AALBORG UNIVERSITY
DENMARK

Master's Thesis

MI10

Deep Learning Approaches to Art Style Recognition in Digital Images

Student name.:	*Study No.:*	*AAU email:*
Rasmus Hove Johnsen	20124665	rjohns12@student.aau.dk
Andrea Gradečak	20167849	agrade16@student.aau.dk

Tuesday 6th June, 2017

Summary

With the increase in digitalised paintings, large collections of artworks are available for tasks such as image recognition. One avenue that has not been explored significantly within image recognition is style classification in fine art paintings. To that effect we have experimented with building a state-of-the-art model, i.e. a Convolutional Neural Network(CNN), for this problem.
We experimented incrementally with three major areas of CNN construction; image representation, structural architecture, and image augmentation, and compared the resulting model with a baseline model based on a pretrained CNN, the VGG16 network and an SVM classifier. We trained and evaluated our model on pairs of painting styles.
During experiments with image representation, we explored the effects of normalisation, methods for resizing, and image sizes, using a network partly inspired by the baseline. We found that scaling the images with *Min-Max* scaling, a normal resizing using Lanczos resampling, and a target size of 224×224 and three colour channels, yielded the highest validation accuracy.
We then experimented with refining the basic network structure, to fit the problem of art style recognition. We investigate two aspects of structural architecture; the depth of the network, i.e. the number of layers, and the width of the network, i.e. the number of feature maps in each layer. In terms of model depth, we combined convolutional layers and max-pooling into convolutional blocks, and found that having 7 of these blocks, with one convolutional layer and a max-pooling layer, gave the highest accuracy. In terms of model width, we found that starting with 32 feature maps in the first convolutional layer, and doubling these after each max-pooling layer, provided the best results.
To combat overfitting and increase classification performance, we employed a series of augmentation techniques. Using a sliding window crop, taking a maximum of 10 crops from the images, together with a Gaussian blur, and horizontal and vertical flipping, we inflated the dataset with 60 times the original samples, and increase the validation accuracy.
For comparing the performance of our network with the performance of the baseline, we used three different datasets:

Set A: *Cubism* and *Neoclassicism*, 1773 training, 580 validation, 585 test

Set B: *Colour Field Painting* and *Magic Realism*, 856 training, 239 validation, 276 test

Set C: *Early Renaissance* and *High Renaissance*, 1265 training, 413 validation, 424 test

And reached the following results:

	VGG16 + SVM	Our Network
Set A	96.1%	91.6%
Set B	97.8%	95.3%
Set C	74.1%	67.7%

Based on the results we concluded that we were not able to perform better than the baseline on this problem, even with a deep architecture and aggressive augmentation.

AALBORG UNIVERSITY
Student Report

Department of Computer Science
Aalborg University
http://www.cs.aau.dk

Title:
Deep Learning Approaches to Art Style Recognition in Digital Images

Theme:
Master's Thesis

Project Period:
Spring Semester 2017

Project Group:
MI108f17

Participant(s):
Rasmus Hove Johnsen
Andrea Gradečak

Supervisor(s):
Manfred Jaeger

Copies: 3

Page Numbers: 74

Date of Completion:
Tuesday 6th June, 2017

Abstract:

Convolutional Neural Networks(CNNs) have become state-of-the-art image recognition models, but have not been used to significant effect for art style recognition in fine art paintings.
Through incremental experimentation with a number of aspects of CNNs, we have build a model for this task. The model has 7 blocks, containing one convolutional layer with rectified linear units and a max-pooling layer. It has 32 feature maps in the first convolutional layer, which is doubled after each max-pooling layer. At the end of the network there is one fully connected layer with 8 neurons and a softmax output layer. As baseline we have used the VGG16 network with a Support Vector Machine(SVM) as classifier. To compensate for the relatively small size of the dataset, we employed a sliding window cropping technique, taking a maximum of 10 crops from the original image to inflate the dataset.
Testing on three pairs of styles we reached the highest test accuracy of 95.3% on *Color Field Painting* and *Magic Realism*. However, this was not enough to beat the baseline, that reached a test accuracy of 97.8%.
To conclude, we find that without aggressive augmentation, training purely on fine art paintings for style recognition, is not viably better than using a pretrained CNN and an SVM classifier.

Foreword

Computation for the work described in this report was supported by the DeiC National HPC Centre, SDU. The authors will be referring to themselves in first person plural.

Counselling and the original project proposal has been delivered by Manfred Jaeger.

Contents

Part I

Introduction

1 Introduction

In the art research categorising artworks by style has always been difficult. The image is analysed and its style is identified based on the determined characteristics. To avoid the need for a human expert, it is advantageous to execute this process automatically on a machine. This type of task falls into the image classification domain, for which solutions are lately utilising deep learning techniques.

After the inception of deep learning, many problems that were challenging for machines have been successfully solved. The main obstacle in solving them was finding a good representation of the large amount of data which would help the machines learn and infer about distinctions among the data. Using deep learning approaches it is possible to generate suitable data representations which led to deep learning systems becoming the state-of-the-art in a wide range of domains.

Although being broadly applicable, deep learning mostly contributed to the area of computer vision. Before the emergence of deep learning, traditional approaches to representing images was using a technique called *bag-of-features* [12], where the image features are obtained through extraction methods. The main drawback of this approach is that the process involves selecting the best features and finding possible ways for combining and improving them. Additionally, in an application specific problem such as art style recognition this also often requires expert knowledge.

In the 2012 ImageNet challenge [10], Convolutional Neural Networks (CNNs), a type of deep learning model, had its breakthrough providing significant results which made a shift in the approach to image recognition tasks. The model is based on convolutional layers with local receptive fields each processing only small portions of the input. It generates features by modifying its weights using backpropagation making the feature extraction an automatic process. These characteristics make CNNs suitable for image data and showed to yield better results compared to the traditional approaches.

Recently, collections of artworks have been digitised, making classifying art an important task. The problem is especially demanding when classifying art by style. Some styles may share certain characteristics due to historical influences. Moreover, different artists who belong to the same style have different methods of painting which creates subtle dissimilarities within the style. This causes various complications when determining the distinctions between the art styles. Implementing a CNN as a part of a system for art style recognition could solve this problem and it would not require any special domain expertise.

However, utilising CNNs has several difficulties, most considerably, the complexity of the model. As a result, there is a trade-off between generally long training time and the required computational power. Complexity is also the reason why some of the aspects of CNNs are not fully understood. Certain conventions are adopted when designing a CNN model, but they do not guarantee that the resulting architecture will be adequate for a specific problem. Furthermore, the classification efficiency of a CNN is affected by the representation of the input data. Methods like pre-processing and augmentation of the data can improve the predictions. Since they have little theoretical background, the best practices are mostly found empirically through experiments.

In the domain of art style classification, CNNs have mostly been used for extracting features, leaving the predictions for other classification models, such as Support Vector Machines (SVMs). Known as autoencoders, the fully connected layers at the top of the networks are removed and the output from the last convolutional layers is used as features in another classification model. Conversely, complete CNNs are not used often for art style classification. Therefore, it is beneficial to conduct experiments to attain knowledge necessary for designing a CNN model for this specific problem.

1.1 Problem Statement

The aim of this Master Thesis is to develop a convolutional neural network for art style recognition. Experiments with different input representations, model architectures and parameter tweaking will be conducted. Through the incremental experiments we will research the possibilities of the CNN model for art style recognition and learn how individual components influence its overall performance. The outcome will help us to address the guidelines for developing and improving such models.

1.2 Tools

For developing the CNN we use a framework called *Keras* and for training them we use an HPC supercomputer called *Abacus*. Both of these will be explained in this section.

1.2.1 Keras

For designing and implementing the CNN we make use of the framework *Keras* [3]. Keras is a frontend built to work with two of the most popular neural network frameworks, Tensorflow [2] and Theano [17]. By providing a more readable syntax, Keras was designed to make it easier to experiment with CNNs, while giving the user the means to access the other frameworks for finer control. These aspects align well with the purpose of this project.
To illustrate the difference between Tensorflow and Keras, Listings 1.1 and 1.2 shows a convolutional layer in Tensorflow and Keras, respectively.

Listing 1.1: A convolutional layer implemented in Tensorflow. Taken from [8]

```
1  import tensorflow as tf
2
3  CROP = 64
4
5
6  def weight_variable(shape):
7      initial = tf.truncated_normal(shape, stddev=0.1)
8      return tf.Variable(initial)
9
10
11 def bias_variable(shape):
```

```
12      initial = tf.constant(0.1, shape=shape)
13      return tf.Variable(initial)
14
15
16  def conv2d(x, W):
17      return tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='SAME')
18
19
20  def convolutional_layer(input, stride, channels, output_depth):
21      W_conv = weight_variable([stride, stride, channels, output_depth
            ])
22      b_conv = bias_variable([output_depth])
23
24      h_conv = tf.nn.relu(conv2d(input, W_conv) + b_conv)
25
26      return h_conv
27
28
29  x_image = tf.reshape(x, [-1,CROP,CROP,1])
30
31
32  conv1 = convolutional_layer(x_image, 5, 1, 64)
```

Listing 1.2: A convolutional layer implemented in Keras.

```
1  from keras.layers import Convolution2D, Input
2
3  inputs = Input((128, 128, 3), name='Input')
4  conv = Convolution2D(8, (7, 7), strides=2, activation='relu',
       padding='same', name='1_conv')(inputs)
```

As we can see, defining a convolutional layer in Tensorflow requires the designer to manually define methods to keep the code readable, and many of the parameters can be hard to understand for people that have not worked with Tensorflow before. An advantage of this is that the designer has more control over what happens and can change smaller details in the layers.
In contrast, Keras allows the designer to define a layer in a single line, where most of the details are handled by parameters. In Listing 1.2 we make use of the functional API that allows us to pass previous layers as input when defining new layers. This further compresses the code, and makes it more elegant, because we are only concerned with a few variables at any given time.

1.2.2 Abacus

CNNs are known to be very complex and time consuming to train due to the sheer amount of parameters [10]. With that in mind, we decided to make use of the *Abacus* HPC supercomputer, offered to Aalborg University by Danish e-infrastructure Corporation (DeiC) and maintained by Syddansk University (SDU) [15].

The Abacus supercomputer provides a number of computing nodes that offer increased computing power. On [16], the full specifications can be viewed, but what we find the most important is that each node contains two 12-core CPUs with around 480 GFlop/s of theoretical performance. In addition to this there are three different types of nodes; slim, fat, and GPU. The slim nodes are exactly as the base configuration, while the fat nodes have 512 GB of memory, as opposed to 64 GB in the slim. The GPU nodes have two Nvidia K40 graphics cards, each with a theoretical performance of 1.43 TFlob/s, and gives us the opportunity to utilise GPU computing. In terms of software, the Abacus supercomputer uses the *Slurm* workload manager, for scheduling script jobs on the computing nodes [13]. Slurm works by allocating requested resources to a user, specified in bash scripts, and offer means to monitor and interact with running jobs, if necessary.
As mentioned in [10], CNN training is vastly improved by using GPUs instead of CPUs. Therefore, we will use the GPU nodes for this project.
The Abacus supercomputer is shared between research institutions, meaning we are not the only users of it. Aalborg University has a branch, *aauhpc*, to which we are assigned, and thus share it with other researchers and students at Aalborg University. Aalborg University has bought time on the supercomputer, and as such we need to be mindful how much time we spend on the computations. For the period between March 17^{th} and June 30^{th} there are 2000 hours for the GPU clusters allotted to *aauhpc*, which are counted as node hours, meaning time spent is multiplied by the amount of nodes requested.
In addition, there are only 72 GPU nodes across all the branches and the usage is based on a *fair use* principle, meaning if we have used the nodes for a long time we might have lower priority for later computations. This means that we are constrained by how much time we have used, how much time we have left and how many people are using the supercomputer at the time.

Part II

Experiments

2 Experiment Strategy

In this Chapter we will outline our general approach to the experiments in this project. These experiments form the foundation for the choices we make regarding the development of the CNNs. First we will explain CNNs in general, to get a sense of the different aspects we need to consider when designing our own. Then, how the experiments will be structured, the dataset, our use of the Abacus supercomputer, the metrics we use, and some criteria for success.

2.1 Convolutional Neural Networks

The images in this section are taken from [11].
CNNs are specialised versions of Deep Neural Networks, which in turn is an Artificial Neural Network with multiple hidden layers. The difference between artificial neural networks and CNNs, is that while former is fully connected, i.e. all neurons in one layer is connected to all neurons in the previous and succeeding layers, as Figure 2.1 shows, CNNs are more loosely connected.

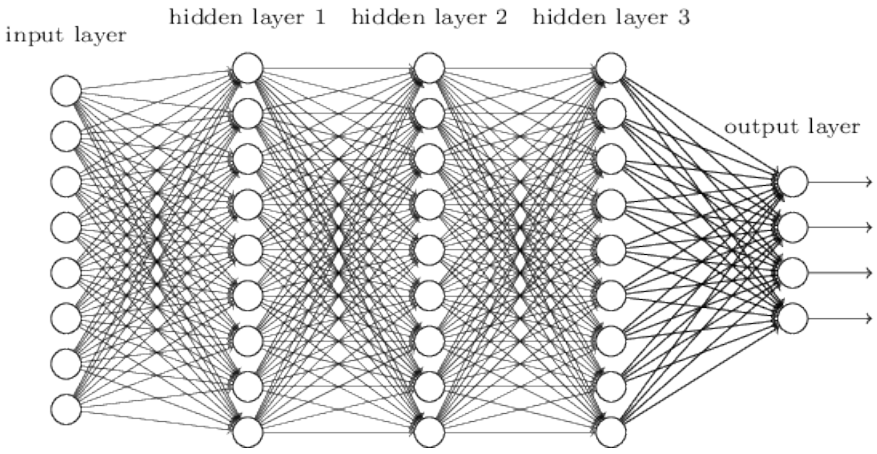


Figure 2.1: A fully connected Artificial Neural Network.

As seen in the figure, all neurons are connected in an acyclical, directed graph. This works well for data with smaller dimensions, but with images this type of network would have too many parameters and overfit.
In contrast, CNNs connect local areas in the input to single neurons in the succeeding layer though local receptive fields, hence forth called filters, as shown in Figure 2.2, on page 8. This is inspired by how the human brain processes visual information, and it reduces the complexity of the input.

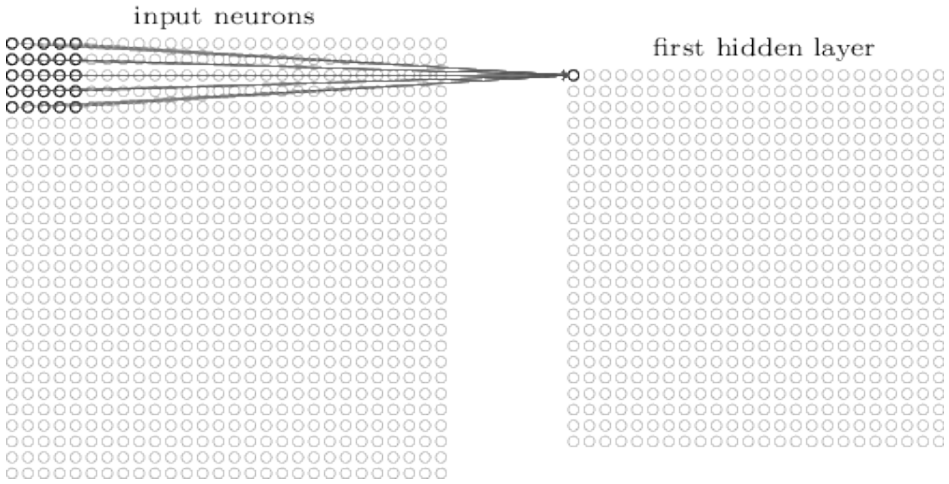


Figure 2.2: A filters for the first convolutional layer.

Another difference is how the weights are distributed. In artificial neural networks weights are unique to all connections between neurons, however with CNNs they are tied to the filters, shown as the highlighted connections in Figure 2.2. This makes weights shared across the input, instead of unique for each edge, giving the intuition that what is important to look for in one area of the input, might also be important elsewhere. The result of applying the weights in this manner is a feature map, that finds a single feature in the input. By making more feature maps, with each their associated filters, the CNN can find multiple features. So each feature map has a filter that indicates what the CNN is looking for.
Lastly, to further reduce dimensionality, CNNs employ a technique called max pooling. By only propagating the maximum activation in a local area of a layer, it effectively reduces the dimensionality by a factor of the size of the area. For instance, max pooling a 2×2 area will halve the dimensions, as can be seen in Figure 2.3. Since the size of the input is reduced, the amount of feature maps usually doubles in response.

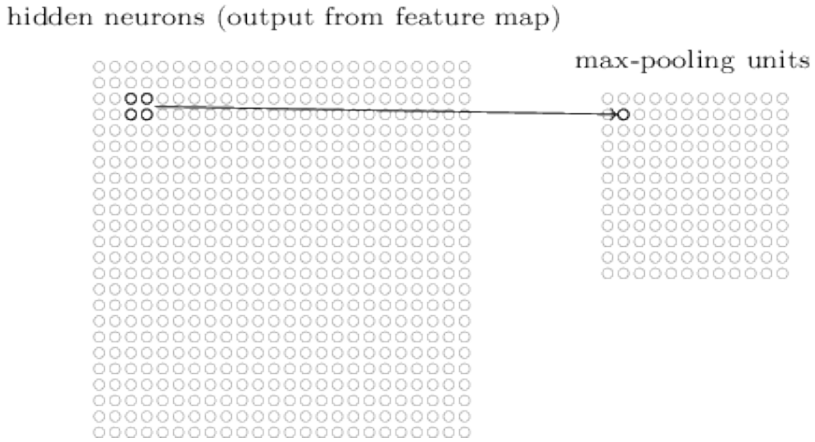


Figure 2.3: An example of a 2×2 max pooling layer.

As mentioned in Section 1.1, the purpose of this project is to design and build a CNN for art style classification. To get the best network structure, we will experiment with different architectures and hyper-parameters. In addition to these two aspects, we also experiment with pre-processing techniques, i.e. input representation and data augmentation, which are neither architecture nor hyper-parameters. In the following sections we will highlight the various features of CNNs that we have to consider while experimenting.

2.1.1 Architecture

The architecture of a CNN pertains to the structure of the various layers, how they are made and how they are connected. The intuition behind a deeper architecture is, the deeper it is the more sophisticated the extracted features become. In terms of architecture we consider the following aspects:

Convolutional layer These are the layers responsible for extracting the features from the input image. The more convolutional layers we have, the more sophisticated the features become. However, having too many of these layers will risk overfitting, because of added parameters. Convolutional layers have a number of parameters that we look at when designing the network:

- **Feature maps** The amount of feature maps used in a convolutional layer determines how many different features the network looks for in the image. The intuition is that fewer feature maps necessitates more general features, while more feature maps makes them more specialised.
- **Filters** The dimensions of the filter affect the granularity at which the network scans the input image. This means that a larger filter finds broader patterns, and reduces the complexity, while a smaller filter looks for more detail, which could increase classification performance.
- **Stride** The rate at which the filter moves across the image. The same intuition applies as with the filters, larger strides give broader features, but reduces parameters.
- **Activation function** This affects how each neuron handles the data it receives from the previous layer. Different activation functions, and their respective parameters, shape how the network sees the input, and what to focus on.

Max pooling layer These layers are used to reduce dimensionality by only propagating the maximum activation in a local area of the previous layer. They work largely like convolutional layers by having a filter scan the previous layer, and then for each field choose the neuron that has the highest activation. There are only two parameters that are significant for max pooling:

- **Size** The size of the filter decides how much the dimensions are reduced. The larger the field the less data is propagated, which could affect the classification performance.
- **Stride** While the stride normally is the size of the filter, so the pooling does not overlap, having overlapping pooling, as mentioned in [10], can have some benefits.

