Face Recognition from Ancient Roman Coins

Using Convolutional Neural Networks

Imanol Schlag

Supervisor: Dr Ognjen Arandjelović

Department of Computer Science
University of St Andrews

This dissertation is submitted for the degree of
Master of Science

August 2016

This thesis is dedicated to my parents for their love, endless support, and encouragement.

Declaration

I, Imanol Schlag, hereby certify that this thesis, which is approximately 10'500 words in length, has been written by me, that it is the record of work carried out by me, and that it has not been submitted in any previous application for a higher degree. Credit is explicitly given to others by citation or acknowledgement.
This project was conducted by me at The University of St Andrews from May 2016 to August 2016 towards the fulfilment of the requirements of the University of St Andrews for the degree of Master of Science in Artificial Intelligence under the supervision of Dr Ognjen Arandjelović.
In submitting this project report to the University of St Andrews, I give permission for it to be made available for use in accordance with the regulations of the University Library. I also give permission for the title and abstract to be published and for copies of the report to be made and supplied at cost to any bona fide library or research worker, and to be made available on the World Wide Web. I retain the copyright in this work.

[illegible]

Imanol Schlag
August 2016

Acknowledgements

I would like to thank Ognjen Arandjelović, my supervisor at the University of St Andrews, for friendly and open-minded supervision. Next, I want to thank the School of Computer Science for providing me with a workplace and the necessary hardware. I'm also grateful to Tom Kelsey, Manuel Kehl, and Francis Priestland for technical discussions and companionship.

Abstract

In recent years, large-scale object recognition challenges have shown that Convolutional Neural Networks achieve state-of-the-art performance on Computer Vision problems like object detection and image classification. Convolutional Neural Networks benefit from datasets with many images per class which helps in learning robust and invariant features. In this thesis, we aim at applying Convolutional Neural Networks to the problem of identifying the emperor depicted on the obverse of an ancient Roman imperial coin. For this task, we have collected and processed an unbalanced and challenging dataset of 22’824 samples spanning 83 different classes. In contrast to popular belief, our results demonstrate that the class imbalances in our data are not impacting the networks classification performance. On the contrary, using the unbalanced training set we achieve the best top-1 mean accuracy of 77.04% and a top-3 mean accuracy of 91.35%. We make use of Stochastic Gradient Descent with momentum, Batch Normalization, Rectified Linear Units, and Dropout. Our results show that Convolutional Neural Networks perform very well on a wide variety of coins including coins in poor condition.

Table of Contents

List of Figures

List of Tables

Nomenclature

Acronyms / Abbreviations

BN	Bach Normalization
CNN	Convolutional Neural Network
GPU	Graphics Processing Unit
MSE	Mean Squared Error
NN	Artificial Neural Network
ReLU	Rectified Linear Unit
RL	Reinforcement Learning
SL	Supervised Learning
tanh	Hyperbolic Tangent
UL	Unsupervised Learning

Chapter 1

Introduction

The Roman currency has been used widely during the Roman Republic and the Roman Empire and consisted of a coinage made out of different metals such as gold, copper, and others. After the transformation of the Roman Republic into the Roman Empire in 27 BC, the ruling emperor usually became the motive of the obverse of the coin. However, due to several issuers, manual production, and wear and tear those ancient coins sometimes only share very high-level features and sometimes do not even look very similar.

The identification of those ancient coins is an important task for collectors and dealer of such. For amateur collectors and laymen this can be a significant and time-consuming challenge. Experts, on the other hand, usually develop an intuitive skill after visually classifying numerous coins. Their time is, however, limited and an automatic tool would be of great use for novices, museums, and auction sites.

Deep Learning and Artificial Neural Networks (NN) have been around for many years but have now with recent advancements become highly popular in the Computer Vision community. Especially Convolutional Neural Networks (CNN) or some derivative are currently achieving the best accuracy on several Computer Vision challenges. In this work, we want to investigate the effectiveness of CNNs on classifying the emperor shown on the obverse of the coin. For this task, we build a dataset of 20’000 coins spanning 83 classes. Unlike popular Computer Vision datasets, this particular dataset we have created varies a lot in the number of samples per class due to the rarity of different coins. Furthermore, it is not clear, how good CNNs are able to generalise the refined distinctions even human experts can struggle with.

1.1 Problem Statement

The goal of this project is the identification of the emperor depicted on the obverse of an ancient Roman coin. Naturally, we focus on imperial coins due to the depiction of Roman emperors and exclude any coins presenting deities, symbols, or objects.
The thesis aims at investigating the following:

- How well are CNNs suited for identifying the personalities depicted on the obverse of ancient imperial Roman coins?
- How does the grade of the coin influence the classification accuracy?

1.2 Thesis Overview

This section concludes the introduction of this work. The next section deals with the theoretical aspects of NNs in general and CNNs in particular. Afterwards, we introduce the reader to the dataset we created and our CNN architecture including its memory requirements and number of weights. This is followed by the actual performance of the network visualizing the optimization procedure. The last section deals with a discussion of the results, as well as a visualization of hard examples and the features learned by the network.

Chapter 2

Background

A NN is a universal function approximator [5][18]. It consists of hundreds, thousands, or even millions of little parameters, also called weights, which are iteratively tweaked by a *Learning Algorithm*. Due to its universality, it can be applied to all kinds of problems, but it usually is performing the best when it is used with very high-dimensional data such as audio, images, or text. It can be intuitively described as a high-level feature extractor which is able to e.g. recognise a cat in an image or the positive or negative nature of a short product review on a website. Especially in an image classification task where the number of classes is kept at a reasonable size (e.g. 1000 classes consisting of animals, planes, etc.) a NN is sometimes even able to surpass human level performance [7]. It is therefore not surprising that NNs have found their way into the products of many IT companies such as Google, Facebook, and Microsoft and that such companies are also performing bleeding-edge research in this field.

The learning of very high-level features from data is loosely defined as *Deep Learning* and almost always refers to the iterative training of a computationally expensive NN on a vast amount data. It belongs to the field of Machine Learning which itself is part of the field of Artificial Intelligence.

2.1 Artificial Neural Networks

NNs have been around since 1943 when McCulloch and Pitts introduced a mathematical model based on logic [14]. At first, they were not trainable but subsequent research focused on Learning Algorithms like Supervised Learning (SL), Unsupervised Learning (UL), and Reinforcement Learning (RL). Such Learning Algorithms allowed NNs to change their adaptive weights in order to approximate the non-linear functions in the data it was trained on. Feed-forward NNs consist of artificial neurones which are connected in an acyclic graph and therefore do not contain any kind of recurrent connections. One the most common architectures is that of the multi-layer perceptron which organises the neurones in layers. The artificial neurone and the feed-forward architecture was inspired by the neuroscience of the mid-20th century. Nowadays we know, that the artificial neurone and the architecture of NNs have very little in common with the biological neural networks in our brains.

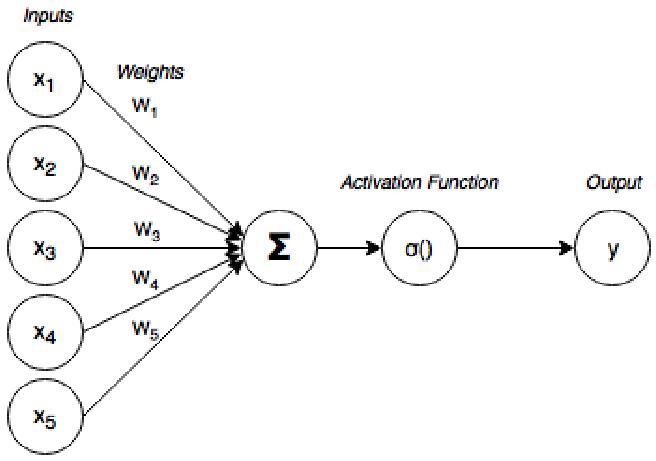


Fig. 2.1 The artificial neurone is based on Rosenblatt's perceptron [15] and consists of the input vector x, the according weight matrix W, the bias b and the non-linear activation function σ.

In a feed-forward NN, the artificial neurones are grouped in layers. The input of the first layer consists of the raw data and the input of every subsequent layer consists of the output of the previous layer. If it is also fully-connected then the input of every artificial neurone in a layer consists of the output of every artificial neurone in the previous layer.

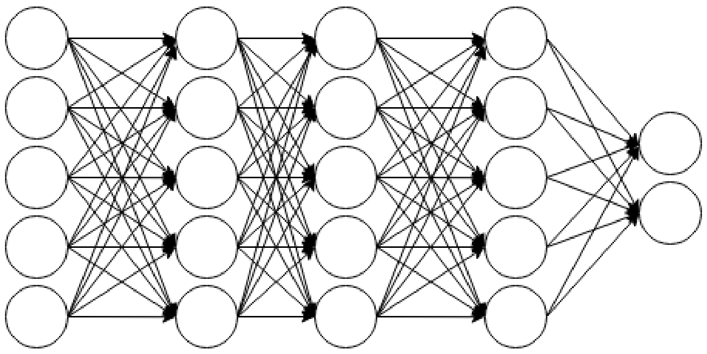


Fig. 2.2 A fully-connected feed-forward NN with two output neurones. Every circle encompasses the sum and activation function. Labels left out for clarity.

Among different types of Learning Algorithms, Supervised Learning (SL) has achieved the best results so far. In a SL algorithm, the NN is trained on a data set where the correct response of the network is known and available. We call such data labelled data. The power of labelled data is that we can measure the error of the NN for every sample. Using NNs, this error measure is calculated using the cost function C, the correct class or prediction y from our data, and the prediction or classification from the NN $\hat{y}$. A widely used error in Machine Learning is the mean-squared error (MSE) but other measures exist and using NNs which perform classification or prediction tasks the cross-entropy error usually performs slightly better than the MSE. In the equation 2.1, $X = \left\{x^{(1)},\ldots,x^{(n)}\right\}$ is the set of input samples in the training data and $Y = \left\{y^{(1)},\ldots,y^{(n)}\right\}$ is the corresponding set of labels.

$$C(X,Y) = -\frac{1}{n}\sum_{i=1}^{n} y^{(i)} \ln \hat{y} + \left(1 - y^{(i)}\right) \ln\left(1-\hat{y}\right)) \tag{2.1}$$

Now in order to train a NN, we are going to treat it as an optimisation problem. The cost of our NN, i.e. the error we try to minimise, depends on all of its parameters. There exists a set of parameters which achieve the lowest mean error over our labelled training data. In other words, the Learning Algorithm tries to find the global minimum in the parameter space of the chosen NN architecture. Optimisation is a branch of mathematics which is very well studied. Unfortunately, the training of a NN is a non-convex optimisation problem which is the main reason why such training is difficult. In order to optimise a non-convex function i.e. our NN, we have to fall back to iterative algorithms. The most commonly used Learning Algorithm is a form of Gradient Descent (GD). GD makes use of the fact that the function which represents our NN is derivable. Thanks to Calculus and the Chain Rule in particular, we can calculate the gradient of our current position in parameter space. The gradient tells us for every dimension in parameter space which way we have to move to decrease the error for that specific sample. Here the *learning rate* is an important *hyperparameter* of the NN. The learning rate simply stretches the size of the step that we take down the hill in our parameter space landscape. Hyperparameters are all the parameters of our network which are not optimised by GD - ergo everything except the weights and biases of the artificial neurones. Such hyperparameters are usually chosen based on empirical experiments and experience.

2.2 Convolutional Neural Networks

In 1959, the two neuroscientists Hubel and Wiesel published their findings on simple and complex cells in a cat's visual cortex. Specific cells fired in response to different visual inputs. Some of them were simple edge detectors while other cells fired based on positional invariant features observed by the cat's eyes. These findings, among others, lead to the development of the Neocognitron by Fukushima in 1979 [6]. It inspired CNNs and was already very similar in its general architecture to current state-of-the-art CNNs. This new type of NN, in contrast to the computationally expensive fully-connected NNs, convolves a *kernel* or *filter* over the two-dimensional input of an image for every specific pattern it tries to learn. The Neocognitron is also one of the first NNs which can be classified as *deep* due to the first successful use of multiple layers.

A classical CNN architecture is made up of *convolutional layers* and *pooling layers*. In contrast to fully-connected layers, in a convolutional layer, every artificial neurone is only connected with respect to its local receptive field i.e. a small two-dimensional area, our kernel. The discrete convolution of the kernel with input image is sliding the small two-dimensional kernel over every pixel of its input which leads to a filtered image where every pixel equals the activation of that artificial neurone at that position. Essentially we are sharing the weights

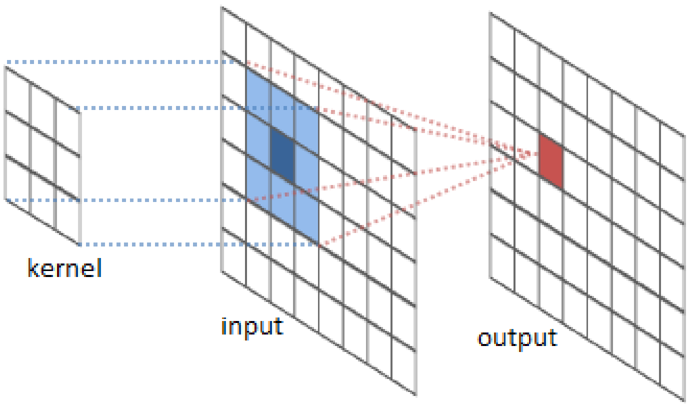


Fig. 2.3 A visualization of the discrete convolution i.e. the sliding window procedure. Image taken from cs231n.github.io.

and biases of a specific kernel among all artificial neurones.This *weight sharing* feature allows CNNs to be computationally cheaper than fully-connected NNs. The size of the kernel and the number of kernels are the most important hyperparameters of such a layer. So for every neurone $a_{i,j}$ in the filtered image we perform eq. 2.2 where the two dimensional kernel W is multiplied with just a small section of the image. After the convolution we usually add a bias b and transform the input using the non-linear activation function σ. However, for this thesis we will later introduce and add a Batch Normalization Transformation before the activation.

$$\sigma\left(b+\sum_{m}\sum_{n}a_{i+m,j+n}W_{m,n}\right) \tag{2.2}$$

Once the weights have been trained the kernels of the convolutional layer become positional invariant feature extractors. The first layer, in particular, is surprisingly similar to the findings in neuroscience where the neurones in the first layer of our visual cortex behave similarly to edge detectors. Similar to the convolutional layer, the pooling layer consists of a sliding

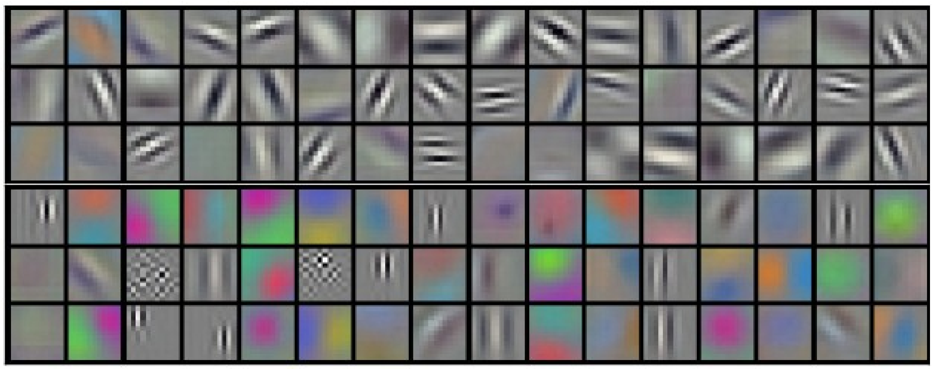


Fig. 2.4 A visualization of the first layer filter bank of AlexNet take from [12]

window operation. The main goal of the pooling layer is to reduce the dimensionality of the processed image. Usually, the stride of the sliding window in a convolutional layer and the padding of the input image is set such that the filtered image has the same dimensions as the input image. This leads to as many filtered images as there are kernels in the filterbank. Hence, the dimensionality of the input after a convolutional layer has increased dramatically. The goal of the pooling layer is to reduce the dimensionality of every filtered image. Several pooling mechanisms exist. A popular pooling algorithm is called *max-pooling*. With max-pooling, we perform a sliding window operation but only keep the maximum value at every position of that sliding window, effectively subsampling the filtered image. If a filtered image has the size of 100×100 pixels and we can perform max-pooling using a sliding window of 2×2 pixels which then effectively reduces the filtered image by four to 50×50 pixels.

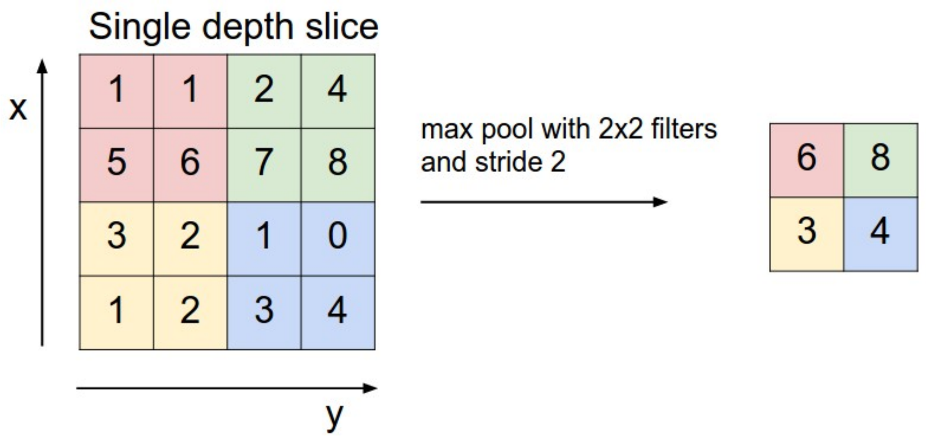


Fig. 2.5 The max pooling operation with a stride of 2. Image taken from cs231n.github.io/convolutional-networks/.

One of the first modern CNNs was Yann LeCunns LeNet in 1994 [13]. It was trained to recognise the handwritten numbers of the MNIST dataset and helped propel the field of Deep Learning. The main insight was that image features are distributed across the image and convolutions with learnable weights are an effective way of extracting them.

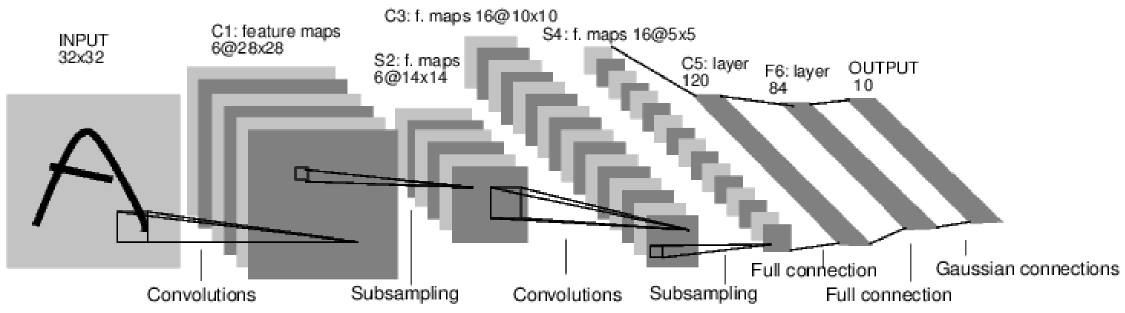


Fig. 2.6 The architecture of LeNet-5 from [13]. A CNN for handwritten digit recognition. Each plane is a filtered image, called filter map in the paper.

In 2012 then came the breakthrough for CNNs in the form of AlexNet by Alex Krizhevsky [12]. AlexNet introduced several new techniques in order to deal with the Vanishing Gradient Problem and help the network to better generalise the features learned from the training data. Nevertheless, it follows the same insights of LeNet and scaled them up greatly thanks to the increased computational power and memory of graphical processing units (GPU) and the big training data available for that specific task. Since then, CNNs have been dominating various Computer Vision challenges.

2.3 Deep Learning

The key feature of Deep Learning is about learning and generalising high-level features from training data. Using NNs, this means to process the data through several layers with non-linear activation functions. In 1991, Hochreiter et al. identified the *unstable gradient problem* which hinders NNs to effectively train networks with more than just a few non-linear activation layers [9]. The unstable gradient problem emerges from the multiplicative nature of the derivative of our NN architecture. In order to calculate the gradient of a shallow layer, i.e. a layer close to the input, we have to backpropagate the error through all the deeper layers using the Chain Rule. As a result, the error is multiplied by several factors before reaching the appropriate parameter. This makes the gradient very unstable. If the factors tend to be higher than 1.0 the error will increase exponentially to infinity resulting in unusable step sizes. And if the factors tend to be smaller than 1.0 the error quickly decrease towards zero which doesn't tweak the shallow layers and therefore the Learning Algorithm is not able to train the NN.

A simple NN with a single hidden layer is theoretically capable of describing any type of function as long as there are an infinite number of artificial neurones available. However, the number of neurones needed for any function can be reduced exponentially by using deeper network architectures. This is the reason why the research community is always interested in developing techniques to train deeper NNs where the data processed goes through several non-linear activation functions. In this work, we make use of some popular techniques which have been shown to push back the unstable gradient problem and help CNNs to generalise well on the training data.

2.3.1 Gradient Descent with Momentum

In GD, the algorithm has to run through all the samples in the training set to do a single update for the parameters in a particular iteration. This is not always possible due to the size of training sets, the huge memory requirements of big CNNs, and the limited amount of memory available on a GPU. This type of GD is sometimes also called batch GD and can be quite costly for just a single step. Stochastic Gradient Descent (SGD), on the other hand, doesn't accumulate the weight updates over all samples before updating. Instead, it performs a single iteration by averaging only a subset of the available samples. The gradient of SGD is based on a stochastic approximation of the "true" cost gradient. Due to its stochastic nature that path towards the global cost minimum is not as "direct" as in GD. However, it has been shown that it can reliably converge to a global minimum for some optimization problems [4]

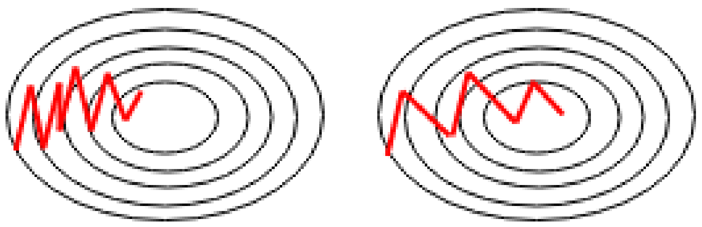


Fig. 2.7 Left: A standard GD algorithm oscillating back and forth. Right: GD optimized with momentum to speed up convergence.

In this work, we randomly draw samples with replacement and perform data augmentation on them before adding them into a mini-batch. The iterative update for the weights and biases of our NN, is the mean of the partial derivative of our cost function C with respect to the specific weight $W^{(l)}$. The learning rate η is the step size and a hyperparameter of the NN. k is defined as the size of the batch. If $k = 1$ we are taking small steps based on a single sample. This is also called on-line learning. If k equals the number of samples in the training set then we are processing the whole batch which equals the classical batch GD. Everything in between is a mini-batch and also introduces a stochastic element into the optimization process.

$$W^{(l)} = W^{(l)} - \frac{\eta}{k} \sum_{i}^{k} \frac{\partial C(W,b)}{\partial W^{(l)}} \tag{2.3}$$

GD and SGD run into problems if the landscape of our cost function resembles a long shallow ravine with steep walls on the sides. Both will tend to oscillate across the sides of the ravine since the negative gradient will mostly point down the steep side rather than along the ravine towards the optimum. Momentum is one method of amplifying a weak but constant gradient in order to improve convergence speed in that direction. Equation 2.4 and 2.5 describe the update for a single weight $W^{(l)}$. The update for the biases or any other parameter is equivalent.

$$v_t = \gamma v_{t-1} + \frac{\eta}{k} \sum_{i}^{k} \frac{\partial C(W,b)}{\partial W^{(l)}} \tag{2.4}$$

$$W^{(l)} = W^{(l)} - v_t \tag{2.5}$$

More sophisticated first-order optimisation algorithms have shown to be faster in optimising the non-convex function of NNs. Some of them are e.g. RMSProp, AdaGrad, AdaDelta, and Adam among others. Almost all of them make use of the concept of momentum which is taking knowledge of previous steps to predict where it is heading. While being very similar, most of these algorithms only differ in the use of an adaptive learning rate and an adaptive momentum, as well as, the exponential decay of such attributes. This, while faster, is automatically also a big disadvantage because they need to keep track of more numbers per parameter which leads to a significantly bigger memory consumption. SGD with a simple momentum has shown to be very effective in combination with Batch Normalization. It allows us to trade the memory consumption of our algorithm with bigger networks while embracing the possibility of longer training times. For this project, thanks to the use of Batch Normalization the training times have been reasonable which is why we have decided to stick with SGD with momentum instead of a more sophisticated optimisation algorithm.

2.3.2 Batch Normalization

Training deep NNs without Batch Normalization (BN) requires very careful tuning of the weight initialization and learning parameters. This is because, in a deep NN, the input of every layer is affected by the parameters of the previous layers. This leads to a complex system where even small changes in the more shallow layers can have a strong effect on the deeper layers of the network. So far this has been dealt with small learning rates and careful initialization which lead to longer training times. But in a paper from December 2015, Sergey Ioffe and Christian Szegedy introduced a technique called Batch Normalization to accelerate learning [10]. They suggest that a change in the distribution of activations because of parameter updates slows down learning. They call this change the *internal covariate shift*. Their paper proposes a computationally cheap BN layer which is put in front of every non-linear activation layer. The goal of the layer is to whiten the mini-batch i.e. the mini-batch is centred to zero-mean and unit variance. For most Machine Learning algorithms covariance shift isn't a big deal. But deep NNs, by its very nature, are parametrized in a hierarchical fashion, and therein lies the root of the problem.

A more intuitive way to describe this problem could be a children's game which some of us might have played when they were young. The participants are sitting in a circle. A person whispers a sentence into the ear of the person to its right. The message continues to travel from person to person changing slightly every time. When the message reaches the last person (to the left of the first person) the message has experienced a covariance shift - it has changed in meaning. BN helps to normalise the message every time before it is heard by an ear based on the statistical mean and variance of all the messages passed through.

The input of the BN layer is the mini-batch $B = \{x_1, \ldots, x_k\}$. The normalized values are labelled $\hat{x}_{1 \ldots k}$ and the linear transformation for the scale and shift are $y_{1 \ldots k}$. y is the output the BN layer and, therefore, the result of the *Batch Normalizing Transform*. Equation 2.6 and 2.7 calculate the mean and variance over the mini-batch B. Equation 2.8 normalizes x_i and scales and shifts the inputs in equation 2.9 according to the learned parameters γ and β. The parameters are unique for every BN layer in the network.

$$\mu_B \leftarrow \frac{1}{k} \sum_{i=1}^{k} x_i \tag{2.6}$$

$$\sigma_B^2 \leftarrow \frac{1}{k} \sum_{i=1}^{k} (x_i - \mu_B)^2 \tag{2.7}$$

$$\hat{x}_i = \leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}} \tag{2.8}$$

$$y_i = \leftarrow \gamma \hat{x}_i + \beta \tag{2.9}$$

As a result, a higher learning rate can be chosen for deeper NNs which speeds up their training. Learning also benefits from BN due to the increased flexibility on the mean and variance value of every dimension in every layer, hence accuracy is increased. Another advantage is the possibility to use non-linear activation functions which can otherwise be saturated such as the sigmoid or tangent hyperbolic activation functions. While BN does enhance the overall deep network training task, its precise effect on gradient propagation is still not well understood.

2.3.3 Rectified Linear Unit

The activation function in a NN has very specific purpose. Imagine a NN without activation function but with a simple multi-layer structure as in figure 2.8. If we leave out the activation function, we can calculate the networks output as follows.

$$y = b * W_{by} + h_1 * W_{h_1 y} + h_2 * W_{h_2 y} \tag{2.10}$$

h_1 and h_2 can be computed in a similar way.

$$h_1 = x_1 * W_{x_1 h_1} + x_2 * W_{x_2 h_1} \tag{2.11}$$

$$h_2 = x_1 * W_{x_1 h_2} + x_2 * W_{x_2 h_2} \tag{2.12}$$

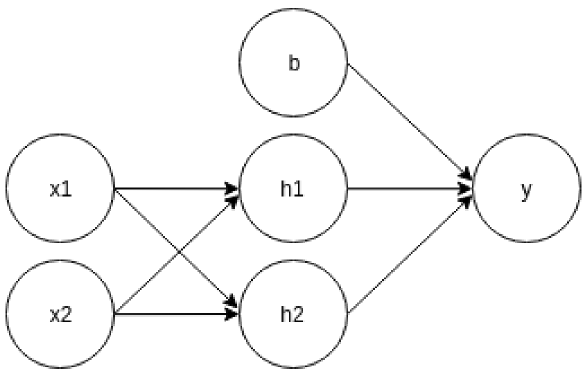


Fig. 2.8 This is just a simple network to show how a deep NN without activation functions will reduce to a linear regression.

Next, we can substitute in h_1 and h_2 and simplify the expression.

$$y = b * W_{by} + (x_1 * W_{x_1 h_1} + x_2 * W_{x_2 h_1}) * W_{h_1 y} + (x_1 * W_{x_1 h_2} + x_2 * W_{x_2 h_2}) * W_{h_2 y} \tag{2.13}$$

$$y = b * W_{by} + x_1 (W_{x_1 h_1} * W_{h_1 y} + W_{x_1 h_2} * W_{h_2 y}) + x_2 (W_{x_2 h_1} * W_{h_1 y} + W_{x_2 h_2} * W_{h_2 y}) \tag{2.14}$$

As we can see, if we leave out the activation function, the network, no matter its depth, will reduce to a simple linear regression. When the activation function was conceived for the first time, its purpose was to squash the output of the network into a continuous range between 0 and 1 to enable an iterative optimisation. The continuity is a necessary property to be able to calculate the first derivative used when backpropagating the error through the network during training. The *sigmoid* and *hyperbolic tangent* (tanh) functions were used extensively for this purpose. However, a undesired property of the sigmoid and tanh function is the saturation of the neurones activation at its tails close to 0 and 1 for the sigmoid and -1 and 1 for the tanh. The local gradient at its tails is very small and will rapidly decrease the backpropagated error, thus eliminate training in shallow layers.
In recent years, the Rectified Linear Unit (ReLU) has become very popular. The ReLU is a non-linear function since it doesn't satisfy the property of a linear function (eq. 2.15).

$$f(ax + b) = af(x) + f(b) \tag{2.15}$$

The function is very close to being linear which preserves many properties that make linear functions easy to optimise with gradient based algorithms. It was found to be very computationally cheap and to greatly accelerate training [12].

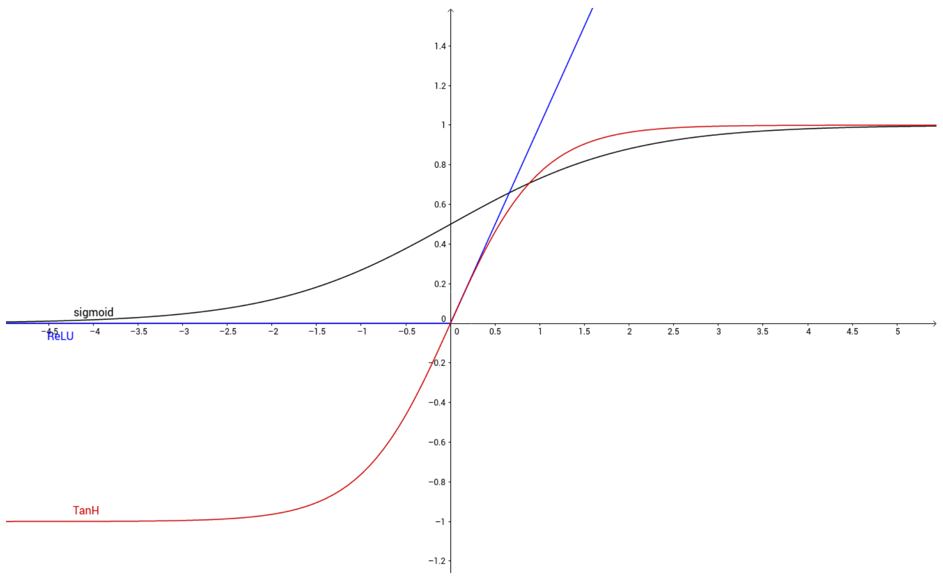
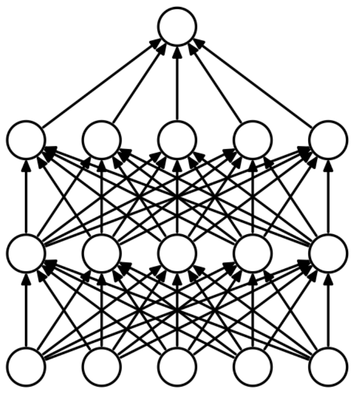


Fig. 2.9 A visualization of the Sigmoid, Hyperbolic Tangent (Tanh), and Rectified Linear Unit activation functions.

2.3.4 Dropout

Dropout is a popular regularisation technique used in NNs to improve its generalisation capabilities [17][8]. A regularisation technique helps the network to not overfit the noise in the data but to instead learn more general patterns. Dropout is usually only used in a fully-connected network. During training on each training sample, the network randomly omits 50% of the neurones of every layer. This prevents the emergence of co-adaptations where a specific feature is dependent on several neurones instead of just one. This forces an artificial neurone to not be relying on other artificial neurones in the same layer. Another way to interpret the Dropout method is to imagine every different configuration of active neurones as an own model during training. This way, Dropout simply can be understood as an effective way of performing model averaging. During testing, all artificial neurones in the network are used but halved in order to compensate for the double amount of neurones - effectively averaging the results of many simpler NNs. Dropout has shown to be also very effective when used over the fully-connected part of a CNN. It is very popular and has been used successfully for many classification and regression tasks.


(a) Standard Neural Net

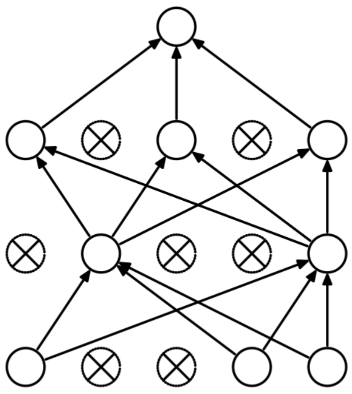

(b) After applying dropout.

Fig. 2.10 A Dropout NN model. On the left is a standard fully-connected feed-forward NN. On the right is a thinned NN by applying the dropout procedure where randomly chosen neurones are deactivated.

2.4 Ancient Roman Coins

In Roman times, coin-minting techniques were shrouded in secrecy which is why we don't know precisely how they were produced. The weights of the coins were typically standardised and started off as discs of metals called blanks. The blanks were produced by pouring molten metal into a mould to create a number of them. The final step of this process transforms a blank into currency. The heated and therefore malleable blank is placed in between two metal dies just like in figure 2.11 and struck by a heavy hammer leaving an impression of the dies on both sides of the coin. For many emperors, multiple dies existed effectively producing many subclasses of the same coin type. Additionally, some coins have endured many years of use, were worn down and have lost distinctive features due their exposure to natural forces. These individual differences make a coin classification solely based on the obverse a very challenging task.

Fig. 2.11 The coin blank was positioned on the anvil die with the punch positioned over it. A graphic from littletoncoin.com.

2.5 Related Work

Classifying ancient Roman coins has shown to be a very difficult problem. In recent years there has not been much work on specifically classifying Roman imperial coins. All of the work we have found made use of the obverse and reverse side of the coin using specialised Computer Vision algorithms such as Arandjelović in [2]. Others, like Kim and Pavlovic used more general purpose Computer Vision algorithms such as SIFT descriptors to a certain success but evaluated them only on 2'800 coins presumably in good condition. In their more recent paper [Kim and Pavlovic Rutgers], they make use of a big pre-trained but then fine-tuned CNN. They evaluate their network on a private dataset consisting of 4'526 coins and achieve significantly better results compared to a Support Vector Machine.

Chapter 3

Methodology

While there are several websites with labelled high-quality ancient Roman coins, we have not found a public dataset of imperial Roman coins with more than 1000 samples and more than 20 classes. We decided to extract the images and labels from a collectors website and perform some image processing in order to build our own dataset from which we then subset or train and test sets.

Once the data has been collected, we used the Machine Learning Library TensorFlow to implement our CNN algorithm [1]. In TensorFlow the image input can be build using input pipelines. We build two separate ones. The *unbalanced* input pipeline builds batches by randomly choosing a sample from the whole train set while the *uniform* input pipeline was designed to deal with possible class imbalance issues and oversamples from classes of smaller size.

A completely separate set of images was provided by Prof. Arandjelović. We performed the same image processing and build a validation set. However, the validation set doesn't include samples for all classes and only a small number of samples per class. The main goal of the validation set, however, is not to perform model comparison but to evaluate the performance of the network based on the grade of the coin.

3.1 Data Set

We collected all the coins annotated as imperial Roman coins from a numismatic research website called CoinProject.com. The website and its database was built and is maintained by dozens of volunteers including experts. While there were also a few other websites providing coin data, CoinProject.com was by far the biggest with the best quality of coins and labels.

At the time of download, the website listed 29’807 coins. Be aware that this number will now decrease as the data continues to be processed. In addition to the identity of the emperor on the coin, we also downloaded four further interesting data points such as a Unicode inscription and an English description using coded language. The inscription and English description are very interesting and could potentially be used for other Machine Learning algorithms which go beyond image classification such as automatic caption generation.

The web scraping was performed using a tool called Data Scraping written in .NET but any other Regex based web scraping tools could have been used. Using Regex and an automated link generation script we were able to collect all the image URLs and extract the correct labels from the HTML source of each web page related to imperial coins. Due to a few network problems and some missing images, the number of images decreased to 28’273.

All of the images that we have downloaded successfully contained the obverse side of the coin on the left side of the image and the reverse side of the coin on the right side of the image. The image is a close up and usually had a plain white background like in fig. 3.2.

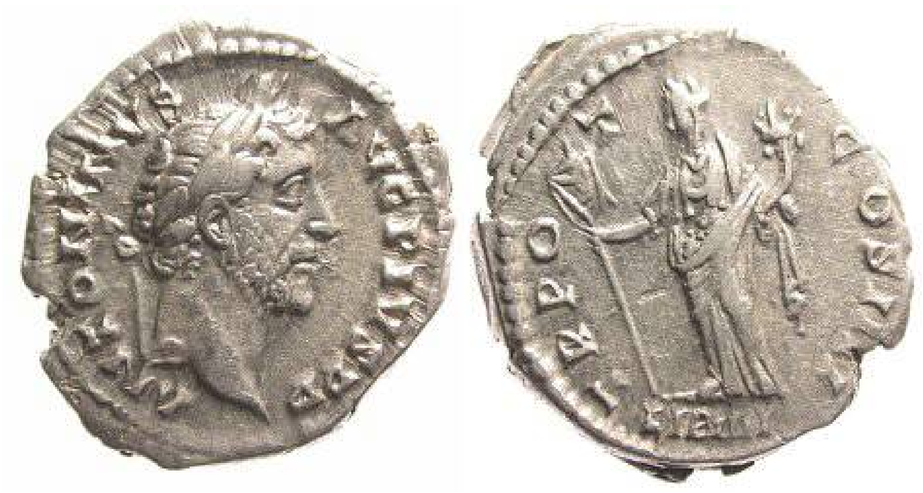


Fig. 3.1 Almost all images from the CoinProject.com website are of high quality and feature the obverse side of the coin on the left and the reverse side on the right with the correct upright rotation, a uniform white background, and a standardised perspective.

In order to extract the obverse and reverse side from the image, we used c++ and the OpenCV3 library to find the coins in the image and to cut them out. Due to the uniform white background in most coins we were able to extract the obverse and reverse image by performing a simple thresholding operation, followed by an morphological opening operation to find the contours of the two coin images. The script was designed in a way to make sure that there will be no bad samples in our data set which made it skip all the images with a non-white background or where OpenCV's *findContours* algorithm failed to locate coins with appropriate shapes, sizes, and positions. This reduced our full data set to 25'890 obverse coin images.

Certainly, there are other websites with similar coin classes and similar image quality. There are also more sophisticated methods for extracting the obverse and reverse side from a single image but we have decided to continue with this thesis and leave out further improvements for future researchers. The number of coins and classes is big enough to train a CNN. Next,

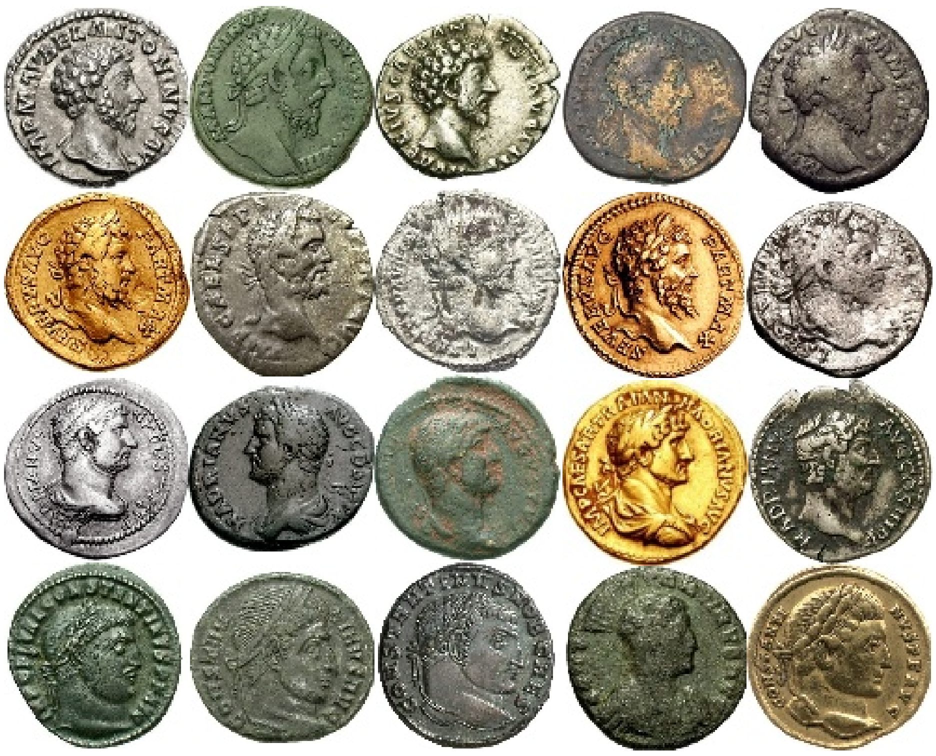


Fig. 3.2 These images are 100x100 pixels in size and are random examples from the training set. Every row depicts the same emperor. From top to bottom we have Aurelius Maximus, Septimus Severus, Hadrian, and Constantine I.

we resized all the images so that they match the input dimensions of our CNN architecture. We first filled the image into a square and made sure not to distort the coins before decreasing their size to 100×100 pixels. This smaller and uniform width and length decreases the computational complexity of the network while the coin is still identifiable by an expert and all the necessary features should be preserved. At last, we remove all classes with less than 60 images and use 30 images of every class for our test set while we use the remaining coins for the training set. This finally leaves us with 22’824 samples encompassing 20’334 train images and 2’490 test images spanning 83 classes.

3.2 Implementation

3.2.1 Framework

Training big NNs can be very computationally expensive. For this reason, many researchers and engineers use powerful GPUs to speed up their algorithms. However, programming efficiently on a GPU is not a straight forward task and requires in-depth knowledge of the hardware architecture used. Among other reasons, this has been a key motivation for the emergence of many Deep Learning or Neural Network libraries. TensorFlow, in this regard, is a rather new Open Source Machine Learning library by Google with a somewhat similar concept to other known libraries like Caffe or Torch. It was released in November 2015 and it uses a Graph made up of mathematical operations and data arrays (tensors) to create a flexible architecture which abstracts away the underlying hardware and allows the programmer to focus on the algorithms itself. We chose TensorFlow over any other library due to its clean API, extensive documentation, and huge popularity among researchers, as well as, engineers. TensorFlow provides a Python and a c++ API of which we used the former for this thesis.

Fig. 3.4 The logo of TensorFlow. A popular Open Source Machine Learning library by Google with a Python and c++ API.

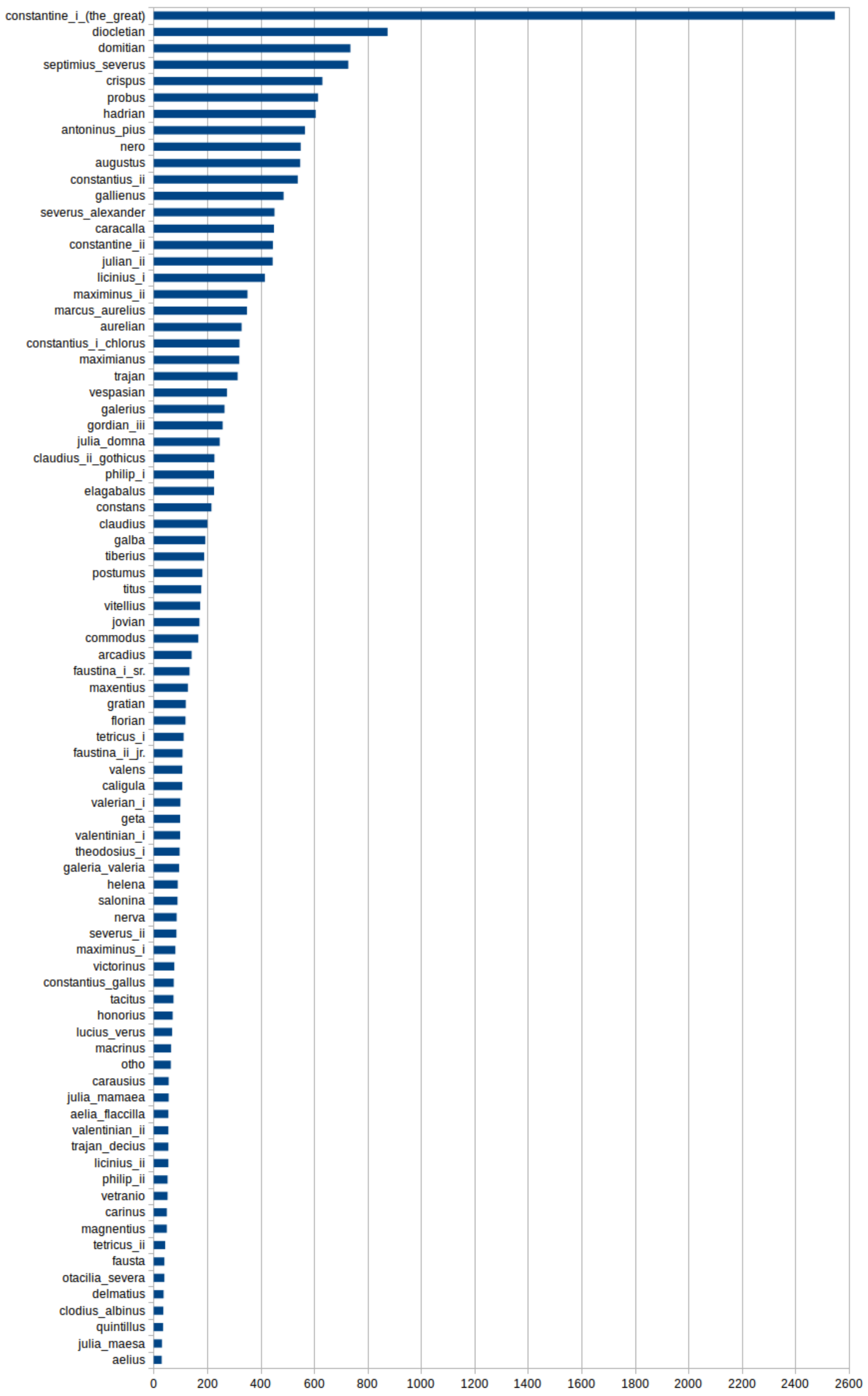


Fig. 3.3 Shows the number of samples per class in the training set. While the test set contains an even number of 30 samples per class, the training set is very unbalanced. The seven classes with most samples make up 33.14% of all coin samples.

3.2.2 Input Pipelines and Dealing with Class Imbalance

An input pipeline in TensorFlow is one way to insert data into our computational graph i.e. the CNN. Using this TensorFlow functionality, we can build threaded queues to make sure that the GPU is never idle. We have created two input pipelines which we call *unbalanced* and *uniform*. The unbalanced input pipeline randomly draws samples from all the training data. This means that the coin classes which have a much higher number of samples will be overrepresented when batches are constructed. The uniform input pipeline, on the other hand, first randomly chooses a class before it randomly chooses a sample from that class. This will balance out the average number of samples per class during training through oversampling.

Another way of dealing with class imbalances is by multiplying fixed weights to the output vector of the network before they are included in the cost function. With NNs, however, this would be a mistake as the network will over time learn to neutralise those fixed weights. The correct approach would be to weight the individual sample gradient i.e. step size based on the cost of each sample. But so far this use case has not been implemented in TensorFlow and to our knowledge is also very difficult to do in other Deep Learning libraries.

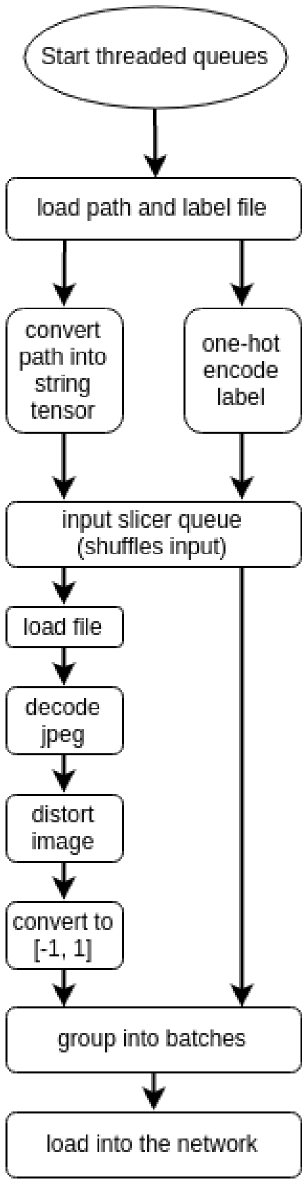


Fig. 3.5 TensorFlows input pipelines are made up of concurrent queues which can perform any type of preprocessing on the images and labels. TensorFlow makes it easy to run these operation over several CPU cores while the GPU is, at the same time, training the NN with previous batch.

3.2.3 Data Augmentation

Whenever either input pipeline reads an image it performs some image operations with randomised parameters to create a unique training sample. Theoretically, this input pipeline can produce an infinite number of training samples. The operations we perform are changes in saturation, contrast, and hue. While cropping and rotation are often performed for data augmentation, it is not necessary in this case due to the already rotational invariance of our training and test data. This would mean that the images fed into the network are expected to be upright. This is easier for the network to learn. Also, vertical and horizontal flips were not performed during data augmentation since left facing faces could be interpreted as a distinctive feature of a specific class. Random translation of the input image is relatively irrelevant in CNNs due to the translation invariance inherent in the convolutional architecture.

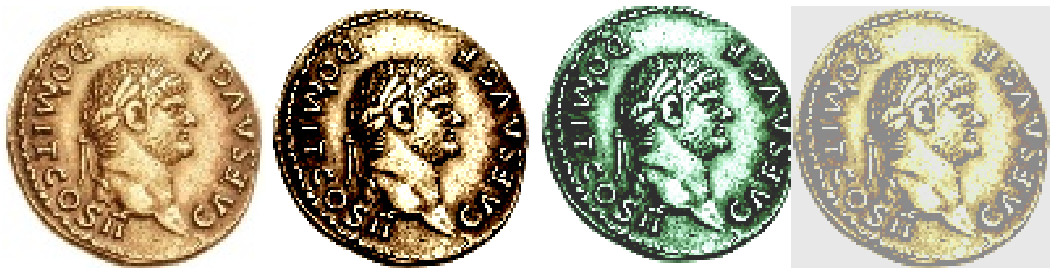


Fig. 3.6 The first coin on the left is without any data augmentation steps. The other three are random changes in saturation, contrast and hue. The emperor depicted is Domitian.

3.2.4 Network Architecture

Deep CNNs consist of different types of layers such as convolutional layers, pooling layers, fully-connected layers, etc. Each one of these layers comes with its own parameters which make up the weights and biases of the network. The parameters used specifically during the training procedure are called hyperparameters. The number of layer types with their individual parameters and the hyperparameters with their effect on training quality and speed make it difficult to choose a good performing architecture. For this reason, we rely on published architectures which performed well on different Computer Vision benchmarks.

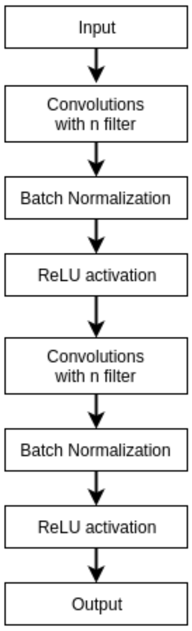


Fig. 3.7 A ConvBlock contains two consecutive sets of n convolutions, Batch Normalization, and ReLU activation. The stride and padding is set in a way that there is no dimensionality reduction. The only parameter of a ConvBlock is the number of filters for both convolutional layers.

Our architecture is inspired by the finding of Simonyan and Zisserman who presented the VGG network which was build out of very small 3×3 kernels [16]. They showed how a few small stacked kernels are superior to bigger kernels in describability and computational cost. Our network architecture is made up of what we call convolutional blocks or *ConvBlocks*. Every ConvBlock uses the same hyperparameters with the exception of the number of kernels used. Every ConvBlock is made up of two sets of convolutional layer, BN, and ReLU activation.

The final architecture is made up of five consecutive ConvBlock and max-pooling pairs. The number of filters is doubled after every pooling layer with the exception of the last layer. The output of the last pooling layer is flatten and then processed by 3 fully-connected layers of 4096 neurones followed by a soft-max output layer consisting of 83 outputs. Dropout is applied on the fully-connected layers except the output. The initial values of the weights and biases are drawn from a Gaussian distribution.

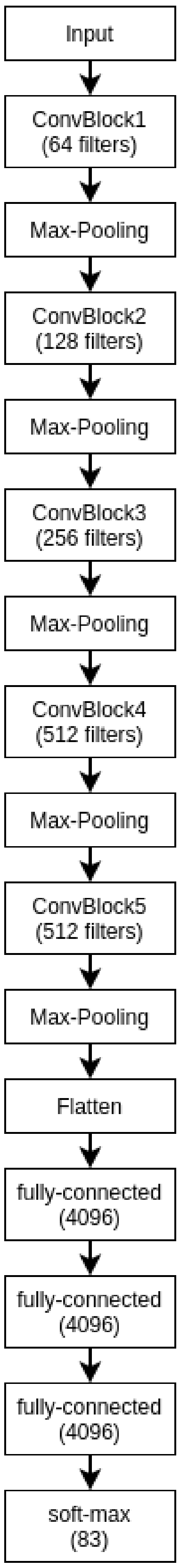


Fig. 3.8 The ConvBlock refers to figure 3.7. Overall the network consists of 14 layers with ReLU activations.

3.2.5 Complexity and Memory Requirements

This chapter breaks down the network in more detail. The convolutions we perform are all 3×3 convolutions with a stride of 1 and zero padding around the image. This leads to an equal number of dimensions in the input and output of the convolutional layer. The pooling layers use 2×2 windows with no padding around the image and a stride of 2 which results in no overlapping windows during the pooling operation. For each layer we can track the memory requirements and number of weights. As it is common with CNNs, we can observe a heavy memory usage in the shallow layers, while the deep fully-connected (FC) layers use most of the weights.

Table 3.1 The complexity and memory requirements of our 14 layer CNN. Every ConvBlock is split into its first and second layer. FC refers to the fully-connected layers. BN layers are left out for simplicity.

Layer	Dimensions	Memory	Weights
Input	[100x100x3]	100*100*3=30K	0
ConvBlock1-1	[100x100x64]	100*100*64=640K	(3*3*3)*64=1.7K
ConvBlock1-2	[100x100x64]	100*100*64=640K	(3*3*64)*64=36.8K
Max-Pool	[50x50x64]	50*50*64=160K	0
ConvBlock2-1	[50x50x128]	50*50*128=320K	(3*3*64)*128=73.7K
ConvBlock2-2	[50x50x128]	50*50*128=320K	(3*3*128)*128=147.5K
Max-Pool	[25x25x128]	25*25*128=80K	0
ConvBlock2-1	[25x25x256]	25*25*256=160K	(3*3*128)*256=294.9K
ConvBlock2-2	[25x25x256]	25*25*256=160K	(3*3*256)*256=1’769.5K
Max-Pool	[12x12x256]	12*12*256=36.9K	0
ConvBlock2-1	[12x12x512]	12*12*512=73.7K	(3*3*256)*512=1’179.7K
ConvBlock2-2	[12x12x512]	12*12*512=73.7K	(3*3*512)*512=2’359.3K
Max-Pool	[6x6x512]	6*6*512=18.4K	0
ConvBlock2-1	[6x6x512]	6*6*512=18.4K	(3*3*512)*512=2’359.3K
ConvBlock2-2	[6x6x512]	6*6*512=18.4K	(3*3*512)*512=2’359.3K
Max-Pool	[3x3x512]	3*3*512=4.6K	0
FC-1	[1x1x4096]	4096	3*3*512*4096=18’874.4K
FC-2	[1x1x4096]	4096	4096*4096=16’777.2K
FC-3	[1x1x4096]	4096	4096*4096=16’777.2K
FC-4	[1x1x83]	83	4096*83=340K
Total memory: 15.1M * 4 bytes ~= 60MB per image[1]			
Total number of parameters: ~64M			

[1]This only encompasses the forward pass. The backward pass performed during training is roughly double that.

3.2.6 Evaluation

The network is trained using the cross-entropy error. We measure the performance of the network on the train, test, and validation set. For every set, we measure the top-1 and top-3 accuracy. Top-1 accuracy is achieved if the correct class has the highest probability. Top-3 accuracy is achieved if the correct class is among the three classes with the highest probabilities.

If during training the network doesn't manage to increase the accuracy four times in a row, the learning rate is decreased by a factor of 2. Training stops after the learning has been decreased four times. The final model is the weight and parameter configuration which achieved the best top-1 accuracy on the test set. This procedure is called *early-stopping* and makes sure to not end up with a network which was trained for too long and as a result is overfitting the training data.

Chapter 4

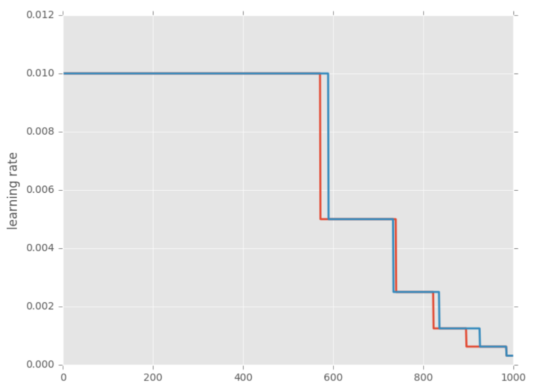
Results

We make use of a Nvidia GTX 1080 to benefit from faster training speeds and larger memory. The mini-batch size, learning rate, and the standard deviation of the initialization procedure have been chosen by a mix of grid-search and random-search. Grid search describes the selection of evenly spaced possible configurations while random-search chooses random values between an upper and lower bound. Random-search was used because it has been shown to be superior [3]. In our experiments, a mini-batch size of 128, a learning rate of 0.01 with a Gaussian standard deviation of 0.01 for the initialization procedure achieved the fastest convergence over the first four epochs. The learning rate is decreased five times by a factor of 2 whenever the test set top-1 accuracy stalls. The accuracy on all three sets is evaluated after every epoch.

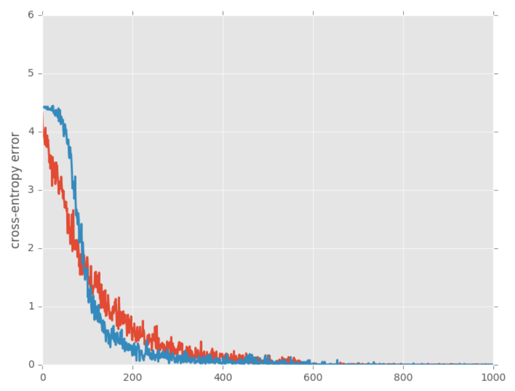
The different input pipelines had no substantial effect on the number of epochs and training time. The average number of epochs until convergence was between 34 and 57 and the average training time was between around 3h 30min and 4h 50min. However, if the network was trained on the unbalanced input pipeline it converged slightly faster than the uniform pipeline.

Table 4.1 Mean accuracy and standard deviation over five runs.

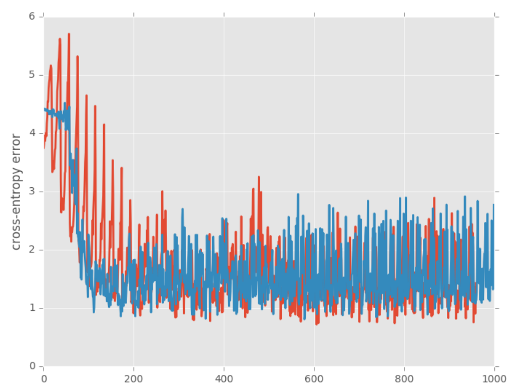
Set	Unbalanced		Uniform	
	Top-1	Top-3	Top-1	Top-3
Train	99.96% (±0.00)	100% (±0.00)	99.37% (±0.77)	99.94% (±0.09)
Test	77.04% (±1.04)	91.35% (±0.56)	75.68% (±1.75)	89.14% (±1.14)
Validation	82.53% (±2.66)	93.23% (±1.31)	79.79% (±2.80)	92.32% (±1.31)



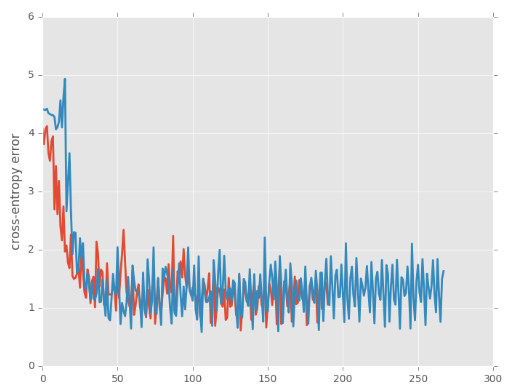
(a) learning rate decay



(b) train loss

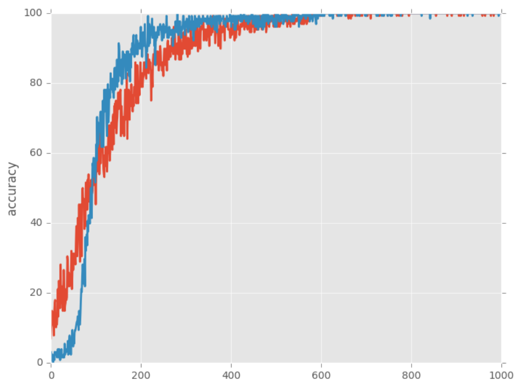


(c) test loss

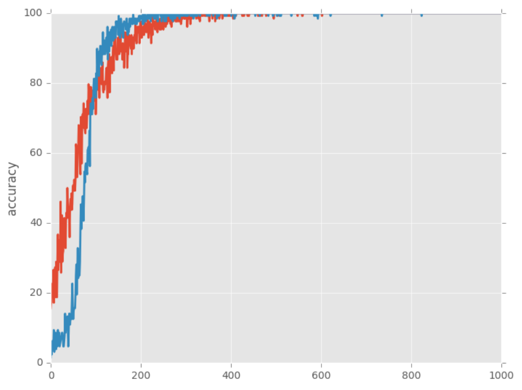


(d) validation loss

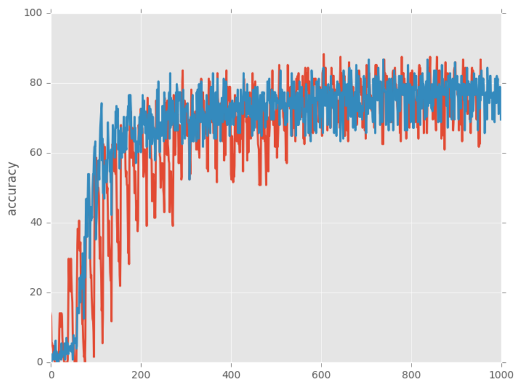
Fig. 4.1 Examples of the learning rate decay and training loss using the uniform input pipeline (blue) and the unbalanced input pipeline (red).



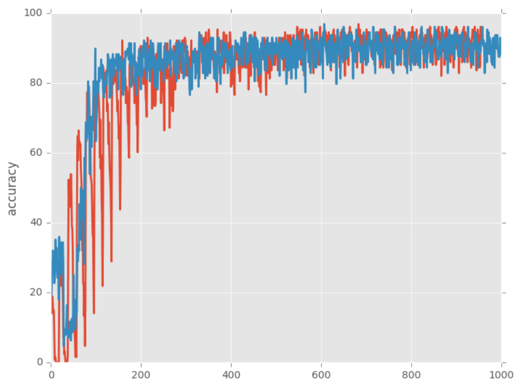
(a) train top-1 accuracy



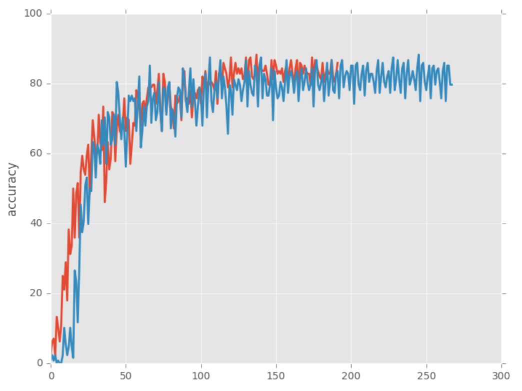
(b) train top-3 accuracy



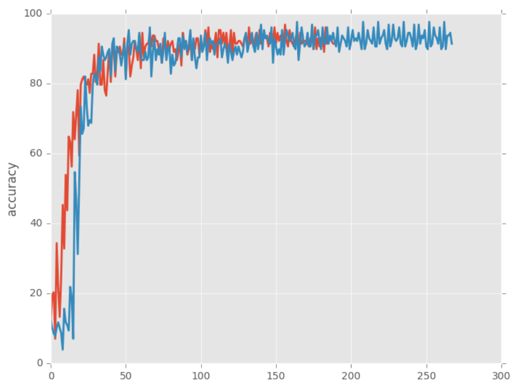
(c) test top-1 accuracy



(d) test top-3 accuracy



(e) validation top-1 accuracy



(f) validation top-3 accuracy

Fig. 4.2 Top-1 and top-3 accuracies on the train, test, and validation set. The unbalanced input pipeline is red and the uniform input pipeline is blue.

Chapter 5

Discussion

5.1 Result Analysis

The network has achieved very good results despite the very strong variety and low number of coins of some classes. The network manages to successfully classify even coins which are severely damaged or worn off. Surprisingly, the unbalanced input pipeline outperforms the more balanced uniform pipeline. The network has no problem to learn important features for every class and converges rather quickly thanks to the use of Batch Normalization and proper hyperparameter initialisation.

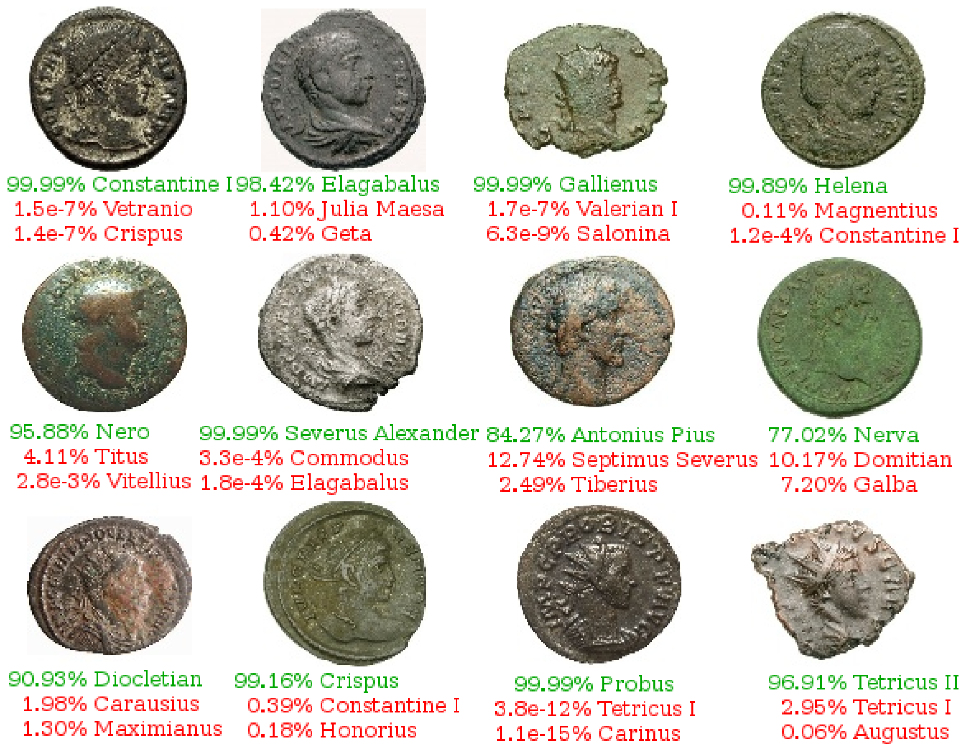


Fig. 5.1 Hand-picked coins in poor condition which the network managed to successfully classify.

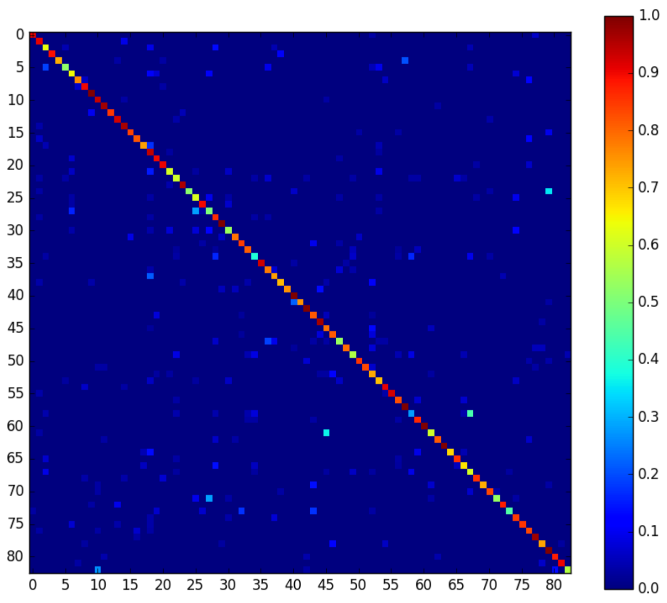


Fig. 5.2 The confusion matrix over all 83 classes of the test set.

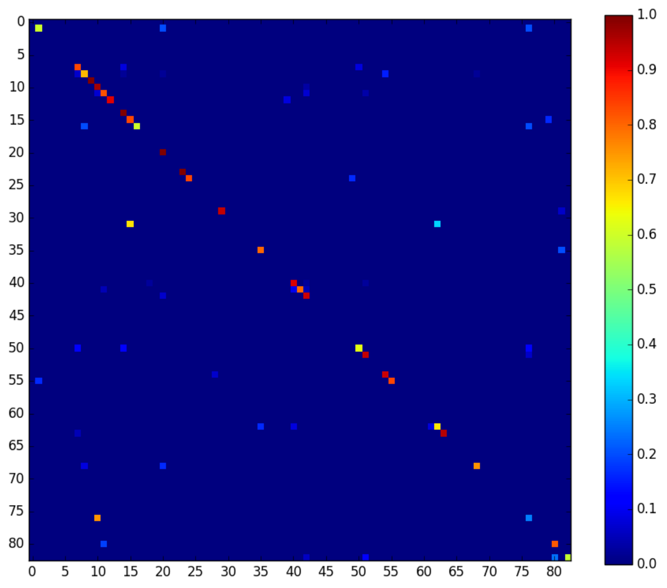


Fig. 5.3 The confusion matrix over the 44 classes of the validation set. The remaining 39 classes are set to 0.

5.2 Feature Visualisation

In order to understand the visual features important for a specific class we follow a similar technique as introduced in [19]. We can identify which regions in the image lead to a high class probability by replicating an image many times and placing an occluder at different positions in the image. The resulting discrepancy in the prediction can be traced in a heat map to show the importance of a visual feature for a specific sample. In our experiments, we noticed that in some cases there are many features used for the classification and sometimes no severe drops in certainty are visible.

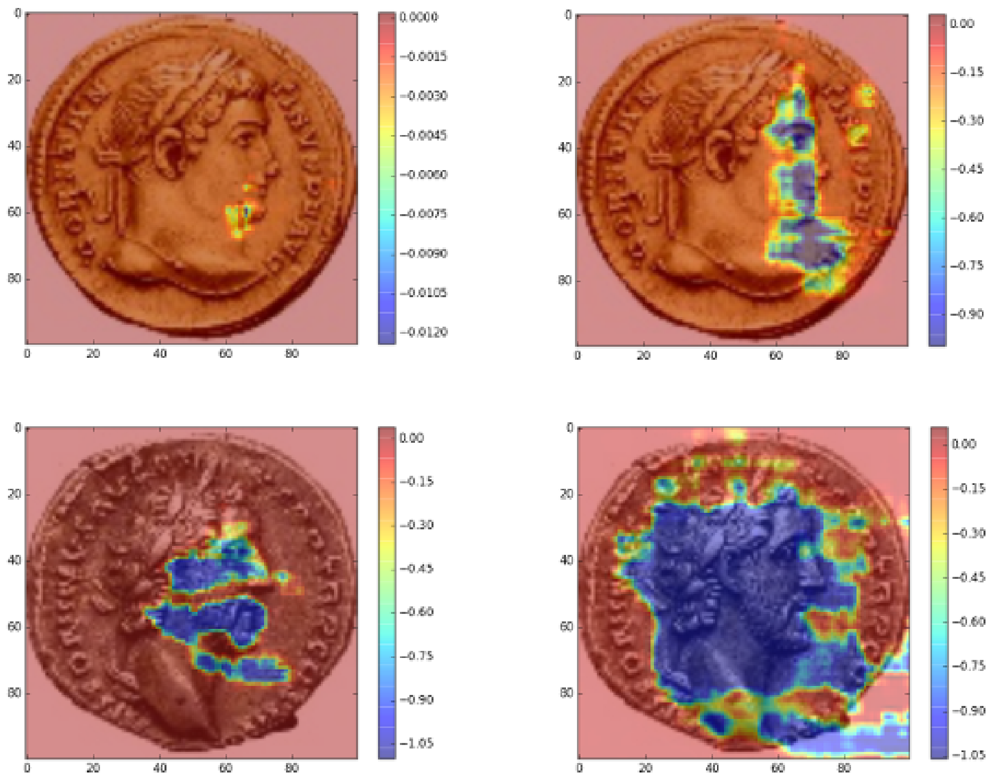


Fig. 5.4 Different occluder sizes have different effects on different samples. On the left side, we have an occluder of size 20×20. On the right side, we have an occluder of size 44×44. Every pixel in the heat map refers to a change in the class probability after placing an occluder at that specific position. The two coins in the top row depict Constantine I. This sample is very reliable to occlusion. We assume this is the case due to the high number of samples in the data. The bottom row shows the emperor Antonius Pius. Here a smaller occluder size is more insightful. Red means that an occluder placed on that spot will result in no loss in accuracy or sometimes even in higher accuracy. Blue refers to a loss in accuracy. Keep in mind that the occluder is sometimes bigger than the coloured spots and that the range of the colour space is not uniform across different images.

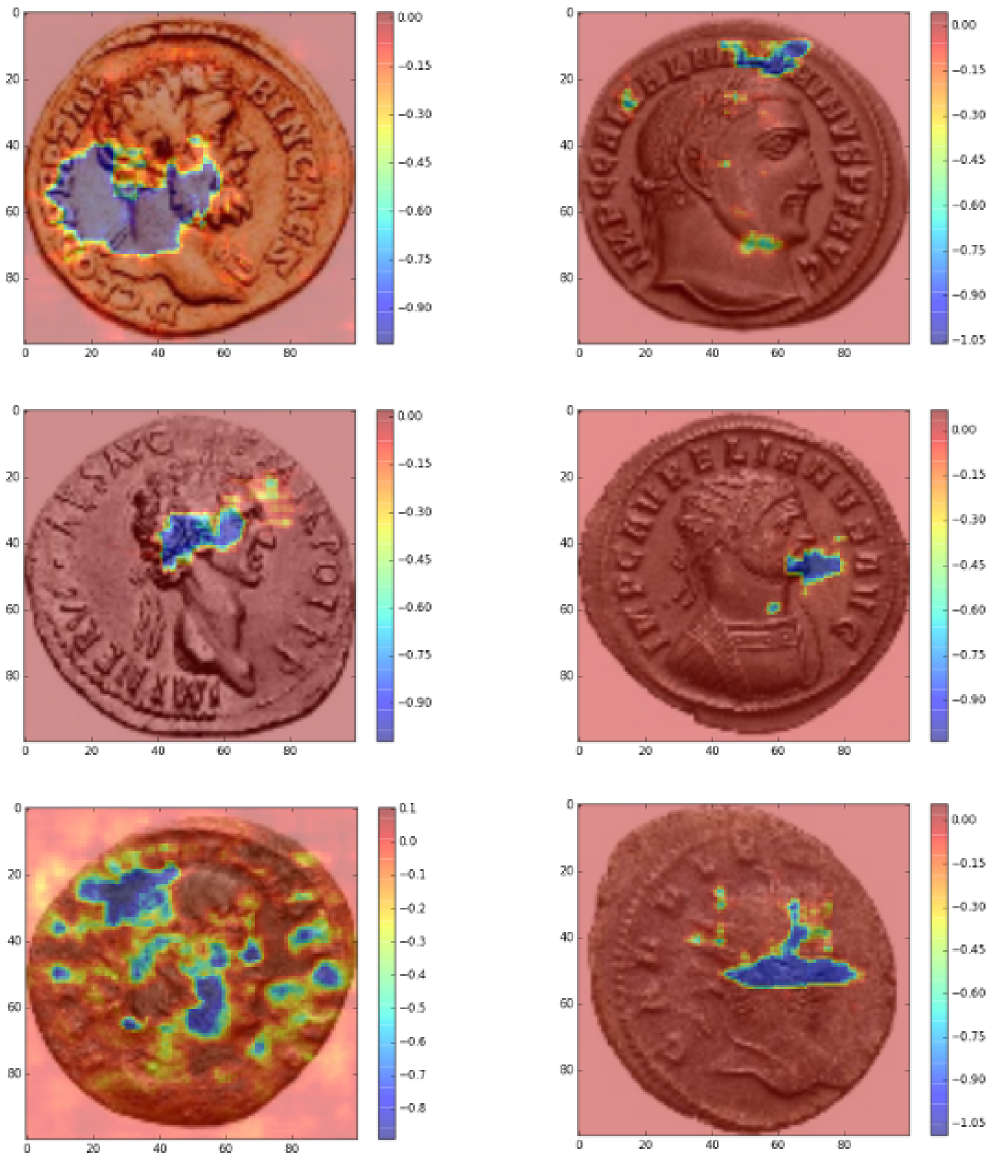


Fig. 5.5 All images used a square occluder. In the following list the size of a side of the occluder is added in brackets. From the top-left to the bottom-right we have Clodius Albinus(20), