Dive into Deep Learning

xAI-Proj-B: Bachelor Project Explainable Machine Learning

Florian Gutbier[*]
Otto-Friedrich University of Bamberg
96047 Bamberg, Germany
florian.gutbier@stud.uni-bamberg.de

Marius Ludwig Bachmeier[†]
Otto-Friedrich University of Bamberg
96047 Bamberg, Germany
marius-ludwig.bachmeier@stud.uni-bamberg.de

Andreas Schwab[‡]
Otto-Friedrich University of Bamberg
96047 Bamberg, Germany
andreas-franz.schwab@stud.uni-bamberg.de

Abstract

This report details our approach at training and optimizing neural Networks using data from the MNIST and MedMNIST datasets. We explore different architectures like Alexnet, ResNet and Xception. We detail the theoretical foundations of our methods and approaches, explaining why we used them and show our results and findings from experiments with the different approaches outlined. At the end we discuss and reflect on our approaches and findings.

1 Introduction

The emergence of deep learning has led to unprecedented advances in various fields, including medical image analysis. This project seeks to explore the fundamental principles of deep learning and to leverage its potential in a practical setting. Our investigation is divided into two parts: First, we focus on the classification of handwritten digits using the MNIST dataset (Lecun et al., 1998), followed by a more complicated challenge of classifying nine different tissue patterns within the PathMNIST dataset (Kather et al., 2018, 2019), a subset of MedMNIST (Yang et al., 2021). These tasks not only serve as a basis for understanding the mechanisms of deep learning, but also highlight the impact of the application of neural networks in medical diagnostics, emphasizing the relevance of our research.

The initial phase of our project was dedicated to learning the basics using the MNIST dataset, which was chosen for its many resources and tutorials to help us get started with deep learning. Our goal was to establish a basic understanding of the field by examining and comparing the results obtained from tuning various hyperparameters. In this phase, we developed a basic convolutional neural network (SimpleCNN) to introduce us to the architectures of neural networks and their ability to classify different digits.

[*]Degree: B.Sc. AI, matriculation #: 2018161
[†]Degree: B.Sc. AI, matriculation #: 2045368
[‡]Degree: B.Sc. AI, matriculation #: 2017990

The transition to the MedMNIST dataset, in particular the PathMNIST subset, represented a significant increase in the complexity of our project. This phase was crucial as it enabled us to apply and refine advanced techniques, such as experimenting with pre-trained models, testing a wide range of optimizers, and exploring different strategies for data pre-processing and augmentation. The PathMNIST subset was chosen to emphasize the critical importance of neural networks in medical image analysis. By tackling the classification of tissue patterns, we have not only explored some technical intricacies of deep learning, but also contributed to an area where such technologies could potentially revolutionize diagnostic methods.

This report is structured by first presenting some theoretical foundations that are crucial for understanding the methods used, including dropout layers, batch normalization and ReLU. Next, we explain the architectures of different deep learning models that were investigated in the project, from our initial SimpleCNN to more complex pretrained models such as AlexNet, ResNet and Xception (Chollet, 2017). Subsequently, a brief overview of the used datasets MNIST and MedMNIST is given, which form the basis for a deeper investigation. We then present the results obtained with each architecture, addressing the specific challenges. The discussion section provides a critical evaluation of our results and leads to a reflective conclusion about the lessons learned and the potential impact of our research on medical image analysis.

2 Methods

2.1 Theoretical Foundations

2.1.1 Deep Learning Pipeline

A generic deep learning pipeline is presented in the "Review of deep learning" by Alzubaidi et al. depicting a sample process of how a deep learning task could be approached step by step (Alzubaidi et al., 2021). In Figure 1 the pipeline proposed by Alzubaidi et al. can be seen. Various steps will be picked up again throughout this project report.

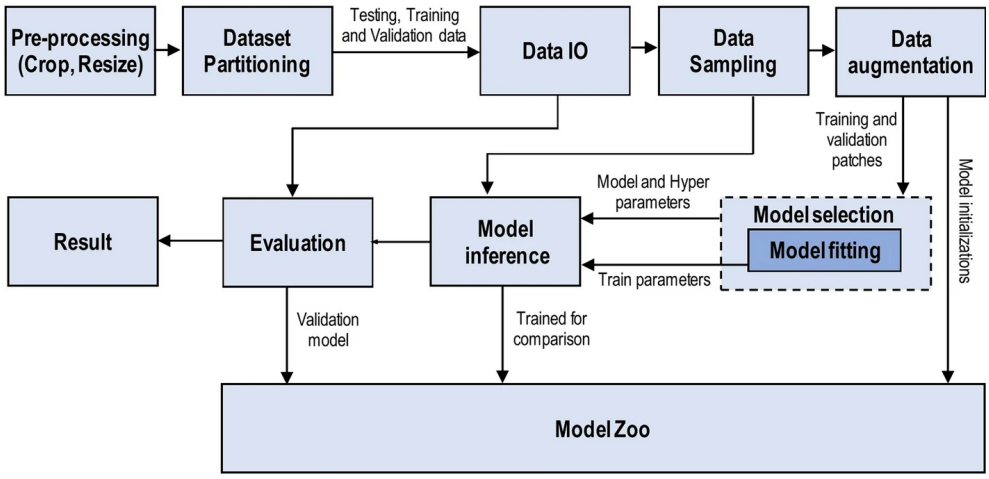


Figure 1: Generic deep learning pipeline as outlined byAlzubaidi et al.

2.1.2 Normalisation of the data

When working with pre-trained models, normalisation is commonly carried out to adjust for the distribution of the dataset that the model was originally trained on. This can be paired with cropping and resizing the images of the given dataset to fit the dimensions required as input to the pre-trained model. Such normalisation can be related to the first step in the pipeline proposed by Alzubaidini et al., namely "Pre-processing (Crop, Resize)" in Figure 1. The normalisation statistics for pre-trained models loaded via PyTorch are generally given as is the case for AlexNet(Hub, 2024). As an implicit

hypothesis, the normalisation of the data according to the model that's employed should enable a frictionless usage of said model.

2.1.3 Balancing of the Dataset

As Kaur et al. elucidate in their Systematic Review on Imbalanced Data Challenges in Machine Learning, an imbalanced dataset can lead to classifications being skewed in favour of the majority class. When working with multi-class classification problems, this applies to the classes with clearly more data points than the other classes. While measures like the Imbalanced Ratio (Kulkarni et al., 2020) do exist to quantify how unbalanced a dataset is, there commonly doesn't appear to be a strict cut-off value for when a dataset is deemed "unbalanced". Nonetheless, it's evident that unbalanced datasets must be dealt with, as Kaur et al. describe how the accuracy in these favour the majority class, yielding potentially better accuracy for the majority class while yielding lower accuracy for the minority class. Kaur et al. also present various methods which essentially rely on oversampling or undersampling classes, sometimes according to some measure (Kulkarni et al., 2020). In the generic pipeline for deep learning by Alzubaido et al., Data Sampling can also be found in the early stages of the proposed pipeline, see Figure 1 PyTorch provides various samplers, among them WeightedRandomSamplerdoc (2024), which allows to sample elements according to some given probabilities. In order to balance out a dataset using WeightedRandomSampler provided by PyTorch, the probabilities with which data from a class is drawn needs to be calculated. These probabilities can be determined by dividing 1 by the number of samples in a class. Following this theoretical foundation, the hypothesis is that a balancing of the data improves accuracy as compared to a model which learned on unbalanced data.

2.1.4 Data Augmentation

Small data size and lack of variation in the data is associated with false predictions and worse accuracy by Maharana et al.. Data quality issues plague many areas of Machine Learning. Maharana et al. note that by means of data augmentation, more data can be generated from limited amounts of data and there's also the potential to combat overfitting. Data augmentation is commonly assigned to the data preprocessing part of the deep learning pipeline. Common data augmentation techniques listed by Maharana include: flipping, rotation, noise, shifting, cropping, PCA jittering GAN, and WGAN. All these methods listed could, in theory, increase variation in the data and thus avoid overfitting, producing a model more capable of generalisation and capturing a plethora of features instead of being limited to a smaller pool of features of an unaugmented dataset. Considering this, the hypothesis that arises is that data augmentation will help with overall accuracy and overfitting, possibly leading to less false positives and more true positives across for instance a confusion matrix.

