are color images of size $32 \times 32 \times 3$. Furthermore, the SVHN images do not show handwritten digits but digits from house numbers as they were extracted from photos taken from Google Streetview[2]. The CIFAR-10 images, on the other hand, are color images of size $32 \times 32 \times 3$ as well, but they do not show a simple object such as a digit. Instead, the images of the CIFAR-10 dataset show a natural image object belonging to one of 10 classes (e.g., horse, deer, ship) in front of a natural image background (e.g., meadow, forest, sea). However, before I could search for clusters within the layer activations from each of these three datasets, I needed to obtain these activations. To obtain the activations, I first had to train a model for each dataset.

In order to train the MNIST model, I chose a simple Convolutional Neural Network-based model architecture that consisted of 3 consecutive convolutional layers (Conv) followed by the fully-connected linear output layer (FC). The ReLU function (Rectified Linear Unit) [39] was used as the activation function for the pre-activation outputs of the layers. The complete model architecture of the MNIST model is as follows: Conv1 (number of filters: 128, kernel size: 6, stride: 2) - Conv2 (number of filters: 128, kernel size: 6, stride: 2) - Conv3 (number of filters: 128, kernel size: 5, stride: 1) - FC (layer size: 10). I initialized the weights of each model layer using the Kaiming Uniform[3] initialization method [45]. Then, I trained the model for 6 training epochs using the Adam[4] optimizer [65] and a learning rate of 0.001. I used the whole official MNIST training dataset for model training (60,000 data samples). After model training, I tested the resulting MNIST model on the test dataset (9,250 data samples), which consisted of the images of the official MNIST test dataset (10,000 data samples) except for the 750 images that I used for the calibration dataset. The obtained MNIST model achieved an accuracy of 99.05% on the test dataset.

In order to train the SVHN model, on the other hand, I used the same Convolutional Neural Network-based model architecture that I already used for the MNIST model. Furthermore, I also initialized the weights of each model layer using the Kaiming Uniform initialization method. However, as SVHN is more complex than MNIST, I chose a slightly different training setup in order to train the SVHN model. I also used the Adam optimizer, but I needed to train the model for 18 training epochs to obtain a sufficient

---

[2] `https://www.google.com/streetview`

[3] `https://pytorch.org/docs/stable/nn.init#torch.nn.init.kaiming_uniform_`

[4] `https://pytorch.org/docs/stable/generated/torch.optim.Adam.html`

classification performance of the model. Furthermore, I did not use a fixed learning rate value for model training. Instead, I trained the SVHN model with a multi-step[5] learning rate schedule. I set the initial learning rate to 0.001. Then, after epoch 10, 14 and 16, I decreased the learning rate value by a factor γ of 0.1. I used the whole official SVHN training dataset for model training (73,257 data samples). After model training, I tested the resulting SVHN model on the test dataset (25,282 data samples), which consisted of the images of the official SVHN test dataset (26,032 data samples) except for the 750 images that I used for the calibration dataset. The obtained SVHN model achieved an accuracy of 89.94% on the test dataset.

Finally, in order to train the CIFAR-10 model, I chose the 20-layer ResNet (Residual Network) model architecture [46] from Zhang et. al. [150]. I initialized the weights of each model layer using the Fixup initialization method [150] instead of Kaiming Uniform, which allowed me to have a faster model training. However, as CIFAR-10 is more complex than MNIST and SVHN, I needed to adjust my training setup again. To obtain a model achieving a sufficient classification performance for CIFAR-10, I used a training setup based on the training setup[6] used by Zhang et. al. [150]. I trained the model for 200 training epochs using the SGD[7] optimizer (Stochastic Gradient Descent) [122]. Furthermore, I used a learning rate schedule instead of a fixed learning rate value again. The best classification performance of the model was obtained by a cosine-annealing[8] learning rate schedule [91] with an initial learning rate value of 0.1. I used the whole official CIFAR-10 training dataset for model training (50,000 data samples). Additionally, I also used variations of the training data samples that I obtained through different data augmentation techniques [128]. These techniques included randomly flipping training images horizontally, randomly cropping training images and applying the Mixup augmentation technique [149] to the training images. After model training, I tested the resulting CIFAR-10 model on the test dataset (9,250 data samples), which consisted of the images of the official CIFAR-10 test dataset (10,000 data samples) except for the 750 images that I used for the calibration dataset. The obtained CIFAR-10 model achieved an accuracy of 92.42% on the test dataset.

After obtaining the model for the respective dataset, I fed the training images into the model again in order to create the activations of these images at the model layers. Then, I fetched the activations from each model layer that I chose to use for detecting out-of-distribution samples. Within the activations of the selected model layers, I aimed to search for the clusters. From the MNIST and SVHN model, I fetched the activations from all model layers, i.e., the activations from the convolutional layers and the activations from the fully-connected linear output layer. From the CIFAR-10 model, however, I did not fetch the activations from all model layers as the CIFAR-10 model had a high number of layers. A high number of layers may lead to a long computation time when searching for clusters within the activations from all of these model layers. Therefore, I chose a subset of the layers from the CIFAR-10 model. Nevertheless, this subset contained both,

---

[5] `https://pytorch.org/docs/stable/generated/torch.optim.lr_scheduler.MultiStepLR`

[6] `https://github.com/hongyi-zhang/Fixup`

[7] `https://pytorch.org/docs/stable/generated/torch.optim.SGD.html`

[8] `https://pytorch.org/docs/stable/generated/torch.optim.lr_scheduler.CosineAnnealingLR`

lower model layers (i.e., the layers closer to the input layer) and higher model layers (i.e., the layers closer to the output layer). I used the activations of the first convolutional layer, the third, sixth and ninth ResNet block, the global average pooling layer [88], and the output layer.

After obtaining the activations from a layer, I searched for clusters within these activations using my approach described in Chapter 6.2.1. My approach to finding clusters within the layer activations includes three steps: (1) Transforming the activations of each convolutional layer from cube form to vector form (for linear layers this step is omitted), (2) projecting the resulting activation vectors from a high-dimensional space to a two-dimensional space, and (3) searching for clusters within this resulting two-dimensional space. The baseline configuration $conf_b$ of my clustering approach (described in Chapter 6.2.1) simply flattens the three-dimensional activation tensor obtained from each convolutional layer in order to receive the activations in vector form. This approach[9] was also used by Papernot and McDaniel [113, 114]. The resulting activation vectors are then projected to a two-dimensional space using my approach that combines the linear projection technique PCA (Principal Component Analysis) [116] and the non-linear projection technique UMAP (Uniform Manifold Approximation and Projection) [98]. To project the activations, I first use PCA to project the activations to 50 dimensions and then, I use UMAP to further project the activations to 2 dimension. Finally, I search for clusters within the projected activations using k-Means [95]. Therefore, my baseline configuration $conf_b$ is as follows: Flatten − PCA+UMAP − k-Means.

To test alternatives of my baseline configuration $conf_b$ (Chapter 6.2.1), I simply adjusted that baseline configuration. For instance, exchanging the clustering method of my baseline configuration $conf_b$ would result in one alternative, while exchanging the projection method (but not the clustering method at the same time) would result in another alternative. Hereinafter, I simply refer to an alternative of my baseline configuration $conf_b$ as $conf_a$. To obtain a configuration $conf_a$, I can adjust the baseline configuration $conf_b$ either by changing the method that transforms an activation tensor into an activation vector, by changing the projection method or by changing the clustering method. For transforming an activation tensor into an activation vector, I examined one alternative method. This alternative method uses the pooling operation [110] that is applied to each channel of the three-dimensional activation tensor of each convolutional layer using a (2×2) kernel (for more information about pooling, see Chapter 2.1). As a result, the pooling operation downsamples the activation tensor with respect to its width and height by a factor of 2. After pooling, I flatten the resulting downsampled activation tensor in order to receive the activation vector. This activation vector is smaller compared to the activation vector obtained by my baseline configuration $conf_b$ due to the pooling operation. I aimed to examine whether this slightly lower-dimensional activation vector is a better representation of the data due to the potential translational invariance capabilities of the pooling operation. By using pooling combined with flattening as an alternative method to transform the activation tensor into an activation vector, I obtained

---

[9] `https://github.com/cleverhans-lab/cleverhans/tree/master/cleverhans_v3.1.0/cleverhans/model_zoo/deep_k_nearest_neighbors`

the following alternative configuration $conf_a$ of my clustering approach: Pool+Flatten – PCA+UMAP – k-Means. For projecting the activations, on the other hand, I examined two different alternative projection methods. The first alternative projection method consisted also of a combination of PCA and a non-linear dimensionality reduction technique. Instead of UMAP, however, I used the t-SNE projection technique (t-Distributed Stochastic Neighbor Embedding) [138]. The t-SNE method was also used by Nguyen et. al. [109]. In contrast to Nguyen et. al. [109], however, I could not use the standard t-SNE method [139] as it does not provide a projection model. However, I needed a projection model because I needed to apply it to the images at inference (for more details, see Chapter 6.2.3 or Chapter 6.2.4). Therefore, I used a parametric version [138] of the t-SNE method instead, which does provide a projection model. By using t-SNE as an alternative projection method, I obtained the following alternative configuration $conf_a$ of my clustering approach: Flatten – PCA+t-SNE – k-Means. In addition to t-SNE, I also examined using only PCA as my second alternative projection method. By using only PCA as an alternative projection method, I obtained the following alternative configuration $conf_a$ of my clustering approach: Flatten – PCA – k-Means. Finally, for searching the clusters, I examined the DBScan clustering algorithm [33] as an alternative clustering method. In contrast to k-Means, DBScan does not require setting the number of clusters to search for as a hyperparameter beforehand. By using DBScan as an alternative clustering method, I obtained the following alternative configuration $conf_a$ of my clustering approach: Flatten – UMAP+PCA – DBScan. Additionally, I also examined other alternative clustering methods such as OPTICS [8] or Agglomerative Clustering [1], but they did not work at all on the activation data used in my experiments.

After obtaining the alternative configurations $conf_a$ of my clustering approach, I tested each of those configurations to find out whether it achieves a better clustering result than my baseline configuration $conf_b$. To test an alternative configuration $conf_a$, I first searched for clusters within the activations of each model layer using $conf_a$. After receiving the clusters, I computed the silhouette score [121] of the identified clusters of each model layer in order to obtain an evaluation of the tested alternative configuration $conf_a$. The silhouette score is a cluster quality metric, whose value ranges between -1 and 1 (for more details, see Chapter 3.3). A silhouette score of 1 represents a perfect clustering result, while a silhouette score of -1 represents the worst possible clustering result. According to Chen et. al. [19], the silhouette score works best for evaluating clusters in activation data. Therefore, I already used the silhouette score to find a good value for hyperparameter k of the k-Means algorithm (for more details, see Chapter 6.2.1). After computing the silhouette scores, I obtained one silhouette score from the clusters of each model layer. As a result, I received multiple silhouette scores for the tested alternative configuration $conf_a$ (one for the clustering result of each model layer). However, comparing different alternative configurations is difficult when using multiple scores. Instead, I needed a total score for each alternative configuration $conf_a$. To obtain a total silhouette score, I simply took the median of the silhouette scores over all model layers. The results of my comparison are shown in Table 6.1. It turned out that my baseline configuration $conf_b$ obtained the best clustering approach.

Table 6.1: Comparison of different configurations of my approach to finding clusters within the layer activations with respect to the MNIST [76] dataset, the SVHN dataset [106], and the CIFAR-10 [70] dataset. The obtained results are given as median of the silhouette scores obtained from each selected model layer (a higher score is better).

Configurations		MNIST	SVHN	CIFAR-10
Vector Transform	Flatten	**0.733**	**0.590**	**0.677**
	Pool+Flatten	0.721	0.561	0.657
Dimension. Reduct.	PCA+UMAP	**0.733**	**0.590**	**0.677**
	PCA+t-SNE	0.432	0.369	0.382
	PCA	0.423	0.349	0.378
Clustering	k-Means	**0.733**	**0.590**	**0.677**
	DBScan	0.699	0.535	0.319

6.3.3 Testing on Simple Datasets

In my first experiment, I aimed to find out whether my proposed method LACA (Layer-wise Activation Cluster Analysis) is able to detect out-of-distribution samples at all. Furthermore, I also aimed to compare LACA to the DkNN method (Deep k-Nearest Neighbors) [113, 114] with respect to detection performance and runtime. The setup of my experiment is described in Chapter 6.3.1. For my experiment, I used the three datasets that I already used for the comparison of different clustering approaches in Chapter 6.3.2: The MNIST [76], SVHN [106], and CIFAR-10 [70] dataset. Further information about these datasets can be found in Chapter 6.3.2.

