Università degli Studi di Milano

Data Science for Economics

Algorithms for Massive Data

Image Classification with Convolutional Neural Networks

by
Jan Philip Richter
Matriculation Number: 20547A

submitted to
Prof. Dr. Dario Malchiodi

April 1, 2024

Abstract

This report uses convolutional neural networks to classify paintings of Spanish artists from the Prado Museum.

Various preprocessing procedures are implemented to enhance the usability of the data for the model and methods of dealing with class imbalance are utilised to obtain more stable predictions.

Three architectural approaches are used for experimentation: VGG, ResNet and MobileNet, all of which achieve ca. 95% predictive accuracy on unseen test images. The VGG framework follows a standard sequential approach. To further improve its predictive capabilities, better regularisation techniques, different activation functions and a superior kernel initialisation method are implemented. ResNet and MobileNet both implement the residual learning framework allowing for a significant increase in depth. The MobileNet architecture additionally implements depthwise separable convolutions, resulting in a considerable reduction of parameters.

Furthermore, transfer learning is performed to compare the achieved results to state-of-the-art pre-trained models.

Contents

1 Introduction **1**
- 1.1 Task . . . 1
- 1.2 Dataset . . . 1
- 1.3 Data Preprocessing . . . 1
  - 1.3.1 Image Conversion and Resizing . . . 2
  - 1.3.2 Horizontal Flipping . . . 2
  - 1.3.3 Image Standardisation . . . 2
  - 1.3.4 PCA Colour Augmentation . . . 2
- 1.4 Class Imbalance . . . 3
  - 1.4.1 Class Weights . . . 3
  - 1.4.2 Output Biases . . . 3
- 1.5 Learning Rate Scheduling . . . 4
- 1.6 Implementation . . . 4

2 VGG **5**
- 2.1 Architecture . . . 5
  - 2.1.1 Regularisation . . . 6
  - 2.1.2 Activation Functions . . . 7
  - 2.1.3 Kernel Initialisation . . . 7
- 2.2 Execution Specifications . . . 8
- 2.3 Results . . . 8

3 Residual Neural Networks **10**
- 3.1 Residual Learning . . . 10
- 3.2 Implementation . . . 10
- 3.3 ResNet29 . . . 11
  - 3.3.1 Architecture . . . 11
  - 3.3.2 Results . . . 12

4 MobileNet V2 **14**
- 4.1 Depthwise Separable Convolutions . . . 14
- 4.2 Inverted Residuals . . . 14
- 4.3 Linear Bottlenecks . . . 15
- 4.4 Architecture . . . 16
- 4.5 Results . . . 17

5 Transfer Learning **18**
5.1 Models 18
5.2 Execution Specifications 19
5.3 Results 19

6 Conclusion **21**
6.1 General Evaluations 21
6.2 Potential Improvements 21

Bibliography **22**

1 Introduction

1.1 Task

The goal of this project is to implement a deep-learning-based system to classify paintings of Spanish artists from the Prado Museum in Madrid. Furthermore, the introduced techniques should scale well for an application on larger data sets.

3 different convolutional neural network architectures will be utilised for experimentation, to investigate which approach yields the most desirable results. The choice of network architecture, as well as its advantages and disadvantages will be thoroughly discussed in the subsequent chapters.

1.2 Dataset

The dataset[1] considered for this report consists of JPEG images of paintings of 5 Spanish artists from the Prado. Namely, the painters, whose artworks are considered for the analysis are: Francisco Goya, Francisco Bayeu y Subías, Carlos de Haes, Cecilio Pizarro and Carlos Luis de Ribera y Fieve. There are a total of 2,364 images, divided into a training and a testing set, as shown in the table below:

	Goya	**Bayeu**	**Haes**	**Pizzaro**	**Ribera**	**Total**
Train	851	376	259	240	166	1,892
Test	229	70	67	50	56	472
Total	1,080	446	326	290	222	**2,364**

The test set, also referred to as the validation set, will be used to get an estimation of the models' performances on unseen data which allows for better model comparison.

1.3 Data Preprocessing

Data Preprocessing is an important and necessary step to ensure a satisfying performance of the models. The goal of the data preprocessing steps performed in this report is to both transform the data into a form, where the models can handle and learn from the data, as well as to enhance specific features and introduce diversity in the data, which can lead to a more stable and faster training run and the reduction of overfitting.

The preprocessing procedures which are introduced in this chapter are applied for all models presented in this analysis.

[1]Dataset obtained from: https://www.kaggle.com/datasets/maparla/prado-museum-pictures

1.3.1 Image Conversion and Resizing

For a neural network to learn from image data, it is necessary to transform their original JPEG format into a numerical representation. The images are thus converted into tensors, which is a multidimensional array of numbers. In our case, each image is represented as a 3-dimensional tensor, where the first 2 dimensions represent the image's height and width, and the 3rd dimension represents its three colour channels, red, green, and blue. The values for each element in the tensors range from 0 to 255. This number indicates the intensity of a given colour channel for a given pixel of the image.

Additionally, a given neural network model requires the input tensors to have a constant shape. As the images in the data set have varying sizes, a resizing procedure is performed via bilinear interpolation. The images are resized to have a dimension of $(224, 224, 3)$, meaning each image has a shape of 224×224 pixels with 3 colour channels.

All models receive the input data in batches containing 32 tensors during the training run.

1.3.2 Horizontal Flipping

Images are randomly flipped around their horizontal axis with a probability of 50%. This ensures a higher robustness of the models with respect to the orientation of the images.

1.3.3 Image Standardisation

Similar to other machine learning algorithms, neural networks also benefit from receiving standardised data as input. The images in our data are standardised to ensure their values have the same scaling, which allows for a more robust learning performance and reduction of the risk of overfitting.