2.1.5 Batch Normalisation

Santurkar et al. describe how for Batch Normalisation additional layers are added to the architecture of the network which set the mean and variance of the output to zero and one respectively (Santurkar et al., 2018). In order to retain model expressivity the batch normalised inputs are also shifted and scaled based on trainable parameters. The idea of the Internal Covariate Shift is described by Santurkar et al. as one of the reasons batch normalisation was invented. Internal Covariate Shift, it is speculated, essentially is a change of the distribution of inputs to a layer in the network caused by an update of parameters to the previous layers. Furthermore, Internal Covariate Shift is hence presumed to have as a consequence a constant shift of the underlying training problem. However, this remains a controversial concept, as Santurkar et al. actually argue that batch normalisation does not reduce Internal Covariate Shift at all. The authors identify the key effect of batch normalisation to be the reparametrisation of the underlying optimisation problem in order to make it more stable. Despite presumably not dealing with the Internal Covariate Shift, batch normalisation thus potentially is a useful addition to a deep learning architecture. Extrapolating from that one arrives at the hypothesis that a network with batch normalisation performs better across given metrics as compared to a network without batch normalisation, ceteris paribus.

2.1.6 Evaluation Metrics

Hand et al. define various evaluation metrics (Hand et al., 2021) which are commonly found throughout the literature. To facilitate legibility, the following abbreviations are used: total number of

positives (P), total number of negatives (N), true positives (TP), true negatives (TN), false positives (FP) and false negatives (FN). A quick definition of these metrics shall be given here as proposed by Hand et al. (2021).

$$\text{Accuracy} = \frac{TP+TN}{P+N} \qquad \text{Precision} = \frac{TP}{TP+FP} \qquad \text{Recall} = \frac{TP}{TP+FN}$$

$$\text{F1-Score} = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall}$$

2.2 Model Architectures

2.2.1 SimpleCNN

Our SimpleCNN model contains two primary convolutional layers. The choice of this architecture was motivated by the goal of understanding the basic mechanisms of neural networks in processing and classifying image data. The model uses LeakyReLU activation to avoid the vanishing gradient problem and to ensure effective backpropagation even at small gradient values. The first convolution layer uses 32 filters with a kernel size of 3×3, a stride of 1 and 'same' padding, which preserves the dimension of the input images. This is followed by a max-pooling layer with a kernel size of 2, which aims to reduce the spatial dimensions while keeping the important features to optimize the model's ability to detect patterns without considerable data loss. Subsequently, a second convolution sequence increases the depth to 64 filters, improving the model's ability to extract more features. This setup is again followed by LeakyReLU activation and max-pooling, further refining the feature extraction process. The architecture concludes with a linear layer that maps the high-level features to the output classes, with a softmax function to interpret the outputs as class probabilities.

As our project progressed and our understanding deepened, we enhanced the initial SimpleCNN model by introducing an additional convolutional layer and incorporating batch normalization and dropout techniques, aiming for improved accuracy and generalization. The modified SimpleCNN architecture begins with a convolutional layer designed for single-channel (grayscale) images to match the format of the MNIST dataset. This layer, consisting of 32 filters with a kernel size of 3×3 and 'same' padding, is followed by batch normalization and ReLU activation, which promotes nonlinearity while maintaining normalization across the inputs of the network. A dropout rate of 0.25 after max pooling aims to mitigate overfitting by randomly omitting a portion of the features during the training process. As the model progresses, the second and third convolutional layers increase the filter count to 64 and then 128, respectively, each augmented with batch normalization, ReLU activation, max pooling, and a consistent dropout rate of 0.25. This architectural depth ensures the extraction of increasingly complex features essential for recognizing diverse patterns in handwritten digits. The concluding segment of the model comprises a fully connected layer transitioning from the convolutional output to 256 units, followed by batch normalization and a higher dropout rate of 0.5, further combating overfitting. The final linear layer maps these processed features to the ten class outputs corresponding to the MNIST dataset's digit categories.

The transition from the SimpleCNN model, which was tailored for the MNIST dataset, to the model customized for the PathMNIST challenge required some adjustments to account the different characteristics of the two datasets. The most important change was to adapt the input layer to include 3-channel RGB images for PathMNIST. This change is important to take advantage of the color information that is critical in medical imaging for identifying different tissue types. Despite this adaptation, the core architecture - consisting of convolutional layers, batch normalization, ReLU activation, and dropout layers - remained consistent between models.

For a detailed implementation of all versions of the model, refer to the Appendix Section A.1.

2.2.2 AlexNet

When the AlexNet architecture was published in 2012 (Krizhevsky et al., 2012), it (re-) ignited the field of Deep Learning. An illustration of the network's architecture as presented by Krizhevsky et al. in the original paper introducing AlexNet to the world is given in Figure 16. At the time, AlexNet was unusually deep, with eight layers in total, consisting of five convolutional layers and three fully connected layers. Back then, a network with this many layers represented a big computational challenge. The authors describe how the training using 90 cycles with a training set of 1.2 million

images and using two NVIDIA GTX 580 3GB GPUs took five to six days. Part of the new look consisted of the use of the Rectified Linear Unit (ReLU) function as the activation function and using dropout. The Rectified Linear Unit superseded the previously used tanh activation function. Using ReLU as opposed to tanh enabled the network to be trained several times faster. For the dropout method, the output of each hidden neuron is set to zero according to a probability - in the case of the original AlexNet paper 0.5 was chosen (Krizhevsky et al., 2012). The neurons which are subject to having their weight set to zero subsequently neither participate in the forward pass nor in the backward pass. Generally, the authors describe that dropout is intended to yield a model which learns more robust features, albeit at a cost of leading the iterations needed to converge to almost double. However, Krizhevsky et al. state that without dropout their network would exhibit substantial overfitting.

2.2.3 ResNet

The Residual Network architecture (ResNet) was first introduced by (He et al., 2015) in their paper "Deep Residual Learning for Image Recognition". This marked a big advancement in deep learning for computer vision, as previous model architectures did not have the ability to effectively scale with more layers. These models would, among other issues, run into a vanishing or exploding gradient problem, making deeper layers less usefull and impacting performance. To combat this, ResNet introduces a solution using residual blocks, as shown in Figure 2. The input X goes through two paths. One is called a skip connection and carries the original input to a junction point. The other carries the input through multiple convolutional layers to process it. Following the first layer, the input is passed through a ReLu function, introducing non-linearity to the model. Then it is passed trough a second convolutional layer, which returns a residual function. In the end, the learned residual function is added to the original input.

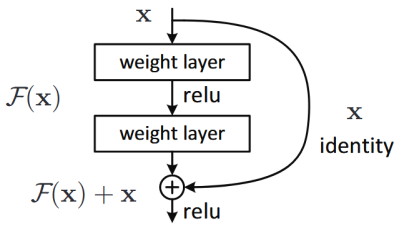


Figure 2: Residual Block as outlined by He et al..

ResNet18 and ResNet32: Using the block shown in Figure 2, the researchers built two networks, that are now commonly used. ResNet18, containing 18 and ResNet32, containing 32 layers, made of these two-layer residual blocks. These networks are designed for simpler tasks and limited training resources. They offer great performance relative to their size.

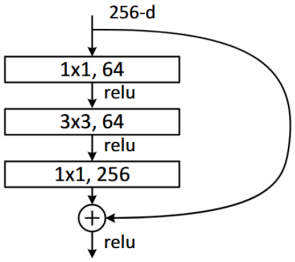


Figure 3: Bottleneck Block as outlined by He et al..