In order to obtain the clusters required by LACA, I first needed to train a Convolutional Neural Network-based (CNN) image classification model using the training images of the respective dataset (e.g., MNIST). To train such a model for each dataset, I used the same training setup that I already used for the comparison of the different clustering approaches in Chapter 6.3.2. Thus, further information about the training setup for each model can be found there. After model training, I fed the training dataset into the resulting model again in order to create the activations at the model layers. Then, I fetched the activations from each model layer that I aimed to use for detecting out-of-distribution samples. In my experiment, I used the activations of the same model layers that I already used to compare the different clustering approaches (Chapter 6.3.2). For MNIST and SVHN, I selected all model layers. For CIFAR-10, on the other hand, I only selected a subset of the model layers in order to decrease the computation time for identifying the clusters. Nevertheless, this selected subset contains both, lower model layers (i.e., the layers closer to the input layer) and higher model layers (i.e., the layers closer to the output layer). Further information about the chosen layers can be found in Chapter 6.3.2. Finally, I searched for clusters within the activations from each model layer. To search for clusters, I used the approach described in Chapter 6.2.1, which turned out to generally obtain the best clustering result, according to my comparison in Chapter 6.3.2 (configuration $conf_b$). I observed that my approach typically finds approximately as many clusters as there are classes within the activations of the higher model layers, while it usually finds a lower number of clusters than there are classes within the activations of the lower model layers. For instance, from the SVHN model (10 classes), I obtained 7 clusters at the first layer and 5 clusters at the second layer, while I obtained 12 clusters from the third layer and 10 clusters from the fourth layer. I made a similar observation for CIFAR-10 (10 classes). Only for MNIST (10 classes), I found approximately 10 clusters at every layer, which corresponds to the number of MNIST classes. Furthermore, the clusters of the first layer of the MNIST model seemed to be well-separated already, according to the obtained silhouette score. This might indicate that MNIST is relatively easy to classify. Finally, after identifying the clusters, I obtained different in-distribution statistics from these clusters. I needed these in-distribution statistics together with the obtained clustering models in order to detect out-of-distribution samples (for more details, see Chapter 6.2.2).

Table 6.2: Parameters of the adversarial attacks FGSM, BIM and PGD that I used for creating the adversarial out-of-distribution datasets with respect to the MNIST [76] dataset, the SVHN [106] dataset, and the CIFAR-10 [70] dataset. I applied each attack to the test dataset of each respective dataset.

Attack Method	MNIST	SVHN	CIFAR-10
FGSM	$\epsilon = 0.25$	$\epsilon = 0.05$	$\epsilon = 0.1$
BIM	$\epsilon = 0.25$, $\alpha = 0.01$	$\epsilon = 0.05$, $\alpha = 0.005$	$\epsilon = 0.1$, $\alpha = 0.05$
PGD	$\epsilon = 0.20$, $\alpha = 2/255$	$\epsilon = 0.04$, $\alpha = 2/255$	$\epsilon = 0.3$, $\alpha = 2/255$

After obtaining the required clustering models and in-distribution statistics from each model layer, I applied LACA to the images of different test datasets ds_{test} in order to test LACA. Each of these test datasets ds_{test} contains images of a specific type that potentially occur at inference. I distinguished between three different types of these test datasets ds_{test}: An in-distribution dataset, a natural out-of-distribution dataset, and different adversarial out-of-distribution datasets. As in-distribution dataset, I used the respective test dataset X^T of each dataset: The MNIST test dataset (9,250 data samples) for the MNIST model, the SVHN test dataset (25,282 data samples) for the SVHN model, and the CIFAR-10 test dataset (9,250 data samples) for the CIFAR-10 model. As the natural out-of-distribution dataset, on the other hand, I used the test dataset $X^{T'}$ of a different dataset (no data samples were removed for a calibration dataset): The KMNIST test dataset (10,000 data samples) for the MNIST model, the CIFAR-10 test dataset (10,000 data samples) for the SVHN model, and the SVHN test dataset (26,032 data samples) for the CIFAR-10 model. Furthermore, to obtain the adversarial out-of-distribution datasets X^T_{adv} for the MNIST model, the SVHN model and the CIFAR-10 model, I applied the following adversarial attacks on the respective test datasets X^T of these models (MNIST: 9,250 data samples, SVHN: 25,282 data samples, CIFAR-10: 9,250 data samples): FGSM (Fast Gradient Sign Method) [41], BIM (Basic Iterative Method) [73], and PGD (Projected Gradient Descent) [96]. The hyperparameters for each attack method are shown in Table 6.2. All attacks were applied to each test image x^T using the Python library *torchattacks* [64]. The classification performances of the models on each test dataset ds_{test} are shown in Table 6.3.

Table 6.3: Classification accuracies of the MNIST [76] model, the SVHN [106] model and the CIFAR-10 [70] model regarding the respective in-distribution dataset (InDist), natural out-of-distribution dataset (NaOOD) (MNIST: KMNIST [21], SVHN: CIFAR-10, CIFAR-10: SVHN), and adversarial out-of-distribution datasets (FGSM, BIM, PGD). A higher accuracy is better.

Model	InDist	NaOOD	FGSM	BIM	PGD
MNIST	0.9905	0.0759	0.0805	0.0004	0.0246
SVHN	0.8994	0.0924	0.0272	0.0079	0.0242
CIFAR-10	0.9242	0.0935	0.1321	0.0084	0.0093

I applied LACA to each test dataset ds_{test} using different values for hyperparameter p_{thres} (values: 0.01, 0.05 and 0.1). As a result, I obtained a credibility score $credib_{laca}$ (Chapter 6.2.4) for each image of the respective test dataset ds_{test}. To obtain the total credibility score for the whole test dataset ds_{test}, I computed the average over the obtained credibility scores of all images of ds_{test}. Furthermore, I computed the difference between the total credibility score of the respective in-distribution dataset and the total credibility score of each out-of-distribution dataset to better evaluate the results of LACA (for more information, see Chapter 6.3.1). A high difference value indicates that LACA is able to confidently distinguish between the in-distribution samples and the respective out-of-distribution samples, while a low difference value indicates that LACA is not able to distinguish between the in-distribution samples and the respective out-of-distribution samples. The resulting difference values are shown in Table 6.5, while the total credibility scores of each test dataset ds_{test} are shown in Table 6.4. Additionally, I also measured the runtimes of LACA on each test dataset ds_{test}, which are shown in Table 6.6. Alternatively, LACA also allows to compute a binary similarity scores $simDet$ (Chapter 6.2.3) for an image. However, as already pointed out in Chapter 6.3.1, I focused in my experiments on evaluating LACA using the credibility score instead of the binary similarity score because the credibility score generally achieves a higher detection performance than the similarity score. Furthermore, the credibility score calculated by LACA is comparable to the credibility score calculated by DkNN, while the similarity score is not comparable to the DkNN credibility score.

Table 6.4: Mean credibility scores obtained by LACA and DkNN for the MNIST [76], SVHN [106] and CIFAR-10 [70] model regarding the respective in-distribution dataset (InDist), natural out-of-distribution dataset (NaOOD) (MNIST: KMNIST [21], SVHN: CIFAR-10, CIFAR-10: SVHN), and adversarial out-of-distribution datasets (FGSM, BIM, PGD). For an in-distribution dataset, a higher score is better, while for an out-of-distribution dataset, a lower score is better.

Dataset	Method	InDist	NaOOD	FGSM	BIM	PGD
MNIST	DkNN	0.799	**0.081**	**0.136**	**0.085**	**0.087**
	LACA ($t = 0.01$)	**0.888**	0.164	0.237	0.209	0.153
	LACA ($t = 0.05$)	0.881	0.125	0.178	0.166	0.123
	LACA ($t = 0.1$)	0.880	0.124	0.173	0.166	0.121
SVHN	DkNN	0.501	**0.146**	0.236	0.309	0.296
	LACA ($t = 0.01$)	**0.702**	0.445	0.574	0.651	0.635
	LACA ($t = 0.05$)	0.634	0.233	0.390	0.500	0.474
	LACA ($t = 0.1$)	0.452	**0.146**	**0.226**	**0.289**	**0.274**
CIFAR-10	DkNN	0.522	0.229	0.168	0.179	0.221
	LACA ($t = 0.01$)	**0.848**	0.565	0.297	0.521	0.571
	LACA ($t = 0.05$)	0.751	0.453	0.176	0.445	0.461
	LACA ($t = 0.1$)	0.358	**0.149**	**0.042**	**0.025**	**0.031**

Table 6.5: Difference values between the mean credibility scores of the respective in-distribution dataset (InDist) and each out-of-distribution dataset obtained by LACA and DkNN with respect to the MNIST [76], SVHN [106] and CIFAR-10 [70] model. I tested a natural out-of-distribution dataset (NaOOD) (MNIST: KMNIST [21], SVHN: CIFAR-10, CIFAR-10: SVHN) and three adversarial out-of-distribution datasets (FGSM, BIM, PGD) for each model. A higher difference value is better.

Dataset	Method	NaOOD	FGSM	BIM	PGD
MNIST	DkNN	0.718	0.663	0.714	0.712
	LACA ($t = 0.01$)	0.724	0.651	0.679	0.735
	LACA ($t = 0.05$)	**0.757**	0.703	**0.716**	0.758
	LACA ($t = 0.1$)	0.756	**0.706**	0.713	**0.759**
SVHN	DkNN	0.355	**0.265**	**0.192**	**0.205**
	LACA ($t = 0.01$)	0.257	0.128	0.051	0.067
	LACA ($t = 0.05$)	**0.401**	0.245	0.134	0.161
	LACA ($t = 0.1$)	0.307	0.227	0.163	0.178
CIFAR-10	DkNN	0.293	0.354	**0.343**	0.301
	LACA ($t = 0.01$)	0.282	0.550	0.327	0.276
	LACA ($t = 0.05$)	**0.298**	**0.575**	0.307	0.290
	LACA ($t = 0.1$)	0.209	0.315	0.333	**0.326**

Table 6.6: Runtimes (in seconds) of the credibility score calculation of LACA and DkNN for the MNIST [76], SVHN [106] and CIFAR-10 [70] model regarding the respective in-distribution dataset (InDist), natural out-of-distribution dataset (NaOOD) (MNIST: KMNIST [21], SVHN: CIFAR-10, CIFAR-10: SVHN), and adversarial out-of-distribution datasets (FGSM, BIM, PGD). A lower runtime value is better.

Dataset	Method	InDist	NaOOD	FGSM	BIM	PGD
MNIST	DkNN	314.7	267.2	308.0	303.0	321.6
	LACA ($t = 0.01$)	16.5	17.9	18.7	21.4	20.6
	LACA ($t = 0.05$)	16.2	**17.2**	**17.7**	**17.8**	**17.5**
	LACA ($t = 0.1$)	**16.0**	17.5	20.0	18.6	18.5
SVHN	DkNN	1171.7	340.9	1109.3	1168.7	1197.5
	LACA ($t = 0.01$)	57.1	28.3	60.2	53.5	**46.2**
	LACA ($t = 0.05$)	**38.6**	**21.3**	**43.4**	**42.8**	50.1
	LACA ($t = 0.1$)	42.7	23.5	59.7	61.6	53.0
CIFAR-10	DkNN	684.1	1620.7	710.6	700.2	684.5
	LACA ($t = 0.01$)	47.4	137.7	52.7	53.3	55.0
	LACA ($t = 0.05$)	**42.3**	**136.0**	**49.5**	**49.8**	**51.1**
	LACA ($t = 0.1$)	44.4	142.7	55.6	55.5	51.4

To be able to compare LACA to the DkNN method, I also applied DkNN to each test dataset ds_{test}. As a result, I obtained a credibility score $credib_{dknn}$ for each image of the respective test dataset ds_{test}. To obtain the total DkNN credibility score for the whole test dataset ds_{test}, I computed the average over the obtained DkNN credibility scores of all images of ds_{test}. Furthermore, I computed the difference between the total DkNN credibility score of the respective in-distribution dataset and the total DkNN credibility score of each out-of-distribution dataset again. The resulting difference values are shown in Table 6.5, while the total DkNN credibility scores of each test dataset ds_{test} are shown in Table 6.4. As shown in Table 6.5, LACA slightly outperforms the DkNN method. Additionally, I also measured the runtimes of the DkNN method on each test dataset ds_{test}, which are shown in Table 6.6. As shown in Table 6.6, LACA is significantly faster than the DkNN method on all test datasets ds_{test}.