2.1.2 Hyper-parameters

These parameters affect the way the network learns and behaves. They are called hyper-parameters to set them apart from model parameters, e.g. individual weights, which we as designers do not have control over. The hyper-parameters we work with are the following:

Learning function The function used to correct the weights in the network through backpropagation. Changing the learning function can affect how fast the network converges, or whether that will happen at all. Furthermore, we can tweak the learning rate as well to regulate this. A higher learning rate will make the network learn faster, but risk not converging on a global minimum, while a lower learning rate can risk getting stuck in local minima.

Loss function The function for calculating the error measure between expected and actual output. Different loss functions can alleviate problems such as slow learning at the start of training, that can affect how long the network needs to train for.

Regularisation To reduce overfitting we can penalise over-complex models by introducing regularisation parameters into the loss computation. The more complex the network becomes, the higher the loss will be when regularisation is applied, thus it is forced to generalise rather than overfit the training data.

Epochs How many iterations the network runs over the training data. When training a CNN, we use a technique called backpropagation, where weights in the network are updated based on an error term. However, the weights are updated very slowly, which means the network needs to see the same input multiple times. The more epochs the network runs, the longer it takes, but the more it learns. Having too many epochs, however, risks overfitting, since the network only sees the training data.

Weight initialisation The way the initial weights are chosen at the start of training. This affects how fast the network learns from the start. A good initialisation can speed up learning considerably and reduce the risk of getting stuck in local minima.

2.1.3 Input

In addition to the architecture and the hyper-parameters, we will also experiment with pre-processing the input. Since neural networks in general rely on static input, e.g. the dimensions of the input have to be the same for all samples, it is important to standardise the input data. Inevitably, standardising data loses some information, because the data is either aggregated or truncated to force it to a certain shape. Therefore, choosing the right technique is important, and as such we will experiment with it.
Changing the way the input data is represented can affect the structure of the network. For instance, if we choose to resize images to a large size, such as $1,024 \times 1,024$, we can change a number of things in the architecture. We could add more pooling layers, making the network deeper, or we could increase the size of the filter, making it coarser to more aggressively reduce the dimensionality. Configuring convolutional layers would have similar effect, though not as impactful as max pooling.

Furthermore, we will look at data augmentation, as a means to reduce overfitting and to increase classification performance. Augmentation pertains to various methods of artificially inflating a dataset, by transforming the existing data to create new data. In terms of images this could be distortion, rotation, sharpening or blurring, inverting colours, and so on. This adds more samples and forces the network to look for robust features across the styles. In theory, the more data we have, the less we need techniques such as regularisation. However, adding too much artificial data can end up as noise, since it is not unique samples but just mutations of the same basic samples.

2.2 Experiment Setup

In this section we will describe the general setup for the experiments conducted during this project. We will outline the dataset we use for training and testing, along with a benchmark, based on a pretrained CNN we wish to compare our results to. Then, a description of the various metrics we use to evaluate our results. Lastly, we will explain our use of the Abacus supercomputer and how the experiments will be performed.

2.2.1 Dataset

The following section is taken from [8], with some alterations.
For our dataset we have 79,420 paintings from 136 styles, from the competition *Painter by Numbers* on *Kaggle.com* [1]. The competition was about determining if pairs of paintings are from the same artist. Since the competition is focused on artists, the style of a painting is not always stated. Additionally, the styles are imbalanced. For instance, *Impressionism* has 8220 paintings, while a less common style like *Muralism* has 124 paintings. In addition to the style imbalance, the images are also of varying shapes and sizes.

2.2.2 Baseline Method

For our experiments we need a baseline to compare results to. Using this as a guideline we can know when to stop optimising for this iteration and go to the next. In art style classification, a common approach is to use a pretrained CNNs as a feature extractor and a prediction model like SVM. As such, we can use something similar for our baseline.
Keras offers a number of pretrained CNNs that can be used and *Scikit learn* has implementations for traditional models. We will use the VGG16 network from [14] as the feature extractor and an SVM as our classifier. Figure 2.4, on page 12, shows table 1 from [14] which contains the specifications for the VGG networks. The architecture implemented in Keras is the *D* column and will be the network we will use for our baseline.

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

Figure 2.4: The VGG CNN architectures, taken from [14].

As for the SVM, we use the implementation from *Scikit learn* with a linear kernel and a penalty parameter of $C = 10^{-5}$. The linear kernel means that the hyperplane separating the classes is a linear regression. The penalty parameter decides the hyperplane margin by controlling how much the model should be penalised for mis-classifying samples. This means that a low penalty allows the SVM to have a large margin, since a small number of samples on the wrong side of the hyperplane is acceptable, if the model generalises well. A higher value would make the model more strict about separating the samples and would risk overfitting.
This setup will serve two purposes. First, the test accuracy of the classifier will be the benchmark value our own network needs to be at least comparable to. Second, the architecture of the VGG16 network can be a guideline for how our own network should be structured. For instance, if we look at the number of feature maps, the first convolutional layer in the VGG16 network has 64, meaning that can be a control value, where we know that 64 feature maps give good results.

2.2.3 Metrics

When we evaluate the performance of our CNN, we do so using a number of metrics. For the incremental experiments we use classification accuracy on a validation set to gauge the network. The reason why we chose accuracy is that it is simple and easy to interpret and the dataset is not overly imbalanced. In addition to validation accuracy, we can also look at training accuracy, to

determine whether the network is overfitting for experiments where that could be an issue. By looking at the difference between these two measures, we can see if the network generalises well; a high training accuracy and low validation accuracy is an indication of overfitting. To further support that conclusion, we can also look at training and validation losses, which for overfitting should show same, but reversed, traits. For certain types of experiments we look at training time, which could tell us whether a change is worth implementing.
For the final comparison with the baseline method we will be using a distinct test set, as opposed to a validation set.

Graphical Analysis

When we report on the above metrics it is only a single measure, taken at the end of the experiment. However, it could also be useful to look at the training process to see how well the network learns. One way of doing this is to plot the losses or accuracies from each epoch. For example, if we have a network that, after running for 100 epochs, has a final training loss of around 0.23, we could say that it could still learn. But, if we then look at the loss curve, we may find that for the last 20 epochs it had close to the same loss, which could indicate it was stuck in a local minima. If the network consistently exhibits this behaviour, we could try with different initialisation methods or change the learning rate. An example of a loss curve can be seen in Figure 2.5.

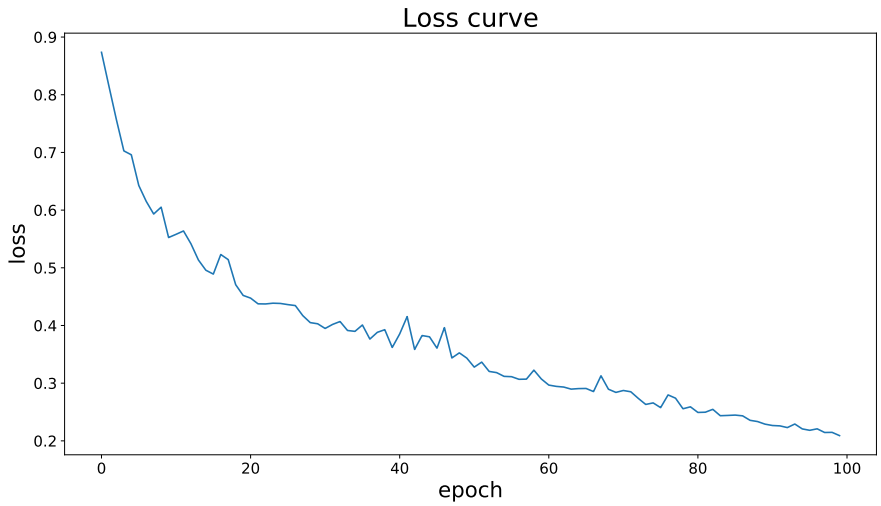
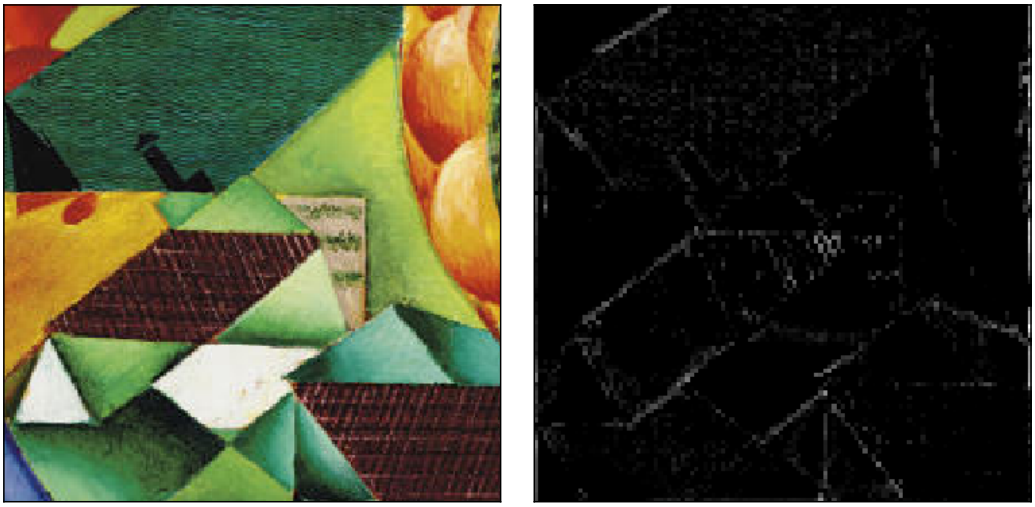


Figure 2.5: An example of a loss curve, plottet over 100 epochs.

To gain additional insight into how the CNNs see the input and what they learn, we can investigate the feature maps to see where the network focuses. Since one feature map is a two dimensional matrix, we can visualise this as a single-channel image, i.e. a grey-scale image, where the intensity of the pixel denotes the value of the activation. Figure 2.6, on page 14, shows an example of a gray-scaled representation of a feature map on a sample image.



(a) An example image.

(b) The associated activations for a single feature map.

Figure 2.6: An illustration of a convolutional feature map.

By comparing the feature map to the original image we can speculate where the CNN focuses, and maybe understand what it learns.
Another way of understanding how the network generally learns is to visualise the filters. As explained in Section 2.1, weights are associated with each convolutional feature map, meaning, in addition to the activations, we can also look at the weights to see the specific pattern that the network is searching for. Figure 2.7 shows a visualisation of the filters in the first layer of a network.

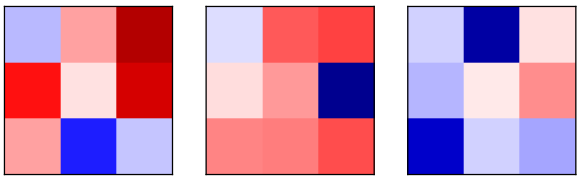


Figure 2.7: An example of the filters for the first convolutional layer in a network.

These weights correspond to the three colour channels of the input image, meaning from left to right it is red, green, and blue.
Since the weights in a CNN can be negative, we need to be able to tell when it is positive and when it is negative. We do this by using the converging spectrum shown in Figure 2.8, on page 15. We centre the range around zero, so that if we see a red tint, we know it is positive and vice versa for negatives.

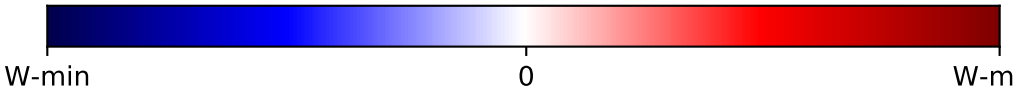


Figure 2.8: The converging colour spectrum used to illustrate the filters. *W-min* and *W-max* are the minimum and maximum values of the filter.

2.2.4 Use of Abacus

As mentioned in Section 1.2.2, we have access to a supercomputer for this project and as such, the majority of our experiments will be performed on that supercomputer. Since we have multiple computing nodes available, we can run several experiments at the same time which would most often be one experiment with various configurations. For instance, if we wish to investigate the effect convolutional feature maps have on the prediction accuracy, we could have, for example, 4 networks with different amount of feature maps. By then analysing the results, we can see which of the configurations had the best effect. This, along with the increased computation, means that we can experiment with multiple parameters at the same time.
To efficiently separate different configurations while concurrently running experiments, the parameters are passed with command line arguments. As mentioned in Section 1.2.2, Abacus uses bash scripts to execute jobs and it uses *Slurm* for scheduling. Listing 2.1 shows an example of a Bash script.

Listing 2.1: An example of how we structure the Bash scripts.

```
1  #! /bin/bash
2  #
3  #SBATCH --account aauhpc_gpu
4  #SBATCH --nodes 3
5  #SBATCH --time 12:00:00
6
7  . ~/venv/bin/activate
8
9  module purge
10 module load cuda/8.0.44
11 module load cudnn/5.1.10
12
13 srun -N 1 python run_network.py [PARAMETERS] > [CAPTURE OUTPUT] 2>&1
       &
14 srun -N 1 python run_network.py [PARAMETERS] > [CAPTURE OUTPUT] 2>&1
       &
15 srun -N 1 python run_network.py [PARAMETERS] > [CAPTURE OUTPUT] 2>&1
       &
16 rm -f slurm-*
17 wait
```

Slurm reads the comments in the header for the keyword *SBATCH* and evaluate the following arguments. In this example we request the GPU nodes associated with *aauhpc*, then 3 of those

nodes and lastly, we want to run the job for no longer than 12 hours.
Since we are not super users on the Abacus, we need a python virtual environment to install special packages, such as Keras and Tensorflow. In this script we activate this environment on line 7. To reduce bloating computations with unnecessary software, Abacus packages several general purpose frameworks and APIs in modules that can be loaded at any time during the job. In this case, we first remove all modules that may already have been loaded and load two modules; *CUDA* and *cuDNN*. These are two frameworks developed by Nvidia; the former is for parallelism on their GPUs, and the latter is made especially for running CNNs on *CUDA* GPUs. Both of these are necessary for Tensorflow to work.
On the following three lines we have the actual networks. The *srun* command is for Slurm to execute a parallel job. The argument *-N* is the amount of nodes this job should take, which here is one. Then we run the python script *run_network.py* that takes a number of parameters to specify the configuration. When running jobs, Slurm captures standard output and standard error and saves it in an out file, named *slurm-<job id>.out*. However, to better distinguish between the different networks, we can redirect this output to a specific file instead. Lastly, we detach this process to run in the background, which is how we make the networks run in parallel. To avoid bloating our workspace, we remove all the Slurm output files, since these are empty because we redirect all the output. We add the *wait* command to leave the job running until all background processes have terminated, otherwise the job would immediately terminate.
We execute this script using the *sbatch* command from a Bash shell. This command offers a number of parameters to further specialise the job, but we choose to only configure the header field in the script.

While Slurm provides several more sophisticated concurrency handling, we chose this simpler approach so that we can focus on building the networks.
When Tensorflow is running on a GPU, it optimises the computation by allocating all visual memory on the GPU, which ensures it does not need to dynamically allocate and manage memory, increasing efficiency. This has the negative effect that only one Tensorflow process can run on a GPU at a time, for risk of memory exhaustion. This is the reason why each network is assigned an entire GPU node. Furthermore, while we have two GPUs on each node, theoretically allowing us to run two networks per node instead of one, it would require that we specify exactly what node and which GPU each of the networks should use.
Alternatively, we could handle it within a python script. Tensorflow provides means for controlling which device, if multiple devices are found, the networks should use. By default, Tensorflow just uses the first GPU it finds, but we can specify that it should use both devices for various tasks. The reason we have not done this is that it effectively doubles the amount of parameters needed for the script, making it more difficult to differentiate the networks.

2.2.5 Incremental Experiments

When we experiment with constructing a CNN, we do so incrementally. We will start with experimenting with input representation to explore the effects this has on the classification accuracy. The result of these experiments will then create the foundation for the experiments with structural architecture. Lastly, we will experiment with augmentation, since it is likely that we will

have a more complex network by the end of the architecture experiments, which will overfit.
During each of these stages we will experiment with as many of the parameters mentioned in Sections 2.1.1 and 2.1.2. To simplify the experiments, we will use only a subset of the dataset, explained in Section 2.2.1. We will use the classes *Cubism* and *Neoclassicism*.
Additionally, we will be using the same static split of the dataset; 60% for training, 20% for test, and 20% for validation. For the incremental experiments we will be training on the test set and evaluate on the validation set. Then when we make the final experiment, comparing our network with the baseline, we will be using the test set.

3 Input Representation

One of the most significant disadvantages of CNNs is the fact that they require static input. For problems such as image classification that often means to resize the images in the dataset to one unified size. The problem with this is that it can change how the content in the image is presented, obscure or remove information, and add artefacts that are not present in the original image. This can all negatively affect the classification performance, since only some images would be affected by these problems.
In this Chapter we will explore the different ways of representing the input. First we will look at how to normalise the input, beyond how images naturally are normalised, and then look at how to scale images to a uniform size. Lastly, we will experiment with various image sizes to see how that affects the predictions.

All the experiments in this Chapter are made with a network with the following architecture:

Layer	Filter size	# feature maps	Stride	Output
Input				$224 \times 224 \times 3$
Conv	3×3	8	1	$224 \times 224 \times 8$
Max pool	2×2		2	$112 \times 112 \times 8$
Conv	3×3	16	1	$112 \times 112 \times 16$
Max pool	2×2		2	$56 \times 56 \times 16$
Conv	3×3	32	1	$56 \times 56 \times 32$
Max pool	2×2		2	$28 \times 28 \times 32$
Conv	3×3	64	1	$28 \times 28 \times 64$
Max pool	2×2		2	$14 \times 14 \times 64$
Conv	3×3	128	1	$14 \times 14 \times 128$
Max pool	2×2		2	$7 \times 7 \times 128$
Fully connected				8
Fully connected				# of classes

Table 3.1: The model for the input representation experiments.

This model is partly inspired by the VGG16 model, mentioned in Section 2.2.2, where we use fewer feature maps in the convolutional layers. In addition, to this we use rectified linear units in all but the final layer, which uses the softmax activation function. We use the log likelihood loss function and represent labels in a one-hot encoding.
As mentioned in Section 2.2.5, we do these experiments with a static split of the dataset; 60% for training set, 20% validation and 20% for testing set, where we will not be using the testing set in this Chapter. We employ early stopping, to limit the amount of time the networks train and to reduce the risk of overfitting. The stopping criteria is when the validation accuracy does not increase by more than 0.5% over 10 epochs.

3.1 Normalisation

In machine learning, data normalisation is used to potentially decrease unwanted noise in the input data. Input can have a wide range of values, making it difficult for prediction models to learn relevant patterns for classification. While images have a natural normalisation, where pixel values are between 0 and 255, it can still be useful to further normalise.
Through preliminary experimentation we observed that, at times the network would seemingly not learn the training set and producing the same prediction for all samples. By reviewing the output from each layer in the network, we found that an entire layer outputted zeros, resulting in the softmax layer always receiving the same input and accordingly made the same output. This happened because we used the raw images as input. Each of these images had a range of 255, from 0 to 255, that is then multiplied by a weight matrix, which potentially contained negative values. When these high pixel values were multiplied with a negative weight and then summed as input to the succeeding layer, it could result in a negative sum. Since we use rectified linear units, these would output zero. If the sum of all input to one layer is zero or negative, the output of this layer would be exactly zero. For this reason, we decided to experiment with various normalisation methods.
There exists a number of normalisation techniques that can be used for this problem. We are experimenting with four basic methods:

Min-Max Scaling Also known as feature scaling, this is a very simple normalisation done as follows: $X' = \frac{X - X_{min}}{X_{max} - X_{min}}$, where X is the sample data, X_{min} and X_{max} are the minimum and maximum values in the sample, respectively. The minimum and maximum can also be calculated on the whole dataset. Typically in images, the minimum is *0* and maximum is *255*, effectively making the above equation look as follows: $X' = \frac{X - 0}{255 - 0} = \frac{X}{255}$.