ResNet50, ResNet101 and ResNet152: In order to further upscale this technology, the researchers introduced a modification to the two-layer structure. As seen in Figure 3, the researchers created a bottleneck block. This block features two convolutional layers with a 1×1 convolution and a middle

layer with a 3×3 convolution. The first layer down scales the input, which is then forwarded to the next layer with a 3×3 convolution. This middle layer is the core of the bottleneck block, where the actual feature extraction happens. Then the third layer upscales the input back to its original size and returns the output for the block. These Models work well for large and complicated datasets but are more computationally demanding than their two-layer counterparts.

2.2.4 Xception

To understand the Xception model, one must become familiar with depth-separable convolutions. This technique is a convolution operation that divides into two different stages to increase computational efficiency. Initially, a depth-wise convolution applies a single filter to each input channel. Subsequently, a point-wise convolution - characterized by a 1×1 kernel - combines the outputs from the depth-wise step across the channels. This factorization significantly reduces the number of parameters and calculations and enables more efficient training. Batch normalization follows each convolution and promotes stable learning by normalizing the ReLU activations of the layer.

As seen in Figure 17 in the Appendix the architecture of Xception, as detailed by (Chollet, 2017), is structured into three primary flows. The entry flow prepares the network with initial convolutions and pooling to create condensed feature maps from the input images. It starts with two standard convolutions, followed by a series of separable convolutions that increase the depth while compressing spatial dimensions. This is achieved using a combination of 1×1 convolutions for channel-wise feature processing and 3×3 convolutions for capturing spatial information, each followed by max pooling to halve the feature map dimensions progressively. This step reduces the initial input image size from 299×299×3 to 19×19×728. Central to Xception's design is the middle flow, which repeatedly applies depthwise separable convolutions to process and refine the features. This part of the network is designed to be repeated eight times, allowing the model to learn increasingly complex patterns without affecting the dimensions. The exit flow then expands the feature maps through additional separable convolutions, incorporating a mix of channel-wise and spatial feature extraction before concluding with global average pooling. This step reduces each map to a single vector, capturing the essence of the input data in a form suitable for classification. The network concludes with a logistic regression layer, such as softmax, to output the final class probabilities.

3 Experiments

Our experimental framework focuses on the use of two central datasets: MNIST and MedMNIST, each of which is briefly introduced here. This section also describes the experiments performed with different model architectures and presents the resulting findings.

3.1 MNIST

The "Modified National Institute of Standards and Technology" dataset (Lecun et al., 1998) comprises a collection of 70,000 handwritten digits carefully divided into a training set of 60,000 images and a test set of 10,000 images. Each digit is represented in a grayscale image of 28×28 pixels, and offers a wide range of styles and shapes. This dataset is widely recognized for its simplicity and effectiveness in benchmarking classification algorithms, making it an ideal starting point for those new to deep learning. Some examples of the dataset can be seeen in Figure 4

Figure 4: Example grayscale images of the MNIST dataset

3.2 MedMNIST

MedMNIST, a more specialized and challenging dataset than MNIST, is tailored for medical image classification tasks. It extends the concept of handwritten digit classification to a diverse range of medical imaging modalities, including dermatology or radiology. Unlike MNIST's uniform format,

MedMNIST encompasses 12 subsets for 2D and 6 subsets for 3D data. For our project, we focused on the PathMNIST (Kather et al., 2018, 2019) subset, which includes "100,000 non-overlapping image patches from hematoxylin and eosin stained histological images, and a test dataset [...] of 7,180 image patches from a different clinical center" (Yang et al., 2021). The images could be classified into nine different types of tissues.

Initially, the images were of high resolution ($3 \times 224 \times 224$ pixels), but the authors of MedMNIST resized them to $3 \times 28 \times 28$ pixels. The 100,000 training images were then originally divided into training and validation sets in a 9:1 ratio. Some examples of the various images can be seen in Figure 5.

Figure 5: Example images of the MedMNIST dataset.

The PathMNIST subset provides a unique challenge by introducing the complexity of medical image analysis. It requires the use of advanced deep learning techniques and models to accurately classify different types of tissue, making it an excellent progression from the simpler MNIST dataset.

3.3 SimpleCNN

After establishing the basic architecture of our SimpleCNN (as described in Section 2.2.1), our next goal was to effectively classify the first digits of the MNIST dataset. This required the selection of initial hyperparameters to begin the training and evaluation process. Without prior benchmarks, we initially decided on a batch size of 8, set the number of epochs to 5 and a learning rate of 0.1. However, this initial configuration led to less satisfactory results, which made us to revise our hyperparameters to improve performance.

The training process was completed relatively quickly so that we could implement a function to test the hyperparameters. This required us to set default values for each training and evaluation session, with certain values being adjusted at each iteration. For example, we set the batch size to 32, the learning rate to 0.001 and the epochs to 15. In subsequent runs, we changed one hyperparameter at a time, while keeping the others constant. This means that we ran training sessions with learning rates of 0.001, 0.01 and 0.1, respectively, while keeping the default values for batch size and epochs. Similarly, we experimented with different batch sizes (16, 32, and 64) and varied the number of epochs (5, 10, 15, and 20), to further understand their individual contributions to the model's accuracy and training efficiency.

After analyzing the experimental data (see Appendix, Table 5), we discovered the most effective hyperparameters for our SimpleCNN model, and interestingly, we found that the default values were already optimal. We obtained these results with the original SimpleCNN architecture. Subsequently, we applied these hyperparameters to train an updated version of the SimpleCNN. To assess the impact of the changes, we compared the performance of the original and updated architectures. The comparative analysis is shown in Figure 6.

For the second challenge, which focused on the PathMNIST dataset, our approach exclusively used the adapted version of our SimpleCNN architecture due to its better performance. To combat overfitting, we introduced an early stopping mechanism. The core of this experiment was to assess the impact of incorporating batch normalization and dropout techniques. We conducted three separate trials: one with both batch normalization and dropout, one with only dropout, and another with only batch normalization. The outcomes of these trials can be seen in Figure 7.

3.4 AlexNet

First of all must be noted that mainly AlexNet was chiefly used here in order to test the hypotheses about balancing the data set, data augmentation and finally batch normalisation. This is due to the fact that even a simple ResNet18 achieves accuracies so high, that it becomes difficult to discern

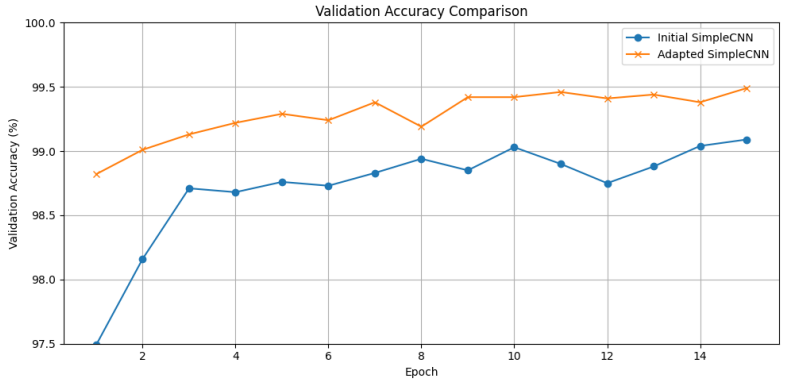


Figure 6: Comparison of validation accuracy across epochs between the initial and the adapted SimpleCNN model, highlighting the overall improved performance of the adapted model.