6.3.4 Omitting Lower Network Layers

In my experiment in Chapter 6.3.3, I used lower model layers (i.e., the layers closer to the input layer) as well as higher model layers (i.e., the layers closer to the output layer) in order to compute the credibility scores using LACA. To compute the credibility score $credib_{laca}(x^I)$ of an image x^I at inference using LACA, I need to find at each model layer l the cluster $h^l_{x^I}$ into which this image falls in feature space of the layer (for more information, see Chapter 6.2.4). The cluster $h^l_{x^I}$ at a certain model layer l can be identified by first applying the projection model p^l followed by applying the clustering model g^l to the feature representation $a^l(x^I)$ of image x^I at model layer l. Both models,

r^l and g^l, have been obtained from the activations of the training dataset at model layer l (for more information, see Chapter 6.2.1). However, applying the projection model to a lower layer usually requires a significantly higher runtime than applying the projection model to a higher layer. This is caused by the different number of activations that are obtained from a lower model layer compared to a higher model layer. The lower model layers typically contain a significantly higher number of activations than the higher model layers. From layer to layer, the amount of activations keeps decreasing. Therefore, the feature representations of the lower model layers are significantly higher-dimensional than the feature representations of the higher model layers. As a result, the projection model of a lower model layer needs to project the activations from a significantly higher number of dimensions down to 2 dimensions than the projection model of a higher layer. Therefore, projecting activations of a lower layer requires significantly more time than projecting activations of a higher layer.

As a result, I conducted another experiment to examine whether some of the lower model layers can be omitted for computing the credibility score $credib_{laca}$ in order to further decrease the total runtime of LACA at inference without significantly reducing its detection performance. To conduct this experiment, I used the same setup as in my experiment in Chapter 6.3.3 except for the list of selected layers. For MNIST and SVHN, I omitted the first layer from the list of selected layers, which is the first convolutional layer of the model. For CIFAR-10, on the other hand, I omitted the first two layers from the list of selected layers, which are the first convolutional layer and the third ResNet (Residual Neural Network) block of the model. After reducing the list of selected model layers, I computed all credibility scores of LACA and DkNN (Deep k-Nearest Neighbors) [113, 114] again, using this reduced list of selected model layers. The total credibility scores of LACA and DkNN are shown in Table 6.7. To be able to better evaluate LACA and DkNN, I also computed the difference value of the total credibility score of the respective in-distribution dataset and the total credibility score of each corresponding out-of-distribution dataset for LACA and DkNN again. The resulting difference values are shown in Table 6.8. Additionally, I also measured the runtimes of LACA and DkNN, which are shown in Table 6.9. The column named *Layers* of all three tables specifies whether I removed no layer, the first layer or the first two layers from the layers that I used in my experiment in Chapter 6.3.3. As shown in Table 6.8, removing the first layer of the MNIST and SVHN model significantly reduced the detection performance. Removing the first or the first two layers of the CIFAR-10 model, however, slightly increased the detection performance at least on the adversarial out-of-distribution datasets. At the same time, omitting these lower model layers significantly decreased the runtimes, as shown in Table 6.9. I also examined to remove some of the higher model layers only. Removing only higher model layers is not expected to significantly decrease the runtime of LACA at inference because the higher model layers do not contain a high number of activations. However, I aimed to test whether these higher model layers are crucial for the out-of-distribution detection. It turned out that removing higher model layers significantly decreased the detection performance of LACA and also the detection performance of DkNN.

Table 6.7: Mean credibility scores regarding a varying number of layers. Column *Layers* specifies the number of previously chosen layers to be omitted, starting from the lower layers. For an in-distribution dataset (InDist), a higher score is better, while for the out-of-distribution datasets (NaOOD, FGSM, BIM, PGD), a lower score is better.

Dataset	Method	Layers	InDist	NaOOD	FGSM	BIM	PGD
MNIST	DkNN	0	0.799	**0.081**	**0.136**	**0.085**	**0.087**
		1	0.876	0.124	0.201	0.171	0.132
	LACA	0	0.881	0.125	0.178	0.166	0.123
		1	**0.963**	0.396	0.449	0.425	0.288
SVHN	DkNN	0	0.501	**0.146**	**0.236**	**0.309**	**0.296**
		1	0.535	0.155	0.245	0.349	0.325
	LACA	0	0.634	0.233	0.390	0.500	0.474
		1	**0.705**	0.280	0.462	0.607	0.568
CIFAR-10	DkNN	0	0.522	0.229	0.168	0.179	0.221
		1	0.519	0.176	0.173	**0.171**	**0.200**
		2	0.518	**0.134**	**0.160**	0.174	0.203
	LACA	0	0.751	0.453	0.176	0.445	0.461
		1	0.839	0.480	0.205	0.481	0.498
		2	**0.842**	0.481	0.206	0.486	0.502

Table 6.8: Difference values between each in-distribution dataset (InDist) and its respective out-of-distribution datasets (NaOOD, FGSM, BIM, PGD) regarding a varying number of layers. Column *Layers* specifies the number of previously chosen layers to be omitted, starting from the lower layers. A higher difference value is better.

Dataset	Method	Layers	NaOOD	FGSM	BIM	PGD
MNIST	DkNN	0	0.718	0.663	0.714	0.712
		1	0.752	0.675	0.705	0.744
	LACA	0	**0.757**	**0.703**	**0.716**	**0.758**
		1	0.567	0.514	0.538	0.675
SVHN	DkNN	0	0.355	0.265	**0.192**	0.205
		1	0.381	**0.291**	0.187	**0.211**
	LACA	0	0.401	0.245	0.134	0.161
		1	**0.425**	0.243	0.098	0.136
CIFAR-10	DkNN	0	0.293	0.354	0.343	0.301
		1	0.343	0.346	0.348	0.319
		2	**0.384**	0.358	0.345	0.315
	LACA	0	0.298	0.575	0.307	0.290
		1	0.359	0.634	**0.358**	**0.341**
		2	0.361	**0.636**	0.356	0.340

Table 6.9: Runtimes (in seconds) of the credibility score calculation with respect to the MNIST [76], SVHN [106] and CIFAR-10 [70] model regarding a varying number of layers. Column *Layers* specifies the number of previously chosen layers to be omitted, starting from the lower layers. I tested for each model an in-distribution dataset, a natural out-of-distribution dataset (NaOOD) (MNIST: KMNIST [21], SVHN: CIFAR-10, CIFAR-10: SVHN) and three adversarial out-of-distribution datasets (FGSM, BIM, PGD). A lower runtime value is better.

Dataset	Method	Layers	InDist	NaOOD	FGSM	BIM	PGD
MNIST	DkNN	0	314.7	267.2	308.0	303.0	321.6
		1	90.1	63.0	78.3	75.8	82.0
	LACA	0	16.2	17.2	17.7	17.8	17.5
		1	**12.0**	**12.3**	**12.5**	**12.9**	**13.3**
SVHN	DkNN	0	1171.7	340.9	1109.3	1168.7	1197.5
		1	348.7	116.2	319.7	344.2	334.8
	LACA	0	38.6	21.3	43.4	42.8	50.1
		1	**26.8**	**16.6**	**28.0**	**28.1**	**28.5**
CIFAR-10	DkNN	0	684.1	1620.7	710.6	700.2	684.5
		1	398.9	1107.4	465.5	439.8	445.9
		2	172.5	447.3	182.1	182.2	182.3
	LACA	0	42.3	136.0	49.5	49.8	51.1
		1	**39.0**	**100.2**	**44.8**	**43.6**	**44.6**

6.3.5 Testing on Complex Datasets

In my experiment in Chapter 6.3.3, I tested LACA in comparison to DkNN (Deep k-Nearest Neighbors) [113, 114] only on three simple datasets (MNIST [76], SVHN [106], CIFAR-10 [70]). Furthermore, Papernot and McDaniel [113, 114] also only used simple datasets in their experiments in order to test DkNN (MNIST, SVHN, GTSRB [52]). Therefore, I aimed to test LACA in comparison to DkNN on more complex datasets in a third experiment. The general setup of my experiment is described in Chapter 6.3.1. For my experiment, I used the following two complex datasets: The Imagenette[10] and the Imagewoof[11] dataset. Both datasets contain color images of 10 classes, which are taken from the ImageNet [25] dataset (1000 classes). The images of both datasets, Imagenette and Imagewoof, show a natural image object of one of the 10 classes in front of a natural image background. However, the classes of Imagenette are different from the classes of Imagewoof. The 10 classes of Imagenette represent 10 different objects such as *cassette player*, *parachute*, or *golf ball*. The 10 classes of Imagewoof, on the other hand, represent 10 different dog breeds. Therefore, Imagewoof is more difficult to classify than Imagenette because different dog breeds are significantly harder to distinguish than general objects that are visually very different from each other, such as a *parachute* and

---

[10] `https://github.com/fastai/imagenette#imagenette-1`

[11] `https://github.com/fastai/imagenette#imagewoof`

a *golf ball*. Furthermore, in contrast to the images of the simple datasets used in my experiment in Chapter 6.3.3, the images of the Imagenette dataset and the Imagewoof dataset are not of a unique image size. However, LACA and DkNN require all images of a dataset to be of the same size (for more details, see Chapter 6.2). Therefore, I needed to resize all images to a fixed image size before running any tests. I chose an image size of $128 \times 128 \times 3$. For each of those resized images, I aimed to compute the credibility score $credib_{laca}$ using LACA and the credibility score $credib_{dknn}$ using DkNN. However, before computing any credibility scores for these two datasets, I first needed to train a model for each dataset.

In order to train the Imagenette model and the Imagewoof model, I used a similar training setup for both datasets. I chose a standard 18-layer ResNet (Residual Network) model architecture [46]. Then, I initialized the weights of each model layer using the Kaiming Uniform[12] initialization method [45]. Finally, I trained each model using the Adam optimizer [65] and a one-cycle learning rate schedule [132] with a maximum learning rate of 0.006. However, as Imagewoof is more complex than Imagenette, I had to train the Imagewoof model for more training epochs than the Imagenette model. In order to obtain a sufficient classification performance of each model, I trained the Imagenette model for 40 training epochs and the Imagewoof model for 60 training epochs. I used the whole official training dataset for the respective model training (Imagenette: 9,469 data samples, Imagewoof: 9,025 data samples). Additionally, I also used variations of the training images that I obtained through different data augmentation techniques. These techniques included randomly flipping training images horizontally and randomly cropping training images. After model training, I tested the resulting model on the respective test dataset (Imagenette: 3,175 data samples, Imagewoof: 3,179 data samples), which consisted of the images of the official test dataset of the respective dataset (Imagenette: 3,925 data samples, Imagewoof: 3,929 data samples) except for the 750 images that I used for the calibration dataset. The obtained Imagenette model achieved an accuracy of 86.17% on its respective test dataset, while the obtained Imagewoof model achieved an accuracy of 75.02% on its respective test dataset.

After model training, I fed the training images into the resulting model again in order to create the activations at the model layers. Then, I fetched the activations from each model layer that I chose to use for detecting out-of-distribution samples. Within the activations of each selected model layer, I aimed to search for the clusters that are required by LACA. From both models, Imagenette and Imagewoof, I did not fetch the activations from all model layers as both models contained a high number of layers. A high number of layers may lead to a long computation time when searching for clusters within the activations from all of these model layers. Therefore, I chose a subset of the layers from each model. Nevertheless, this subset contained both, lower model layers (i.e., the layers closer to the input layer) and higher model layers (i.e., the layers closer to the output layer). From both models, I used the activations of the first convolutional layer, the activations of the subsequent maxpooling layer, the activations from the 8 ResNet blocks, and the activations of the global average pooling layer. Finally, I searched for

---

[12] `https://pytorch.org/docs/stable/nn.init#torch.nn.init.kaiming_uniform_`

clusters within the activations from each selected model layer. To search for clusters, I used the same approach as in Chapter 6.3.3. As I already observed in my experiment in Chapter 6.3.3, my approach typically finds approximately as many clusters as there are classes within the activations of the higher model layers, while it usually finds a lower number of clusters than there are classes within the activations of the lower model layers. For both datasets, Imagenette and Imagewoof (both have 10 classes), I found between 5 and 6 clusters at the lower model layers, while I found around 10 clusters at the higher model layers. Finally, after identifying the clusters, I obtained different in-distribution statistics from these clusters. I needed these in-distribution statistics together with the obtained clustering models in order to detect out-of-distribution samples (for more details, see Chapter 6.2.2).