Standard Score A common machine learning normalisation method, where the mean of the dataset is subtracted from the sample value and divided by the standard deviation, as follows: $X' = \frac{X - \mu}{\sigma}$, where X is the sample, μ is the mean, and σ is the standard deviation. This places all samples in a normal distribution with zero mean and a standard deviation of one, making comparisons easier. For high dimensional data, such as images, this has to be done on multiple dimensions separately meaning for images, there are individual means and standard deviations for red, green, and blue channels, respectively.

Subtract Mean This is a part of the *Standard Score* computation and we investigate how subtracting the mean performs in isolation. While this method does not limit the variance, it will centre the image around zero, making comparisons easier.

Divide by Standard Deviation While this is also a part of the *Standard Score*, we are also interested in simply dividing by the standard deviation. By both scaling the pixel values and decreasing the range of the variance, this should help the network learn. This will force each sample into a normal distribution centred around its own mean, as opposed to zero, meaning comparing samples would still be difficult.

Since we already know how the network behaves without normalisation, we will only experiment with the above four methods.

Experiment

As mentioned in Section 2.2.5, we use the two styles *Cubism* and *Neoclassicism*, and we also use early stopping, with a maximum of 100 epochs. We resize the input to 224×224, as described in Table 3.1, on page 18. We calculate the mean and standard deviation on the training set and apply it on the entire dataset. We run the networks 5 times and average the results to reduce the randomness introduced by initialisation. For this reason we also look at the deviation in accuracy between the runs to see if a result is due to the technique or a subset of the runs just did better than the rest. Since we use early stopping, the accuracy and loss curves are not averaged and instead the run that reached the highest validation accuracy is used for plotting.

Results

Following are the results of the normalisation experiments:

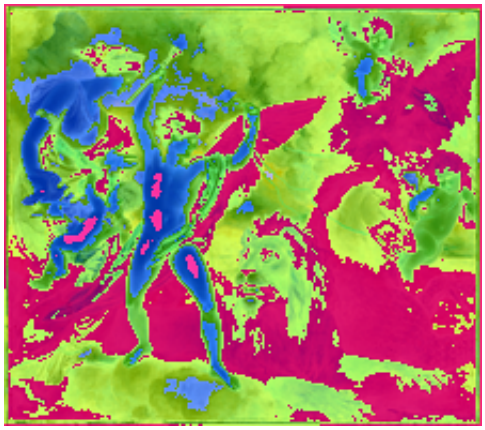
	Train Loss	**Validation Loss**	**Train Accuracy**	**Validation Accuracy**
Min-Max	0.2552	0.3795	89.8% $\pm$ 1.2%	**84.3%** $\pm$ 1.4%
Mean	0.1072	0.5726	96.5% $\pm$ 6.2%	80.8% $\pm$ 2.9%
Standard Deviation	0.2080	0.3815	91.9% $\pm$ 1.4%	83.5% $\pm$ 2.3%
Standard Score	0.2111	0.4004	92.1% $\pm$ 1.5%	83.3% $\pm$ 1.9%

Table 3.2: The results of the normalisation experiment.

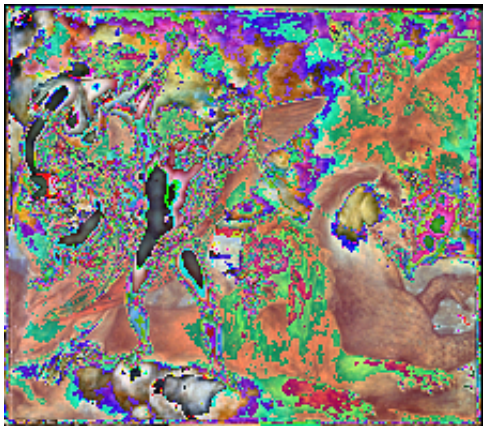
We see that the *Min-Max* scaling is the best in terms of validation accuracy and, when comparing the validation with the training accuracy, we see that it overfits less than the other techniques. Subtracting the mean does the worst of all the techniques, while dividing by the standard deviation and the *Standard Score* both have very similar results. If we compare the validation deviation between the two bottom results, we see that the standard deviation varies more across runs, whereas the *Standard Score* was more uniform. Interestingly, subtracting the mean seems to show signs of overfitting, due to the very high deviation on training, but when looking at the deviation on validation, it is comparable to the other techniques. This suggests that, even if the network was overfitting, the features it learned were able to be generalised to the validation set.
It is surprising that the *Standard Score* does not do as good as the *Min-Max*, considering it is a common practice. To analyse this further, we will illustrate how these techniques affect an image using Figure 3.1, on page 21, as an example.

Figure 3.1: The Neoclassicism painting *The Power of Love in the Three Elements*, by Omnia Vincit Amor, 1809.

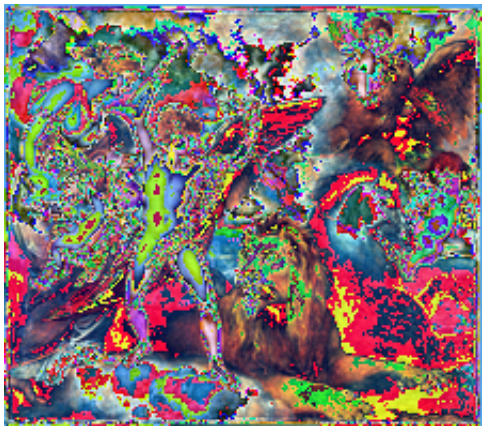
(a) The *Min-Max* technique.



(b) Subtracting the per channel mean.



(c) Dividing by the per channel standard deviation.



(d) The resulting *Standard Score* image.

Figure 3.2: The effect of the normalisation techniques.

Figures 3.2a through 3.2d show the effects of the normalisation techniques on the example image. Figures 3.2b and 3.2d have negative pixel values, so it is beneficial to know how these are handled. The images are in PNG format, which does not natively handle negative values. When saving the images it offsets the pixel values by the lowest value, if that value is negative, and writes the offset into the header of the image file. When the image engine in LaTeXreads this header it offsets the pixels in reverse, to give the same negative values.
Figure 3.2a shows the example image normalised with *Min-Max*. As we can see, the image is not changed, since the only difference is that the values are between zero and one, as opposed to zero and 255. Figure 3.2b shows the effect of subtracting the mean from the original image. The first thing to notice is that the colours are very different, which is because the mean is calculated per channel, meaning channels with more uniform distributions on values are closer to zero in the resulting image. The background becomes very green, indicating that both red and blue channels have very uniform distributions. We can see that by subtracting the mean, the objects and shapes in the image are more pronounced and there is a higher contrast between the colours, making this technique very useful for object recognition, which is why the VGG16 network uses it [14]. However, for style recognition, the actual colours used and how they blend can be more important than what objects and shapes are in the painting, which might be why this technique does so poorly.
Figure 3.2c shows the effects of dividing by the standard deviation. It seemingly adds a great deal of noise to the image, but it also seems to pick out a lot of the texture of the image. It seems to, at times, trace the edges of most of the objects, such as the outer edges of the angel's wing and the mane of the lion. The reason why this technique might has done so well, compared to subtracting the mean, is that the resulting image has tracings of objects. Considering how dissimilar *Neoclassicism* and *Cubism* are, in terms of shapes, it might be able to differentiate the styles based on that.
Lastly, by combining the two above techniques, we get the *Standard Score*, as shown in Figure 3.2d. As expected, this basically shows the tracings in Figure 3.2c on Figure 3.2b. If we look at the horse's body in the right of the image, we see that for subtracting the mean it becomes an almost continuous magenta colour, while for the standard deviation it blends with the background. However, on the *Standard Score* image, some of the texture of this horse is highlighted as the primarily red colours which means that the two techniques that constitute the *Standard Score* compliment each other to pick out more detail.

Another advantage of normalisation is that it increases convergence rate during learning which is especially desirable for neural networks. By plotting the accuracies and losses during training we can see if this also happens for these experiments. Figures 3.4, on page 24, and 3.3, on page 23, show the losses and accuracies of the four methods.

Since these plots are made using the best of the five training runs in terms of validation accuracy the curves do not necessarily reflect the results from Table 3.2. If we go through the graphs in turn, we see that *Min-Max* shows both training and validation curves following each other well, suggesting that this network is not overfitting. We can also see how, towards the end, the two lines start to separate. We see the validation curve starts to plateau while the training keeps decreasing, indicating that the early stopping took effect at the right time.

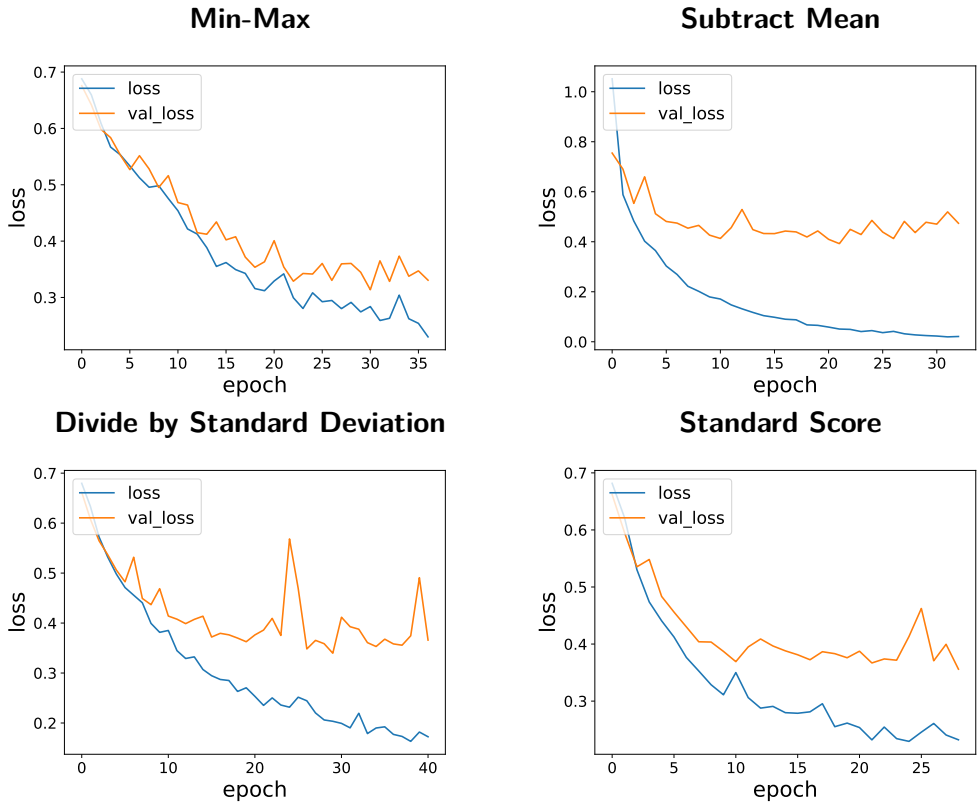


Figure 3.3: The losses for the normalisation experiment.

If we look at *Subtract Mean*, we see that the model starts to overfit very early, at the 5^{th} epoch. If we only consider the loss curve, the early stopping might have stopped the training between epoch 10 and 15 but the accuracy might have increased beyond the stopping criteria.
Dividing by Standard Deviation has some sporadic movement around the middle, but only for the validation set. In addition to this, it is also the model that trains for the longest time, suggesting that it struggled to converge, since it did not reach a higher validation accuracy than the other models. This, again, shows similar behaviour as *Subtract Mean*, in that the validation loss becomes very flat between the values 0.4 and 0.6.
The *Standard Score* looks very similar to *Dividing by Standard Deviation*, only with a smoother validation curve. The validation loss plateaus around 0.4, while the training loss keeps decreasing, suggesting overfitting. As expected of this technique, it is the one that converges the quickest, at around 27 epochs.
In addition to these losses, if we look at the accuracies, we might see why the networks are allowed to plateau in the loss without being stopped by the early stopping.

We see some similar tendencies in the accuracy curves, as we saw in the loss curves. The *Min-Max* scaling still shows very nice validation and training curves.
If we look at subtracting the mean, we see why the model was allowed to keep training. Though it does not increase by much, it is increasing more than 0.5%, which would be 0.005 on the y-axis. The overfitting issue is also very pronounced, where the training accuracy keeps increasing steadily

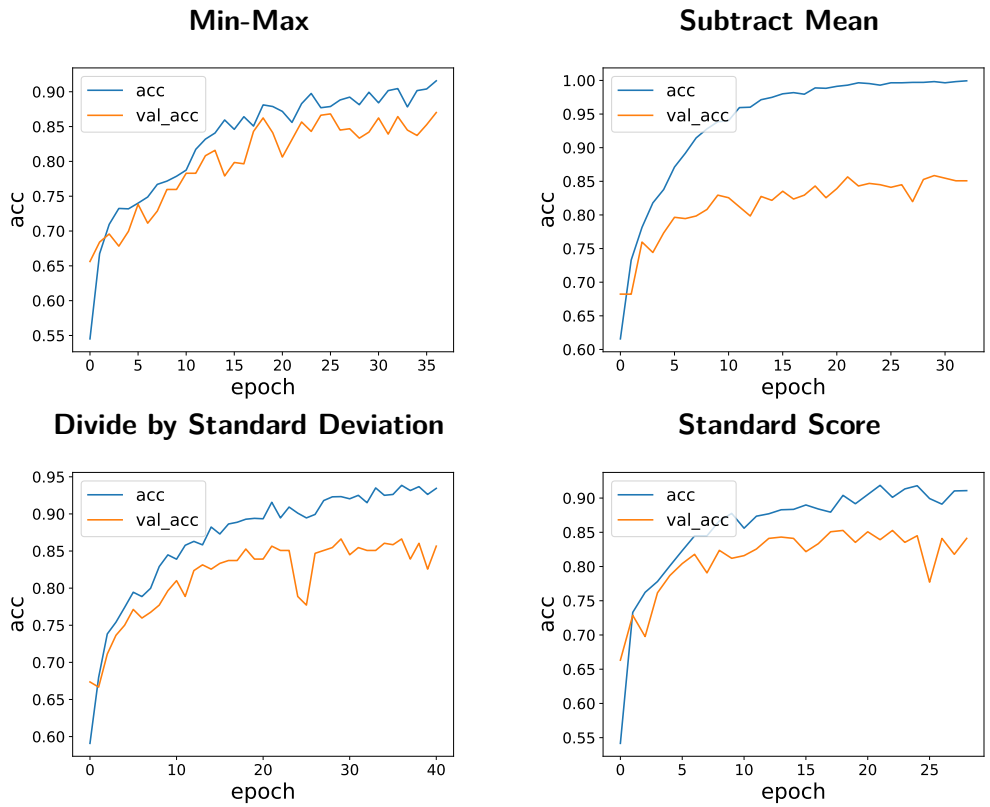


Figure 3.4: The accuracies for the normalisation experiment.

past epoch 5, while the validation accuracy starts to become flat.
For the standard deviation, the validation and training curves also follow each other at a decent pace. The drop in accuracy around epoch 25 coincides with the spike in loss, meaning that epoch had trouble with generalising.
Lastly, the *Standard Score* shows again that it converges quickly around epoch 27, and the validation curve follows the training curve well, indicating little overfitting.

Since the *Min-Max* normalisation reached the highest validation accuracy, we will be using this for future experiments.

3.2 Input Method

In this section we will describe the various ways of reshaping images to the same uniform shape and outline the experiments we did to find the best one for our network.

For handling the images we make use of Python Imaging Library (PIL). We are considering two overall methods; resizing, and a hybrid between resizing and zero padding. To explain how these techniques affect an image we will use Figure 3.5, on page 25, as an example.

Figure 3.5: The Cubism painting La Tricoteuse, by Jean Metzinger, 1919.
Dimensions: 447x641

Resizing

Resizing is a common processing technique in the literature [10, 9]. It involves down- or upsampling the image to reach a certain size. We use PIL for handling images, as it provides a number of resizing methods. To show how the resizing affects an image we will use Figure 3.6a, since Figure 3.5 already has been downsized by LaTeX.

(a) A 200×200 centre crop of Figure 3.5.

(b) Figure 3.6a resized to 100×100, using Lanczos resampling.

Figure 3.6: The effects of downsizing.

For the experiments, we are using the most sophisticated resizing method available in PIL,

known as Lanczos resampling. It works by approximating the largest eigenvalue and eigenvector of the pixel values and samples the new values from these. Figure 3.6b shows an image resized using Lanczos.

As we can see, the image is downsized with little artefacts apart from a small amount of blurring. One disadvantage is that Lanczos does some heavy calculations to approximate these eigenvalues, meaning using this technique too often might increase computation time.

Hybrid

Resizing an image to a square shape can affect the result for all images that are not originally square, such as Figure 3.5, on page 25. Resizing this to a 100×100 image like Figure 3.6b, would result in Figure 3.7.

Figure 3.7: An example of resizing a rectangular painting to a square shape.

What we can see from the image is that the shapes in the image are stretched, changing how they look which could negatively affect predictions because the more rectangular a painting is, the more stretched the result would be. Therefore, we are considering a method that resizes the image but keeps the ratio between the dimensions, which we do by first calculating the ratio between the dimensions. Then we resize the largest dimension to the desired size, and multiply this with the ratio to find the smallest dimension. This creates a rectangular image with the same ratio as the original image. We then pad the rest of the image in zeros to get a square image for the network. Figure 3.8, on page 27, shows an example of downsizing Figure 3.5, on page 25, to 200×200.
As we can see, the actual painting has the same shape as the original and the width is padded with black to fill out the rest of the image. By doing this we avoid distorting the shapes in the image at the cost of some added noise in the form of zero padding.

Experiment

As with the previous experiment we use the two classes, *Cubism* and *Neoclassicism*, as the dataset. Again, we employ early stopping, with a maximum of 100 epochs, and average the results across 5 runs. We start with the original images and resize them to 224×224 and we use the *Min-Max* scaling from the previous experiment to normalise the values. Since the experiment with normal resizing is the same as the *Min-Max* approach from the previous experiment, we will compare the hybrid technique with the *Min-Max* results.

Figure 3.8: An example of the hybrid method.

Results

Table 3.3 shows the results from the input shape experiment.

	Train Loss	**Validation Loss**	**Train Accuracy**	**Validation Accuracy**
Resize	0.2552	0.3795	89.8% ± 1.2%	**84.3%** ± 1.4%
Hybrid	0.2983	0.4237	87.4% ± 2.2%	80.0% ± 2.4%

Table 3.3: The results of the input experiment.

Interestingly, the hybrid approach did not do as good as the normal resizing. While it may keep the aspect ratio in the image, the fact that for some images there are more zeros than others may have affected how the weights are updated. In addition to this, there could be areas of the input that are very rarely activated thus the weights associated with the input neurons in those areas are rarely trained. Therefore, even if the shapes become stretched, the resizing did the best. In contrast with the hybrid approach, normal resizing has no dead areas in the input and all weights are being trained.
Figures 3.9, on page 28, and 3.10, on page 28, show the loss and accuracies, respectively, of the input experiments.

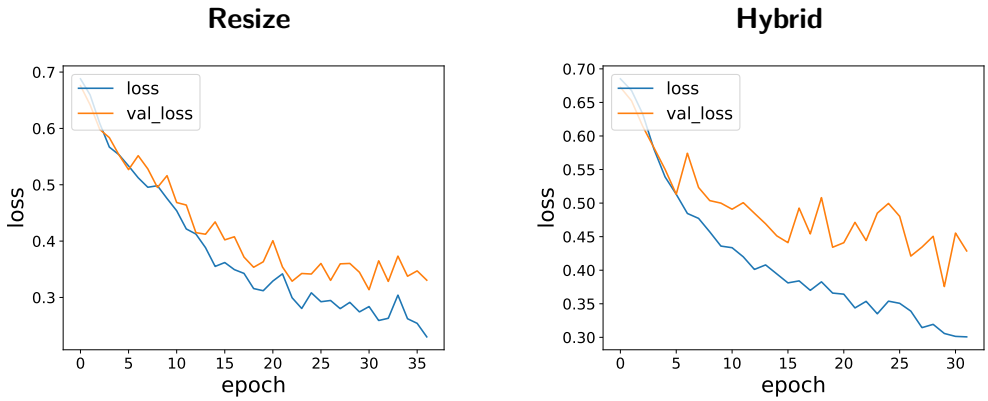


Figure 3.9: The losses for the input representation experiment.