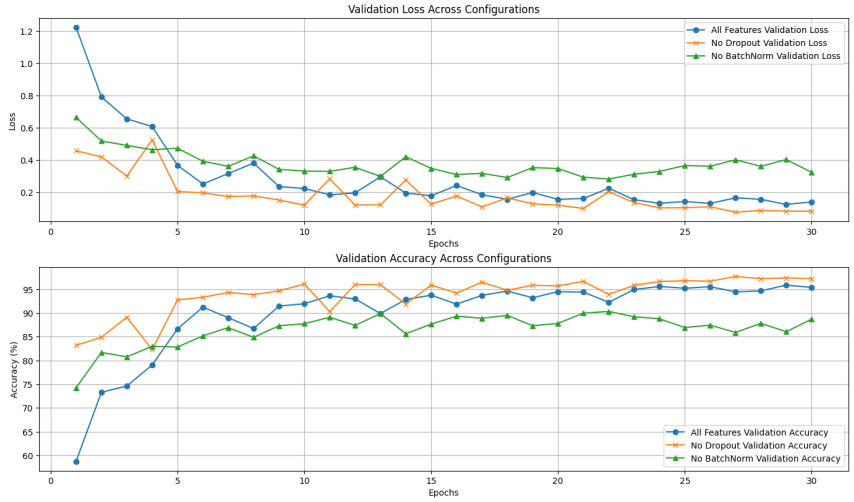


Figure 7: Performance comparison of SimpleCNN variations on the PathMNIST dataset over 30 epochs.

effects in for example a confusion matrix. In confusion matrices, the row number corresponds to the (index of a) class of the true label, while the column number corresponds to the (index of a) class of the predicted label. On the trace, which includes the diagonal elements, the true positives can be found. A column, excluding the element which belongs to the trace, contains the false positives. For the false negatives, one simply needs to look at a row and exclude the element which belongs to the trace of the confusion matrix. As can be seen in Fig. 8 almost all the elements, excluding the trace, range from single-digit numbers, including many zeros, to the low double-digit numbers. This model used a simple ResNet18 implementation, using the pre-trained version provided by PyTorch. The model achieved an accuracy of about 97.57%, with a training split size of 0.8, a learning rate of 0.01, SGD as the optimiser, batch size 64 and early stopping with patience 5 that was triggered in epoch 41. While the confusion matrix would certainly display the effects of, for example, balancing out the dataset, with such high accuracies it's difficult to assign meaning to relative changes which seem large What seems like a meaningful improvement will yield a barely changed accuracy. Furthermore, it's much harder to visualise the effect the application of a method has when almost all the predictions are correct already, so to speak. Taking these aspects into consideration, AlexNet proved to provide accuracies which, from a qualitative standpoint, seemed to be fit for visualising effects in a confusion matrix but also not too low as to provide inconclusive results to whether the network has actually learned anything. Thus, AlexNet seems an appropriate choice for evaluating the hypotheses.

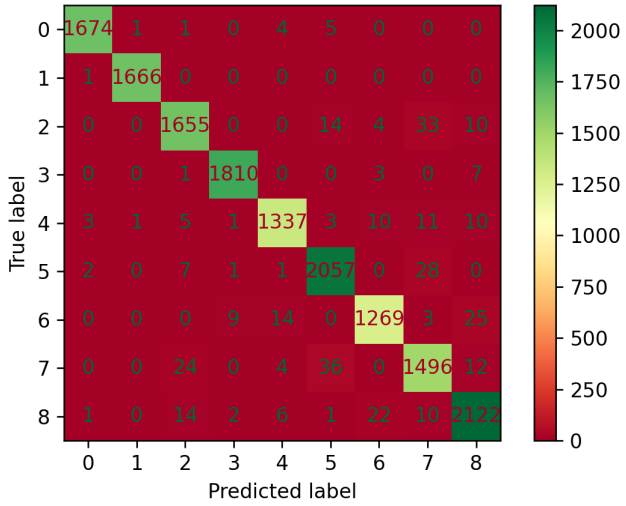


Figure 8: Confusion Matrix for simple ResNet 18

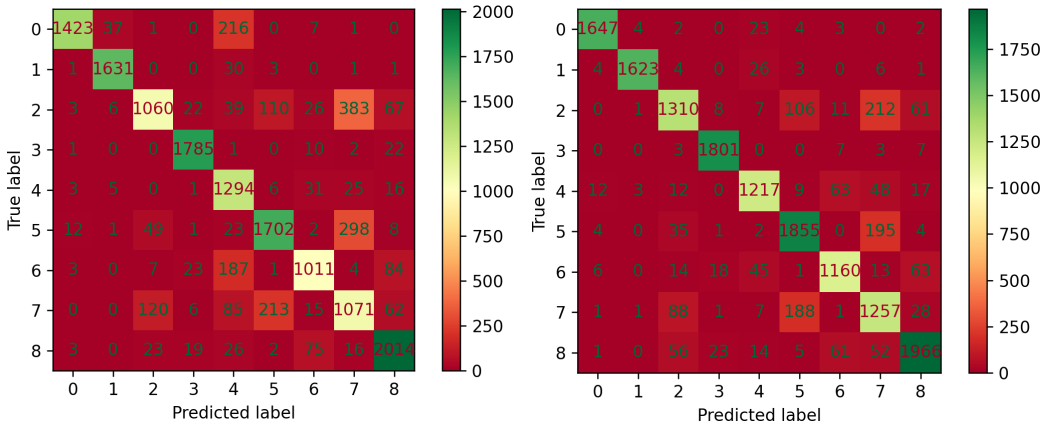
3.4.1 Experiment Balancing

Various AlexNet models were trained in order to gather evidence regarding the hypothesis that balancing of the data improves accuracy compared to a model which learned on unbalanced data. For this, the only adjustments that were made were to either enable or disable balancing and use early stopping or leave it out. All the models that were trained used a learning rate of 0.01, Adagrad as the optimiser, a split size of 0.8 and a batch size of 64. The models that didn't use early stopping were trained for 20 fixed epochs. For the early stopping version, a patience of 8 was used while a maximum number of epochs was given with 50. The results as well as the configuration for the balancing and early stopping are listed in Table 1.

Table 1: AlexNet before and after balancing out the class distribution

Validation Accuracy (in%)	Balancing	Epochs	Early Stopping
84.16	False	20	False
89.63	True	20	False
88.94	False	44	True
91.18	True	20	True

Evidently, the validation accuracy improved with balancing as compared to using no balancing both for the case of early stopping not employing early stopping. For the version with the fixed number of epochs set at 20, an improvement in validation accuracy of 5.47% was observed. In the case of employing early stopping, an improvement in validation accuracy of 2.24% was observed. It is to be noted that for early stopping, the unbalanced version stopped after more than twice as many epochs as the balanced version while still achieving a lower validation accuracy. The confusion matrices for the models without early stopping are given in Fig. 9. A reduction in false predictions can be observed broadly across the board. It's especially noticeable for the true label 0 and the predicted label 4. Here, the false prediction shrinks by almost an entire order of magnitude from 216 to 24. Some peculiarities remain as the false predictions for true label 2 and the predicted labels 5 and 7 remain large compared to the rest, with the false predictions for 5 barely changing from 110 to 106 and the false predictions for 7 almost halving from 383 to 212. For true class 7 and predicted classes 2 and 5 the false positives also remain large relative to the other ones. In conclusion, this experiment yields evidence supporting the initial hypothesis that balancing of the data improves accuracy as compared to a model which learned on unbalanced data. Improvement is observed to happen broadly across the board, with some outliers remaining.