Table 6.10: Parameters of the adversarial attacks FGSM, BIM and PGD that I used for creating the adversarial out-of-distribution datasets with respect to the Imagenette dataset and the Imagewoof dataset. I applied each attack to the test dataset of each respective dataset.

Attack Method	Imagenette	Imagewoof
FGSM	$\epsilon = 0.25$	$\epsilon = 0.3$
BIM	$\epsilon = 0.25$, $\alpha = 0.01$	$\epsilon = 0.1$, $\alpha = 0.005$
PGD	$\epsilon = 0.10$, $\alpha = 2/255$	$\epsilon = 0.1$, $\alpha = 2/255$

After obtaining the required clustering models and in-distribution statistics from each model layer, I applied LACA to the images of different test datasets ds_{test} in order to evaluate LACA. Each of these test datasets ds_{test} contains images of a specific type that potentially occur at inference. I distinguish between three different types of these test datasets ds_{test}: An in-distribution dataset, a natural out-of-distribution dataset, and different adversarial out-of-distribution datasets. As in-distribution dataset, I used the respective test images X^T of both datasets: The Imagenette test images (3,175 data samples) for the Imagenette model and the Imagewoof test images (3,179 data samples) for the Imagewoof model. As the natural out-of-distribution dataset, on the other hand, I used the test images $X^{T'}$ of the other dataset (no data samples were removed for a calibration dataset): The Imagewoof test images (3,929 data samples) for the Imagenette model and the Imagenette test images (3,925 data samples) for the Imagewoof model. Furthermore, to obtain the adversarial out-of-distribution samples X^T_{adv} for the Imagenette model and the Imagewoof model, I applied the following adversarial attacks on the respective test dataset X^T (Imagenette: 3,175 data samples, Imagewoof: 3,179 data samples): FGSM (Fast Gradient Sign Method) [41], BIM (Basic Iterative Method) [73], and PGD (Projected Gradient Descent) [96]. The hyperparameters for each attack method are shown in Table 6.10. All attacks were applied to each test image x^T using the Python library *torchattacks* [64]. The classification performances of the models on each test dataset ds_{test} are shown in Table 6.11.

Table 6.11: Classification accuracies of the Imagenette and Imagewoof model regarding the respective in-distribution dataset (InDist), natural out-of-distribution dataset (NaOOD) (Imagenette: Imagewoof, Imagewoof: Imagenette), and adversarial out-of-distribution datasets (FGSM, BIM, PGD). A higher accuracy is better.

Model	InDist	NaOOD	FGSM	BIM	PGD
Imagenette	0.8617	0.0926	0.1231	0.0394	0.0545
Imagewoof	0.7502	0.1248	0.1217	0.0239	0.0009

I applied LACA to each test dataset ds_{test} using different values for hyperparameter p_{thres} (values: 0.01, 0.05 and 0.1). As a result, I obtained a credibility score $credib_{laca}$ (Chapter 6.2.4) for each image of the respective test dataset ds_{test}. To obtain the total credibility score for the whole test dataset ds_{test}, I computed the average over the obtained credibility scores of all images of ds_{test}. Furthermore, I computed the difference between the total credibility score of the in-distribution dataset and the total credibility score of each out-of-distribution dataset to better evaluate the results of LACA, as I already did in my experiment using the simple datasets (for more information, see Chapter 6.3.3). A high difference value indicates that LACA is able to confidently distinguish between the in-distribution samples and the respective out-of-distribution samples, while a low difference value indicates that LACA is not able to distinguish between the in-distribution samples and the respective out-of-distribution samples. The resulting difference values are shown in Table 6.13, while the total credibility scores of each test dataset ds_{test} are shown in Table 6.12. Additionally, I also measured the runtimes of LACA on each test dataset ds_{test}, which are shown in Table 6.14.

To be able to compare LACA to the DkNN method, I also applied DkNN to each test dataset ds_{test}. As a result, I obtained a credibility score $credib_{dknn}$ for each image of the respective test dataset ds_{test}. To obtain the total DkNN credibility score for the whole test dataset ds_{test}, I computed the average over the obtained DkNN credibility scores of all images of ds_{test}. Furthermore, I computed the difference between the total DkNN credibility score of the in-distribution dataset and the total DkNN credibility score of each out-of-distribution dataset again. The resulting difference values are shown in Table 6.13, while the total DkNN credibility scores of each test dataset ds_{test} are shown in Table 6.11. As shown in Table 6.13, LACA is able to detect out-of-distribution samples also on these two complex datasets, Imagenette and Imagewoof. However, the detection performance is decreased compared to the detection performance on the simple datasets in my experiment in Chapter 6.3.3. Moreover, a value of 0.1 for hyperparameter p_{thresh} was already too high for Imagewoof in order to obtain a useful credibility score. However, DkNN is not able to detect out-of-distribution samples with respect to these two complex datasets at all. The calculated credibility scores by DkNN for the in-distribution dataset and the out-of-distribution datasets are approximately equally high. As a result, in contrast to LACA, DkNN is not able to distinguish between in-distribution samples and out-of-distribution samples. Furthermore, as shown in Table 6.14, LACA obtained significantly lower runtimes than DkNN on all test datasets ds_{test}.

Table 6.12: Mean credibility scores obtained by LACA and DkNN for the Imagenette model and the Imagewoof model regarding the respective in-distribution dataset (InDist), natural out-of-distribution dataset (NaOOD) (Imagenette: Imagewoof, Imagewoof: Imagenette), and adversarial out-of-distribution datasets (FGSM, BIM, PGD). For an in-distribution dataset, a higher score is better, while for an out-of-distribution dataset, a lower score is better.

Dataset	Method	InDist	NaOOD	FGSM	BIM	PGD
Imagenette	DkNN	**0.757**	0.746	0.749	0.763	0.749
	LACA ($t = 0.01$)	0.610	0.429	0.293	0.364	0.364
	LACA ($t = 0.05$)	0.476	0.295	0.218	0.204	0.193
	LACA ($t = 0.1$)	0.320	**0.167**	**0.014**	**0.109**	**0.110**
Imagewoof	DkNN	**0.901**	0.905	0.924	0.912	0.911
	LACA ($t = 0.01$)	0.676	0.415	0.353	0.586	0.582
	LACA ($t = 0.05$)	0.578	**0.235**	**0.134**	**0.435**	**0.477**
	LACA ($t = 0.1$)	-	-	-	-	-

Table 6.13: Difference values between the mean credibility scores of the respective in-distribution dataset (InDist) and each out-of-distribution dataset obtained by LACA and DkNN with respect to the Imagenette and Imagewoof model. I tested a natural out-of-distribution dataset (NaOOD) (Imagenette: Imagewoof, Imagewoof: Imagenette) and three adversarial out-of-distribution datasets (FGSM, BIM, PGD) for each model. A higher difference value is better.

Dataset	Method	NaOOD	FGSM	BIM	PGD
Imagenette	DkNN	0.011	0.007	-0.006	0.008
	LACA ($t = 0.01$)	**0.181**	**0.317**	0.246	0.246
	LACA ($t = 0.05$)	**0.181**	0.258	**0.272**	**0.283**
	LACA ($t = 0.1$)	0.154	0.307	0.211	0.210
Imagewoof	DkNN	-0.004	-0.023	-0.011	-0.009
	LACA ($t = 0.01$)	0.261	0.323	0.090	0.094
	LACA ($t = 0.05$)	**0.343**	**0.444**	**0.143**	**0.101**
	LACA ($t = 0.1$)	-	-	-	-

Table 6.14: Runtimes (in seconds) of the credibility score calculation of LACA and DkNN for the Imagenette model and the Imagewoof model regarding the respective in-distribution dataset (InDist), natural out-of-distribution dataset (NaOOD) (MNIST: KMNIST [21], SVHN: CIFAR-10, CIFAR-10: SVHN), and adversarial out-of-distribution datasets (FGSM, BIM, PGD). A lower runtime value is better.

Dataset	Method	InDist	NaOOD	FGSM	BIM	PGD
Imagenette	DkNN	2613.5	2902.8	2193.6	2336.7	2714.0
	LACA ($t = 0.01$)	196.2	281.2	207.8	**208.1**	212.5
	LACA ($t = 0.05$)	**188.9**	**277.3**	**206.7**	216.1	**194.2**
	LACA ($t = 0.1$)	193.8	278.9	211.5	212.5	207.2
Imagewoof	DkNN	1955.5	2368.2	1960.1	1989.7	1970.2
	LACA ($t = 0.01$)	**212.1**	**285.0**	219.3	212.2	199.6
	LACA ($t = 0.05$)	213.6	286.5	**200.2**	**192.1**	**197.8**
	LACA ($t = 0.1$)	-	-	-	-	-

6.4 Discussion

Papernot and McDaniel [113, 114] proposed a method named DkNN (Deep k-Nearest Neighbors) to detect out-of-distribution samples at inference with respect to a Convolutional Neural Network-based (CNN) image classification model. In order to check if an image is an out-of-distribution sample, DkNN runs a k-nearest neighbor classification at each model layer. By running the k-nearest neighbor classification, DkNN finds out which training images of the model are close to the image in question in feature space of a layer. If the nearby training images are of the same class at every model layer, DkNN concludes that the image is an in-distribution sample. If the nearby training images are not of the same class at every model layer, however, DkNN concludes that the image is most likely an out-of-distribution sample.

However, due to the k-nearest neighbor classification, the DkNN method has a high runtime and memory consumption at inference. To address these issues, I proposed a method named LACA (Layer-wise Activation Cluster Analysis), which is based on clustering rather than a k-nearest neighbor classification. By using clustering, I showed that LACA has a lower memory consumption at inference than the DkNN method (Chapter 6.2.2). To evaluate runtime and detection performance of LACA, I conducted several experiments. My first research goal was to examine whether LACA is able to detect out-of-distribution samples at all. In Chapter 6.3.3, I showed using different simple datasets that LACA is indeed able to detect out-of-distribution samples. Therefore, my second research goal was to examine how LACA performs in comparison to the DkNN method. In Chapter 6.3.3, I also showed that LACA is able to achieve a similar detection performance as the DkNN method on different simple datasets. This suggests that clustering is suitable to find nearby training images of the image in question (in feature space of a layer) in order to check if the image is an out-of-distribution sample. However, the clustering approach (Chapter 6.2.1) and the hyperparameter p_{thres} (Chapter 6.2.2) of LACA are crucial to its detection performance. In Chapter 6.3.2, I showed that a combination of PCA and UMAP along with the k-Means clustering algorithm is best suited to identify the clusters. A good value for the hyperparameter p_{thres}, on the other hand, seems to be 0.05 in most cases, as shown by my experiment in Chapter 6.3.3. However, LACA not only achieves a similar detection performance as DkNN. LACA also has a significantly lower runtime at inference compared to DkNN. A fast detection of out-of-distribution samples is important in practice, especially for safety-critical applications running in real-time, such as driving assistance systems.

To further reduce the runtime of LACA, I also examined whether it is possible to omit some of the lower model layers (i.e., the layers closer to the input layer) for detecting out-of-distribution samples without significantly reducing the detection performance of LACA. The lower model layers are the most expensive ones for LACA to process in terms of runtime. In my experiment in Chapter 6.3.4, I showed that it is indeed possible to omit some of the lower model layers to further reduce the runtime of LACA without significantly reducing its detection performance, at least for the detection of adversarial out-of-distribution samples. However, lower model layers can only be omitted if the model has a high number of layers in total. Higher model layers (i.e., the layers closer to the output layer), on the other hand, cannot be omitted at all. I assume this is caused by the typical behavior of an adversarial out-of-distribution sample when fed into a Convolutional Neural Network-based image classification model. I have observed visually that, in feature space of a lower model layer, an adversarial out-of-distribution sample is still close to training images (i.e., in-distribution samples) of the same class as the class of the in-distribution sample from which the adversarial out-of-distribution sample was created. In feature space of higher model layers, on the other hand, adversarial out-of-distribution samples are suddenly close to training images of a different class (to visualize images in feature space I simply project them to 2 dimensions to be able to plot them). Thus, if there are many layers, not all lower model layers might be needed for detecting an adversarial out-of-distribution sample using LACA, as long as the model layers are kept where the adversarial out-of-distribution sample changes its position in feature space towards training images of a different class.

Finally, I also tested LACA in comparison to the DkNN method on more complex datasets. I showed in my experiment in Chapter 6.3.5 that LACA again had a significantly lower runtime than DkNN. Furthermore, LACA was also able to detect out-of-distribution samples with respect to these more complex datasets, while the DkNN method failed to detect any out-of-distribution samples at all. However, the detection performance of LACA was lower compared to its detection performance on the simple datasets (Chapter 6.3.3). Nevertheless, I have shown that layer activations of a Convolutional Neural Network-based image classification model can be exploited by finding clusters within these layer activations in order to detect when a model fails at inference due to the occurrence of out-of-distribution samples.