The standardisation is performed on a per-image basis, using the following formula.

$$image_{std} = \frac{image - \mu}{sd_{adj}}$$

where μ denotes the mean value of all elements in the image and $sd_{adj} = \max(sd, 1/\sqrt{N})$ where sd is the standard deviation of all elements in the image and N is the number of all elements[2].

1.3.4 PCA Colour Augmentation

Colour augmentation has the goal of enhancing specific features in the image data, which allows a model to better learn the patterns which distinguish the target classes. With the goal of reducing overfitting for their huge network architecture Krizhevsky

[2] https://www.tensorflow.org/api_docs/python/tf/image/per_image_standardization

et al. (2012) introduced a colour augmentation procedure which changes the intensities of the colour channels in an image using the principal components of the RGB pixel values.

Specifically, to each pixel $I_{xy} = [I^R_{xy}, I^G_{xy}, I^B_{xy}]^T$ the colour intensity gets altered by adding the following quantity:

$$[\boldsymbol{p_1}, \boldsymbol{p_2}, \boldsymbol{p_3}][\alpha_1\lambda_1, \alpha_2\lambda_2, \alpha_3\lambda_3]^T$$

where $\boldsymbol{p_i}$ and λ_i are the i-th eigenvector and eigenvalue of the 3×3 covariance matrix of the pixel values and α_i is a random variable from the Gaussian distribution $N(0, 0.1)$ which is drawn once for each pixel in a given training image.

1.4 Class Imbalance

Class imbalance is an obstacle for almost every machine learning scenario, as highly imbalanced data can often result in poor predictive performance, as there is a tendency to predict the majority class. In our case, the paintings by Francisco Goya make up more than 45% of the entire dataset. This may result in the neural networks mostly learning the patterns that make up paintings from Goya and not focusing enough on the remaining data. To avoid a prediction bias towards Goya paintings, 2 methods are utilised to deal with the class imbalance present in the data.

1.4.1 Class Weights

During the models' training run we specify class weights according to the proportion of a class in the data set. The class weights indicate by which factor the loss value will be multiplied.

The weight for each class is calculated using the following formula:

$$weight_c = \frac{1}{n_c}\frac{n}{|c|}$$

where $weight_c$ denotes the weight of class c, n_c is the number of observations of class c, n is the total number of data points and $|c|$ denotes the number of different classes.

The different class weights ensure, that misclassifications of the model made on minority classes get penalised more heavily during the training run to guarantee that the model learns to detect all classes and not just the majority class.

1.4.2 Output Biases

A second way to deal with class imbalance is through the initialisation of biases in the output layer of the model. The biases in the output layer represent an initial guess of the model and can be altered to reflect the class imbalance in the data.

To calculate the output biases, we solve the following system of equations for the vector of biases:

$$
\begin{aligned}
f_1 &= e^{b_1}/(e^{b_1} + e^{b_2} + ... + e^{b_5}) \\
f_2 &= e^{b_2}/(e^{b_1} + e^{b_2} + ... + e^{b_5}) \\
&\vdots \\
f_5 &= e^{b_5}/(e^{b_1} + e^{b_2} + ... + e^{b_5})
\end{aligned}
$$

where f_i denotes the frequency of class i with respect to the data set, and b_i denotes the bias of neuron i in the output layer of the network[3].

1.5 Learning Rate Scheduling

The learning rate of an optimiser determines the size of the step towards a minimum of the loss function, which it tries to optimise. For a stable training run it is advisable to let the optimiser take large steps towards the loss function minima in the early stage of the training run, when the model's parameters are still very far away from their optimal values and to reduce the learning rate over time to allow for a finer tuning of weight updates in the later stages of the training run when the model has already learned its core features.

Thus, an exponential decay of the learning rate is implemented for the training runs of the models in this report. After the 10th epoch of a training run, the learning rate gets adjusted using the following formula:

$$lr_t = lr_{t-1} \times exp(-0.05)$$

Where lr_t denotes the learning rate at epoch t.

1.6 Implementation

The project is implemented in Python using the TensorFlow framework (version 2.15). The code can be found in this GitHub repository. For faster execution time in the training phase of the networks, the V100 GPU provided by the Google Colaboratory environment is utilised.

[3]Formula obtained from: https://stackoverflow.com/questions/60307239/setting-bias-for-multiclass-classification-python-tensorflow-keras

2 VGG

The first architecture considered for this analysis is inspired by the VGG architecture introduced by Simonyan and Zisserman (2014).

The main contribution of this architecture was to introduce models having a significantly increased depth with up to 19 weight layers. For this, they consider convolutional layers with filters that are relatively small with a size of (3×3). This has the positive effect of significantly reducing the number of parameters in the model compared to architectures which implement larger filter sizes[1].

The VGG-architecture also utilises (1×1) convolutions which were originally introduced by Lin et al. (2013) and are described as a linear transformation of the input channels. (1×1) convolutions can be used for dimensionality reduction and depth scaling, but in the VGG setup, their main contribution is to introduce further non-linearity. The (1×1) convolutions map the input tensors into a space of the same dimensionality, as input and output channels stay the same. However, the non-linearity of the activation function results in a higher discriminative power of the output function of the model.

The filter size increases throughout the model while the dimensionality is reduced by pooling layers. This setup allows the convolutional layers of the model to capture more general patterns in the earlier stages when the tensors have a larger size, and focus detecting on more complex features in the later stages when the tensor size is relatively small. Additionally, the downsampling reduces the number of parameters in the model, thus limiting the computational costs, as well as the risk of overfitting by helping to extract the main patterns of the images.