(a) No Balancing (b) Balancing

Figure 9: Comparison confusion matrices no balancing vs balancing

3.4.2 Experiment Data Augmentation

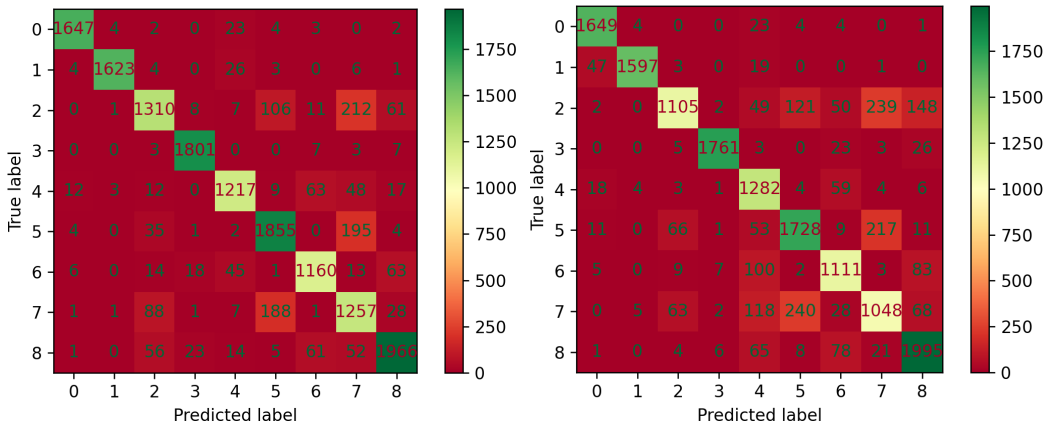
Going back to Section 2.1.4, the hypothesis suggested is that data augmentation will aid with overall achieving more true positives. To test this hypothesis, several models were trained, with balancing having been applied to all of the models. A distinction is again made between using early stopping or not. A patience of 8 has been used and a maximum number of 50 epochs for the models with early stopping. The best model is saved during training and the results are according to the best model. For the hyperparameters, the learning rate is 0.01, Adagrad is again being used as optimiser, the batch size is 64, split size 0.8. An overview over the results is given in Table 2. As can be seen, a reduction

Table 2: AlexNet before and after augmenting the data

Validation Accuracy (in%)	Augmentation	Epochs	Early Stopping
89.63	False	20	False
86.00	True	20	False
91.18	False	20	True
92.88	True	49	True

in accuracy has actually occurred when a fixed number of epochs was used. A reduction of 3.63% occurred there. On the other hand, for the early stopping, accuracy increased by 1.7%. However, it is to be noted that for the model with early stopping but without augmentation, coincidentally the training went on for 28 epochs with the best model being found in epoch 20, while for the model with augmentation the entire epochs were exhausted and the bets model found in epoch 49. These results do not unequivocally support or undermine the initial hypothesis. Looking at the confusion matrix, no clear conclusion can be drawn either. From the confusion matrix without early stopping, presented in Figure 10, it can be seen that the variation introduced by the data augmentation seemingly leads to drastically worse classification, for example for the true label 7 and predicted label 4. Here, without data augmentation only 7 images are incorrectly classified to be of label 4 when in reality they're label 7, but with data augmentation 118 images are incorrectly classified to be of label 4 when in reality they're label 7. That's a increase of more than 16 times, granted that the number of incorrectly classified data points there was initially just 7, a single-digit number. However, one could speculate whether the augmentation in this case lead to the model learning wrong features, thus causing a decline in validation accuracy.

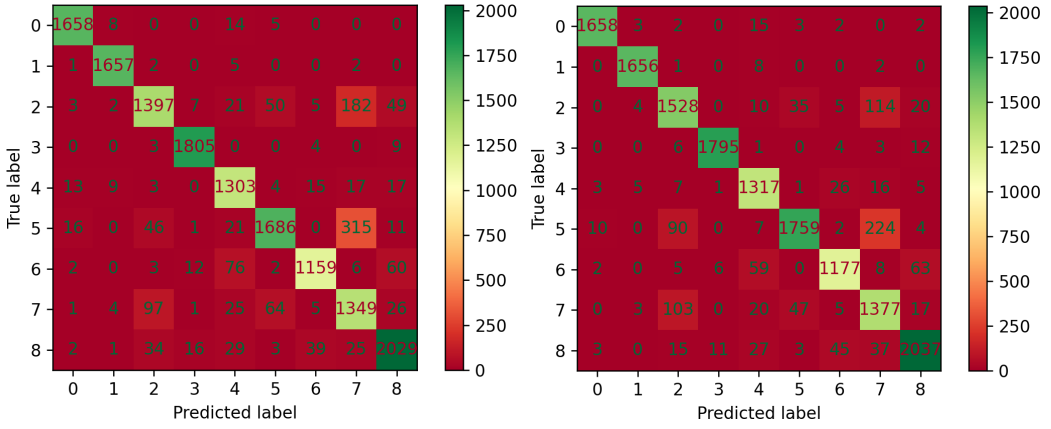
Taking a look at the models with early stopping, largely a broad decrease in false classified images can be observed. A suspicious rise in misclassifications is also present. Here, for the true label 5 and predicted label 4, before augmentation 46 images were predicted and after augmentation 90, which is almost an exactly two-fold increase. This might be an indication for a wrong feature being learned but



(a) No Augmentation (b) Augmentation

Figure 10: Comparison confusion matrices no augmentation vs augmentation

apart from this one entry, no other entry provides evidence in favour of this speculation. In conclusion it can be noted that the results for models without early stopping and with early stopping are contrary, with the former providing evidence against the initial hypothesis that data augmentation will aid with overall achieving more true positives while the latter would support such a claim.



(a) No Augmentation (b) Augmentation

Figure 11: Comparison confusion matrices no augmentation vs augmentation with early stopping

3.4.3 Experiment Batch Normalisation

For this experiment, the AlexNet architecture provided by PyTorch Hub (2024) was modified to include batch normalisation layers after every convolution layer. The code for this is included in the appendix. Continuing from Section 2.1.5, the hypothesis is that when training a model (in this case AlexNet) without batch normalisation layers and with batch normalisation layers, the performance across the evaluation metrics should improve ceteris paribus. To put this hypothesis to test, one AlexNet model was trained without batch normalisation layers and two models implemented to include batch normalisation layers were trained. Training was done using a learning rate of 0.001, a balanced and augmented dataset, split size of 0.8, batch size of 64, using Adagrad as the optimiser, having determined the maximum number of epochs as 150 and including early stopping with patience 8. During training, the early stopping criterion was hit for all the models 1,2,3 and the best model was found in epoch 34, 63 and 27 respectively. The results of this training can be seen in Table 3. While

model 2 achieved an increase of validation accuracy of 0.21% as compared to model 1, model 3 actually had a lower validation accuracy as compared to model 1, by 0.87%. All in all, the increase or decrease doesn't exceed 1% and also doesn't fall short of 0.1%. Qualitatively speaking, one might call these changes to the validation accuracy small. This begs the question whether batch normalisation has really affected what the model learned here in a meaningful way. Such a change can't be made out in the validation accuracy. A possible explanation for why no such change is seen could be that AlexNet simply is not deep enough for batch normalisation to exert a big influence. Santurkar et al. describe applying batch normalisation to networks with a depth of 25 layers (Santurkar et al., 2018). AlexNet falls short of even 10 layers. In conclusion, the experiment of applying batch normalisation to the AlexNet architecture did not result in evidence supporting the intial hypothesis that batch normalisation would enhance performance.

Table 3: AlexNet before and after augmenting the data

Validation Accuracy (in%)	Batch Normalisation	Model #
97.29	False	1
97.50	True	2
96.42	True	3

3.5 ResNet

Setup and reproducibility: All ResNet models presented here were trained on the same PC using Windows 11, Python 3.10.11 and an Nvidia RTX4090 graphics card. In order to ensure a consistent split for training and test data, the random seeds for numpy and torch were set to 22. No such steps were taken to make the training itself, using cuda, more reproducible, leading to small but noticeable variations in the results between different training runs using the same parameters.