7 Conclusion and Future Perspective

Convolutional Neural Network-based (CNN) image classification models are the current state-of-the-art for solving image classification problems [46, 71]. However, training and using such a model to solve a specific image classification problem presents several challenges. In order to train the model, we need to find good values for the hyperparameters that need to be set for model training, such as the learning rate hyperparameter or the initial model weights. Finding good values for these training hyperparameters, however, is usually a non-trivial process, as discussed in Chapter 4. Furthermore, another issue with respect to model training is that the datasets that should be used for the training are often class-imbalanced in practice. A class-imbalanced dataset contains images that belong to a specific set of classes, but the images are not uniformly distributed among the classes. This class imbalance usually has a negative impact on model training, as discussed in Chapter 5. However, not only is it challenging to train a Convolutional Neural Network-based image classification model, but also to use the model after model training. After training, the model might be applied to images that were drawn from a data distribution that is different from the data distribution the training data was drawn from. These images are typically referred to as out-of-distribution samples. Unfortunately, Convolutional Neural Network-based models typically fail to predict the correct class for out-of-distribution samples without warning, as discussed in Chapter 6. Thus, an out-of-distribution sample poses a serious threat when using such a model for safety-critical applications (e.g., driving assistance systems, medical diagnosis systems).

The goal of my work was to examine whether the information from the layers of a Convolutional Neural Network-based image classification model (pixels and activations) can be used to address the aforementioned challenges and thereby improve the classification process, as pointed out in Chapter 1. As a result, I suggested a method for initializing the model weights based on image patches (Chapter 4.2), a method for balancing a class-imbalanced dataset based on layer activations (Chapter 5.2), and a method for detecting out-of-distribution samples, which is also based on layer activations (Chapter 6.2). All of these methods search for clusters within the information obtained from the model layers (pixels and activations). The identified clusters are then exploited for the proposed methods in order to improve the classification process. To test the proposed methods, I conducted extensive experiments using different datasets. The experiments have shown that the layer information (pixels and activations) can indeed be used to address the aforementioned challenges and thereby improve the classification process in various ways. In Chapter 4, it was shown that image patches extracted from the training images (input layer information) can be used to initialize the model weights. Moreover, using these image patches to initialize the model weights made the choice of the learning rate hyperparameter for model training less critical in my experiments.

When using a suboptimal learning rate value, the model initialized with my proposed method achieved a superior classification performance compared to the models initialized with a state-of-the-art weight initialization method. Suitable image patches for the proposed patch-based weight initialization method are identified by using clustering in image space of a set of candidate patches that are extracted from the training images of the model. However, it has also been shown that not only the information obtained from the input layer of the model (i.e., the image pixels) can be exploited to improve the classification process but also the information from other model layers containing the intermediate representations of the input images in the form of the activations (hidden layer information). In Chapter 5, it was shown that the activations of a higher model layer (i.e., a layer closer to the output layer) can be used to balance a class-imbalanced dataset. To balance a dataset, my proposed method exploits cluster information within the activations of such a layer. By balancing the dataset with my proposed method, it was possible to improve the classification performance of the resulting model in my experiments compared to a model that was trained using the original class-imbalanced dataset and various models that were trained using a state-of-the-art method to address the class imbalance problem. Furthermore, in Chapter 6, it was also shown that layer activations can be used not only to improve model training but also model inference by detecting out-of-distribution samples. To detect out-of-distribution samples, my proposed method exploits cluster information within the activations from multiple layers. Detecting out-of-distribution samples is especially important when using a Convolutional Neural Network-based image classification model for safety-critical applications (e.g., driving assistance systems, medical diagnosis systems). However, my proposed method could not only detect out-of-distribution samples in my experiments but could also detect them quickly, which is critical for applications running in real-time, such as a driving assistance system.

In future work, my suggested methods based on layer information should be further improved or even tested for other domains. It could be investigated, for instance, whether it is possible to combine the image patch-based weight initialization method from Chapter 4.2 with a state-of-the-art weight initialization method in order to improve model training with an optimal learning rate value as well. The proposed method for balancing a class-imbalanced dataset from Chapter 5.2, on the other hand, can be applied to other domains such as medical images. Medical image datasets are frequently imbalanced due to the gender-based imbalance of medical data [74] or the rare occurrence of certain diseases [119]. Furthermore, it could be investigated whether the proposed approach to balancing a dataset using cluster information within the activations of a higher model layer can also be used for detecting noisy training data samples. Noisy training data samples are training data samples that have been mislabeled during data gathering. These mislabeled training data samples have a negative impact on model training as well [4, 143]. Finally, the detection performance of the suggested out-of-distribution detection method from Chapter 6.2 needs to be further improved for more complex datasets.

Bibliography

[1] Marcel Rudolf Ackermann, Johannes Blömer, Daniel Kuntze, and Christian Sohler. Analysis of Agglomerative Clustering. *Algorithmica*, 69:184–215, 2014.

[2] Charu C. Aggarwal. *Neural Networks and Deep Learning: A Textbook*. Springer, Cham, Switzerland, 2018.

[3] Astha Agrawal, Herna L. Viktor, and Eric Paquet. SCUT: Multi-class Imbalanced Data Classification using SMOTE and Cluster-based Undersampling. In *2015 7th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management (IC3K)*, volume 1, pages 226–234, Lisbon, Portugal, 2015. IEEE.

[4] Görkem Algan and Ilkay Ulusoy. Image Classification with Deep Learning in the Presence of Noisy Labels: A Survey. *Knowledge-Based Systems*, 215:106771, 2021.

[5] Shin Ando and Chun Yuan Huang. Deep Over-sampling Framework for Classifying Imbalanced Data. In Michelangelo Ceci, Jaakko Hollmén, Ljupčo Todorovski, Celine Vens, and Sašo Džeroski, editors, *Machine Learning and Knowledge Discovery in Databases, ECML PKDD 2017*, Lecture Notes in Computer Science, pages 770–785, Cham, Switzerland, 2017. Springer.

[6] Alexandr Andoni and Piotr Indyk. Near-Optimal Hashing Algorithms for Approximate Nearest Neighbor in High Dimensions. In *2006 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS '06)*, pages 459–468, Berkeley, CA, USA, 2006. IEEE.

[7] Plamen Angelov and Eduardo Soares. Towards Explainable Deep Neural Networks (xDNN). *Neural Networks*, 130:185–194, 2020.

[8] Mihael Ankerst, Markus M. Breunig, Hans-Peter Kriegel, and Jörg Sander. OPTICS: Ordering Points to Identify the Clustering Structure. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, SIGMOD 1999, page 49–60, New York, NY, USA, 1999. ACM.

[9] Randall Balestriero, Mark Ibrahim, Vlad Sobal, Ari Morcos, Shashank Shekhar, Tom Goldstein, Florian Bordes, Adrien Bardes, Gregoire Mialon, Yuandong Tian, Avi Schwarzschild, Andrew Gordon Wilson, Jonas Geiping, Quentin Garrido, Pierre Fernandez, Amir Bar, Hamed Pirsiavash, Yann LeCun, and Micah Goldblum. A Cookbook of Self-Supervised Learning. *arXiv preprint* `arXiv:2304.12210`, 2023.

[10] Yoshua Bengio, Pascal Lamblin, Dan Popovici, and Hugo Larochelle. Greedy Layer-Wise Training of Deep Networks. In Bernhard Schölkopf, John Platt, and Thomas Hoffmann, editors, *Advances in Neural Information Processing Systems 19: Proceedings of the 2006 Conference*, volume 19 of *NIPS 2006*, pages 153–160, Cambridge, MA, USA, 2007. MIT Press.

[11] Battista Biggio, Igino Corona, Davide Maiorca, Blaine Nelson, Nedim Šrndić, Pavel Laskov, Giorgio Giacinto, and Fabio Roli. Evasion Attacks against Machine Learning at Test Time. In Hendrik Blockeel, Kristian Kersting, Siegfried Nijssen, and Filip Železný, editors, *Machine Learning and Knowledge Discovery in Databases, ECML PKDD 2013*, Lecture Notes in Computer Science, pages 387–402, Berlin - Heidelberg, Germany, 2013. Springer.

[12] Tom B. Brown, Dandelion Mané, Aurko Roy, Martín Abadi, and Justin Gilmer. Adversarial Patch. *arXiv preprint* `arXiv:1712.09665`, 2018.

[13] Mateusz Buda, Atsuto Maki, and Maciej A. Mazurowski. A Systematic Study of the Class Imbalance Problem in Convolutional Neural Networks. *Neural Networks*, 106:249–259, 2018.

[14] Roberto Caldelli, Rudy Becarelli, Fabio Carrara, Fabrizio Falchi, and Giuseppe Amato. Exploiting CNN Layer Activations to Improve Adversarial Image Classification. In *2019 IEEE International Conference on Image Processing (ICIP)*, pages 2289–2293, Taipei, Taiwan, 2019. IEEE.

[15] Kaidi Cao, Colin Wei, Adrien Gaidon, Nikos Arechiga, and Tengyu Ma. Learning Imbalanced Datasets with Label-Distribution-Aware Margin Loss. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Proceedings of the 2019 Conference*, volume 32 of *NeurIPS 2019*, pages 1565–1576, Red Hook, NY, USA, 2019. Curran Associates Inc.

[16] Fabio Carrara, Fabrizio Falchi, Roberto Caldelli, Giuseppe Amato, and Rudy Becarelli. Adversarial Image Detection in Deep Neural Networks. *Multimedia Tools and Applications*, 78(3):2815–2835, 2019.

[17] Ivan Castillo Camacho and Kai Wang. A Simple and Effective Initialization of CNN for Forensics of Image Processing Operations. In *Proceedings of the ACM Workshop on Information Hiding and Multimedia Security*, IH&MMSec 2019, pages 107–112, New York, NY, USA, 2019. ACM.

[18] Nitesh V. Chawla, Kevin W. Bowyer, Lawrence O. Hall, and W. Philip Kegelmeyer. SMOTE: Synthetic Minority Over-Sampling Technique. *Journal of Artificial Intelligence Research*, 16(1):321–357, 2002.

[19] Bryant Chen, Wilka Carvalho, Nathalie Baracaldo, Heiko Ludwig, Benjamin Edwards, Taesung Lee, Ian Molloy, and Biplav Srivastava. Detecting Backdoor

Attacks on Deep Neural Networks by Activation Clustering. In Huáscar Espinoza, Seán Ó hÉigeartaigh, Xiaowei Huang, José Hernández-Orallo, and Mauricio Castillo-Effen, editors, *Proceedings of the AAAI Workshop on Artificial Intelligence Safety 2019 co-located with the 33rd AAAI Conference on Artificial Intelligence 2019 (AAAI 2019)*, volume 2301 of *CEUR Workshop*, Honolulu, HI, USA, 2019. CEUR.

[20] Tongfei Chen, Jiri Navratil, Vijay Iyengar, and Karthikeyan Shanmugam. Confidence Scoring Using Whitebox Meta-models with Linear Classifier Probes. In Kamalika Chaudhuri and Masashi Sugiyama, editors, *The 22nd International Conference on Artificial Intelligence and Statistics (AISTATS)*, volume 89 of *Proceedings of Machine Learning Research*, pages 1467–1475, Naha, Japan, 2019. PMLR.

[21] Tarin Clanuwat, Mikel Bober-Irizar, Asanobu Kitamoto, Alex Lamb, Kazuaki Yamamoto, and David Ha. Deep Learning for Classical Japanese Literature. *arXiv preprint* `arXiv:1812.01718`, 2018.

[22] Gilad Cohen, Guillermo Sapiro, and Raja Giryes. Detecting Adversarial Samples Using Influence Functions and Nearest Neighbors. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 14441–14450, Seattle, WA, USA, 2020. IEEE.

[23] Francesco Crecchi, Davide Bacciu, and Battista Biggio. Detecting Adversarial Examples through Nonlinear Dimensionality Reduction. In *27th European Symposium on Artificial Neural Networks (ESANN)*, Bruges, Belgium, 2019.

[24] Yann N Dauphin and Samuel Schoenholz. MetaInit: Initializing Learning by Learning to Initialize. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Proceedings of the 2019 Conference*, volume 32 of *NeurIPS 2019*, pages 12645–12657, Red Hook, NY, USA, 2019. Curran Associates Inc.

[25] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Fei-Fei Li. ImageNet: A Large-scale Hierarchical Image Database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 248–255, Miami, FL, USA, 2009. IEEE.

[26] Thomas G. Dietterich. Approximate Statistical Tests for Comparing Supervised Classification Learning Algorithms. *Neural Computation*, 10(7):1895–1923, 1998.

[27] Carl Doersch, Abhinav Gupta, and Alexei A Efros. Unsupervised Visual Representation Learning by Context Prediction. In *2015 IEEE International Conference on Computer Vision (ICCV)*, pages 1422–1430, Santiago, Chile, 2015. IEEE.

[28] Pedro Domingos. A Few Useful Things to Know about Machine Learning. *Communications of the ACM*, 55(10):78–87, 2012.

[29] Qi Dong, Shaogang Gong, and Xiatian Zhu. Class Rectification Hard Mining for Imbalanced Deep Learning. In *2017 IEEE International Conference on Computer Vision (ICCV)*, pages 1869–1878, Venice, Italy, 2017. IEEE.

[30] Vincent Dumoulin and Francesco Visin. A Guide to Convolution Arithmetic for Deep Learning. *arXiv preprint* `arXiv:1603.07285`, 2016.

[31] Dumitru Erhan, Yoshua Bengio, Aaron Courville, Pierre-Antoine Manzagol, Pascal Vincent, and Samy Bengio. Why Does Unsupervised Pre-Training Help Deep Learning? In Yee Whye Teh and Mike Titterington, editors, *The 13th International Conference on Artificial Intelligence and Statistics (AISTATS)*, volume 9 of *Proceedings of Machine Learning Research*, pages 201–208, Sardinia, Italy, 2010. PMLR.

[32] Linus Ericsson, Henry Gouk, Chen Change Loy, and Timothy M. Hospedales. Self-Supervised Representation Learning: Introduction, Advances, and Challenges. *IEEE Signal Processing Magazine*, 39(3):42–62, 2022.

[33] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *Proceedings of the 2nd International Conference on Knowledge Discovery and Data Mining (KDD)*, KDD 1996, page 226–231, Palo Alto, CA, USA, 1996. AAAI Press.

[34] Kevin Eykholt, Ivan Evtimov, Earlence Fernandes, Bo Li, Amir Rahmati, Chaowei Xiao, Atul Prakash, Tadayoshi Kohno, and Dawn Song. Robust Physical-World Attacks on Deep Learning Visual Classification. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1625–1634, Salt Lake City, UT, USA, 2018. IEEE.

[35] Yarin Gal. *Uncertainty in Deep Learning.* PhD thesis, University of Cambridge, 2016.

[36] Yarin Gal and Zoubin Ghahramani. Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning. In Maria Florina Balcan and Kilian Q. Weinberger, editors, *The 33rd International Conference on Machine Learning (ICML)*, volume 48 of *Proceedings of Machine Learning Research*, pages 1050–1059, New York, NY, USA, 2016. PMLR.

[37] Salvador García and Francisco Herrera. Evolutionary Undersampling for Classification with Imbalanced Datasets: Proposals and Taxonomy. *Evolutionary Computation*, 17(3):275–306, 2009.

[38] Xavier Glorot and Yoshua Bengio. Understanding the Difficulty of Training Deep Feedforward Neural Networks. In Yee Whye Teh and Mike Titterington, editors, *The 13th International Conference on Artificial Intelligence and Statistics (AISTATS)*, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256, Sardinia, Italy, 2010. PMLR.

[39] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep Sparse Rectifier Neural Networks. In Geoffrey Gordon, David Dunson, and Miroslav Dudík, editors, *The 14th International Conference on Artificial Intelligence and Statistics (AISTATS)*, volume 15 of *Proceedings of Machine Learning Research*, pages 315–323, Fort Lauderdale, FL, USA, 2011. PMLR.

[40] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning.* MIT Press, 2016. `http://www.deeplearningbook.org`.

[41] Ian Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and Harnessing Adversarial Examples. In Yoshua Bengio and Yann LeCun, editors, *The 3rd International Conference on Learning Representations (ICLR)*, San Diego, CA, USA, 2015.

[42] Scott Gray, Alec Radford, and Diederik P. Kingma. GPU Kernels for Block-Sparse Weights. `https://openai.com/research/block-sparse-gpu-kernels`, 2017.

[43] Kathrin Grosse, Praveen Manoharan, Nicolas Papernot, Michael Backes, and Patrick McDaniel. On the (Statistical) Detection of Adversarial Examples. *arXiv preprint* `arXiv:1702.06280`, 2017.

[44] Jingyu Hao, Chengjia Wang, Heye Zhang, and Guang Yang. Annealing Genetic GAN for Minority Oversampling. In *31st British Machine Vision Conference 2020, BMVC 2020*. BMVA Press, 2020.

[45] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. In *2015 IEEE International Conference on Computer Vision (ICCV)*, pages 1026–1034, Santiago, Chile, 2015. IEEE.

[46] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, Las Vegas, NV, USA, 2016. IEEE.

[47] Dan Hendrycks, Steven Basart, Mantas Mazeika, Andy Zou, Joe Kwon, Mohammadreza Mostajabi, Jacob Steinhardt, and Dawn Song. Scaling Out-of-Distribution Detection for Real-World Settings. In Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvari, Gang Niu, and Sivan Sabato, editors, *The 39th International Conference on Machine Learning (ICML)*, volume 162 of *Proceedings of Machine Learning Research*, pages 8759–8773, Baltimore, MD, USA, 2022. PMLR.

[48] Dan Hendrycks, Steven Basart, Norman Mu, Saurav Kadavath, Frank Wang, Evan Dorundo, Rahul Desai, Tyler Zhu, Samyak Parajuli, Mike Guo, Dawn Song, Jacob Steinhardt, and Justin Gilmer. The Many Faces of Robustness: A Critical Analysis of Out-of-Distribution Generalization. In *2021 IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 8320–8329, Montreal, Canada, 2021. IEEE.

[49] Dan Hendrycks and Kevin Gimpel. A Baseline for Detecting Misclassified and Out-of-Distribution Examples in Neural Networks. In *The 5th International Conference on Learning Representations (ICLR)*, Toulon, France, 2017. OpenReview.net.

[50] Dan Hendrycks, Mantas Mazeika, Saurav Kadavath, and Dawn Song. Using Self-Supervised Learning Can Improve Model Robustness and Uncertainty. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Proceedings of the 2019 Conference*, volume 32 of *NeurIPS 2019*, pages 15637–15648, Red Hook, NY, USA, 2019. Curran Associates Inc.

[51] Dan Hendrycks, Kevin Zhao, Steven Basart, Jacob Steinhardt, and Dawn Song. Natural Adversarial Examples. In *2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 15257–15266, Nashville, TN, USA, 2021. IEEE.

[52] Sebastian Houben, Johannes Stallkamp, Jan Salmen, Marc Schlipsing, and Christian Igel. Detection of Traffic Signs in Real-World Images: The German Traffic Sign Detection Benchmark. In *The 2013 International Joint Conference on Neural Networks (IJCNN)*, number 1288, pages 1–8, Dallas, TX, USA, 2013. IEEE.

[53] Jeremy Howard and Sylvain Gugger. *Deep Learning for Coders with fastai and PyTorch*. O'Reilly Media, Inc., 2020.

[54] Jeremy Howard and Sebastian Ruder. Universal Language Model Fine-tuning for Text Classification. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (ACL)*, volume 1, pages 328–339, Melbourne, Australia, 2018. ACL.

[55] Chen Huang, Yining Li, Chen Change Loy, and Xiaoou Tang. Learning Deep Representation for Imbalanced Classification. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 5375–5384, Las Vegas, NV, USA, 2016. IEEE.

[56] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q. Weinberger. Densely Connected Convolutional Networks. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2261–2269, Honolulu, HI, USA, 2017. IEEE.

[57] Haiwen Huang, Zhihan Li, Lulu Wang, Sishuo Chen, Bin Dong, and Xinyu Zhou. Feature Space Singularity for Out-of-Distribution Detection. In Huáscar Espinoza, John McDermid, Xiaowei Huang, Mauricio Castillo-Effen, Xin Cynthia Chen, José Hernández-Orallo, Seán Ó hÉigeartaigh, and Richard Mallah, editors, *Proceedings of the Workshop on Artificial Intelligence Safety 2021 (SafeAI 2021) co-located with the 35th AAAI Conference on Artificial Intelligence (AAAI 2021)*, CEUR Workshop. CEUR, 2021.

[58] Qian Huang, Isay Katsman, Horace He, Zeqi Gu, Serge J. Belongie, and Ser-Nam Lim. Enhancing Adversarial Example Transferability with an Intermediate Level Attack. In *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 4732–4741, Seoul, South Korea, 2019. IEEE.

[59] Andrew Ilyas, Shibani Santurkar, Dimitris Tsipras, Logan Engstrom, Brandon Tran, and Aleksander Madry. Adversarial Examples Are Not Bugs, They Are Features. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Proceedings of the 2019 Conference*, volume 32 of *NeurIPS 2019*, Red Hook, NY, USA, 2019. Curran Associates Inc.

[60] Sergey Ioffe and Corinna Cortes. Batch Normalization Layers. US 2016/0217368 A1, Jul. 28, 2016. `https://patents.google.com/patent/US20160217368A1/en`.

[61] Sergey Ioffe and Christian Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In Francis Bach and David Blei, editors, *The 32nd International Conference on Machine Learning (ICML)*, volume 37 of *Proceedings of Machine Learning Research*, pages 448–456, Lille, France, 2015. PMLR.

[62] Salman H. Khan, Munawar Hayat, Mohammed Bennamoun, Ferdous Sohel, and Roberto Togneri. Cost Sensitive Learning of Deep Feature Representations from Imbalanced Data. *IEEE Transactions on Neural Networks and Learning Systems*, 29(8):3573–3587, 2018.

[63] Prannay Khosla, Piotr Teterwak, Chen Wang, Aaron Sarna, Yonglong Tian, Phillip Isola, Aaron Maschinot, Ce Liu, and Dilip Krishnan. Supervised Contrastive Learning. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Proceedings of the 2020 Conference*, volume 33 of *NeurIPS 2020*, pages 18661–18673, Red Hook, NY, USA, 2020. Curran Associates Inc.

[64] Hoki Kim. Torchattacks: A PyTorch Repository for Adversarial Attacks. *arXiv preprint* `arXiv:2010.01950`, 2020.

[65] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. In Yoshua Bengio and Yann LeCun, editors, *The 3rd International Conference on Learning Representations (ICLR)*, San Diego, CA, USA, 2015.

[66] Saiprasad Koturwar and Shabbir Merchant. Weight Initialization of Deep Neural Networks (DNNs) using Data Statistics. *arXiv preprint* `arXiv:1710.10570`, 2017.

[67] Michał Koziarski. Radial-based Undersampling for Imbalanced Data Classification. *Pattern Recognition*, 102:107262, 2020.

[68] Michał Koziarski. Two-Stage Resampling for Convolutional Neural Network Training in the Imbalanced Colorectal Cancer Image Classification. In *The 2021 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, Shenzhen, China, 2021. IEEE.

[69] Philipp Krähenbühl, Carl Doersch, Jeff Donahue, and Trevor Darrell. Data-dependent Initializations of Convolutional Neural Networks. In Yoshua Bengio and Yann LeCun, editors, *The 4th International Conference on Learning Representations (ICLR)*, San Juan, Puerto Rico, 2016.

[70] Alex Krizhevsky. Learning Multiple Layers of Features from Tiny Images. Technical report, University of Toronto, 2009.

[71] Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In Fernando Pereira, Christopher J. Burges, Léon Bottou, and Kilian Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25: Proceedings of the 2012 Conference*, volume 25 of *NIPS 2012*, pages 1106–1114, Lake Tahoe, NV, USA, 2012. Curran Associates, Inc.

[72] Miroslav Kubat and Stan Matwin. Addressing the Curse of Imbalanced Training Sets: One-Sided Selection. In Douglas H. Fisher, editor, *The 14th International Conference on Machine Learning (ICML)*, volume 97, pages 179–186, Nashville, TN, USA, 1997. Morgan Kaufmann.

[73] Alexey Kurakin, Ian J. Goodfellow, and Samy Bengio. Adversarial Examples in the Physical World. In *The 5th International Conference on Learning Representations (ICLR)*, Toulon, France, 2017. OpenReview.net.

[74] Agostina J. Larrazabal, Nicolás Nieto, Victoria Peterson, Diego H. Milone, and Enzo Ferrante. Gender Imbalance in Medical Imaging Datasets Produces Biased Classifiers for Computer-aided Diagnosis. *Proceedings of the National Academy of Sciences (PNAS)*, 117(23):12592–12594, 2020.

[75] Gustav Larsson, Michael Maire, and Gregory Shakhnarovich. Learning Representations for Automatic Colorization. In Bastian Leibe, Jiri Matas, Nicu Sebe, and Max Welling, editors, *The 14th European Conference on Computer Vision (ECCV)*, volume 9908 of *Lecture Notes in Computer Science*, pages 577–593, Cham, Switzerland, 2016. Springer.

[76] Yann LeCun, Corinna Cortes, and Chris Burges. MNIST Handwritten Digit Database. *ATT Labs [Online]*, 2, 2010. `http://yann.lecun.com/exdb/mnist`.

[77] Hansang Lee, Minseok Park, and Junmo Kim. Plankton Classification on Imbalanced Large Scale Database via Convolutional Neural Networks with Transfer Learning. In *2016 IEEE International Conference on Image Processing (ICIP)*, pages 3713–3717, Phoenix, AZ, USA, 2016. IEEE.

[78] Kimin Lee, Honglak Lee, Kibok Lee, and Jinwoo Shin. Training Confidence-calibrated Classifiers for Detecting Out-of-Distribution Samples. In *The 6th International Conference on Learning Representations (ICLR)*, Vancouver, Canada, 2018. OpenReview.net.

[79] Kimin Lee, Kibok Lee, Honglak Lee, and Jinwoo Shin. A Simple Unified Framework for Detecting Out-of-Distribution Samples and Adversarial Attacks. In Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 31: Proceedings of the 2018 Conference*, volume 31 of *NeurIPS 2018*, page 7167–7177, Red Hook, NY, USA, 2018. Curran Associates Inc.

[80] Daniel Lehmann and Marc Ebner. Are Image Patches Beneficial for Initializing Convolutional Neural Network Models? In *Proceedings of the 16th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications (VISIGRAPP 2021) - Volume 5: VISAPP*, pages 346–353. INSTICC, SciTePress, 2021.

[81] Daniel Lehmann and Marc Ebner. Layer-Wise Activation Cluster Analysis of CNNs to Detect Out-of-Distribution Samples. In Igor Farkaš, Paolo Masulli, Sebastian Otte, and Stefan Wermter, editors, *Proceedings of the 30th International Conference on Artificial Neural Networks and Machine Learning - ICANN 2021*, Lecture Notes in Computer Science, pages 214–226, Cham, Switzerland, 2021. Springer.

[82] Daniel Lehmann and Marc Ebner. Calculating the Credibility of Test Samples at Inference by a Layer-wise Activation Cluster Analysis of Convolutional Neural Networks. In *Proceedings of the 3rd International Conference on Deep Learning Theory and Applications - Volume 1: DeLTA*, pages 34–43, Lisbon, Portugal, 2022. INSTICC, SciTePress.

[83] Daniel Lehmann and Marc Ebner. Subclass-based Undersampling for Class-imbalanced Image Classification. In *Proceedings of the 17th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications (VISIGRAPP 2022) - Volume 5: VISAPP*, pages 493–500. INSTICC, SciTePress, 2022.

[84] Daniel Lehmann and Marc Ebner. Reliable Classification of Images by Calculating their Credibility Using a Layer-wise Activation Cluster Analysis of CNNs. In Ana Fred, Carlo Sansone, Oleg Gusikhin, and Kurosh Madani, editors, *Deep Learning Theory and Applications - DeLTA 2022 (Revised Selected Papers)*, Communications in Computer and Information Science, pages 33–55, Cham, Switzerland, 2023. Springer.

[85] Oscar Li, Hao Liu, Chaofan Chen, and Cynthia Rudin. Deep Learning for Case-Based Reasoning through Prototypes: A Neural Network that Explains its Predictions. In *Proceedings of the 32nd AAAI Conference on Artificial Intelligence 2018*

(AAAI 2018), volume 32 of *AAAI 2018/IAAI 2018/EAAI 2018*, New Orleans, LA, USA, 2018. AAAI Press.

[86] Xin Li and Fuxin Li. Adversarial Examples Detection in Deep Networks with Convolutional Filter Statistics. In *2017 IEEE International Conference on Computer Vision (ICCV)*, pages 5775–5783, Venice, Italy, 2017. IEEE.

[87] Yu Li, Tao Wang, Bingyi Kang, Sheng Tang, Chunfeng Wang, Jintao Li, and Jiashi Feng. Overcoming Classifier Imbalance for Long-Tail Object Detection with Balanced Group Softmax. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 10988–10997, Seattle, WA, USA, 2020. IEEE.

[88] Min Lin, Qiang Chen, and Shuicheng Yan. Network in Network. In Yoshua Bengio and Yann LeCun, editors, *The 2nd International Conference on Learning Representations (ICLR)*, Banff, Canada, 2014.

[89] Ziqian Lin, Sreya Dutta Roy, and Yixuan Li. MOOD: Multi-level Out-of-distribution Detection. In *2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 15308–15318, Nashville, TN, USA, 2021. IEEE.

[90] Xu-Ying Liu, Jianxin Wu, and Zhi-Hua Zhou. Exploratory Undersampling for Class-Imbalance Learning. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 39(2):539–550, 2009.

[91] Ilya Loshchilov and Frank Hutter. SGDR: Stochastic Gradient Descent with Warm Restarts. In *The 5th International Conference on Learning Representations (ICLR)*, Toulon, France, 2017. OpenReview.net.

[92] David G. Lowe. Distinctive Image Features from Scale-Invariant Keypoints. *International Journal of Computer Vision*, 60(2):91–110, 2004.

[93] Xingjun Ma, Bo Li, Yisen Wang, Sarah M. Erfani, Sudanthi Wijewickrema, Grant Schoenebeck, Michael E. Houle, Dawn Song, and James Bailey. Characterizing Adversarial Subspaces Using Local Intrinsic Dimensionality. In *The 6th International Conference on Learning Representations (ICLR)*, Vancouver, Canada, 2018. OpenReview.net.

[94] Mateus Machado, Evandro Ruiz, and Kuruvilla Joseph Abraham. A New Statistical Approach for Comparing Algorithms for Lexicon-based Sentiment Analysis. *arXiv preprint* `arXiv:1906.08717`, 2019.

[95] J. MacQueen. Some Methods for Classification and Analysis of Multivariate Observations. In L. M. Le Cam and J. Neyman, editors, *Proceedings of the 5th Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, pages 281–297. University of California Press, 1967.

[96] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards Deep Learning Models Resistant to Adversarial Attacks. In *The 6th International Conference on Learning Representations (ICLR)*, Vancouver, Canada, 2018. OpenReview.net.

[97] Anima Majumder, Samrat Dutta, Swagat Kumar, and Laxmidhar Behera. A Method for Handling Multi-class Imbalanced Data by Geometry-based Information Sampling and Class-prioritized Synthetic Data Generation (GICaPS). *arXiv preprint* `arXiv:2010.05155`, 2020.

[98] Leland McInnes, John Healy, and James Melville. UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction. *arXiv preprint* `arXiv:1802.03426`, 2018.

[99] Dongyu Meng and Hao Chen. MagNet: A Two-Pronged Defense against Adversarial Examples. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS 2017, page 135–147, New York, NY, USA, 2017. ACM.

[100] Jan Hendrik Metzen, Tim Genewein, Volker Fischer, and Bastian Bischoff. On Detecting Adversarial Perturbations. In *The 5th International Conference on Learning Representations (ICLR)*, Toulon, France, 2017. OpenReview.net.

[101] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. Mixed Precision Training. In *The 6th International Conference on Learning Representations (ICLR)*, Vancouver, Canada, 2018. OpenReview.net.

[102] Dmytro Mishkin and Jiri Matas. All You Need is a Good Init. In Yoshua Bengio and Yann LeCun, editors, *The 4th International Conference on Learning Representations (ICLR)*, San Juan, Puerto Rico, 2016.

[103] Mohammadreza Mohseni, Jordan Yap, William Yolland, Majid Razmara, and M. Stella Atkins. Out-of-Distribution Detection for Dermoscopic Image Classification. *arXiv preprint* `arXiv:2104.07819`, 2021.

[104] Ajinkya More. Survey of Resampling Techniques for Improving Classification Performance in Unbalanced Datasets. *arXiv preprint* `arXiv:1608.06048`, 2016.

[105] Sankha Subhra Mullick, Shounak Datta, and Swagatam Das. Generative Adversarial Minority Oversampling. In *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 1695–1704, Seoul, South Korea, 2019. IEEE.

[106] Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, and Andrew Y. Ng. Reading Digits in Natural Images with Unsupervised Feature Learning. In *Proceedings of the NIPS Workshop on Deep Learning and Unsupervised Feature Learning*, 2011. `http://ufldl.stanford.edu/housenumbers`.

[107] Wing WY Ng, Shichao Xu, Jianjun Zhang, Xing Tian, Tongwen Rong, and Sam Kwong. Hashing-based Undersampling Ensemble for Imbalanced Pattern Classification Problems. *IEEE Transactions on Cybernetics*, 52(2):1269–1279, 2022.

[108] Anh Nguyen, Jason Yosinski, and Jeff Clune. Deep Neural Networks are Easily Fooled: High Confidence Predictions for Unrecognizable Images. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 427–436, Boston, MA, USA, 2015. IEEE.

[109] Anh Nguyen, Jason Yosinski, and Jeff Clune. Multifaceted Feature Visualization: Uncovering the Different Types of Features Learned by Each Neuron in Deep Neural Networks. *arXiv preprint* `arXiv:1602.03616`, 2016. *Workshop on Visualization for Deep Learning at ICML 2016.*

[110] Rajendran Nirthika, Siyamalan Manivannan, Amirthalingam Ramanan, and Ruixuan Wang. Pooling in Convolutional Neural Networks for Medical Image Analysis: A Survey and an Empirical Study. *Neural Computing and Applications*, 34(7):5321–5347, 2022.

[111] Mehdi Noroozi and Paolo Favaro. Unsupervised Learning of Visual Representations by Solving Jigsaw Puzzles. In Bastian Leibe, Jiri Matas, Nicu Sebe, and Max Welling, editors, *The 14th European Conference on Computer Vision (ECCV)*, volume 9910 of *Lecture Notes in Computer Science*, pages 69–84, Cham, Switzerland, 2016. Springer.

[112] Gökhan Özbulak and Hazim Kemal Ekenel. Initialization of Convolutional Neural Networks by Gabor Filters. In *2018 26th Signal Processing and Communications Applications Conference (SIU)*, pages 1–4, Izmir, Turkey, 2018. IEEE.

[113] Nicolas Papernot. *Characterizing the Limits and Defenses of Machine Learning in Adversarial Settings.* PhD thesis, The Pennsylvania State University, 2018.

[114] Nicolas Papernot and Patrick McDaniel. Deep k-Nearest Neighbors: Towards Confident, Interpretable and Robust Deep Learning. *arXiv preprint* `arXiv:1803.04765`, 2018.

[115] Deepak Pathak, Philipp Krähenbühl, Jeff Donahue, Trevor Darrell, and Alexei A. Efros. Context Encoders: Feature Learning by Inpainting. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2536–2544, Las Vegas, NV, USA, 2016. IEEE.

[116] Karl Pearson. LIII. On Lines and Planes of Closest Fit to Systems of Points in Space. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 2(11):559–572, 1901.

[117] Samira Pouyanfar, Yudong Tao, Anup Mohan, Haiman Tian, Ahmed S. Kaseb, Kent Gauen, Ryan Dailey, Sarah Aghajanzadeh, Yung Hsiang Lu, Shu Ching

Chen, and Mei Ling Shyu. Dynamic Sampling in Convolutional Neural Networks for Imbalanced Data Classification. In *2018 IEEE Conference on Multimedia Information Processing and Retrieval (MIPR)*, pages 112–117, Miami, FL, USA, 2018. IEEE.

[118] Maithra Raghu, Chiyuan Zhang, Jon Kleinberg, and Samy Bengio. Transfusion: Understanding Transfer Learning for Medical Imaging. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Proceedings of the 2019 Conference*, volume 32 of *NeurIPS 2019*, pages 3342–3352, Red Hook, NY, USA, 2019. Curran Associates Inc.

[119] Md Shamim Reza and Jinwen Ma. Imbalanced Histopathological Breast Cancer Image Classification with Convolutional Neural Network. In *2018 14th IEEE International Conference on Signal Processing (ICSP)*, pages 619–624, Beijing, China, 2018. IEEE.