2.1 Architecture

The architecture of the first model used for experimentation is shown in Figure 2.1. The model is composed of 12 weight layers, the first 10 of which are convolutional layers. The model features multiple subsequent convolutional layers after which downsampling is performed by AveragePooling layers with a stride of 2^2. The last 2 weight layers are fully connected layers with 16 and 8 neurons respectively. The output layer contains 5 neurons, one for each class, and the softmax activation function.

The VGG model considered in this analysis features 6,625,101 trainable parameters, making it the computationally most intensive model.

[1] A sequence of 3 convolutional layers with a (3×3) filter size and 16 in- and output channels has $3 \times (3^2 \times 16^2) = 6,912$ weights while a single (7×7) convolution has $7^2 \times 16^2 = 12,544$ weights.

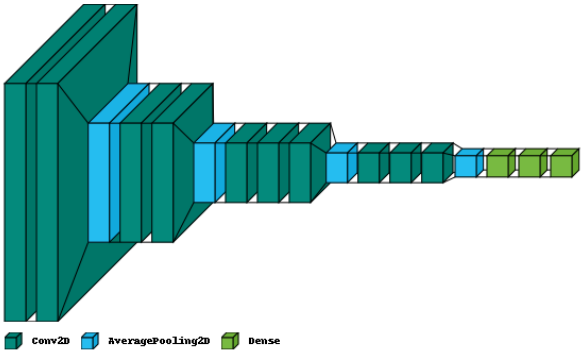
[2] The original setup considered MaxPooling instead of AveragePooling.

Figure 2.1: The convolutional layers are denoted by Conv⟨filter size⟩-⟨number of channels⟩. Activation functions, BatchNormalisation and preprocessing layers are not displayed for simplicity.

(a) VGG Architecture

Layer	Output Size
Input (RGB)	(224, 224, 3)
Preprocessing	(224, 224, 3)
Conv3-64	(224, 224, 64)
Conv3-64	(224, 224, 64)
AvgPool2, /2	(112, 112, 64)
Conv3-128	(112, 112, 128)
Conv3-128	(112, 112, 128)
AvgPool2, /2	(56, 56, 128)
Conv3-256	(56, 56, 256)
Conv3-256	(56, 56, 256)
Conv1-256	(56, 56, 256)
AvgPool2, /2	(28, 28, 256)
Conv3-512	(28, 28, 512)
Conv3-512	(28, 28, 512)
Conv1-512	(28, 28, 512)
AvgPool2, /2	(14, 14, 512)
FC-16	16
FC-8	8
Output (Softmax)	5

(b) VGG Visualisation



Although the model's setup is inspired by the VGG framework, there are a considerable number of modifications in comparison to the original architecture which are introduced in the following.

2.1.1 Regularisation

In contrast to the original VGGNets, which did not feature any form of regularisation after the convolutional layers, the model used in this analysis utilises BatchNormalisation, first introduced by Ioffe and Szegedy (2015). The goal of BatchNormalisation was to address the problem of the internal covariate shift in deep neural networks which describes the phenomenon that the distribution of the layers' input changes throughout the training run of the model caused by the change in parameters of previous layers[3]. BatchNormalisation normalises the features by estimation of the 1st and 2nd moments within each batch. Additionally, the scaling allows for faster convergence of the gradient

[3]The reduction of the internal covariate shift by BatchNormalisation is debatable. For instance, Santurkar et al. (2018) propose that BatchNormalisation does not improve the distributional stability, albeit they uncover other benefits of BatchNormalisation, like a smoother optimisation landscape during the training phase.

descent, thus making the training phase quicker, as well as acting as a regularisation method that makes the use of Dropout layers obsolete. BatchNormalisation layers are implemented directly after every convolutional layer and before the activation function.

2.1.2 Activation Functions

Each convolutional layer is followed by a BatchNormalisation layer and a subsequent ReLU activation function.

The last 2 fully connected layers are also followed by a BatchNormalisation layer and a subsequent parametric-ReLU (PReLU) activation function, which was introduced by He et al. (2015). The PReLU activation function introduces the tunable parameter α_i in the following form:

$$f(y_i) = \begin{cases} y_i, & \text{if } y > 0 \\ \alpha_i y_i, & \text{if } y_i \leq 0 \end{cases}$$

The additional parameter allows for tuning of the slope in the negative part of the function. Substituting ReLU with its parametric counterpart introduces relatively little more complexity, while possibly resulting in improved classification accuracies.

2.1.3 Kernel Initialisation

The default kernel initialisation method in the TensorFlow framework is the Xavier-Initialiser (also Glorot-Initialiser) introduced by Glorot and Bengio (2010). The main improvement of their initialisation method over random weight initialisation is to address the vanishing gradient descent problem. The normalisation method of the Xavier-Initialiser allows for more stable gradients throughout all layers of a network.

However, the Xavier-Initialiser is designed for an environment that uses the tanh activation function, which is centred around a mean of 0.

The weight initialisation method used in this model is thus the He-Initialiser introduced by He et al. (2015), which has been proven paramount when initialising weights in a setup, where the ReLU or PReLU activation functions are in use. Where the He-Initialiser allows for deep neural network architectures to converge, the previously mentioned Xavier-Initialiser does not.

The He-Initialiser draws numbers from the following distribution to initialise weights:

$$N\left(0, \sqrt{\frac{2}{n^l}}\right)$$

where n^l denotes the number of neurons n in layer l. The scaling factor of 2 takes into account that the ReLU/PReLU activation functions are not centred around a 0 mean.

2.2 Execution Specifications

The chosen loss function to be optimised during the training of the model is categorical cross-entropy, which effectively evaluates both the accuracy of label classification as well as quantifying the disparity between predicted class probabilities and actual labels. It is thus a suitable loss function as higher prediction uncertainty results in a higher penalisation.