Hyperparameters: In the first stages of our experimentation, we tried to identify the best hyperparameters by making multiple test runs using ResNet18 and identifying the best performances. With this approach, we aimed to get a better understanding of the impact different parameters have, before starting to use larger models that take significantly more time to train. All of our tests here used the previously discussed data augmentation. As mentioned before, multiple training cycles with the same parameters would lead to marginally different results, and in very few extreme cases, even to an accuracy difference of up to 2%. Due to this, some of the following results might not be representative, especially if the margin between two parameters is very small. All Training was done with enabeled dataset balancing and data-augmentation allready discussed To decide on the optimizer, we compared three different ones: Adagrad, Adam, NAdam and SGD. Prior to this, our expectation was, that Adam would be the optimal choice and that SGD would perform the worst. Our test results (worst to best) were the following: SGD with an average loss = 0.08, followed by NAdam and Adam both having an average loss = 0.06, and finally Adagrad with an average loss of 0.035. Based on this we chose Adagrad for our further training. For our learning rate, we used an adaptive learning rate scheduler for most of the later training. Despite Adagrad inherently adjusting the learning rate, we still found that the results using a learning rate scheduler were either better over many epochs or, at worst, made no difference. In these early stages, we found that an optimal starting learning rate was somwhere between 0.001 and 0.0001. After implementing the learning rate scheduler, we defaulted to a starting learning rate of 0.0005. For our Training and validation split, we chose 80/20. Here we did not do any testing, as the number seemed adequate for the size of the dataset. The batch size did not have any noticeable impact on the performance of the model. We ended up using a batch size of 64 for all the training, as it proved to be more practical with the larger models. Changing it to 100, for example, would lead to a ResNet152 using more than 21 GB of Video RAM. To prevent overfitting but still have the option to train our models until they converge, we implemented an early stopping criterion. Here, the new best model would be saved each time it improved based on validation accuracy and then stop training after a set number of epochs without improvement. As we wanted to make sure our models had converged, the maximum number of epochs for our final models was set to 200 and the early stopping patience to 20 epochs. For the normalization of the images, we calculated the

mean and standard deviation of the whole dataset and normalized it based on these results. Below, in table 4, you can see our final hyperparameters.

Hyper-Parameter	Value
Learning rate	0.0005
Batch size	64
Split	80/20
Optimizer	Adagrad
Max epochs	200

Table 4: Overview of Hyperparameters

ResNet18: Training ResNet18 with our optimized hyperparameters already gave us great results. Our very first attempts, without any optimisations lead to an validation accuracy of about 97% as seen in figure12.

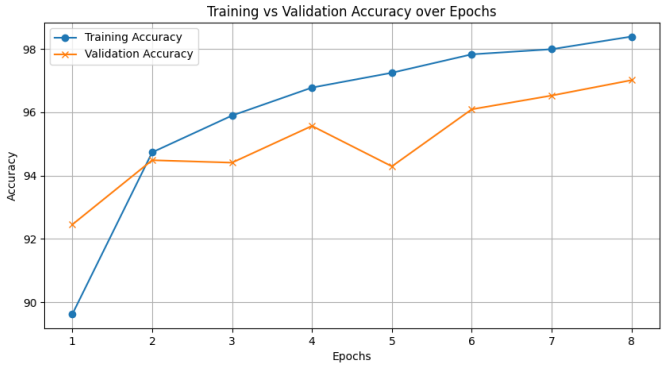


Figure 12: ResNet18 first results.

After training with our optimized hyperparameters over more epochs, we already reached an validation accuracy of 99.10% as can be seen in figure13

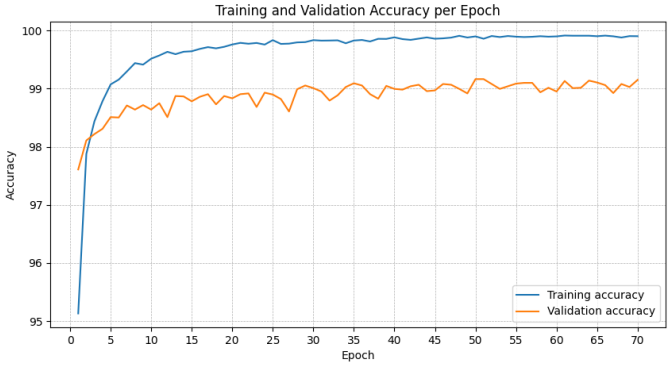


Figure 13: ResNet18 optimised results.

ResNet152: Next, we wanted to use deeper models. For this we mostly skipped the models between ResNet18 and ResNet152, as we realized that, while they provided some performance increase, it was too little to focus on them. For the training of ResNet152 we employed the same hyperparameters (table 4) as for the optimized ResNet18 (figure13). The results here were a clear but small improvement, as we are already over 99% accuracy. In figure14, you can see that our ResNet152 performed at an 99.40% training accuracy.

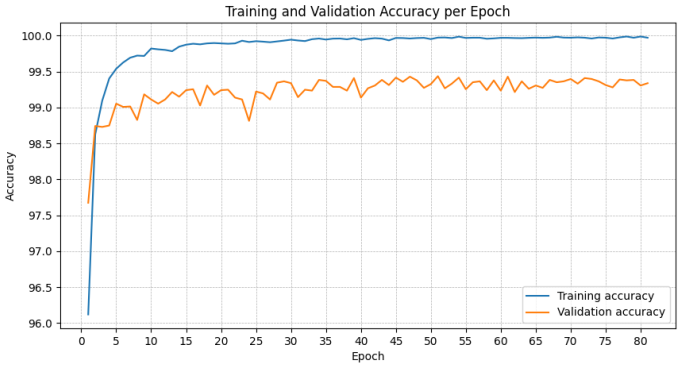


Figure 14: ResNet18 optimised results.

Ensemble Methods: This is a very complex topic with many different approaches. The core idea is to find a way to combine multiple models together in order to achieve better results. This can be done training one model on the output of multiple different models, train multiple models in a row to improve upon the output of the previous model, and many other ways. As we only wanted to test, if we could get a little bit more performance out of our models, we decided on implementing the most simple version, which just averages the output of different models. Optimally, this would be weighted based on the models strengths and weaknesses. Unfortunately, the strengths and weaknesses for our models were consistent trough all that we trained. All of them struggled with the same classes. Never the less, this still brought some improvements. The performance for these ensemble models we only measured on the test data provided for the Kaggle challenge. There our ensemble of two ResNet152 models, trained with the same parameters outperformed the individual models (each about 99.39% accuracy) and was our best submission with an accuracy of 99.45%.

3.6 Xception

The implementation of the Xception architecture presented us with particular challenges right from the start, mainly because Xception is not included in the PyTorch model library. We used the "pretrainedmodels" library (Cadene, 2018) to gain access to a pre-trained Xception model, although this is outdated (last update in April 2020). While the library's pretrained Xception model was operational, its use led to some further complications. One major hurdle was adjusting the input size to the Xception default requirement of 299×299 pixels, which is a significant increase from PathMNIST's original 28×28 pixels. Attempting to upscale images to this size proved to be computationally intensive and resulted in crashes, both on local PCs and when using Google's V100 GPUs in Google Colab (Google, 2024). To mitigate this problem, we initially resized the images to a more manageable 168×168 pixels. This adjustment allowed us to maintain the batch size of 64, which is consistent with our previous models for PathMNIST. However, the training time for a single epoch with a batch size of 64 was still very long - about 16 minutes on a V100 GPU via Colab and about 4 hours on local machines. Given the unreliability of Colab due to possible interruptions when GPUs are not available, as well as the cost of using a V100 GPU, we further reduced the stack size to 32. This adjustment reduced the epoch training time to about 10 minutes and allowed for a smoother local training process.

For training the Xception model, we used the AdamW optimizer (Loshchilov and Hutter, 2019), known for its effectiveness in handling weight decay and improving generalization by modifying the traditional Adam optimizer's handling of the learning rate.