[120] Herbert Robbins and Sutton Monro. A Stochastic Approximation Method. *The Annals of Mathematical Statistics*, 22(3):400–407, 1951.

[121] Peter J. Rousseeuw. Silhouettes: A Graphical Aid to the Interpretation and Validation of Cluster Analysis. *Journal of Computational and Applied Mathematics*, 20(1):53–65, 1987.

[122] Sebastian Ruder. An Overview of Gradient Descent Optimization Algorithms. *arXiv preprint* `arXiv:1609.04747`, 2017.

[123] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[124] Dipankar Sarkar, Ankur Narang, and Sumit Rai. Fed-Focal Loss for Imbalanced Data Classification in Federated Learning. *arXiv preprint* `arXiv:2011.06283`, 2020.

[125] Chandramouli Shama Sastry and Sageev Oore. Detecting Out-of-Distribution Examples with Gram Matrices. In Hal Daumé III and Aarti Singh, editors, *The 37th International Conference on Machine Learning (ICML)*, volume 119 of *Proceedings of Machine Learning Research*, pages 8491–8501. PMLR, 2020.

[126] Andrew M. Saxe, James L. McClelland, and Surya Ganguli. Exact Solutions to the Nonlinear Dynamics of Learning in Deep Linear Neural Networks. In Yoshua Bengio and Yann LeCun, editors, *The 2nd International Conference on Learning Representations (ICLR)*, Banff, Canada, 2014.

[127] Mathias Seuret, Michele Alberti, Marcus Liwicki, and Rolf Ingold. PCA-Initialized Deep Neural Networks Applied to Document Image Analysis. In *2017 14th IAPR International Conference on Document Analysis and Recognition (ICDAR)*, volume 01, pages 877–882, Kyoto, Japan, 2017. IEEE.

[128] Connor Shorten and Taghi M. Khoshgoftaar. A Survey on Image Data Augmentation for Deep Learning. *Journal of Big Data*, 6(1), 2019.

[129] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. In Yoshua Bengio and Yann LeCun, editors, *The 3rd International Conference on Learning Representations (ICLR)*, San Diego, CA, USA, 2015.

[130] Naman D. Singh and Abhinav Dhall. Clustering and Learning from Imbalanced Data. *arXiv preprint* `arXiv:1811.00972`, 2018.

[131] Poulami Sinhamahapatra, Rajat Koner, Karsten Roscher, and Stephan Günnemann. Is it all a cluster game? - Exploring Out-of-Distribution Detection based on Clustering in the Embedding Space. In Gabriel Pedroza, José Hernández-Orallo, Xin Cynthia Chen, Xiaowei Huang, Huáscar Espinoza, Mauricio Castillo-Effen, John A. McDermid, Richard Mallah, and Seán Ó hÉigeartaigh, editors, *Proceedings of the Workshop on Artificial Intelligence Safety 2022 (SafeAI 2022) co-located with the 36th AAAI Conference on Artificial Intelligence (AAAI 2022)*, volume 3087 of *CEUR Workshop*. CEUR, 2022.

[132] Leslie N. Smith. Cyclical Learning Rates for Training Neural Networks. In *2017 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pages 464–472, Santa Rosa, CA, USA, 2017. IEEE.

[133] Robert Sowah, Moses Agebure, Godfrey Mills, Koudjo Koumadi, and Seth Fiawoo. New Cluster Undersampling Technique for Class Imbalance Learning. *International Journal of Machine Learning and Computing*, 6(3):205–214, 2016.

[134] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958, 2014.

[135] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian J. Goodfellow, and Rob Fergus. Intriguing Properties of Neural Networks. In Yoshua Bengio and Yann LeCun, editors, *The 2nd International Conference on Learning Representations (ICLR)*, Banff, Canada, 2014.

[136] Ranjita Thapa, Kai Zhang, Noah Snavely, Serge Belongie, and Awais Khan. The Plant Pathology Challenge 2020 Data Set to Classify Foliar Disease of Apples. *Applications in Plant Science*, 8(9), 2020.

[137] Chih-Fong Tsai, Wei-Chao Lin, Ya-Han Hu, and Guan-Ting Yao. Under-Sampling Class Imbalanced Datasets by Combining Clustering Analysis and Instance Selection. *Information Sciences*, 477:47–54, 2019.

[138] Laurens van der Maaten. Learning a Parametric Embedding by Preserving Local Structure. In David van Dyk and Max Welling, editors, *The 12th International Conference on Artificial Intelligence and Statistics (AISTATS)*, volume 5 of *Proceedings of Machine Learning Research*, pages 384–391, Clearwater Beach, FL, USA, 2009. PMLR.

[139] Laurens van der Maaten and Geoffrey Hinton. Visualizing Data using t-SNE. *Journal of Machine Learning Research*, 9(86):2579–2605, 2008.

[140] Yiru Wang, Weihao Gan, Jie Yang, Wei Wu, and Junjie Yan. Dynamic Curriculum Learning for Imbalanced Data Classification. In *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 5016–5025, Seoul, South Korea, 2019. IEEE.

[141] Donglai Wei, Bolei Zhou, Antonio Torrabla, and William Freeman. Understanding Intra-Class Knowledge Inside CNN. *arXiv preprint* `arXiv:1507.02379`, 2015.

[142] Lei Wu, Zhanxing Zhu, Cheng Tai, and Weinan E. Understanding and Enhancing the Transferability of Adversarial Examples. *arXiv preprint* `arXiv:1802.09707`, 2018.

[143] Tong Xiao, Tian Xia, Yi Yang, Chang Huang, and Xiaogang Wang. Learning from Massive Noisy Labeled Data for Image Classification. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2691–2699, Boston, MA, USA, 2015. IEEE.

[144] Ning Xie, Gabrielle Ras, Marcel van Gerven, and Derek Doran. Explainable Deep Learning: A Field Guide for the Uninitiated. *Journal of Artificial Intelligence Research*, 73:329–396, 2022.

[145] Jingkang Yang, Kaiyang Zhou, Yixuan Li, and Ziwei Liu. Generalized Out-of-Distribution Detection: A Survey. *arXiv preprint* `arXiv:2110.11334`, 2021.

[146] Huaxiu Yao, Yu Wang, Sai Li, Linjun Zhang, Weixin Liang, James Zou, and Chelsea Finn. Improving Out-of-Distribution Robustness via Selective Augmentation. In Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvari, Gang Niu, and Sivan Sabato, editors, *The 39th International Conference on Machine Learning (ICML)*, volume 162 of *Proceedings of Machine Learning Research*, pages 25407–25437, Baltimore, MD, USA, 2022. PMLR.

[147] Show-Jane Yen and Yue-Shi Lee. Cluster-based Under-Sampling Approaches for Imbalanced Data Distributions. *Expert Systems with Applications*, 36(3):5718–5727, 2009.

[148] Matthew D. Zeiler and Rob Fergus. Visualizing and Understanding Convolutional Networks. In David Fleet, Tomas Pajdla, Bernt Schiele, and Tinne Tuytelaars, editors, *The 13th European Conference on Computer Vision (ECCV)*, volume 8689 of *Lecture Notes in Computer Science*, pages 818–833, Cham, Switzerland, 2014. Springer.

[149] Hongyi Zhang, Moustapha Cisse, Yann N. Dauphin, and David Lopez-Paz. mixup: Beyond Empirical Risk Minimization. In *The 6th International Conference on Learning Representations (ICLR)*, Vancouver, Canada, 2018. OpenReview.net.

[150] Hongyi Zhang, Yann N. Dauphin, and Tengyu Ma. Fixup Initialization: Residual Learning Without Normalization. In *The 7th International Conference on Learning Representations (ICLR)*, New Orleans, LA, USA, 2019. OpenReview.net.

[151] Jianping Zhang and Inderjeet Mani. kNN Approach to Unbalanced Data Distributions: A Case Study Involving Information Extraction. In *Proceedings of the ICML Workshop on Learning from Imbalanced Datasets*, volume 126, pages 1–7, 2003.

[152] Richard Zhang, Phillip Isola, and Alexei A. Efros. Colorful Image Colorization. In Bastian Leibe, Jiri Matas, Nicu Sebe, and Max Welling, editors, *The 14th European Conference on Computer Vision (ECCV)*, volume 9907 of *Lecture Notes in Computer Science*, pages 649–666, Cham, Switzerland, 2016. Springer.

[153] Yulu Zhang, Liguo Shuai, Yali Ren, and Huilin Chen. Image Classification with Category Centers in Class Imbalance Situation. In *2018 33rd Youth Academic Annual Conference of Chinese Association of Automation (YAC)*, pages 359–363, Nanjing, China, 2018. IEEE.

[154] Fuzhen Zhuang, Zhiyuan Qi, Keyu Duan, Dongbo Xi, Yongchun Zhu, Hengshu Zhu, Hui Xiong, and Qing He. A Comprehensive Survey on Transfer Learning. *Proceedings of the IEEE*, 109(1):43–76, 2021.

List of Publications

This thesis is based on the following scientific publications:

- Daniel Lehmann and Marc Ebner (2023)
  Reliable Classification of Images by Calculating their Credibility Using a Layer-wise Activation Cluster Analysis of CNNs. In *Ana Fred, Carlo Sansone, Oleg Gusikhin, Kurosh Madani (editors) Deep Learning Theory and Applications - DeLTA 2022 (Revised Selected Papers)*, Communications in Computer and Information Science, Springer Cham, pages 33-55.

- Daniel Lehmann and Marc Ebner (2022)
  Calculating the Credibility of Test Samples at Inference by a Layer-wise Activation Cluster Analysis of Convolutional Neural Networks. In *Proceedings of the 3rd International Conference on Deep Learning Theory and Applications - Volume 1: DeLTA*, SciTePress, pages 34-43. **(Best Paper Award)**

- Daniel Lehmann and Marc Ebner (2022)
  Subclass-based Undersampling for Class-imbalanced Image Classification. In *Proceedings of the 17th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications (VISIGRAPP 2022) - Volume 5: VISAPP*, SciTePress, pages 493-500.

- Daniel Lehmann and Marc Ebner (2021)
  Layer-Wise Activation Cluster Analysis of CNNs to Detect Out-of-Distribution Samples. In *Igor Farkaš, Paolo Masulli, Sebastian Otte and Stefan Wermter (editors) Proceedings of the 30th International Conference on Artificial Neural Networks and Machine Learning - ICANN 2021*, Lecture Notes in Computer Science, Springer Cham, pages 214-226.

- Daniel Lehmann and Marc Ebner (2021)
  Are Image Patches Beneficial for Initializing Convolutional Neural Network Models? In *Proceedings of the 16th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications (VISIGRAPP 2021) - Volume 5: VISAPP*, SciTePress, pages 346-353.

Eigenständigkeitserklärung

Hiermit erkläre ich, dass diese Arbeit bisher von mir weder an der Mathematisch-Naturwissenschaftlichen Fakultät der Universität Greifswald noch einer anderen wissenschaftlichen Einrichtung zum Zwecke der Promotion eingereicht wurde. Ferner erkläre ich, dass ich diese Arbeit selbstständig verfasst und keine anderen als die darin angegebenen Hilfsmittel und Hilfen benutzt und keine Textabschnitte eines Dritten ohne Kennzeichnung übernommen habe.

....

Ort, Datum Unterschrift

Curriculum Vitae

Name:	Daniel Lehmann
Date of Birth:	29.11.1985
Place of Birth:	Rostock

Education

Since 2019	PhD Student in Computer Science *University of Greifswald, Germany*
2006 – 2012	Student in Business Computer Science (Degree: Diplom) *University of Rostock, Germany*
1996 – 2005	High School Student (Degree: Abitur) *Richard-Wossidlo-Gymnasium Ribnitz-Damgarten, Germany*

Professional Experience

2017 – 2023	Research Assistant at Institut für Mathematik und Informatik *University of Greifswald, Germany*
2013 – 2017	Machine Learning Engineer *Gini GmbH, Munich, Germany*
2010 – 2011	Software Engineering Intern *IBM, San José, CA, USA*

Acknowledgement

First of all, I would like to thank Prof. Dr. Marc Ebner for his supervision. Thanks to his intensive support, I was able to acquire comprehensive skills. This allowed me to develop as a researcher in general. I also thank my colleagues, Diclehan Ulucan and Oguzhan Ulucan, for their support and the interesting discussions with them. In addition to my colleagues, I would like to thank Catherine Castling for proofreading the manuscript. Finally, I want to thank the University of Greifswald and my family.