The algorithm used to optimise the loss function is Adam which was introduced by Kingma and Ba (2014). The Adam optimizer is an adaptive learning rate optimization algorithm commonly used in machine learning that combines the advantages of both AdaGrad and RMSProp.

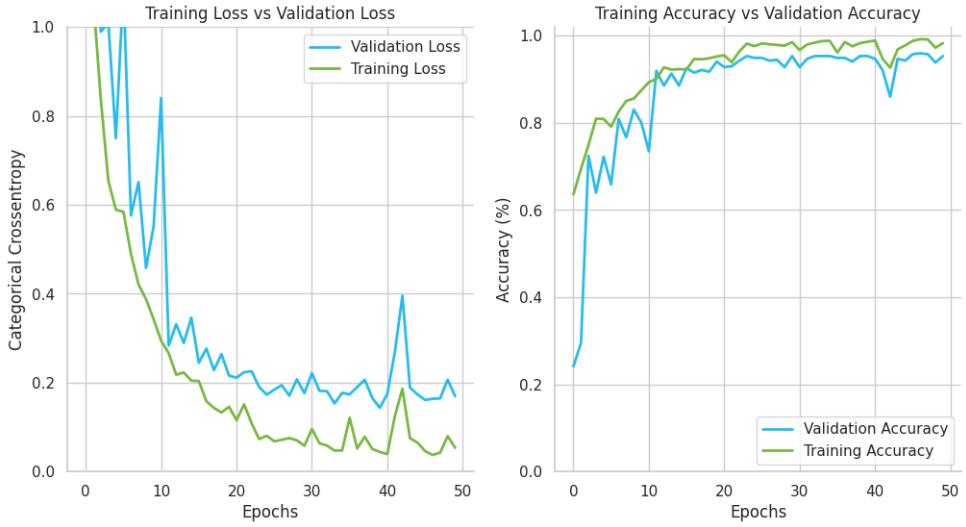
The starting learning rate of the Adam optimiser is set to 0.001. All other parameters are left at their TensorFlow-specific default values. The neural network is trained for 50 epochs on the training set and validated on the test set.

Unless stated differently, this execution setup will be used for all further models.

2.3 Results

The results of the VGG model's training run are displayed in Figure 2.2. We can observe that the model yielded fantastic results both with respect to the loss function as well as predictive accuracy.

Figure 2.2: VGG results



There is no behaviour of systematic overfitting visible, as both training and validation losses follow a similar trajectory, which is a very desirable behaviour. This is very likely

caused by the introduction of BatchNormalisation into the VGG architecture. The variance of both loss and accuracy is in an acceptable range.

The model's predictive accuracy reaches a level of ca. 95% on unseen test images, which is also a very desirable result.

Nevertheless, it is doubtful if this architecture would produce similarly good results on a larger scale of data with a higher number of target classes, as 12-layer architecture is fairly narrow compared to modern state-of-the-art neural networks. Increasing the depth, however, would result in a significant increase in parameters which can be challenging to tune.

Thus, in the following chapters, other architectural approaches are explored which allow for both deeper models while reducing the number of parameters.

3 Residual Neural Networks

Deep architectures allow neural networks to learn more complex patterns, theoretically increasing a model's predictive capability. He and Sun (2015) showed, however, that increasing a model's depth too much leads to stagnation or even reduction of its performance. The introduction of residual neural networks by He et al. (2016a) played a pivotal part in overcoming this limitation, allowing for significant depth increases in deep learning architectures.

3.1 Residual Learning

Very deep architectures suffer not only from a higher computational cost but also from the presence of a "degradation" effect, causing saturation and even degradation of the model's accuracy. Deeper setups should not negatively affect a model's predictive performance as the added layers could theoretically be constructed as identity mappings. However, He et al. (2016a) show that this is not the case and that deep architectures also negatively impact, the training error, meaning that the degradation phenomenon is not caused by overfitting.

Their proposed solution to overcome this limitation is to let the network's layers learn the residual mapping. If $\mathcal{H}(x)$ is the desired underlying mapping of a layer, it is now fit to $\mathcal{F}(x) = \mathcal{H}(x) - x$ and then mapped back into $\mathcal{F}(x) + x$. The residual learning framework allows for a better, faster and deeper learning environment, as it is easier to fit the residual mapping to 0 than to learn the identity mapping. He et al. (2016a) suggest that this is partly due to weights being initialised around a mean of 0 and because most solvers are not suitable for approximating the identity function.

3.2 Implementation

The residual mapping is learned by residual blocks, consisting of a sequence of convolutional layers. The identity mapping is applied by adding the input tensor of the residual block to its output via a shortcut connection, as displayed in Figure 3.1[1].

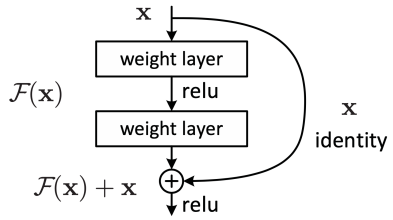
The convolutional layers no longer have to capture everything important about the input tensor that needs to be passed along to subsequent layers. Instead, they learn what information can be added to the input.

This results in an easier learning task for each residual block with better information being available, as inputs keep getting passed forward through the shortcut connections.

[1] $\mathcal{F}(x) + x$ is performed by element-wise addition.

Additionally, gradients follow shorter paths, as they are also passed along the shortcut connections through the backpropagation which allows for better weight updates in the earlier layers of the model.

Figure 3.1: Shortcut connection by He et al. (2016a)



Residual blocks are implemented through the utilisation of *identity blocks* and *projection blocks*.

When the input and output dimensions of a residual block are identical, the shortcut connection can be executed directly by an identity block, using pairwise addition(Identity shortcuts are visualised by solid-line arrows in Figure 3.2b).

Projection blocks are used when input and output dimensions differ. The dimensionality matching is achieved through (1×1) convolutions with a stride of 2 (Projection shortcuts are visualised by dashed-line arrows in Figure 3.2b).

3.3 ResNet29

3.3.1 Architecture

The model proposed for this analysis is a 29-layer residual neural network, denoted ResNet29. Its architecture is displayed in Figure 3.2. Alongside the utilisation of the residual learning framework, there are further alterations from the previously introduced model.

The first difference is the implementation of a large (7×7) convolution, followed by MaxPooling. Those two layers quarter the dimensions of the input tensor.

The model is composed of a sequence of residual blocks with each residual block consisting of 3 convolutional layers. Similar to the VGG-inspired model, each convolutional layer is followed by a BatchNormalisation layer and the ReLU activation function. The 1st and 3rd convolutional layers within each residual block feature a (1×1) convolution, yielding similar benefits as in the VGG setup. This setup is also referred to as a "bottleneck" block. The (1×1) convolutions reduce dimensions for the (3×3) convolution to reduce the number of parameters, and then scale the depth back to the input size. The last convolutional layer in each residual block has a 4-fold increase in the number of channels.

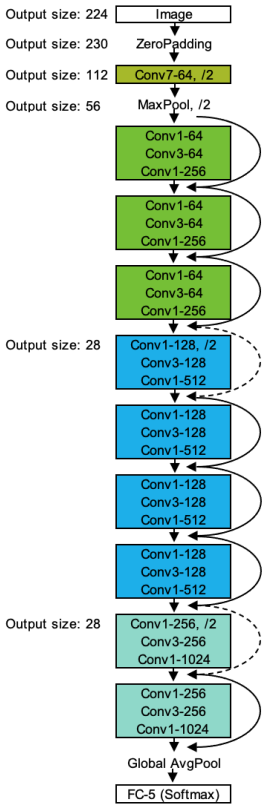
The dimensionality reduction is achieved by a stride of 2 within the convolutional layers and not by pooling layers anymore.

Figure 3.2

(a) ResNet29 Architecture
BatchNorm and ReLU not displayed for simplicity

Output size	29-layer
112×112	7×7, 64, stride 2
56×56	3×3 MaxPool, stride 2
	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$
28×28	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 4$
14×14	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 2$
1×1	Global AvgPool, FC-5

(b) ResNet29 Visualisation



The last modification is the utilisation of a Global AveragePooling layer, which was first introduced by Lin et al. (2013). Global AveragePooling provides an alternative to fully connected layers, previously used in the VGG architecture, which have a high overfitting risk. Instead of vectorising (flattening) the final convolutional layer to connect it with the fully connected layers, Global AveragePooling generates a feature map for every category of the classification task, which is fed directly into the classification function. This results in a significant reduction of parameters.

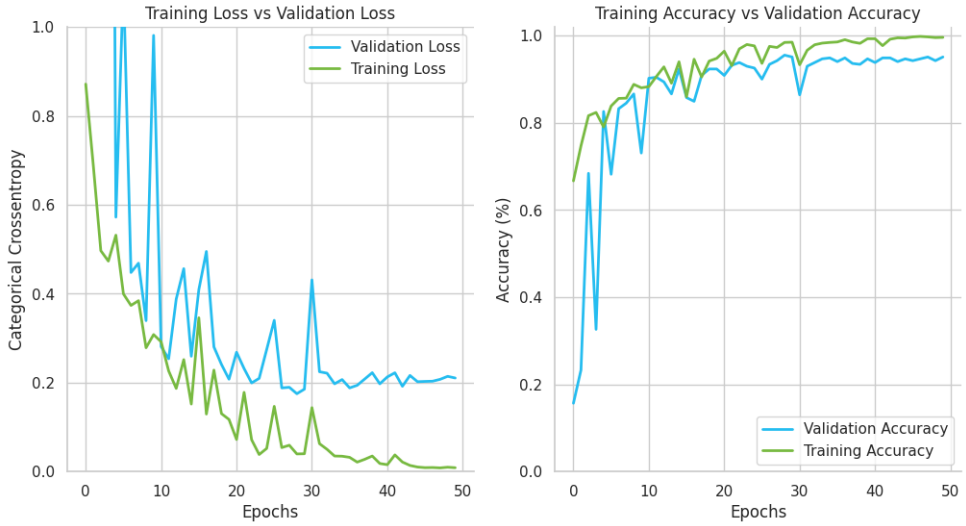
With 29 layers and a total number of 4,088,837 trainable parameters, ResNet29 has a significantly deeper architecture than the previously introduced VGG model, while also reducing the number of tunable parameters. In the context of this analysis, this is a very desirable property, as this allows for training on larger datasets while limiting the computational expenses.

3.3.2 Results

The results of ResNet29 are shown in Figure 3.3. Both training and validation loss follow a downward trajectory with no overfitting behaviour being observable, which is a

desirable characteristic, albeit the validation loss suffers from high variance during the first 30 epochs. The volatility decreases over time and the loss stays around 0.2 for the remainder of the training run.

Figure 3.3: ResNet29 results



The validation accuracy also suffers from a higher fluctuation than the VGG model, although the variance also lowers as more epochs are executed. The model exceeds 94% accuracy on unseen data in several occurrences, reaching accuracies above 95% towards the final epochs.

This is an extremely significant result as it shows that the model has similar predictive capabilities as the VGG model, while containing over 2.5 million fewer parameters, making it a far superior choice when training on a larger scale of data.

4 MobileNet V2

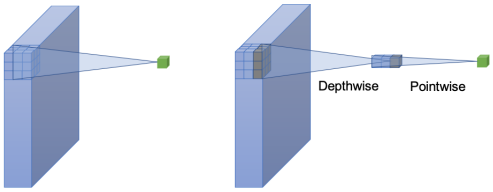
The last model used in this analysis is inspired by the MobileNet V2 architecture, originally introduced by Sandler et al. (2018). This architecture introduces several new methods which allow for deeper than-before setups while reducing the number of parameters in the model even further.

4.1 Depthwise Separable Convolutions

Instead of regular convolutional layers, the MobileNet V2 architecture considers depthwise separable convolutions, which were introduced by Chollet (2017) and split the channel- and spatialwise computations performed by a standard convolution into 2 separate steps.

For a normal convolutional layer, each convolutional filter is applied on all input channels, resulting in a high number of parameters. In a depthwise convolution, there is just one convolutional filter applied per input channel.

Figure 4.1: Standard Convolution vs Depthwise Separable Convolution
by Guo et al. (2019)



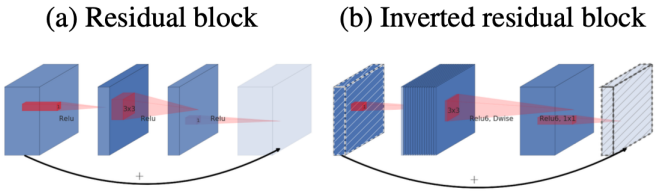
A (1×1) convolution is also referred to as a pointwise convolution. A depthwise convolution, followed by a pointwise convolution is called a depthwise separable convolution. The pointwise convolutions are used for generating a linear combination of the depthwise convolutions' outputs, as well as depth scaling to achieve the desired channel size.

4.2 Inverted Residuals

Classical residual blocks follow a *wide -> narrow -> wide* structure regarding the number of channels, as displayed in Figure 4.2 (a). The input has a high number of channels, which gets compressed by a (1×1) convolution so that the following (3×3)

convolution is performed on a lower number of channels to limit the number of parameters. A (1×1) convolution expands the channel size after that again to match the input dimensions.

Figure 4.2: Inverted Residuals by Sandler et al. (2018)



Inverted residual blocks, displayed in Figure 4.2 (b), follow the opposite approach, characterised by a *narrow -> wide -> narrow* structure. A (1×1) convolution expands the low number of input channels. Then an inexpensive depthwise convolution is applied, followed by a (1×1) convolution to downsample the channel depth back to match the input dimensions. Sandler et al. (2018) describe this setup as inverted residual blocks because the shortcut connections exist between narrow parts of the model.

An inverted residual block, making use of depthwise separable convolutions, yields the advantage of significantly reducing the number of parameters, compared to a classical residual block.

4.3 Linear Bottlenecks

Linear bottlenecks refer to bottleneck blocks, which do not feature an activation function after the final pointwise convolution. Sandler et al. (2018) discuss that using an activation function removes information, and considering bottleneck blocks with linear output improves the predictive accuracy of their model.

Figure 4.3: Bottleneck Blocks by Sandler et al. (2018)

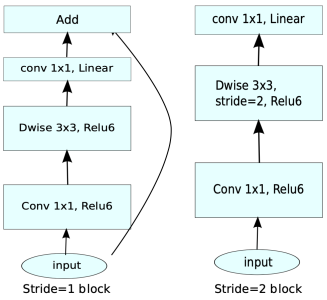


Figure 4.3 shows the 2 different inverted linear bottleneck blocks, which are present in the MobileNet V2 architecture. These blocks are also commonly referred to as MBConv

blocks. They feature depthwise separable convolutions, inverted residuals and linear outputs.

One further peculiarity is the utilisation of the ReLU6 activation function, which is capped at a maximum value of 6. The ReLU6 activation function was already implemented by Howard et al. (2017) in the MobileNet V1 architecture and has the benefit of encouraging the model to learn from sparse input earlier.

Lastly, the shortcut connection is only implemented for linear bottleneck blocks where the first pointwise convolution has a stride of 1, meaning input and output dimensions are equal. When the first pointwise convolution is applied with a stride of 2, no shortcut connection is implemented.

4.4 Architecture

The architecture of the MobileNet V2-inspired model used in this analysis is displayed in Figure 4.4. After an initial regular convolutional layer, the main part of the model consists of multiple bottleneck sequences.

Output Size	Layer	t	c	n	s
(112, 112, 32)	Conv3-32	-	32	1	2
(112, 112, 16)	Bottleneck	6	16	1	1
(56, 56, 24)	Bottleneck	6	24	2	2
(28, 28, 32)	Bottleneck	6	32	2	2
(14, 14, 64)	Bottleneck	6	64	4	2
(14, 14, 96)	Bottleneck	6	96	3	1
(14, 14, 160)	Conv1-160	-	160	1	1
160	Global AvgPool	-	-	1	-
5	FC-5 (Softmax)	-	-	1	-

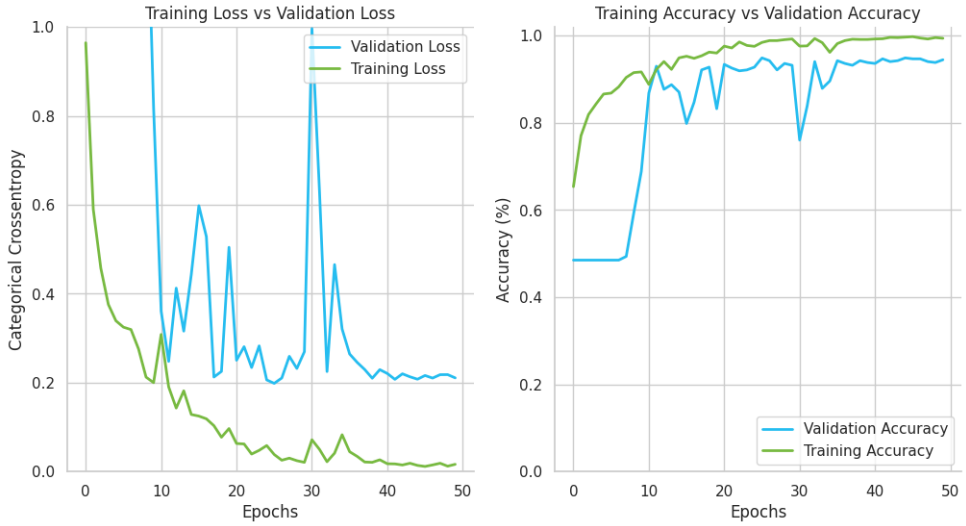
Figure 4.4: MobileNet V2 Architecture: Each line describes a sequence of identical layers, repeated n times. Layers in the same sequence have the same number of c output channels. The first layer of each sequence has stride s, and all following layers have stride 1. The expansion factor t denotes the factor by which the channel size is increased by the first pointwise convolution within a bottleneck block.

In contrast to the original MobileNet V2 architecture, a Global AveragePooling layer is implemented before the output, similar to the ResNet model described in the previous chapter. The model features a total number of 38 weight layers, resulting in 642,709 parameters, which is a significant reduction from both previously considered architectures, while also being the deepest architecture introduced so far.

4.5 Results

The results of the MobileNet model are displayed in Figure 4.5. The validation loss shows the by far highest variance so far, displaying an undesirable behaviour. However, the validation accuracy follows a smoother trajectory and ends up at 95% after the end of the training run, yielding an almost similar predictive power to the previous models.

Figure 4.5: MobileNet Architecture



Given the complex nature and vastly increased depth of this architectural approach, a higher number of epochs may be required before obtaining more stable and robust outcomes. It could further be the case that deeper models like the previously introduced ResNet29 or this MobileNet instance, would benefit significantly from an in-depth exploration of the hyperparameter space. This could allow for both a more stable training run, as well as a potential further increase in accuracy.

However, in the context of increasing models' depths and running them on larger amounts of data, the results achieved by the MobileNet architecture are remarkable. The MobileNet model yields almost the same predictive power as the VGG model while having 6 million fewer parameters, with the same number of epochs of training, underlining the strength of its architecture.

5 Transfer Learning

Transfer learning describes the method of reusing pre-trained machine learning models on new tasks and datasets. In image classification transfer learning can be performed by considering state-of-the-art neural networks, which yielded outstanding results in previous classification tasks, and letting those models generate predictions for our own problem. The rationale is that pre-trained models have learned to detect useful features in their previous classification tasks and can now use those patterns to predict new unseen images for an arbitrary classification task. This approach is sometimes also referred to as feature extraction.

5.1 Models

The models considered for transfer learning have been trained on the ImageNet dataset[1], which consists of over 1.4 million different images coming from 1,000 different classes[2].

Specifically, the following 4 ConvNet models are considered for feature extraction:

- VGG-19
- ResNet50 V2
- MobileNet V2
- Efficientnet B3 V2

The VGG-19 is the deepest VGG model introduced by Simonyan and Zisserman (2014) having 19 weight layers. In contrast to the VGG-inspired architecture introduced previously, this VGG network does not feature BatchNormalisation.

The ResNet50 V2 by He et al. (2016b) represents a 50-layer deep residual neural network, following a similar architecture as previously discussed.

The MobileNet V2 by Sandler et al. (2018) follows the same approach as previously introduced, albeit this model contains a significantly higher number of bottleneck sequences.

The Efficientnet architecture, by Tan and Le (2019), introduced a scaling method which allowed for a significant increase in model complexity while maintaining a low

[1]The dataset can be found at: https://www.image-net.org/download.php

[2]The ImageNet dataset contains a completely different set of classes, but the models should still be able to perform satisfying predictions for our problem.

number of parameters[3]. Model scaling is performed by not only increasing the depth (e.g. number of layers) but also the width (e.g. number of channels), and resolution (e.g. the input size of the images), using a constant ratio.

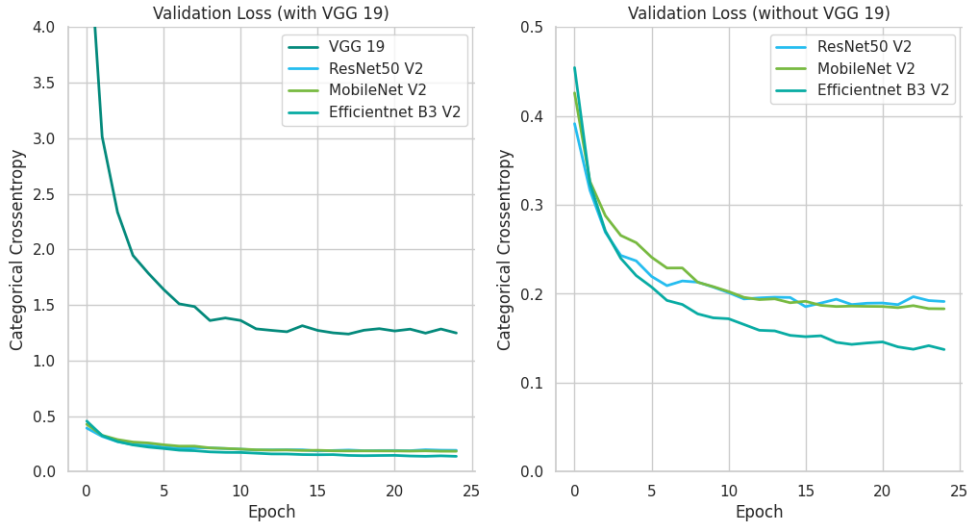
5.2 Execution Specifications

All models' weights have been obtained from a training run for a classification problem with 1,000 target classes. To make the models suitable for our classification task, a fully connected layer containing 5 neurons as the output layer, using the softmax activation function, is added to each model. The models are executed for 25 epochs on the training set and evaluated on the test set. For image preprocessing the respective recommended preprocessing functions of the TensorFlow framework are utilised[4].

5.3 Results

Figure 5.1 displays the models' losses on the test set. With the exception of the VGG-19, all models yielded excellent results.

Figure 5.1: Transfer Learning Losses



The Efficientnet B3 V2 obtains the best results, reaching a final loss value lower than the best loss obtained by the previously used models. It is observable, that all models'

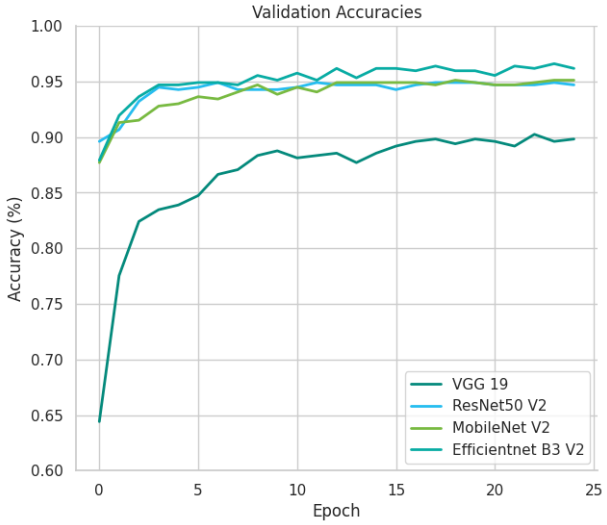
[3]While the ResNet50 has 26 million parameters, Efficientnet B0 has only 5.3 million parameters. The Efficientnet B3 architecture used for feature extraction is based on a 101-layer ResNet and has 12 million parameters.

[4]Though the preprocessing procedures vary slightly from model to model, all preprocessing functions involve some form of centring or standardisation.

losses also follow a very smooth trajectory, displaying little to no variance throughout the entire training run.

Figure 5.2 shows that the models also perform remarkably well with respect to the predictive accuracy, again except for VGG-19. Efficientnet B3 V2 again scores the best value, reaching accuracies above 96% for the majority of epochs.

Figure 5.2: Transfer Learning Accuracies



These outcomes demonstrate how incredibly well modern state-of-the-art models can perform on unseen data. This is especially impressive, given the fact, that almost no training needs to be performed[5] and the transfer learning models reach outstanding results after only a very small number of epochs. This is due to the models' complex nature and the fact that they were trained on a vastly larger dataset for an extremely high number of epochs[6]. Thus, when it comes to achieving high predictive accuracies, with extremely low computational costs, transfer learning is often the preferred procedure.

[5]There is only a small number of parameters that had to be trained, introduced by adding an output layer with 5 neurons to generate predictions for this task.

[6]For instance, the original ResNet model was trained for 60×10^4 epochs.

6 Conclusion

6.1 General Evaluations

The main insights to be drawn from the analysis are:

- The residual learning framework allows for an increased depth, which is not feasible to obtain using regular network architectures.
- However, deeper architectures are more difficult to train and may yield less stable results.
- (1×1) convolutions are versatile building blocks for generating high-performing models. They introduce non-linearity, allow for dimensionality reduction/dimensionality matching and are computationally inexpensive, making them especially useful for bottleneck blocks.
- Depthwise convolutions are a fantastic alternative to regular convolutions, allowing for a significant reduction in trainable parameters.
- Feature extraction is the quickest and cheapest solution to produce high-quality predictions for arbitrary classification problems.

6.2 Potential Improvements

The 2 strongest levers for improving the results shown in this analysis are higher computational resources and more data.

With more computational resources available, longer training runs could likely result in a further increase in predictive accuracy. As previously mentioned, this, in combination with hyperparameter tuning could likely benefit the deeper architectures of the ResNet29 and MobileNet models.

A larger amount of data would allow the models to learn more patterns which may help to better distinguish the artworks of different painters. One possible solution to obtaining more data could be to artificially generate new images through the image augmentation phase. Instead of replacing the original images with their augmented versions, as done in this analysis, it is possible to leave the original images as they are and increase the dataset by adding the augmented versions.

Lastly, more advanced model architectures, like the Efficientnet framework, could be explored to investigate if higher predictive capabilities can be achieved.

Bibliography

François Chollet. Xception: Deep learning with depthwise separable convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1251–1258, 2017.

Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256. JMLR Workshop and Conference Proceedings, 2010.

Yunhui Guo, Yandong Li, Liqiang Wang, and Tajana Rosing. Depthwise convolution is all you need for learning multiple visual domains. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 8368–8375, 2019.

Kaiming He and Jian Sun. Convolutional neural networks at constrained time cost. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 5353–5360, 2015.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016a.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. In *Computer Vision–ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11–14, 2016, Proceedings, Part IV 14*, pages 630–645. Springer, 2016b.

Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. pmlr, 2015.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25, 2012.

Min Lin, Qiang Chen, and Shuicheng Yan. Network in network. *arXiv preprint arXiv:1312.4400*, 2013.

Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4510–4520, 2018.

Shibani Santurkar, Dimitris Tsipras, Andrew Ilyas, and Aleksander Madry. How does batch normalization help optimization? *Advances in neural information processing systems*, 31, 2018.

Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

Mingxing Tan and Quoc Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International conference on machine learning*, pages 6105–6114. PMLR, 2019.

Declaration of Own Work

I declare that this material, which I now submit for assessment, is entirely my own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my work. I understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion or copying. This assignment, or any part of it, has not been previously submitted by me or any other person for assessment on this or any other course of study.