With these changes, we were able to successfully run our experiments with the Xception model. The performance results are shown in Figure 15:

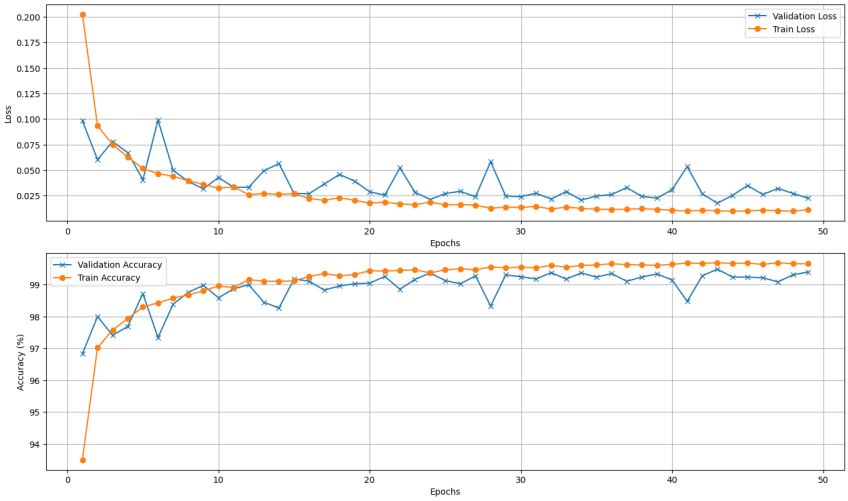


Figure 15

4 Discussion

4.1 SimpleCNN

In our initial experiments with the MNIST dataset using the SimpleCNN model, we compared the effectiveness of the ReLU and LeakyReLU activation functions. We found no significant difference in performance between them, which led us to continue with ReLU due to the simplicity of our model. For the PathMNIST challenge, our experiments with SimpleCNN showed that using only batch normalisation yielded slightly better results than combining it with dropout layers, suggesting that the latter might introduce unnecessary complexity that reduces model performance in this particular task.

4.2 ResNet

During the training we observed a slight but consistent difference between the validation accuracy after training and the final accuracy on our unlabeled dataset for Kaggle. One idea is, that the dataset used for our submission has more instances of the problematic classes our models struggled with. Over all we managed to improve the performance of our models by a lot, when compared to our first attempts. For the future we should put bigger focus on reproducibility and consistency when doing multiple training runs to test differences between hyperparameters. Our choice of optimizer here, while delivering good results, might have been just a statistical anomaly during testing and it would have made sense to do more in-depth testing on different optimizers. One other approach that might be worth looking in to is to minimize the non-deterministic behaviour duric training with cuda. One way would be to set CuDNN in PyTorch to deterministic mode. Here it would be interesting to see how much of an performance impact this has on the training. We also observed that deeper models would take many more epochs than smaller models until they converged. Some instances of ResNet152 trained for 80 epochs, before early stopping was triggered. Here, the question could be raised how important the gained improvement is, compared to smaller models. It would also be interesting to see how these performance differences would change with larger datasets. Another path that we would like to look into more are ensemble methods. A different implementation, or just the combination of more models of different architectures with our current approach might lead to even better results.

4.3 Xception

Using the Xception architecture for the PathMNIST dataset significantly improved the results, demonstrating the value of deeper, more complex models for processing complicated datasets. However, this came at the expense of computational resources, emphasising the need to strike a balance between the complexity of the model and the practicalities of computational effort. This

experience highlighted the critical trade-offs in deep learning between achieving high accuracy and coping with resource constraints, particularly in the context of complex medical imaging tasks.

5 Conclusion

//TODO

References

L. Alzubaidi, J. Zhang, A. J. Humaidi, A. Al-Dujaili, Y. Duan, O. Al-Shamma, J. Santamaría, M. A. Fadhel, M. Al-Amidie, and L. Farhan. Review of deep learning: Concepts, cnn architectures, challenges, applications, future directions. *Journal of big Data*, 8:1–74, 2021.

R. Cadene. Pretrained models for pytorch. `https://github.com/Cadene/pretrained-models.pytorch`, 2018. Accessed: 14.01.2024, PyPI link: `https://pypi.org/project/pretrainedmodels`.

F. Chollet. Xception: Deep learning with depthwise separable convolutions, 2017.

Google. Google colaboratory. `https://colab.research.google.com`, 2024. Accessed: 14.01.2024.

D. J. Hand, P. Christen, and N. Kirielle. F*: an interpretable transformation of the f-measure. *Machine Learning*, 110(3):451–456, 2021.

K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition, 2015.

P. Hub. AlexNet — pytorch.org. `https://pytorch.org/hub/pytorch_vision_alexnet/`, 2024. [Accessed 27-02-2024].

J. N. Kather, N. Halama, and A. Marx. 100.000 histological images of human colorectal cancer and healthy tissue, 2018. URL `https://doi.org/10.5281/zenodo.1214456`.

J. N. Kather, J. Krisam, P. Charoentong, T. Luedde, E. Herpel, C.-A. Weis, T. Gaiser, A. Marx, N. A. Valous, D. Ferber, et al. Predicting survival from colorectal cancer histology slides using deep learning: A retrospective multicenter study. *PLoS Medicine*, 16(1):e1002730, 2019.

H. Kaur, H. S. Pannu, and A. K. Malhi. A systematic review on imbalanced data challenges in machine learning: Applications and solutions. 52(4), aug 2019. ISSN 0360-0300. doi: 10.1145/3343440. URL `https://doi.org/10.1145/3343440`.

A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. Burges, L. Bottou, and K. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012. URL `https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf`.

A. Kulkarni, D. Chong, and F. A. Batarseh. Foundations of data imbalance and solutions for a data democracy. In *Data democracy*, pages 83–106. Elsevier, 2020.

Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998. doi: 10.1109/5.726791.

I. Loshchilov and F. Hutter. Decoupled weight decay regularization, 2019.

K. Maharana, S. Mondal, and B. Nemade. A review: Data pre-processing and data augmentation techniques. *Global Transitions Proceedings*, 3(1):91–99, 2022.

S. Santurkar, D. Tsipras, A. Ilyas, and A. Madry. How does batch normalization help optimization? *Advances in neural information processing systems*, 31, 2018.

WeightedRandomSamplerdoc. torch.utils.data &x2014; PyTorch 2.2 documentation — pytorch.org. `https://pytorch.org/docs/stable/data.html`, 2024. [Accessed 27-02-2024].

J. Yang, R. Shi, and B. Ni. Medmnist classification decathlon: A lightweight automl benchmark for medical image analysis. In *IEEE 18th International Symposium on Biomedical Imaging (ISBI)*, pages 191–195, 2021.

Declaration of Authorship

Ich erkläre hiermit gemäß § 9 Abs. 12 APO, dass ich die vorstehende Projektarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Des Weiteren erkläre ich, dass die digitale Fassung der gedruckten Ausfertigung der Projektarbeit ausnahmslos in Inhalt und Wortlaut entspricht und zur Kenntnis genommen wurde, dass diese digitale Fassung einer durch Software unterstützten, anonymisierten Prüfung auf Plagiate unterzogen werden kann.

Bamberg, February 28, 2024

(Place, Date)	(Signature)

Bamberg, February 28, 2024

(Place, Date)	(Signature)

Bamberg, February 28, 2024

(Place, Date)	(Signature)

A Appendix

A.1 SimpleCNN architectures

Initial version of our SimpleCNN, including two convulutional layers:

```
class SimpleCNN(nn.Module):
    def __init__(self, num_classes=10):
        super(SimpleCNN, self).__init__()
        self.conv1 = nn.Sequential(
            nn.Conv2d(
                in_channels = 1,
                out_channels = 32,
                kernel_size = 3,
                stride=1,
                padding="same"
            ),
            nn.LeakyReLU(),
            nn.MaxPool2d(kernel_size=2),
        )
        self.conv2 = nn.Sequential(
            nn.Conv2d(32,64,3,1,"same"),
            nn.LeakyReLU(),
            nn.MaxPool2d(kernel_size=2),
        )
        self.out = nn.Linear(64*7*7, num_classes)

    def forward(self, x):
        x = self.conv1(x)
        x = self.conv2(x)
        x = x.view(-1, 64*7*7)
        output = self.out(x)
        return torch.log_softmax(output, dim=1)
```

Structure of the improved version of the SimpleCNN using three convolutional layers, Batch normalization and Dropout:

```
class SimpleCNN(nn.Module):
    def __init__(self, num_classes=10):
        super(SimpleCNN, self).__init__()
        self.conv1 = nn.Sequential(
            nn.Conv2d(1, 32, kernel_size=3, stride=1, padding="same"),
            nn.BatchNorm2d(32),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2),
            nn.Dropout(0.25)
        )
        self.conv2 = nn.Sequential(
            nn.Conv2d(32, 64, kernel_size=3, stride=1, padding="same"),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.MaxPool2d(2),
            nn.Dropout(0.25)
        )
        self.conv3 = nn.Sequential(
            nn.Conv2d(64, 128, kernel_size=3, stride=1, padding="same"),
            nn.BatchNorm2d(128),
            nn.ReLU(),
            nn.MaxPool2d(2),
            nn.Dropout(0.25)
        )
        self.fc1 = nn.Linear(128 * 3 * 3, 256)
        self.fc_bn = nn.BatchNorm1d(256)
        self.dropout_fc = nn.Dropout(0.5)
        self.fc2 = nn.Linear(256, num_classes)

    def forward(self, x):
        x = self.conv1(x)
        x = self.conv2(x)
        x = self.conv3(x)
        x = x.view(-1, 128 * 3 * 3)
        x = F.relu(self.fc_bn(self.fc1(x)))
        x = self.dropout_fc(x)
        x = self.fc2(x)
        return torch.log_softmax(x, dim=1)
```

Structure of the improved version of the SimpleCNN for the PathMNIST dataset:

```python
class SimpleCNN(nn.Module):
    def __init__(self, num_classes=10):
        super(SimpleCNN, self).__init__()
        self.conv1 = nn.Sequential(
            nn.Conv2d(3, 32, kernel_size=3, stride=1, padding="same"),
            nn.BatchNorm2d(32),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2),
            nn.Dropout(0.25)
        )
        self.conv2 = nn.Sequential(
            nn.Conv2d(32, 64, kernel_size=3, stride=1, padding="same"),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.MaxPool2d(2),
            nn.Dropout(0.25)
        )
        self.conv3 = nn.Sequential(
            nn.Conv2d(64, 128, kernel_size=3, stride=1, padding="same"),
            nn.BatchNorm2d(128),
            nn.ReLU(),
            nn.MaxPool2d(2),
            nn.Dropout(0.25)
        )
        self.fc1 = nn.Linear(128 * 3 * 3, 256)
        self.fc_bn = nn.BatchNorm1d(256)
        self.dropout_fc = nn.Dropout(0.5)
        self.fc2 = nn.Linear(256, num_classes)

    def forward(self, x):
        x = self.conv1(x)
        x = self.conv2(x)
        x = self.conv3(x)
        x = x.view(-1, 128 * 3 * 3)
        x = F.relu(self.fc_bn(self.fc1(x)))
        x = self.dropout_fc(x)
        x = self.fc2(x)
        return torch.log_softmax(x, dim=1)
```

A.2 Data Transformation and Augmentation

Normalization for ImageNet and Data Augmentation techniques used.

```python
preprocessing = transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
])

# augmentations to be applied
augmentations = v2.Compose([
    v2.RandomHorizontalFlip(0.1),
    v2.RandomVerticalFlip(0.1)
])

augmentations_2 = v2.RandomApply(torch.nn.ModuleList(
    [v2.RandomRotation(30),]), p=0.1)
augmentations_3 = v2.RandomApply(torch.nn.ModuleList(
    [transforms.ColorJitter(),]), p=0.1)

```

A.3 WeightedRandomSampler

Data balancing with WeightedRandomSampler

```python
    # Calculate weights for each class
    class_weights = [1 / count for count in t_occurences]

    # Create a list of weights corresponding to each sample in the dataset
    sample_weights = [class_weights[int(train_dataset[label_idx][1])] for label_idx in train_set.indices]

    # Convert the list of weights to a PyTorch tensor
    weights = torch.DoubleTensor(sample_weights)

    # Use WeightedRandomSampler to balance the classes
    sampler = WeightedRandomSampler(weights, len(sample_weights), replacement = True) #len(train_set) target_number * num_classes
```

A.4 Pretrained Model Architectures

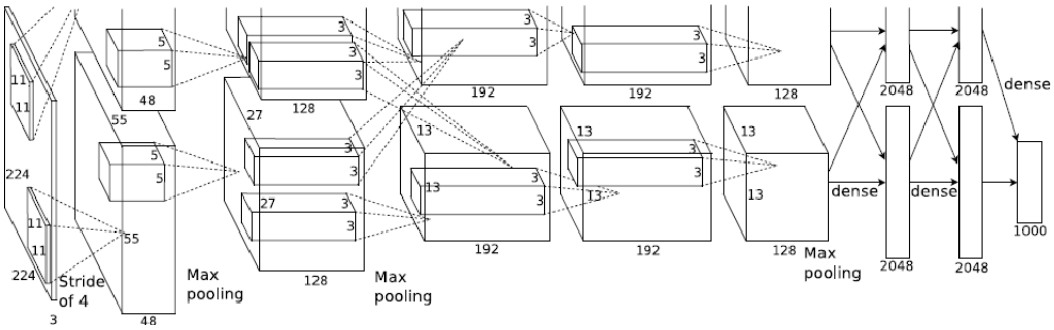


Figure 16: Original AlexNet architecture given by Krizhevsky et al..

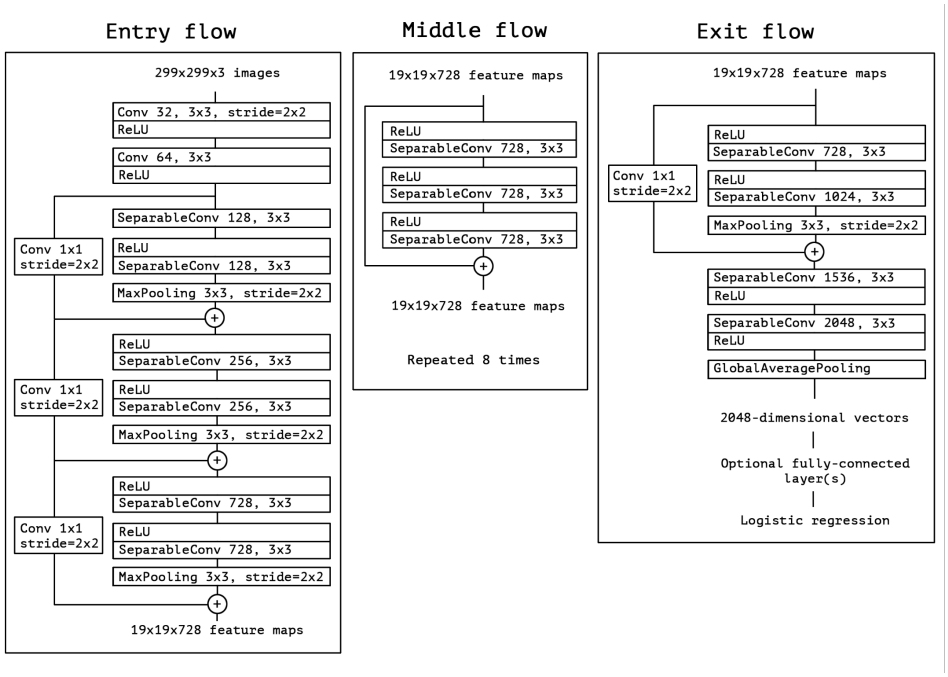


Figure 17: Xceptions architecture as outlined by Chollet.

A.5 Additional Tables

Table 5: Impact of Learning Rate, Batch Size, and Epochs on Accuracy in order to determine the best set of Hyperparameters for the SimpleCNN

Learning Rate		
Learning Rate	Average Train Accuracy (%)	Average Validation Accuracy (%)
0.001	99.6	98.9
0.01	97.7	97.2
0.1	10.3	10.0

Batch Size		
Batch Size	Average Train Accuracy (%)	Average Validation Accuracy (%)
16	99.7	98.9
32	99.5	98.9
64	99.4	98.9

Epochs		
Epochs	Average Train Accuracy (%)	Average Validation Accuracy (%)
5	99.4	98.8
10	99.6	99.0
15	99.6	99.0
20	99.7	99.0