The resize loss curve is the same as *Min-Max* in Figure 3.3, so we will not be reiterating the analysis. Seemingly, the hybrid approach converges faster than resize. However, the training and validation curves start to separate around epoch 5, where the validation loss spikes and then does not decrease as fast as the training loss. The hybrid approach also seems to struggle to generalise the features it finds during training.

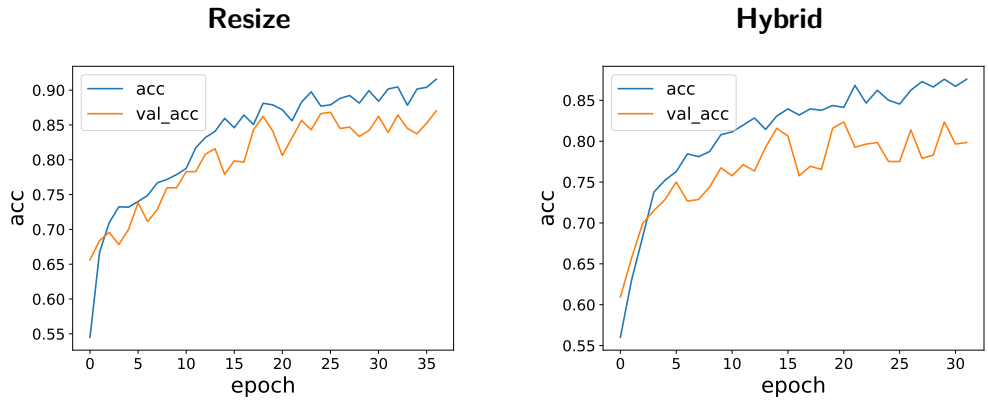


Figure 3.10: The accuracies for the input representation experiment.

As opposed to the losses, the accuracies look fairly similar while still showing that the resizing generalises better to the validation set.

3.3 Input Size

For the experiment in Section 3.2, we resized all the input images to 224×224, because this was the size from our baseline method, explained in Section 2.2.2. In this section we will investigate the effect different input sizes have on the classification performance of the network.
For this experiment we consider 5 image sizes;

- 128×128

- 224×224
- 256×256
- 384×384
- 512×512

Apart from the changed input sizes, the network described in 3.1, on page 18, is still used for these experiments.

Experiment

As with the previous experiments, we use the classes *Cubism* and *Neoclassicism*.
In addition to the other metrics, we will also report overall running time of the training across the 5 runs and the running time per epoch. We look at running time to decide whether a given image size is worth using; if it provides a negligible accuracy increase at a significant computation cost it would not be useful to implement.
Similarly to the previous experiment, the 224×224 size, with *Min-Max* scaling and normal resizing is the exact same as in the previous two experiments so we will repeat these results for easy reference.

Results

Table 3.4 shows the results of the input size experiment.

	Train Loss	**Validation Loss**	**Train Accuracy**	**Validation Accuracy**	**Running Time**	**Running Time (per epoch)**
128x128	0.3542	0.4448	86.0% $\pm$ 3.4%	80.5% $\pm$ 2.3%	05:11	~1s
224x224	0.2552	0.3795	89.8% $\pm$ 1.2%	**84.3%** $\pm$ 1.4%	19:01	~5s
256x256	0.2782	0.4036	88.8% $\pm$ 3.0%	83.0% $\pm$ 1.4%	22:07	~7s
384x384	0.2095	0.4043	92.0% $\pm$ 0.9%	83.5% $\pm$ 2.2%	01:03:10	~15s
512x512	0.2330	0.3942	91.0% $\pm$ 2.0%	83.8% $\pm$ 1.5%	01:27:47	~26s

Table 3.4: The results of the dimensions experiment.

As with the previous two experiments, resizing to 224×224 has the highest validation accuracy. 128×128 does the worst, which could be because we used a network with five max pooling layers, resulting in a 4×4 image, whereas for 224×224 results in a 7×7 image. This means that there is less data to generalise from, resulting in lower accuracy. Conversely, both the overall running time and the per epoch running time is very short, which can either mean that because of the smaller image size the network learns faster or it plateaus faster and is stopped by the early stopping.
As mentioned before, the 224×224 row is the same as the previous experiments, apart from the running times. Since the per epoch running time is 5 times as long as for 128×128, we could

expect the overall running time to also be 5 times as long. However, the overall running time is only 4 times as long, which means that, on average, the model converges faster for 224×224.
For 256×256 the interesting result is the deviation in the validation accuracy compared to the training, and comparing these to 224×224. The deviation in the training accuracy is more than twice as large as the validation accuracy, meaning, for 256×256, there is a lot of variance in the training set, but it still generalises well to the validation set. Another point is that the validation deviation is the same for 224×224 and 256×256 while the accuracy is lower for 256×256, indicating that, on average, it has more mis-predictions.
Looking at 384×384, the overall running time surpasses one hour and the per epoch running time is twice as long as the previous image size. We also notice a slight overfitting, in that the difference between the validation accuracy and the training accuracy is almost 10%. If we combine this with the training deviation, we see that it generally overfits to around the 90% mark, while the validation accuracy is anywhere between 85% and 80%.
The results for 512×512 are very similar to 384×384. Interestingly, it overfits less than 384×384, though not significantly and it also has a higher deviation, meaning for some runs it could potentially overfit less.

Figures 3.11, on page 31, and 3.12, on page 32, show the losses and accuracies for the best runs for the size experiment.

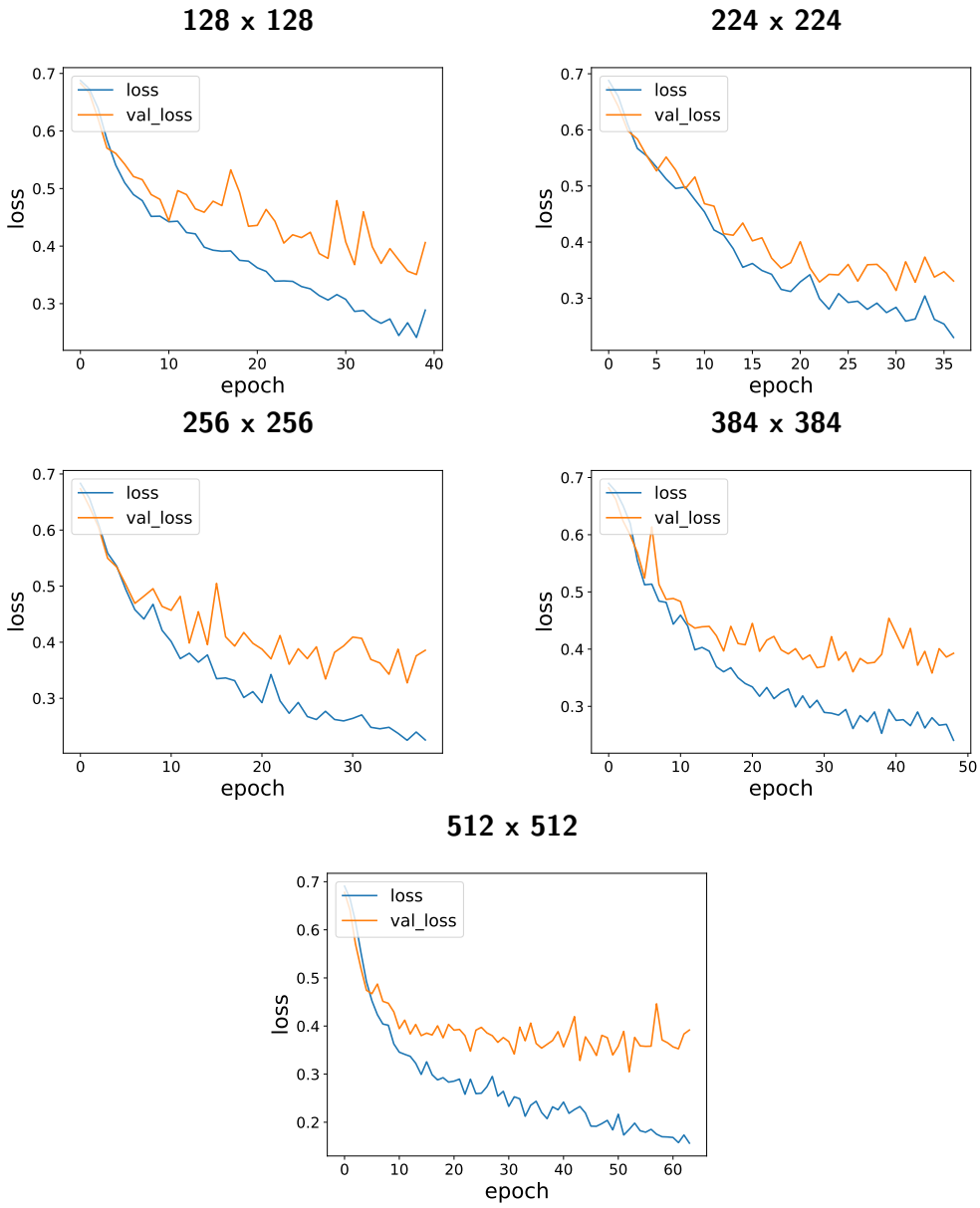


Figure 3.11: The losses for the size experiment.

We can see that for 128×128, 224×224, and 256×256 the network trains for about equal amount of epochs and is stopped between epoch 35 and 40. Most of the curves show that, around epoch 10, the validation loss starts separating from the training curve and is stopped with a fairly large gap between the two curves. Other than that, we see that the networks generally need to train longer the larger the image size is which is to be expected since there is more data that needs to be generalised.

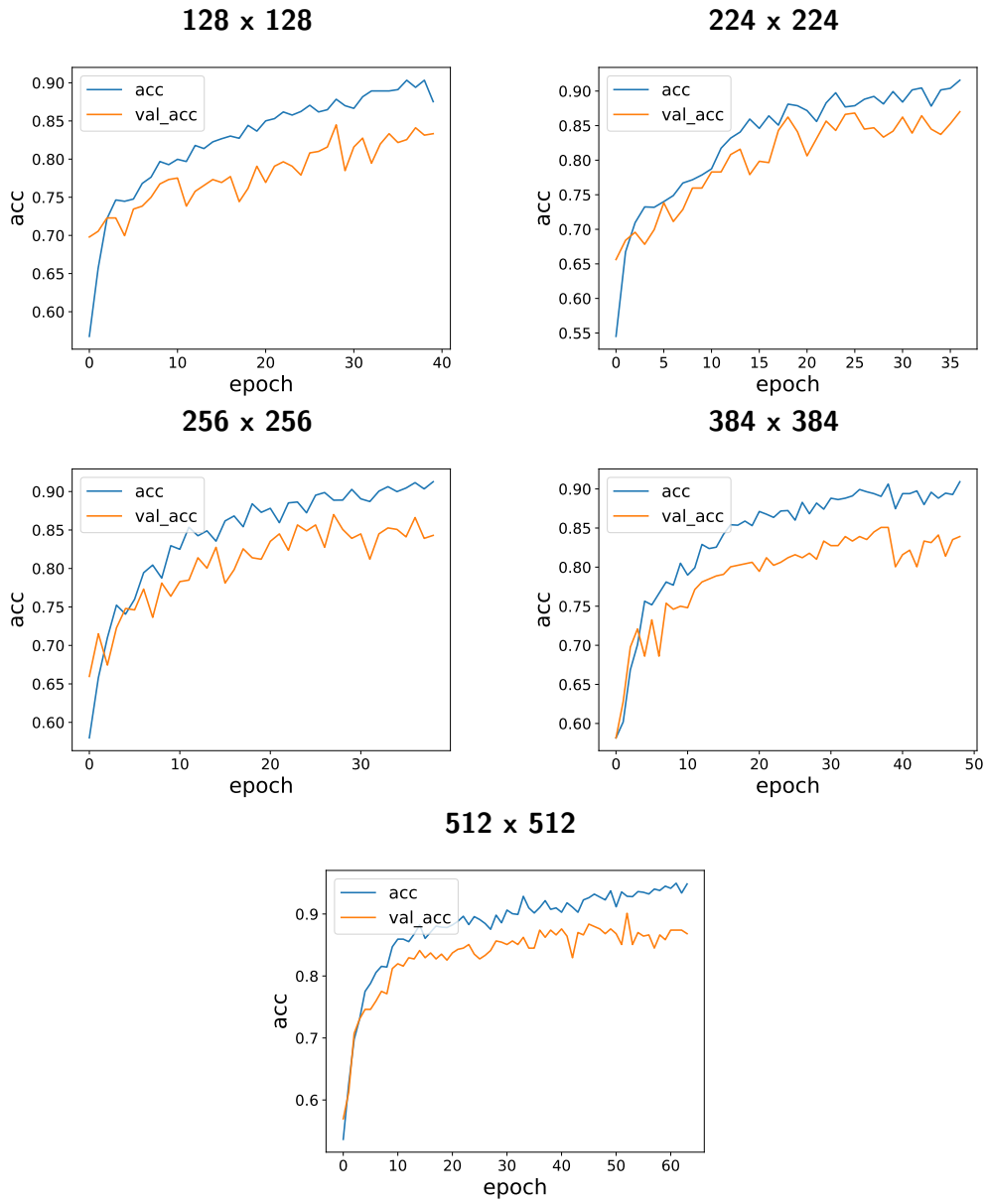


Figure 3.12: The accuracies for the size experiment.

We can see that all the validation curves follow their respective training curves in rate of growth but only for 224×224 are the curves very close. We also see that for 384×384 the training is stopped around epoch 39, where there is a large drop in validation accuracy. When the early stopping compares the last epoch to the one 10 epochs prior, it is below the threshold and training is stopped. It is likely that if the drop did not happen, the training could have continued further.

Summary

Throughout this Chapter we have experimented with various normalisation techniques, resizing methods and image sizes. Using the basic network architecture presented in Table 3.1, on page 18, normalising the input with *Min-Max* scaling gave the best results. Furthermore, a normal resizing method gave better results than padding the input with zeros. Finally, we found that keeping the image size to 224×224 provided the best results, so we will keep using that as the image size for our network.

4 Architecture

The core task in designing a CNN is choosing a fitting architecture for the problem. There are many architecture components that can be changed which can influence the recognition performance in different ways. Most of the relevant ones are mentioned in 2.1.1.

To be able to learn distinctions between the samples of different classes, the network should have enough parameters. This can be achieved by increasing the number of layers or the number of feature maps in the network. Therefore, we will experiment with the amount of layers and feature maps as these changes can have the biggest influence on the recognition performance of the network.

4.1 The Number of Layers

So far, CNNs models provide state-of-the-art performances in visual recognition problems. Deeper models have larger capacity which is beneficial for capturing hierarchical structures found in images. However, more layers increases the complexity of the model which often makes it difficult to train. Depending on the recognition task, smaller network architectures can produce equally good results while avoiding the problems of deeper models.

A common architecture pattern for CNNs is stacking blocks of convolutional layers with max pooling layer at the end of each block. This can also be observed in the architecture of our baseline network, mentioned in 2.2.2.
The number of layers the CNN should have can depend on the spatial size of the input. When applying max pooling or using a stride bigger than 1 in convolutional layers, the spatial output size is reduced. Applying such reductions many times can obtain better features. On the other hand, applying it too many times would scale down the spatial size to the degree when no further processing would make sense as the features might not be useful any more. Therefore, with bigger spatial input sizes we can have more blocks, i.e. layers. In the VGG16 network, for example, the spatial input size is 224×224 and the spatial output size before the fully connected layers is 7×7. After the blocks of convolutions and max pooling, there are usually a few fully connected layers. Fully connected layers are used to construct combinations of the features learned in the convolutional layers. Last fully connected layer is the output layer, producing predictions.

Because of the increased capacity, deeper CNN models produce more expressive features and can capture complex structures in images. It has been found in [5] and [6] that increasing the number of layers in the network leads to a better classification performance. As stated in [5], finding bad local minima is less probable when training deeper models. This is because deeper models have many local minima that are easy to find and they result in test accuracies that have roughly the same values. With many parameters, deeper models can learn more complex functions and, therefore, can learn training set very well reaching high training accuracy. Nevertheless, deeper models tend to have a high variance caused by modelling random noise in the training set, which leads to overfitting and decrease in test accuracy. Employing regularisation techniques can reduce the problem of overfitting, yet it can still be challenging to train deeper model as it requires more time and computational power. Adding more layers at some point results in only a slight improvement in classification performance while the number of parameters can significantly

increase. This trade-off between the classification performance and complexity might not always be profitable if approximately the same results can be obtain by a smaller model, which is less demanding to train. On the other hand, removing the layers and decreasing the number of parameters can result in a CNN model that is too small. Training models that are too small is also problematic because they are more sensitive to initialisation, often getting stuck in local minima with low test accuracy. Training the network multiple times can indicate whether the model is too small by observing the variance in test accuracies. High variance in test accuracy means that the model is too small and dependent on initialisation, while low variance means that the model produces stable solutions and, providing that the test accuracy is high enough, is fitting for the problem.

Experiment

As in the Section 2.2.5, the dataset for this experiment consists of two classes, *Cubism* and *Neoclassicism*. The dataset is split into train, validation and test set with a $60/20/20\%$ split respectively. We employ early stopping with a maximum of 100 epochs to limit the amount of training time and prevent overfitting. The stopping criteria is when the validation accuracy has not increased by at least 0.5% compared to the 10^{th} epoch prior. Networks are run 5 times and we average the result reporting both training and validation loss and accuracy with standard deviation for the runs. Standard deviation will help us to discern models that are too simple and prone to initialisation. We also report running time and per epoch running time to see if the possible accuracy increase is worth the gained increase in training time. We plot the loss and accuracy curves of the run that reached the highest validation accuracy.

Given the results from the experiments in Chapter 3, we use an input image size of 224×224, *Min-Max* scaling and normal resizing for preprocessing. Due to the 224×224 input size, we only consider a model of no more than 7 layers, which can be seen in Table 4.1, on page 36.
The general approach in this experiment is to incrementally increase the number of blocks. For example, a model with 4 blocks consists of the Input layer, the first 4 blocks and the 2 Fully connected layers, in Table 4.1. First, we experiment with one convolutional layer in each block and then with two identical convolutional layers in each block.

Results

Blocks with a single convolutional layer

The results of the first experiment, with one convolutional layer in each block, are shown in the Table 4.2, on page 36.

	Layer	Size	# filters	Stride	Output
	Input				$224 \times 224 \times 3$
Block 1	Conv	3×3	8	1	$224 \times 224 \times 4$
	Max pool	2×2		2	$112 \times 112 \times 4$
Block 2	Conv	3×3	16	1	$112 \times 112 \times 8$
	Max pool	2×2		2	$56 \times 56 \times 8$
Block 3	Conv	3×3	32	1	$56 \times 56 \times 16$
	Max pool	2×2		2	$28 \times 28 \times 16$
Block 4	Conv	3×3	64	1	$28 \times 28 \times 32$
	Max pool	2×2		2	$14 \times 14 \times 32$
Block 5	Conv	3×3	128	1	$14 \times 14 \times 64$
	Max pool	2×2		2	$7 \times 7 \times 64$
Block 6	Conv	3×3	256	1	$7 \times 7 \times 128$
	Max pool	2×2		2	$4 \times 4 \times 128$
Block 7	Conv	3×3	512	1	$4 \times 4 \times 256$
	Max pool	2×2		2	$2 \times 2 \times 256$
	Fully connected				8
	Fully connected				# of classes

Table 4.1: The largest possible model for the architecture experiment

# layers	**Train Loss**	**Validation Loss**	**Train Accuracy**	**Validation Accuracy**	**Running Time (mm)**	**Running Time (per epoch)**
1	0.4598	0.6378	72.0% $\pm$ 20.8%	64.4% $\pm$ 9.5%	12	~3s
2	0.1848	0.4361	94.2% $\pm$ 1.8%	79.9% $\pm$ 2.0%	13	~4s
3	0.2942	0.4432	90.6% $\pm$ 4.8%	79.9% $\pm$ 2.3%	17	~4s
4	0.29	0.4128	88.0% $\pm$ 2.4%	79.9% $\pm$ 2.5%	14	~5s
5	0.2544	0.3654	89.7% $\pm$ 2.1%	83.5% $\pm$ 1.1%	17	~5s
6	0.2754	0.4426	92.0% $\pm$ 2.5%	**84.4%** $\pm$ 1.9%	18	~5s
7	0.2566	0.4683	92.4% $\pm$ 3.2%	82.5% $\pm$ 1.8%	13	~5s

Table 4.2: The results of experiment with model depth.

We can see that by increasing the number of blocks, we achieve higher accuracies, with the exception of the 7 block network. As expected, the model with 1 block achieves poor accuracy and a high variance in both train and validation. It seems that this model is too small to learn to distinguish the two classes. The models with 2, 3 and 4 blocks achieve higher validation accuracy than the 1 block model, with the same accuracy and roughly the same standard deviation. All three models overfitted, meaning that we could have used a more strict early stopping criteria. The same can be concluded for the 5, 6 and 7 block models. But despite overfitting, they reached higher accuracy, with the 6 block model reaching the highest accuracy. Additionally, the increase in blocks did not increase per epoch running time, which means that adding blocks in this case

would be a reasonable architecture choice.
Interestingly, the 7 block network did not achieve higher accuracy than 5 and 6 block networks. In the 7 block network we can also observe that the total running time is much shorter than for the other networks, which means that it stopped earlier. This can indicate that the learning rate is not appropriately set for this model, making it converge faster and ending up in a local minima with lower accuracy, which causes the early stopping. Aside from that, after 7 max-pooling layers, the 7 block model ends up with a spatial size of 2×2 which might be too small and, therefore, does not produce better features than, for example, the 6 block model. This can also be the reason for lower accuracy of the 7 block model.

Loss and accuracy curves for this experiment can be seen in Figures 4.1, on page 38, and 4.2, on page 39, respectively.

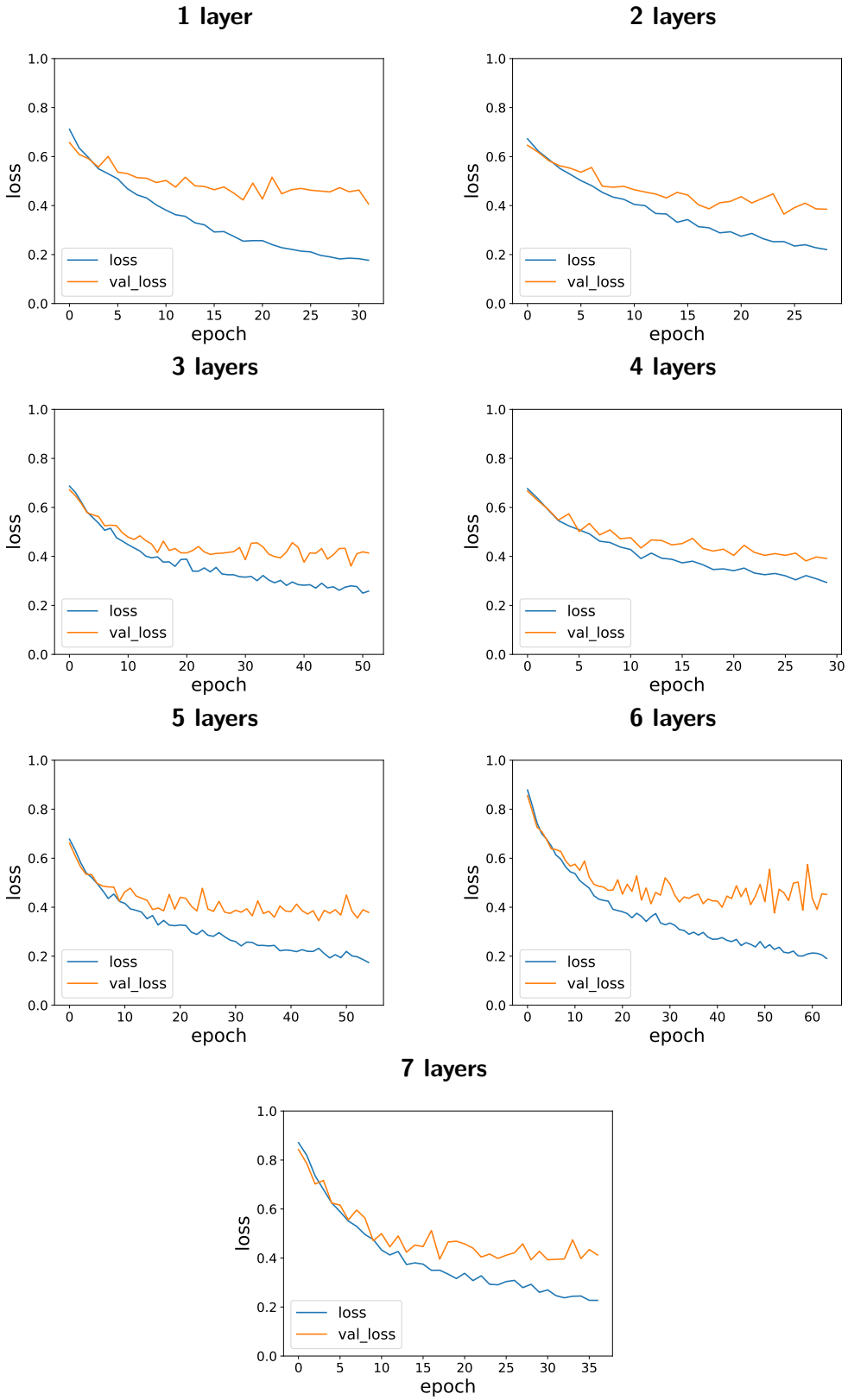


Figure 4.1: The losses for the experiment with model depth.

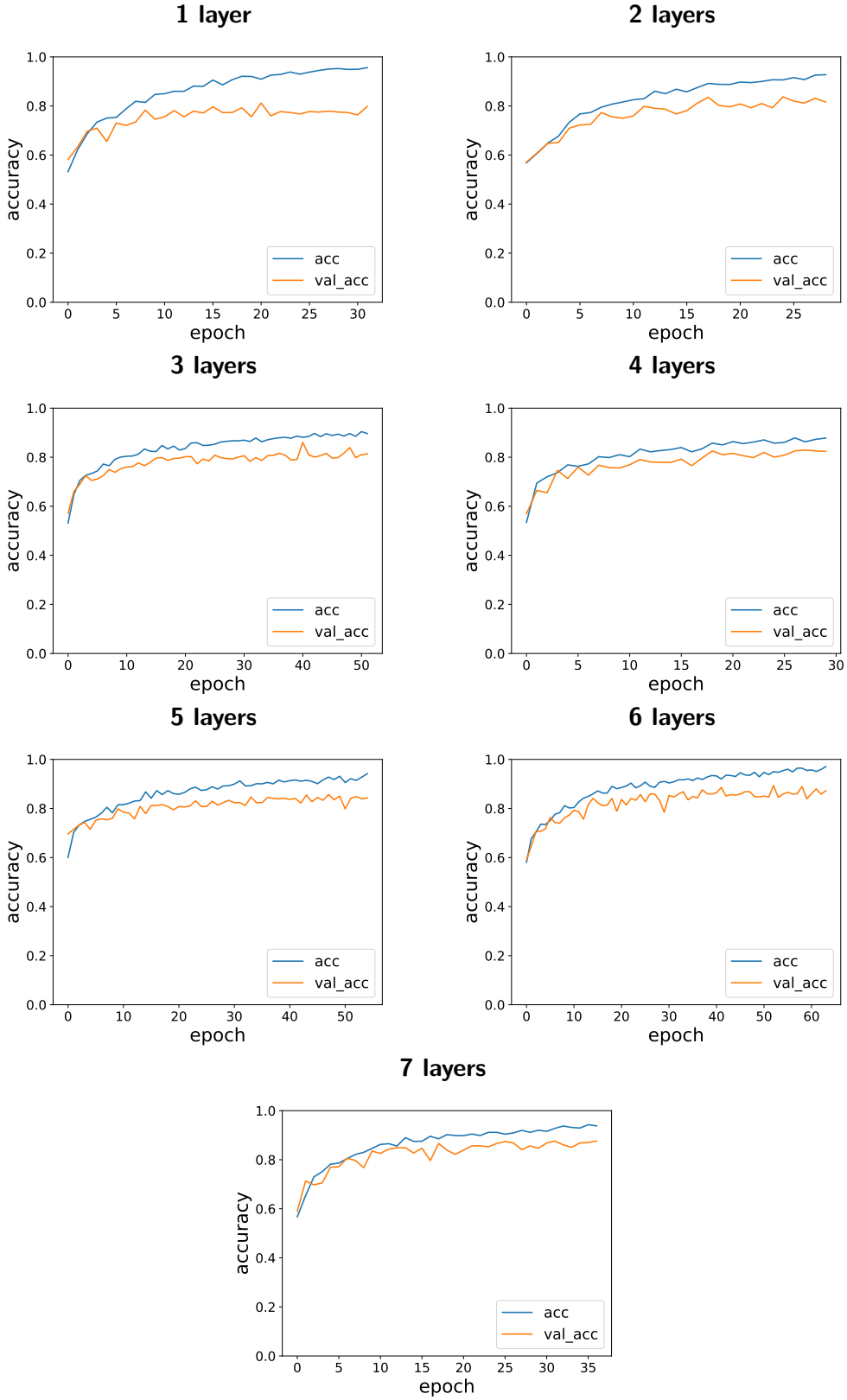


Figure 4.2: The accuracies for the experiment with model depth.

The validation curves show that all models reach almost the same loss, around 0.4. In all of the models, the gap between the train and validation loss increases over time because they overfit the training data, which is also noted in the tables.
As for the accuracy, we can see the strong overfitting in the models with 1 and 2 blocks. The validation curves of the other models mostly seem to follow the training curves. All models reach roughly the same training accuracy, but the 1 and 2 blocks models can not reach the validation accuracy as high as other, deeper models, which corresponds to the results in the Table 4.2.

From the result we can conclude that increasing the number of blocks also increases the validation accuracy with the reasonable trade-off with running time. The models with 5, 6 and 7 blocks provided roughly the same performance and the differences could be caused by the initialisation or early stopping technique. That is why we chose to use these three models in our further experiments.

Blocks with double convolutional layers

The next experiment is with double convolutional layers in each block. Aside from the three models that performed the best in the above experiment, we decided to also include the model with 4 blocks to see if it will provide better accuracy now, with double convolutions. The results of the experiment are given in the Table 4.3.

# layers	Train Loss	Validation Loss	Train Accuracy	Validation Accuracy	Running Time (mm)	Running Time (per epoch)
4	0.271	0.4299	88.9% ± 2.8%	81.3% ± 1.8%	23	~9s
5	0.2799	0.4102	88.1% ± 2.5%	80.9% ± 2.9%	18	~9s
6	0.2771	0.4678	90.2% ± 2.7%	82.1% ± 1.4%	34	~10s
7	0.1696	0.5261	95.3% ± 2.7%	**83.1%** ± 3.5%	38	~11s

Table 4.3: The results of the experiment with model depth with double convolutions

We can see that by increasing the number of blocks, we achieve higher validation accuracy, with the exception of the 5 block network. The validation accuracy of the 5 block network is only 0.4% worse than the 4 block one and has the highest variance, which might be due to initialisation. We can again see that all models overfit, where 7 block network overfits the most. This might be the reason why the 7 block network also has the highest variance and validation loss. Additionally, it runs the longest, around twice as long as the 4 and 5 block networks, which explains why it overfits.
Comparing this table with Table 4.2, the results are a bit strange. Only the 4 and 7 block networks increased in performance. The 4 block network seem to be the only one that benefits from double convolutions. Even though 7 block network recorded increase in validation accuracy with double convolutions, we can see that the variation is much higher compared to the model with 7 blocks

with single convolutions. Taking into the account the strong overfitting and long training time, we can conclude that 7 block model with double convolution is too complex and that is why it exhibits such behaviour. Same assumption could be drawn for the 5 and 6 block network with double convolutions, though they overfit less than their single convolution counterpart, yet have lower validation accuracy. Adding double convolutions resulted in longer running time and running time per epoch. Since the final results are not better than the results with single convolution blocks, we deduce that adding convolutions does not provide significantly better outcome.

Loss and accuracy curves for this experiment can be seen in Figures 4.3 and 4.4, on page 42, respectively.

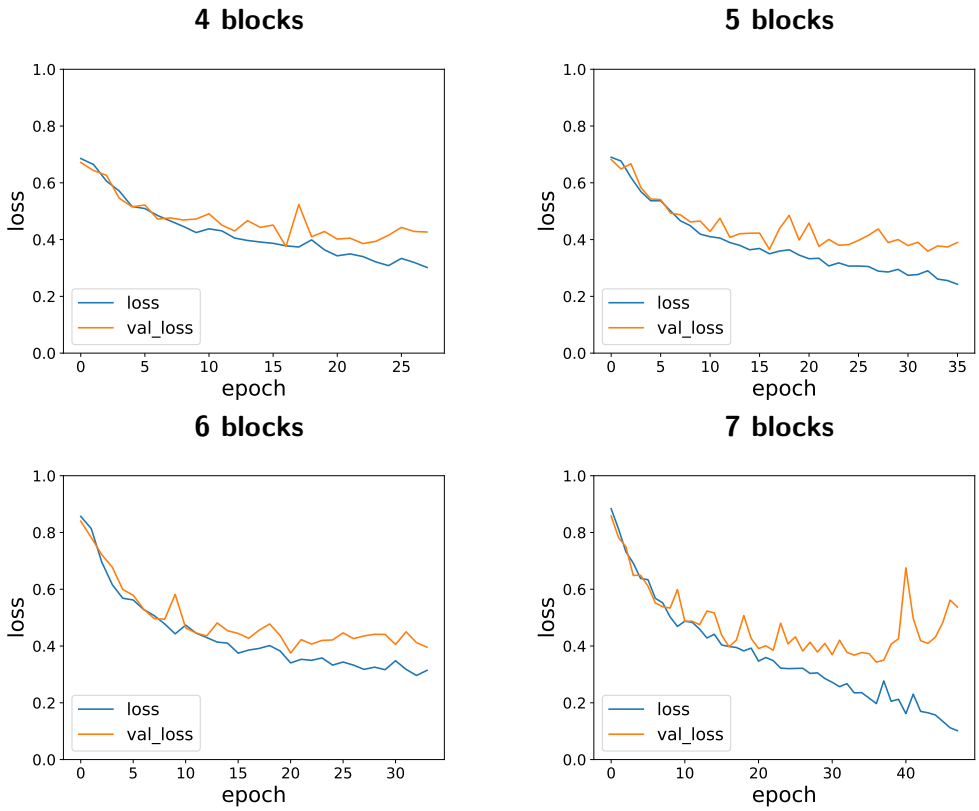


Figure 4.3: The losses for the experiment with model depth with double convolutions in each block.

The loss plots for double convolution blocks are similar to the loss plots with single convolution blocks. The validation loss decreases and follows the training loss, but after a certain point, the training loss keeps decreasing, while validation loss starts to oscillate around 0.5 value, creating a bigger gap between the training loss and the validation loss. More interesting behaviour can be seen in the validation loss of the 7 block network, which had a sudden increase in loss around the 40th epoch and just before stopping. This is probably because of the strong overfitting and rapid decrease in training accuracy, indicating that the model found its global minimum. Despite

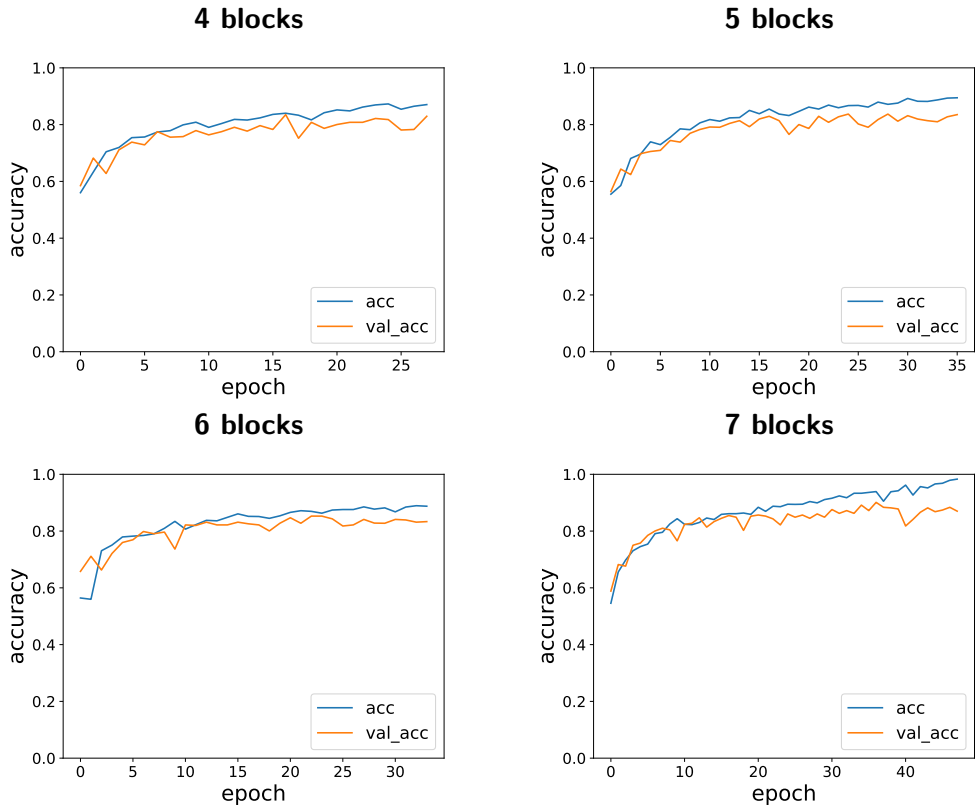


Figure 4.4: The accuracies for the experiment with model depth with double convolutions in each block.

this strange behaviour, the model with 7 blocks achieves higher accuracy than other models. In its accuracy plot we can notice the decrease in validation accuracy at the same point where the sudden increase in validation loss was reported. Other than that, we can see that all models reach over 80% of validation accuracy, where the models with more blocks reach it slightly faster and also train for longer.

Overall, double convolutions did not improve classification performance. Slight increase in validation accuracy is reported in some cases, but with noticeable cost of running time. Furthermore, the results of the experiments with blocks with single convolutions have less variance. Therefore, in our future experiments, we decided to evaluate models with 5, 6 and 7 blocks with single convolutional layer.

4.2 The Number of Feature Maps

Similar to adding layers, adding more feature maps increases the number of parameters as well as the capacity of the CNN. With more feature maps, the CNN can look for more features in the images. The features can then be more specialised for certain characteristics exhibited in the dataset, such as patterns, edges or colours. By having more features in a layer, succeeding

layers can produce higher quality features. As mentioned in 2.1, the number of feature maps is in correlation with the number of blocks. Since the input is scaled down by 2 along each dimension when using default max-pooling, the number of feature maps is doubled to compensate for the decrease in the number of parameters. Furthermore, the deeper the network is, the more features it should have, as the features that are constructed in the later blocks are more specialised. As with more blocks, the drawback for having large number of feature maps is increased complexity which causes the problems with training and overfitting.

Experiment

The settings for this experiment are the same as in Section 4.1, with the exception of the model, where we change the number of feature maps. We report running time and per epoch running time to see if the change in the number of feature maps is worth the potential increase in validation accuracy. We plot the loss and accuracy curves of the run that reached the highest validation accuracy.

We use the input image size of 224×224, *Min-Max* scaling and normal resizing for preprocessing. The architecture is the same as in Figure 4.1, with the exception of the number of feature maps. We experiment with 16, 32 and 64 feature maps in the convolutional layer of the first block and we double the number of feature maps after each block. Considering the results from the experiment with model depth, Section 4.1, we only experiment with 5, 6 and 7 blocks with single convolutional layer in each block.

Results

16 feature maps The results of the experiment with 16 feature maps are shown in the Table 4.4.

# layers	Train Loss	Validation Loss	Train Accuracy	Validation Accuracy	Running Time (mm)	Running Time (per epoch)
5	0.2165	0.4014	91.2% $\pm$ 0.8%	82.6% $\pm$ 1.8%	23	~7s
6	0.2439	0.473	93.8% $\pm$ 2.0%	83.9% $\pm$ 2.0%	26	~8s
7	0.1893	0.4766	96.4% $\pm$ 2.5%	**84.3%** $\pm$ 1.0%	24	~9s

Table 4.4: The results for the experiment with feature maps, starting from 16 feature maps.

32 feature maps The results of the experiment with 32 feature maps are shown in the Table 4.5.

# layers	Train Loss	Validation Loss	Train Accuracy	Validation Accuracy	Running Time (mm)	Running Time (per epoch)
5	0.2266	0.4736	93.7% ± 1.7%	83.0% ± 0.6%	42	~13s
6	0.1155	0.4916	98.1% ± 0.4%	84.3% ± 0.9%	49	~15s
7	0.075	0.5776	99.1% ± 0.5%	**85.2%** ± 0.5%	44	~18s

Table 4.5: The results for the experiment with feature maps, starting from 32 feature maps.

64 feature maps The results of the experiment with 64 feature maps are shown in the Table 4.6.

# layers	Train Loss	Validation Loss	Train Accuracy	Validation Accuracy	Running Time (hh:mm)	Running Time (per epoch)
5	0.161	0.4619	96.4% ± 1.4%	**84.7%** ± 0.9%	01:10	~28s
6	0.083	0.5873	99.1% ± 0.4%	84.3% ± 0.5%	01:21	~36s
7	0.3274	0.7105	81.4% ± 21.5%	73.1% ± 13.3%	01:25	~50s

Table 4.6: The results for the experiment with feature maps, starting from 64 feature maps.

Combining these results with the results of the previous experiment, with 8 feature maps, seen in Table 4.2, on page 36, the highest accuracy is obtained by 7 block model starting from 32 feature maps. Doubling the number of feature maps from 8 to 16 did not perform better in terms of validation accuracy, though the variance decreased. Doubling from 16 to 32 and to 64 mostly reported higher validation accuracy, but also higher validation loss and training accuracy, which indicates stronger overfitting. Running time and running time per epoch is also longer, approximately twice as much. The exception is the 7 block model with 64 feature maps, where in a couple of runs it did not learn, as explained in Section 3.1, probably due to the bad initialisation and training difficulties caused by the complexity of the model. That is why it reported the lowest validation accuracy and a very high variance.
The plots for the losses and the accuracies are provided in Appendix A. The observations are not different from the observations for the previous experiment in 4.1 and therefore we will not describe them individually. The losses and accuracy plots for 16 feature maps experiment are shown in Figures A.1 and A.2, for 32 feature maps experiment in Figures A.3 and A.4 and for 64 feature maps experiment in Figures A.5 and A.6.
Since we got the highest validation accuracy with the 7 block model with 32 feature maps we will continue to use this architecture for further experiments. Even though it had the strongest overfitting, we believe that is not an issue as we will apply augmentation to handle the overfitting.

Summary

Through the experiments in this Chapter, we have investigated the possible configurations of the structural architecture of the CNN and the implications this has on the prediction accuracy. We

have explored two general components of structural architecture; the depth of the network, i.e. the number of blocks, and the width of the network, i.e. the number of feature maps in each block. As a result of the experiments, we have found that the following architecture illustrated in Figure 4.5 was best suited for this problem.

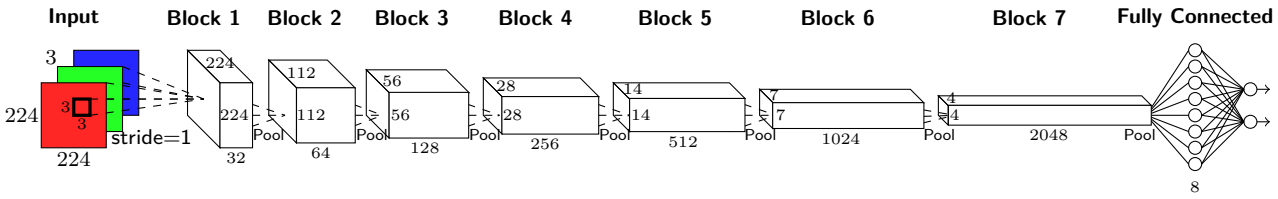


Figure 4.5: An illustration of our network structure.

5 Augmentation

Augmentation is a popular technique for artificially inflating a dataset with transformed training data. By processing the data in various ways, we can create new samples from which a model can learn new features. There exists a number of augmentation techniques for images, but we are considering three methods; flipping, cropping and blurring. In the following section we will explain how each of these affect the images, and why we chose them.
To show the effects of the augmentation we will be using Figure 5.1 as an example image.

Figure 5.1: The Neoclassicism painting *Portrait of the artist Delon* by Jean Auguste Dominique Ingres.

Flipping

For this augmentation, we flip the image along either the horizontal or the vertical axis. The main purpose of it is to add more samples to the dataset. It is simple is a simple way of translating the images but it does not add much in terms of content. When feeding the images to the network, we randomly flip the images either horizontally or vertically.
Figure 5.2 shows how the flipping affects an image.
The network would see Figures 5.1 through 5.2, on page 47 as three different samples, albeit extremely similar. One disadvantage of this is that there is not much changed from the original to the augmented images, which could result in additional overfitting as opposed to alleviating it. Furthermore, since the flipping occurs randomly during an epoch, there is a risk that for some images, only the flipped version is shown, meaning there would be no augmentation. However, the randomness is decided at the start of each epoch and thus, when running for many epochs, the risk of this is very low.

Figure 5.2: Flipped versions of Figure 5.1, horizontal and vertical.

Blurring

By blurring the image, we can obscure shapes and objects in the image, making the network rely on colour regions to differentiate styles. This can make the features the network learns more robust as they also need to cover images where the content is warped. For blurring we use the `GaussianBlur` function of the OpenCV library. This translates pixel values using a gaussian kernel.
Figure 5.3 shows an example of a blurred image.

Figure 5.3: Figure 5.1, on page 46, after the gaussian blur.

The blurring in Figure 5.3 is done with a three by 3 kernel with a standard deviation of 3. The most notable thing is that the texture of the painting is almost completely smoothed, meaning the network cannot look for arbitrary details in the texture. Many of the shapes are still present, but the edges around these shapes are not as harsh as in Figure 5.1, on page 46. This means that the network must rely on regions of colour, which it could have ignored if it was only looking for sudden changes in colour.
On the other hand, some styles are characterised by sharp lines. By blurring their images, it would make them less representative of the style they belong to. This could cause difficulties for the network and obstruct it from learning such style-specific features.

Cropping

In this technique we take small crops out of the original image, label these crops, and use them as the data. We made a sliding window across the images and extracted the crops. Figure 5.4 shows examples of these crops.

Figure 5.4: Eight 224 by 224 crops from Figure 5.1.

We compute how many crops we can possibly make with the size we want which, in this instance, is 224, while allowing a maximum of 60% overlap. Figure 5.1, on page 46, is originally 517×512 pixels, which allows 8 crops of size 224×224 that can be seen in Figure 5.4. There is some translation difficulty with this approach, in that there can be areas of the image that are not seen if the image is large enough. For this reason we resize the largest images so that the larger of the dimensions is exactly 1000 and the other dimension is resized accordingly, to keep the ratio. This resizing is done offline and the resized images are then loaded and cropped on runtime.

The point of the sliding window is that the resulting crops should cover the whole image. By making such crops, we greatly increase the size of the dataset. Previously mentioned techniques are randomly applied to images and, at each epoch, only original or the augmented sample is seen meaning that the number of samples per epoch stayed the same. Unlike that, by cropping we actually increase the number of samples as all the crops are processed each epoch. The only drawback could be that separately processing crops might not have the same effect as processing

the entire image.
Ideally, each crop should be representative of the style of its corresponding image. However, sometimes the crops could show parts of images that do not say much about the style and can cause the CNN to try to learn features that are not relevant. Moreover, some crops could even be more representative of some other style. This would only add confusion to the CNN, increasing the probability of mis-classification.

5.1 Experiment

Due to time constraints we have chosen to use all the augmentation techniques at once, instead of experimenting with each of them individually. We use the two classes, *Cubism* and *Neoclassicism*. When these classes have been split with our static 60/20/20% split, there is $1,773$ images in the training set, from which we take a maximum of 10 crops. This gives us potentially $17,730$ crops that then, in turn, can be flipped and blurred once per epoch, giving us around $17,730 \cdot 3 \cdot 2 = 106,380$ training images in total.
We use the seven block model, starting from 32 feature maps, explained in Chapter 4, and we take 224×224 crops from the images while normalising the crops with *Min-Max* scaling. We train the model for 100 epochs without early stopping. Due to constraints on the *Abacus*, we can only run three of these networks, as opposed to the five runs we have had in previous experiments. Furthermore, we are performing the experiments in two different manners. We train the network on cropped images, but predict the validation images that are not cropped, but only resized. Another approach we will try is to also apply the sliding crop on the validation images and then average the predictions in an ensemble manner.
In this experiment we do not include losses of the CNNs because the losses were incorrectly computed for the augmentation and the ensemble approach.

5.2 Results

The results of the augmentation experiments can be seen in Table 5.1.

	Train Accuracy	**Validation Accuracy**	**Running Time (hh:mm)**	**Running Time (per epoch)**
No Aug	98.1%	84.7%	02:35	~18s
Aug	96.1% ± 0.2%	85.0% ± 1.3%	13:57	~166s
Aug + Ensemble	96.2% ± 0.01%	**91.0%** ± 0.8%	15:28	~185s

Table 5.1: The results of the augmentation and ensemble experiment.

As the results from the previous experiment for the model with 7 blocks and 32 feature maps was done with early stopping, we decided to run that model again without early stopping for this experiment, to better compare the results. After training the model three times, we noticed that

two of the runs exhibited a similar learning issue as the one we explained in Section 3.1, where the networks seemingly did not learn. Therefore, we decided to only include the run that learned, which is why there is no deviation for that row.
The order of these results are to be expected, with the augmentation and ensemble doing the best. The reason why this technique does significantly better than without ensemble, is that there is a higher correlation between the training data and the validation data.
Without ensemble, the network trains on crops of the training images. Based on these crops, it predicts the validation images, which are only resized. Additionally, it gets only one chance of prediction for each validation image as the validation set is not cropped.
Conversely, with the ensemble, it predicts crops from the validation images, which are now cropped and should more closely resemble the training images. Furthermore, if we take 10 crops from a validation image, we get a prediction for each of those crops and then average the predictions to make one prediction which is returned. This means that, for ensemble, the network potentially gets 10 chances to predict a validation image. This can have both positive and negative effect as the CNN gets 10 chances to predict correctly, but also gets 10 chances for mis-prediction.
If we look at the running time, the augmentation takes significantly longer to train, which is to be expected since the total amount of samples per epoch is around 10 times more than without cropping.

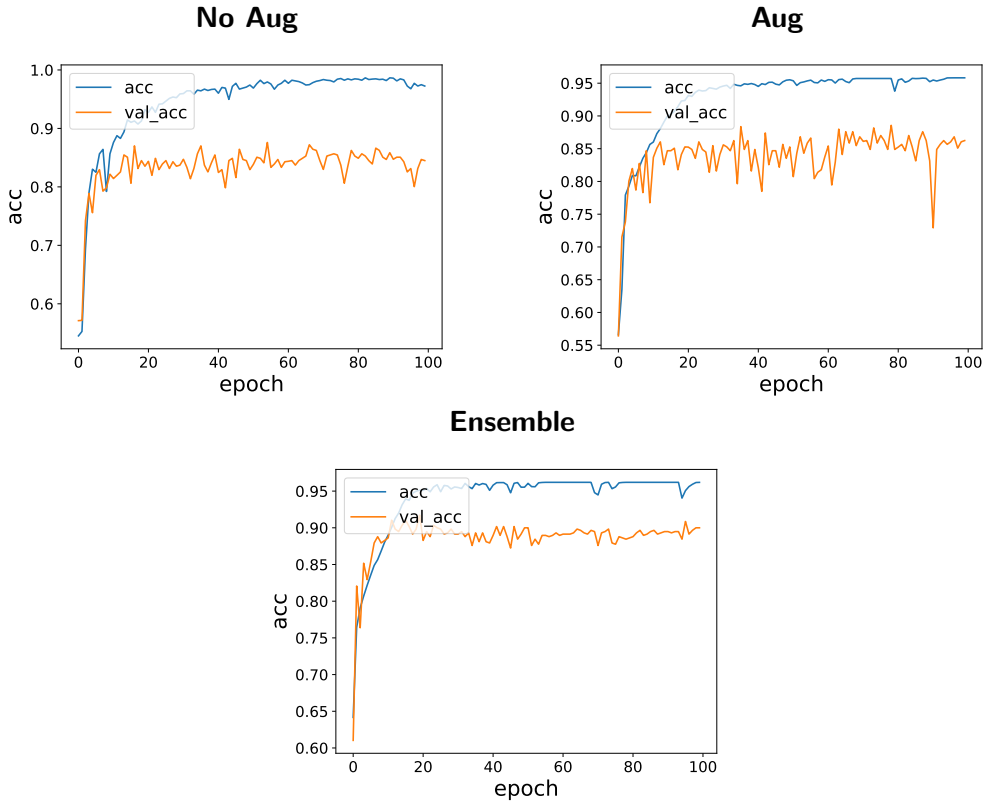


Figure 5.5: The accuracies for the augmentation experiment.

The plots for accuracies for this experiment are shown in Figure 5.5.

The first thing we notice from the accuracies is that the networks do not improve significantly over the majority of the training epochs.
For no augmentation we see that the training accuracy approaches 1, while the validation accuracy varies around the 0.80 mark, indicating that the features it learns from the training set are not adequately generalised to the validation set.
If we shift focus to the augmentation without ensemble, we see a very high variance in the validation accuracy, which could be due to the significant difference between the crops in the training set and the normally resized images in the validation set. In addition to that, we see that the validation accuracy does not seem to reach a much higher value comparing to the experiment without augmentation. This implies that the network does not benefit from the augmentation if the validation set is not augmented in a similar way.
Lastly, the ensemble approach has very little variance and at times has the exact same training accuracy across epochs. The gap between the training accuracy and the validation accuracy is also smaller than for the other experiments, meaning, as mentioned before, that the network is better at generalising the features it learns. Interestingly, it seems like the validation accuracy is increasing a little at the end of training, though that could also just be variance.

Given the significant accuracy increase of the ensemble approach, we will be using that in conjunction with the augmentation techniques.

6 Evaluation

In this Chapter we will be evaluating the final network and compare it to the baseline network explained in Section 2.2.2. Because our CNN is designed throughout the experiments using one set of classes, we will experiment with different sets of classes to investigate if our model is robust for other styles. To attain further knowledge about the learning process, we will visualise the activations in the intermediate layers in the networks. Aside from that, we will look into the mis-predicted images to see if we can understand the reason for mis-predictions. Lastly, we are interested in how painters affect the style and, therefore, the learning process. We experiment with a different data split, where there are no mutual painters between training set and validation and test sets. From this experiment we would like to know if the CNN learns the features of the styles or the painters that belong to the styles.

6.1 Comparative Experiment

Throughout this project we have focused primarily on the two classes *Cubism* and *Neoclassicism* since they had approximately the same amount of samples and were visually distinctive. However, this could result in a network that is specialised only for those two classes and thus, is not useful for additional or different classes. Therefore, we will use different pairs of classes for this experiment and compare our network with the baseline.
Overall, we are considering three pairs of classes:

Dataset A: *Cubism* and *Neoclassicism*, 1,773 training, 580 validation, 585 test

Dataset B: *Colour Field Painting* and *Magic Realism*, 856 training, 239 validation, 276 test

Dataset C: *Early Renaissance* and *High Renaissance*, 1,265 training, 413 validation, 424 test

We have labelled each pair of classes *A* through *C* for easy reference.
We chose *Cubism* and *Neoclassicism* because they were visually distinct, while still having many samples. *Cubism* has 1,316 samples and *Neoclassicism* has 1,622, meaning there is a slight imbalance between the classes.
As with *Cubism* and *Neoclassicism*, we also chose *Colour Field Painting* and *Magic Realism*, due to their significant visual dissimilarity. *Set B* was also chosen because the classes are relatively balanced, where *Colour Field Painting* has 675 samples and *Magic Realism* has 696.
As opposed to *Set A* and *Set B*, we chose the classes of *Set C* because they were visually similar, to investigate how well the network handles difficult classes. As for the amount of samples in each class, *Early Renaissance* has 1,052 samples and *High Renaissance* has 1,050 samples.

For this experiment we used the baseline method, described in Section 2.2.2, and compare this to our network structure, as shown in Table 4.5. We are using the augmentation and preprocessing techniques described in Chapters 5 and 3, respectively. We train our network once, on 50 epochs, without early stopping. The reason why we run it only once and why we removed early stopping is because of the time constraints. Since now we have more data, we expect the learning process to last longer and behave differently. Therefore, it should have different criteria for early stopping, but we did not have enough time to further experiment with it. From the

experience with previous experiments we concluded that 50 epochs should be sufficient for the network to learn and for us to be able to analyse it.

Because of the learning problems mentioned in Section 5.2, we decided to use a seed for weights initialisation. For all of our experiments we are using Glorot normal initializer [7] which draws samples from a truncated normal distribution with the mean $\mu = 0$ and the standard deviation $\sigma = \sqrt{\frac{2}{n_{in}+n_{out}}}$, where n_{in} is the number of inputs feeding into the filter and n_{out} is the number of outputs connected to the filter. By using the seed, we fix the values of the sampling of the weights reducing the risk of the learning problem.

In this experiment we will not report on losses, only accuracies, for the same reason as in the Section 5.1. We will look at all the accuracies, that is, training, validation and test, where test is the most important measure. Since the baseline method is using an SVM as the classifier, there is no training accuracy for this.

6.1.1 Results

The results from the dataset experiment can be seen in Table 6.1.

		Train Accuracy	**Validation Accuracy**	**Test Accuracy**
VGG16 + SVM	Set A	—	96.4%	96.1%
	Set B	—	98.3%	97.8%
	Set C	—	78.7%	74.1%
Our Network	Set A	96.1%	90.5%	91.6%
	Set B	96.2%	93.7%	95.3%
	Set C	94.1%	71.4%	67.7%

Table 6.1: The results of the dataset experiments.

Generally, our network does not perform as well as the baseline, on all the datasets.
Set A was the dataset that was primarily used throughout the project and, as we can see, there is a significant gap between our network and the baseline. Interestingly, the validation accuracy and the test accuracy of the baseline are relatively close, while it is slightly farther in our network, suggesting the baseline generalises better to new images. Nevertheless, both the baseline and our network have the smallest gap in validation and train accuracies for this dataset compared to the other datasets. This implies that the validation and test sets for *Set A* are equally difficult.
As for *Set B*, this was the easiest dataset to distinguish, both for our network and the baseline. It is also the dataset where the gap between the two networks is the smallest in terms of test accuracy. When looking at the differences between the validation and test set, we can see that the baseline has higher accuracy on the validation set than the test set, unlike our network, which performs better on the test set compared to the validation set.
As mentioned before, *Set C* was chosen because of how visually similar the classes are to each

other, which we can see in the accuracies. If we look at the test accuracy of our network, it is a little over random, i.e. 50%, while the validation accuracy is higher. It is also worth noting the training accuracy for this dataset, which is below the training accuracies for the previous two datasets, suggesting that our network has difficulties differentiating the training set. We can see that the baseline also has difficulties with this dataset, dropping significantly in both validation and test accuracies.

6.2 Network Analysis

As mentioned in Section 2.2.3, we can visualise the activations in the feature maps, as well as the filters, to investigate how the network learns.
For visualising the activations in the feature maps, we will use Figure 6.1, on page 54, as an example. We have chosen to look at the activations after the first pooling layer because it is the

Figure 6.1: A crop from an example painting for illustrating the filters and activations.

output of the first block of the CNN, which is the initial step in processing the input.
As for the weights in the filters, they are globally scaled to fit the same range, making the comparison simpler.
We have chosen only to display four of the 32 feature maps from the first pooling layer, which can be seen in Figure 6.2.

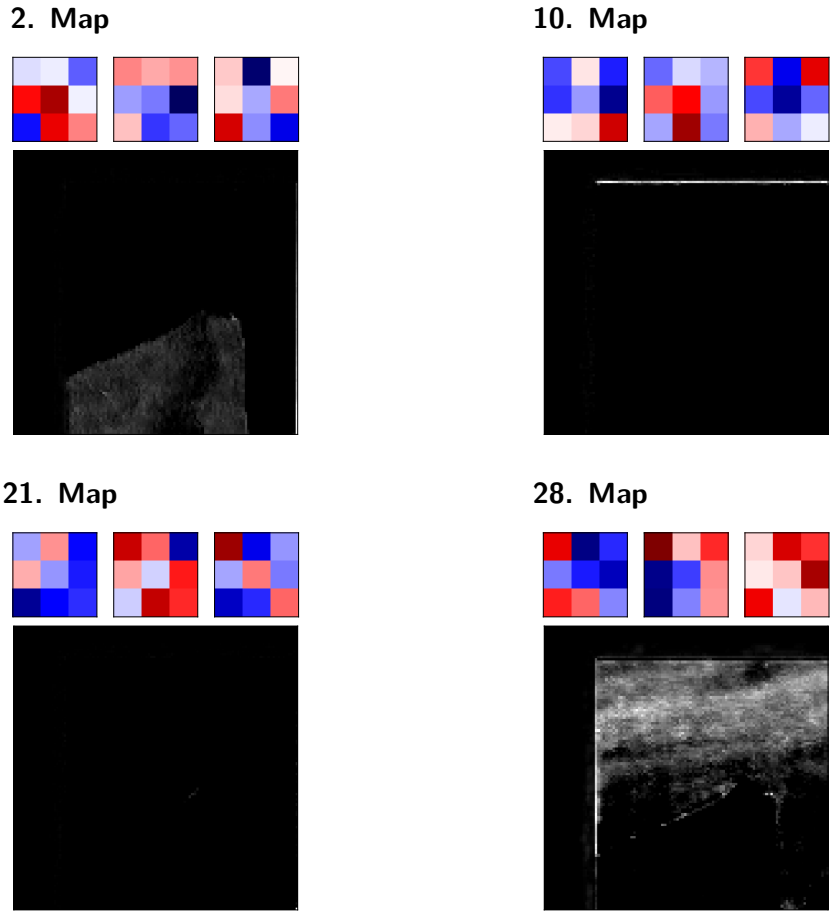


Figure 6.2: The filters and activations of the first layer.

We see that the feature maps we have chosen to display are focused on the three colour channels of the input image, apart from the 10th feature map.
The 2nd feature map seems to have most of the activations on the mountain in the centre of the image. Since this mountain is mostly red, we also see that the first filter has mostly positive weights, denoted by the red colours, and a few negative weights, denoted by the blue colour. There are also some weights that are closer to zero, as these are very white. Combining this with how the other two filters look, we see that if the middle of the filters has a high value in the red channel and smaller values in the other two channels, the sum would likely be above zero, making the rectified linear unit fire.
When choosing these four feature maps, we noticed that many of the maps had many activations around the black border of the image. We chose the 10th map because it had a very distinct horizontal line of activations, which correspond to the border. If we consider how the filters are moved across the image, in a sliding window manner, we can see why this is the only place that has activations. When the bottom row of the filters are on the border, where the blue colour starts, the sum of the weighted values is greater than 0, meaning there would be an activation. We see that there are a few activations above and below the line, but mostly it is a single pixel line that has activation.

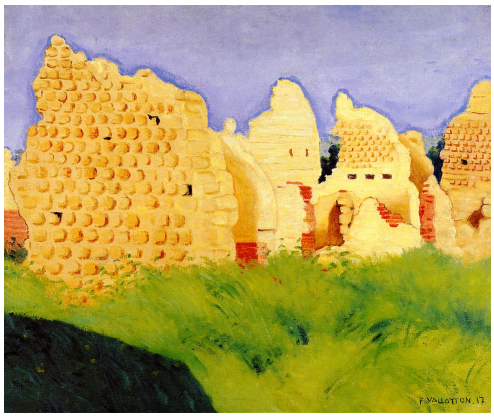
We see that some feature maps, like the 21st feature map, are specialised to look for colours that are not necessarily in all images. In the 21st feature map, the green channel filter has mostly positive weights, while the red and the blue have very negative weights. Since the example image is primarily red and blue, this results in an almost completely empty feature map apart from a small patch of activations that correspond to the peak of the mountain. The reason why there is activations in that particular place might be that the very negative weights of the red filter are in the shadow area of the mountain, and thus the red values are smaller. If, in turn, the large weight in the top left corner of the blue filter is within the sky of the painting, then there should only be some values in the green channel for the sum to be greater than 0.
The 28th feature map seems to be focused on blue colours, which is evident in the fact that the entire sky is activated. We see the areas with high concentration of blue colours are being activated, but the areas of white seem to not activate. This is because, as the colours approach white, the values in the other two channels also increase, and the negative values in those filters ensure that the sum is negative. There is also a particular dark area in the top of the image, which might be because this is the darkest area in the example image as well and thus, the values are the lowest for all channels.

6.3 Mis-Predictions

To better understand why the network might mis-predict some samples we will present some mis-predicted examples and analyse them. For this we use the classes *Color Field Painting* and *Magic Realism*, which were the classes with the highest accuracy.
When predicting the test set, we found that 13 paintings were mis-classified, which constituted the 5% error rate for *Set B* in Table 6.1, on page 53. For simplicity, we chose 5 out of these 13 and presented them in Figure 6.3, on page 57.

When analysing the mis-predictions, we should consider the augmentation approach we use, explained in Chapter 5, where we take 10 crops from the painting, in a sliding window. Sometimes the crops might not be representative of the style. This affects the predictions and, after averaging them, can result in a mis-predicted image.

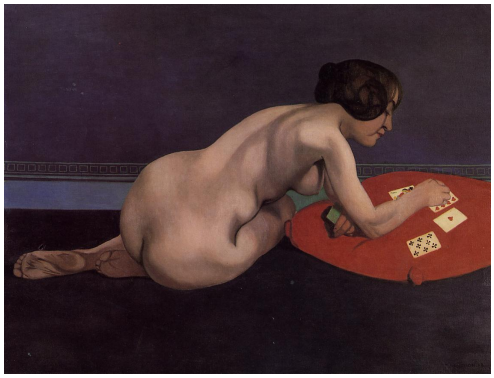
Generally, for Figures 6.3a through 6.3d, the probability for the predicted label is very close to a random guess, i.e. 50%.
If we look at Figure 6.3a, the network seems to just guess the label for this. A reason for this could be the way the painting is segmented in the blue background, the green foreground, and then primarily yellow in between. This clear division of colour regions resembles *Color Field Painting*, which might be why it is mis-predicted. Furthermore, considering that we take crops, some of the crops might display only one color region so their resulting predictions are likely to be *Color Field Painting*. On the other hand, the shapes of the objects seem to be more ordered, which is closer to *Magic Realism*. Similar things can be concluded for the Figure 6.3b.In the Figure 6.3c, we can see that the colours and their arrangement in the painting are very similar to the ones in Figure 6.3a. This could indicate that there are some similarities between which colours are used and where they are applied between the two styles.



(a) True label: Magic Realism
Predicted label: Color Field Painting
Probability: 51.5%

(b) True label: Magic Realism
Predicted label: Color Field Painting
Probability: 55.9%

(c) True label: Color Field Painting
Predicted label: Magic Realism
Probability: 58.3%



(d) True label: Magic Realism
Predicted label: Color Field Painting
Probability: 55.2%

(e) True label: Color Field Painting
Predicted label: Magic Realism
Probability: 95.7%

Figure 6.3: Five paintings that were mis-predicted by the network.

In terms of colour, Figure 6.3d looks similar to the Figure 6.3b. There is a very high contrast between the colours of the shapes, with the background being very dark and the two shapes being light in colour. This results in two distinct regions of colour which, again, could resemble *Color Field Painting*.

The probability shown for the previous four figures have all been close to random but with Figure 6.3e, the network is seemingly very confident in the prediction, even though it is incorrect. One reason for this mis-prediction could be the black and white shapes that contrast significantly with the background, which is a common characteristic of a portrait painting. However, this does not account for the high confidence, which we do not entirely understand. This might make each crop contain one of the background colours and a part of each of the shapes. If all the crops look similar and are predicted as *Magic Realism*, when we average the predictions the overall result would be *Magic Realism*.

6.4 Split by Painters

As mentioned in Chapter 1, classifying paintings by style can be difficult because styles are often based on the fact that a group of painters coincidently paint in similar ways but there are few formal rules to them. This means that there can be a number of different ways of painting styles like *Neoclassicism*, depending on the artist. If the network learns how these painters are different in the training set, it would have an easier time if there are paintings by the same painter in the test set. For this reason we made sure in this experiment that there are no painters with paintings in both the training and test sets, making the classification problem harder since the network cannot rely on the particular style of a painter. We compare how our network does with how the baseline does to see if the harder dataset could close the gap.
As for the dataset, we use *Cubism* and *Neoclassicism*, which is then split such that there are no painters that overlap the sets. The results of the experiment can be seen in Table 6.2.

	Train Accuracy	**Validation Accuracy**	**Test Accuracy**
VGG16 + SVM	—	97.4%	92.1%
Our Network	95.9%	91.4%	79.8%

Table 6.2: The results of the artist experiment.

As we can see, the baseline dropped in test accuracy, from 96.1% in Table 6.1, on page 53, to 92.1%, meaning the split had an effect on the performance. However, it is still better than our network, which dropped significantly in test accuracy. Interestingly, both our network and the baseline increased in validation accuracy. This could be because of the new split, which was not exactly 60/20/20% as before. With the new split the validation set ended up with fewer paintings, making it easier to predict.
By separating the painters, we made the dataset harder which can be seen in the test accuracy. We can see that our network was not able to handle the more difficult set as well as the baseline. Comparing the test accuracies in Table 6.1, on page 53, and Table 6.2, it seems that our network

relies on the painters. Despite that, it still learns some general features of the styles, since the test accuracy is above 50%.
However, as mentioned in Chapter 1, the peculiarities of each painter makes the style what it is. If the network can recognise the painter, it is highly likely that it would also recognise the style and, therefore, yield better results.
Nevertheless, there can be painters who paint in multiple styles. In that case, the network that relies on painters would have trouble distinguishing the styles, which might be the case with *Early Renaissance* and *High Renaissance* dataset.

7 Conclusion

Throughout this project we have been incrementally experimenting with constructing a Convolutional Neural Network for art style classification in fine art paintings. We have focused on three major aspects of neural network design; input representation, input augmentation, and structural architecture.

During experiments with input representation, we explored the effects of normalisation, resizing methods and image sizes. We found that normalising the images with *Min-Max* scaling, using a normal square resizing with Lanczos resampling, and a target image size of 224×224, provided the best results.

After the input experiments, we investigated various network architectures, to find the structure that best suited our problem. The architecture was divided into two areas; the depth of the network, i.e. the number of blocks, and the width of the network, i.e. the number of feature maps. Through experiments with increasing number of blocks and feature maps, we found that 7 blocks, with one convolutional layer and max pooling, starting with 32 feature maps in the first convolutional layer, provided the best results. As for the fully connected layers, we have 8 neurons in the first fully connected layer, and two softmax output neurons.

In terms of augmentation we employ three different kinds; Gaussian blur, horizontal and vertical flip, and a sliding window crop. Applying these techniques, it resulted in up to 60 times more samples.

We compared this network with a baseline, consisting of the VGG16 network from [14] and an SVM classifier, on three different datasets:

Set A: *Cubism* and *Neoclassicism*, 1,773 training, 580 validation, 585 test

Set B: *Colour Field Painting* and *Magic Realism*, 856 training, 239 validation, 276 test

Set C: *Early Renaissance* and *High Renaissance*, 1,265 training, 413 validation, 424 test

The results can be seen in Table 7.1.

	VGG16 + SVM	Our Network
Set A	96.1%	91.6%
Set B	97.8%	95.3%
Set C	74.1%	67.7%

Table 7.1: The test accuracies from the experiments.

Our network did not perform better than the baseline. However, as mentioned in [14], the baseline was trained on 1.3 million images, while ours was at most trained on 1,773 images, in *Set A*. This means that with the aggressive augmentation, and a deeper architecture, we reached comparable performance, with around $\frac{1}{1000}$ of the data.

To conclude, we found that, unless we had aggressive augmentation and a very deep architecture, we were not able to train a CNN purely on digitalised fine art paintings and perform better than using a pretrained network with an SVM.

Part III

Epilogue

8 Discussion

In this Chapter we will be reviewing our work on this project and argue if we could have done some aspects differently.

8.1 Tools Discussion

Over the course of this project we have used two major tools; Keras for building the networks and Abacus for training them. In this section we will be evaluating our experience with these.

Keras

Throughout this project we have been using Keras to design and implement CNNs. Generally, the tool has been simple to work with and offered a number of methods for analysing and investigating how CNNs learn. However, one of the biggest disadvantages that we encountered was the documentation.
On the tool website [4] the documentation more closely resembles a tutorial and it is missing a full API overview. For some of the classes and methods the description is incomplete and some of the parameters are not explained.
Since Keras is a frontend for Tensorflow and Theano it is also not clear when a method is specific to Keras or is a part of the backend. Keras methods usually return Numpy arrays, or other regular datastructures, whereas Tensorflow returns most things as Tensors. This can cause confusion when trying to understand the networks, since Tensors need to be evaluated in a Tensorflow session to get the value.
An example of this is when we have to get the weights associated with a layer. Keras provides a method called `get_weights()` that returns a double, where the first element is the weights and the second is the biases, the latter of which is not explained in the documentation. Conversely, if we use the attribute `weights`, we get a list of Tensorflow `Variables`, which in turn are Tensors, that we then need to evaluate to get the same values as with `get_weights()`.

Abacus

When training the networks during the experiments we have been using the Abacus HPC supercomputer. This has helped us to create increasingly complex networks while still being able to train them in reasonable time, which was not possible in previous work [8]. However, as explained in Section 1.2.2, we have been sharing this supercomputer with a number of people from AAU. Most notably we are sharing the amount of time we have available to run computations, meaning when all the time has been spent we cannot run any more experiments. At one point during the project we ran out of time on the GPU clusters. Therefore, for a time we had to use the CPU clusters when running experiments which increased the running time significantly.
In addition to sharing the time spent with other people from AAU, we share the overall supercomputer with people from other institutes. While these have no effect on the time spent on the supercomputer, it meant that, at times, our jobs were pending in a queue for extended periods of time. This was especially apparent towards the end of the project, where we could be pending for more than 24 hours. This meant that an experiment that would normally take around three to four hours, now took days before we could get the results.

8.2 Dataset Discussion

During the experiments we have chosen images based on two basic criteria; visual similarity and number of images. In this section we will argue whether these were good criteria and if we could have chosen differently.

Class balance

The two classes should be balanced, having roughly the same amount of images, to avoid the CNN being biased towards one class. The classes in our dataset are highly imbalanced which limited our choice when choosing classes. If we wish to add more classes, or choose different ones, we would need to narrow our choice to the classes that have somewhat the same amount of images or to balance the classes by either duplicating or removing samples from certain classes. Even though class imbalance would represent frequency of the styles in the real world, it would be difficult for us to analyse what the CNN learned if its parameters are biased.

The Number of Images

Another relevant factor for choosing classes is the number of images. The number of images should be large enough, since the bigger a dataset is the more a CNNs can learn. Still, we did not have enough original data to compete with the baseline. We needed to artificially inflate the dataset using augmentation to improve the classification performance. Although this boosted the validation accuracy, it still did not do better than our baseline, the VGG16 network, which was trained on 1,3 million images containing 1000 classes. Our training set contains 1,773 images for classes *Cubism* and *Neoclassicism*, which after augmentation resulted in $106,380$ samples. Comparing the VGG16 and our dataset, it is clear that there is a considerable difference in the amount of images.

Although more data would be favourable, we experienced difficulties when training our network on the Abacus with the two classes that have the largest amount of samples, *Impressionism* and *Realism*. At the time of the experiment, we had only CPU nodes available, and for the 24 hours, which is the maximum that we could run a job, the training reached only 10 epochs where loading of the images into memory took 7 hours. When we got back to GPU nodes, we were not able to load all the images into the memory, since it only contained 64GB, compared to 512GB on the CPU nodes. For GPU nodes we implemented a batch loading algorithm. Despite that, it did not perform faster since at each batch the images were loaded from the disk and not from the memory. The fact that we were not able to use bigger dataset on the Abacus also limited our choice of the classes we would experiment with.

Class similarity

Additionally, we decided to choose more dissimilar classes to experiment with rather than the similar ones. If we chose more similar classes, we would expect the network to exhibit more problems during training which would slow down the experiment process. Moreover, the differences between some similar classes, like *Early Renaissance* and *High Renaissance* are difficult to distinguish even

for CNNs. This has been confirmed in our experiments in 6.1, where we can see that even our baseline had difficulties in learning the *Early Renaissance* and *High Renaissance* dataset.
However, our similarity criteria is based on the visual differences we see between the styles. Even though our assumptions are confirmed for *Early Renaissance* and *High Renaissance*, there might be some other set of classes that the CNN could learn which seem similar to us. In general, when looking at the images inside a certain style, we often discovered that many styles included very dissimilar paintings because of the variety of painters included. Our experiment with the painters in Section 6.4 indicates that, from the perspective of the CNN, the style might be determined by the painters and that our CNN is learning the painter-specific features, rather than style-specific features. This makes our similarity criteria questionable and difficult to apply for the decision about the classes.

8.3 Preprocessing Discussion

For our final model with augmentation we resized all images so that the largest dimension was no greater than 1000 and normalised them using *Min-Max* scaling method. For the augmentation of the images we used flipping, cropping and blurring. There are many other possible techniques we could use to further inflate our dataset, but due to the time constraints we were able to experiment with only a few. When choosing the techniques for augmentation it is important to know how they might affect the dataset which then affects the process of learning of the CNN. Some common techniques include changing the values of the colour channels or altering the shape of the lines and objects in the image. For art recognition problem, line and colour are two of the main components that define the style. After applying augmentation techniques, the transformed image might lose the characteristics of its style.

Cropping

When we use cropping, we cut fairly small crops of the image. Some of those crops might not be a good representation of the original image and might confuse, rather than help the CNN learn. For example, in the *Color Field Painting* and *Magic Realism* dataset, some *Magic Realism* paintings contained large areas of certain colours. The crop of that area would display uniform colours without any objects, which is a characteristic of *Color Field Painting*. We manage to alleviate this issue by using an ensemble approach where averaging results of all crops lowered the impact of the possibly confusing crops on the final result.

Blurring

Blurring can also affect our data as blurring the image results in warped shapes and objects. This can alter some styles to the extent where the image might not be representative of that style anymore. This is especially the case for the styles which are characterised by the sharp lines and edges and highly detailed objects. The blurring would make the lines and objects in these paintings more bland which might result in an image resembling to other styles characterised by gradient colours.

Flipping

Flipping the images along the both the horizontal and the vertical axes provided a simple way of creating new samples. One disadvantage of this is that, due to the shared weights in the feature maps, the glscnn is invariant to rotation and location of objects in the image. For instance, if the CNN sees an object in the left of the input image, it would still find that object if the image was flipped horizontally. The only part of the CNN that would be affected by this is the fully connected layer. Since this layer does not have shared weights, it is sensitive to the order of the feature vector produced by the convolutional layers. This feature vector would be in a different order for samples that are flipped, even if the values would be similar.

Since we did not experiment with the chosen techniques individually because of the time constraints, we are not able to say if they affected our CNN as we assume.

8.4 Experiments Discussion

When designing our CNN, we primarily focused on the more significant components, the architecture and the input representation. The components and parameters that we decided are relevant are mostly based on our intuition of what might cause considerable improvements and the problems we confronted during the experiments. Although our experiments were extensive, there are still some other, less dominant factors that we could have considered, mentioned in Sections 2.1.1 and 2.1.2.

Learning Rate

For example, we could experiment with the different values of learning rate or implement the learning rate scheduling. This could speed up the training as well as controlling it when it plateaus. Additionally, in the experiments where we used early stopping that is dependent on the validation accuracy, the training could then last longer which could result in higher accuracy.

Filters

A larger filter size is commonly used when the input image is large. The advantage of a large filter is that it does not propagate much noise from the input, at the cost of only seeing broad features. On the other hand, a small filter finds more detail in the image, but it also allows more noise. By having a deeper network the noise from the small filter is reduced and the network can use the detailed features. With the larger filter the network can be more shallow, but it loses some expressiveness.

Regularisation

Early stopping and augmentation are types of regularisation as both of them deal with the overfitting problem. We could have also used other regularisation techniques, such as dropout or L2-regularisation or even combine it with the early stopping and augmentation. This could be useful for training the models with a very strong overfitting and many more parameters.

Initialisation

In many of the early experiments we encountered the problem where the CNN did not learn. In our later experiments we decided to use fixed variable initialisation to avoid this problem. If we had used it in our early experiments, mainly with architecture, we could have gotten different results and we could have ended up with a different model. At the same time, this might not have given us better results because for most of the models in the architecture experiments the results were roughly the same.

Datasets

Our model is designed while experimenting with the classes *Cubism* and *Neoclassicism*. With different classes we would likely end up with a different model as well. Our experiments could also be focused on other parameters, depending on the problems we encounter. As we could see from the results in Section 6.1, for the classes *Early Renaissance* and *High Renaissance* we reported a considerable decrease in test accuracy. If we designed our CNN with these classes we would have focused our efforts on highlighting the differences between these classes with preprocessing. It is also likely that we would not have found a good way of classifying these classes, considering that the baseline also decreased significantly in test accuracy.

Bibliography

[1] Kaggle - painters by numbers. https://www.kaggle.com/c/painter-by-numbers. accessed: 2016-06-10.

[2] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[3] François Chollet. Keras. https://github.com/fchollet/keras, 2015.

[4] François Chollet. *Keras Documentation*, April 2017. https://keras.io/.

[5] Anna Choromanska, Mikael Henaff, Michaël Mathieu, Gérard Ben Arous, and Yann LeCun. The loss surface of multilayer networks. *CoRR*, abs/1412.0233, 2014.

[6] David Eigen, Jason Tyler Rolfe, Rob Fergus, and Yann LeCun. Understanding deep architectures using a recursive convolutional network. *CoRR*, abs/1312.1847, 2013.

[7] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *In Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS'10). Society for Artificial Intelligence and Statistics*, 2010.

[8] Andrea Gradečak, Mathias Ottosen, and Rasmus Johnsen. Art classification tools and techniques: An experiment-based analysis on art classification tools. Pre-specialisation report, Aalborg University, Selma Lagerløfs Vej 300, Aalborg Ø, January 2017.

[9] Christian Hentschel, Timur Pratama Wiradarma, and Harald Sack. Fine tuning cnns with scarce training data—adapting imagenet to art epoch classification. In *Image Processing (ICIP), 2016 IEEE International Conference on*, pages 3693–3697. IEEE, 2016.

[10] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[11] Michael A. Nielsen. Neural networks and deep learning. Online, 2015. http://neuralnetworksanddeeplearning.com/chap6.html.

[12] Stephen O'Hara and Bruce A Draper. Introduction to the bag of features paradigm for image classification and retrieval. *arXiv preprint arXiv:1101.3354*, 2011.

[13] SchedMD. Slurm workload manager, March 2013. https://slurm.schedmd.com/overview.html.

[14] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[15] SDU Syddansk Universitet. Abacus 2.0, March 2017. https://abacus.deic.dk/.

[16] SDU Syddansk Universitet. Abacus 2.0, March 2017. https://abacus.deic.dk/setup/hardware.

[17] Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, May 2016. http://arxiv.org/abs/1605.02688.

A Appendix

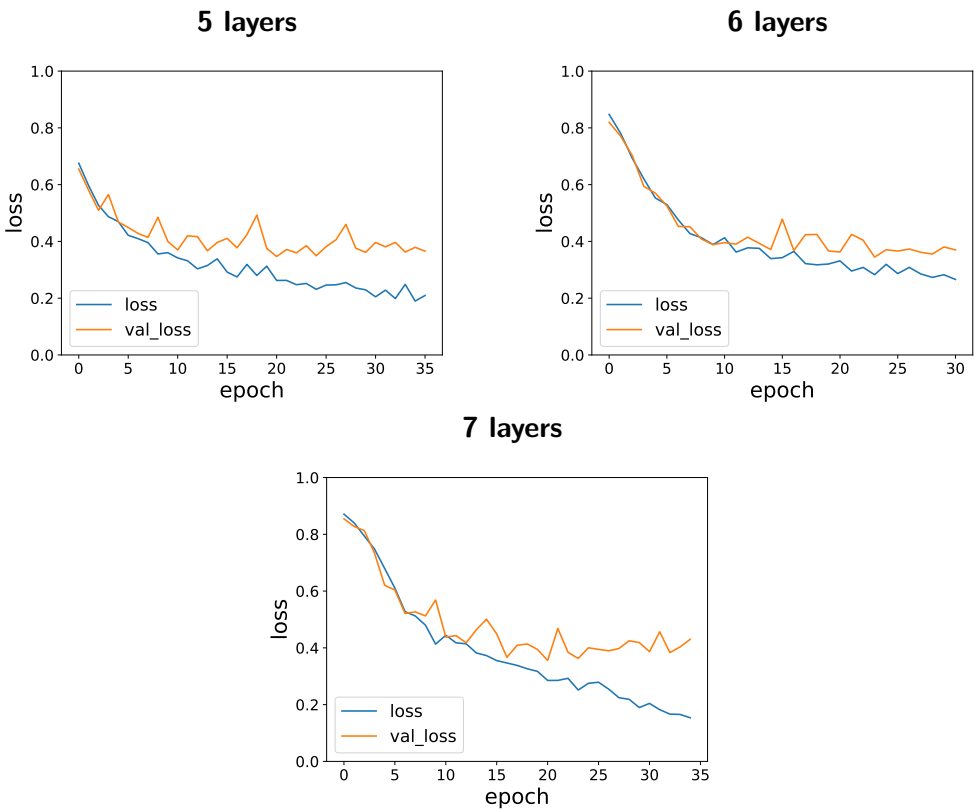


Figure A.1: The losses for the experiment with filters, starting from 16 filters

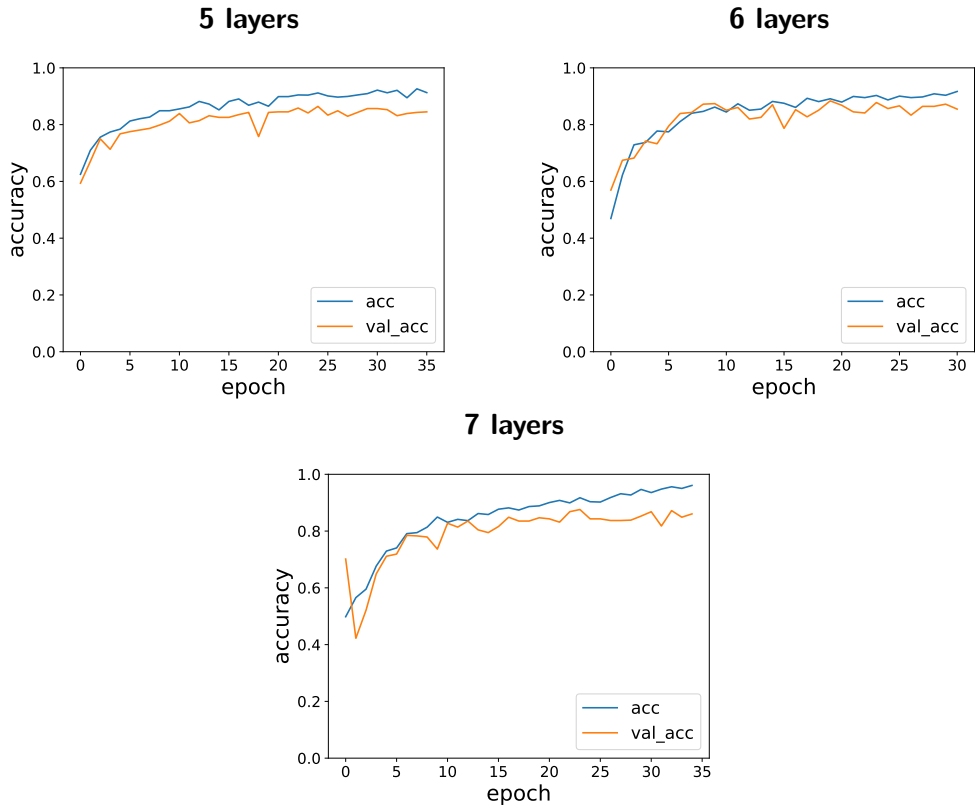


Figure A.2: The accuracies for the experiment with filters, starting from 16 filters

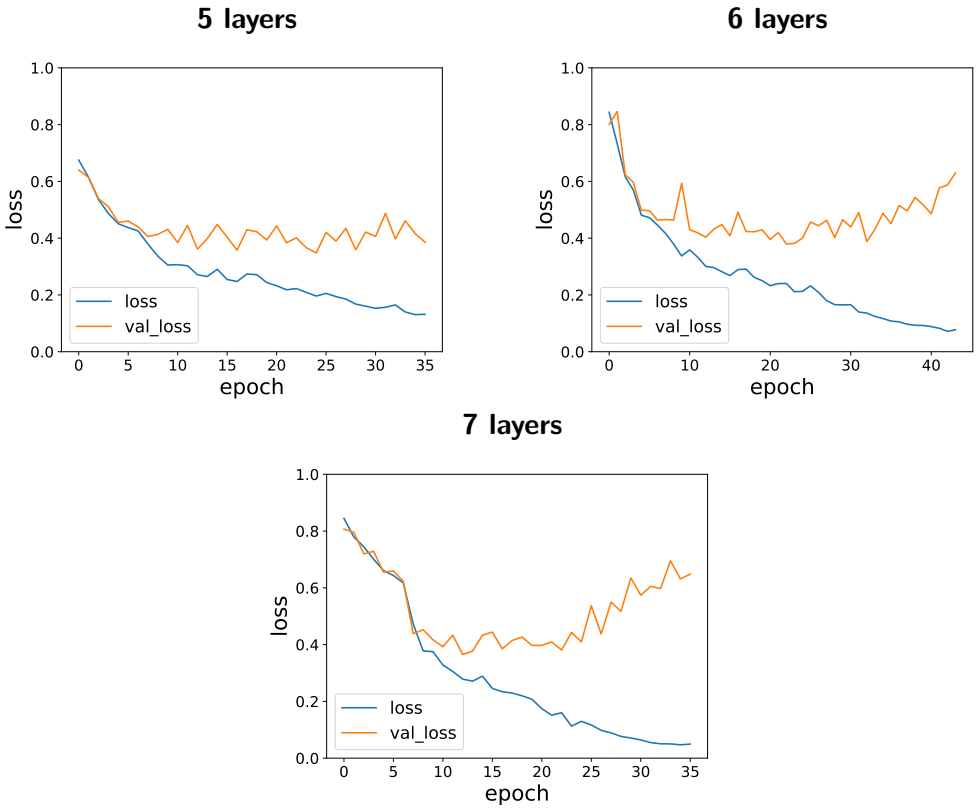


Figure A.3: The losses for the experiment with filters, starting from 32 filters

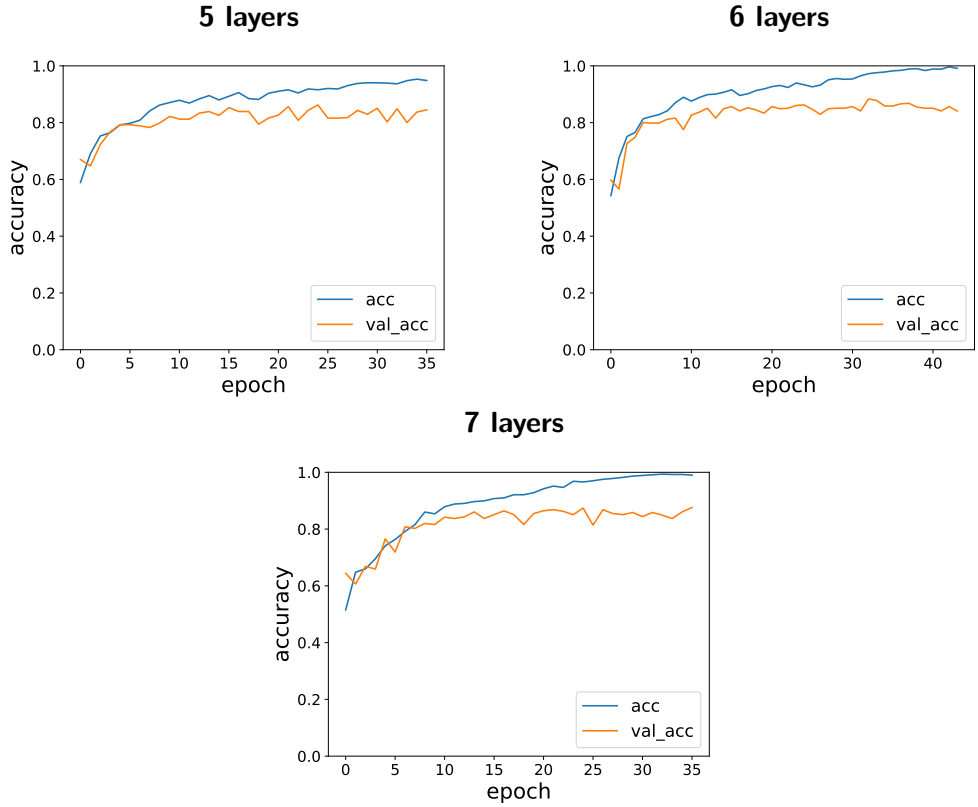


Figure A.4: The accuracies for the experiment with filters, starting from 32 filters

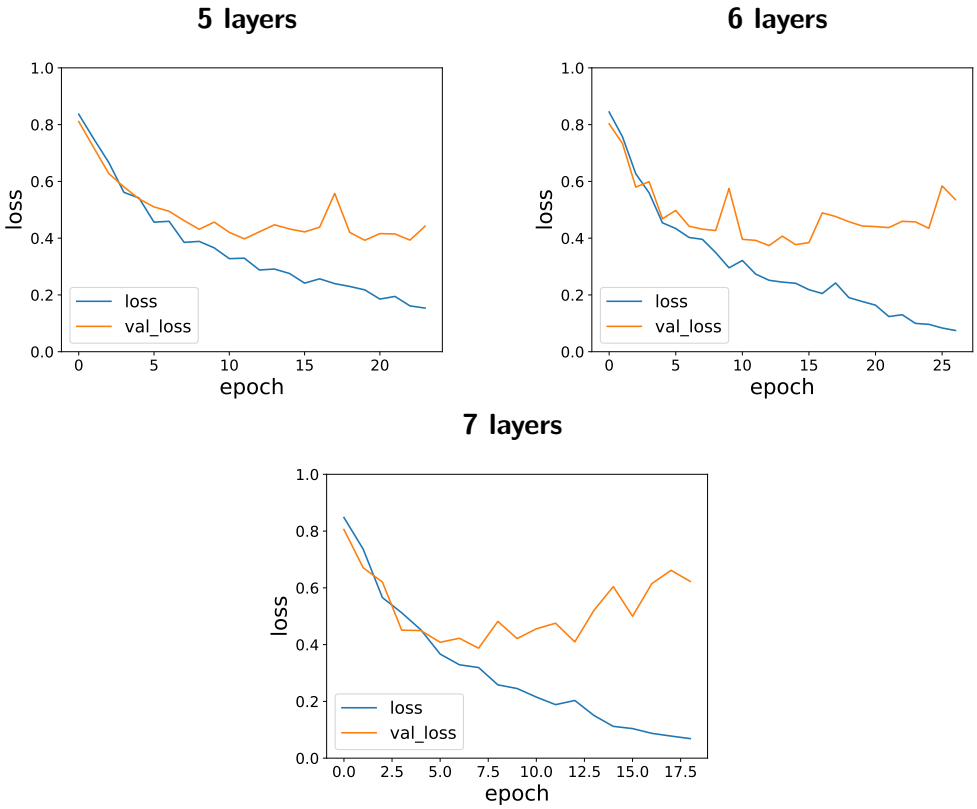


Figure A.5: The losses for the experiment with filters, starting from 64 filters

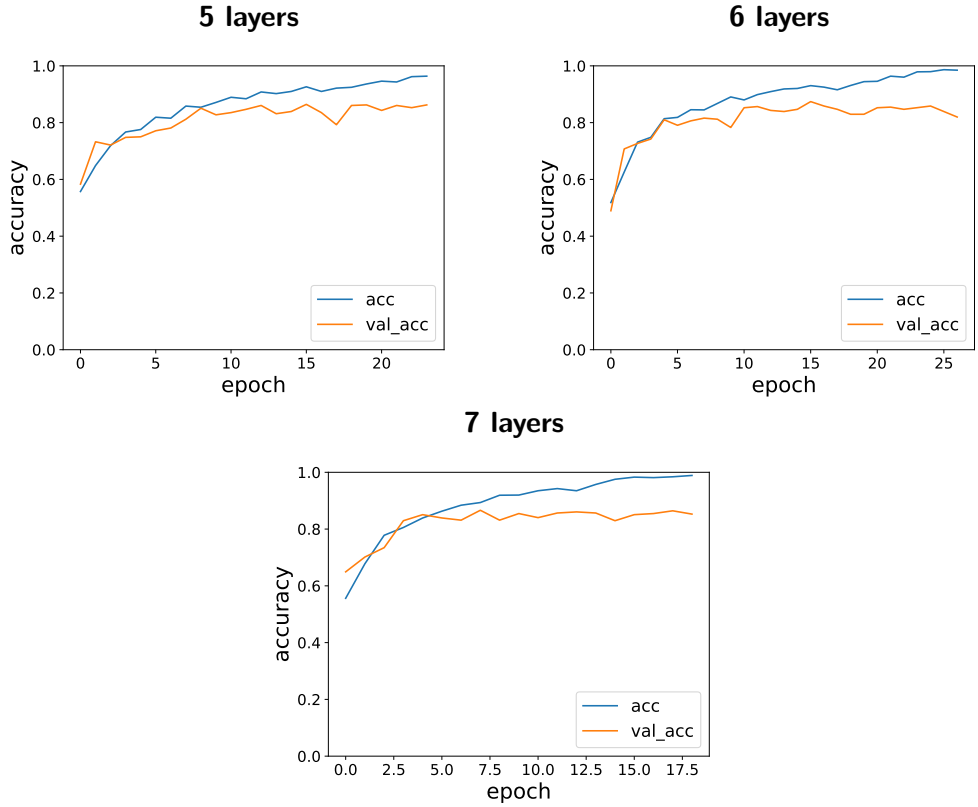


Figure A.6: The accuracies for the experiment with filters, starting from 64 filters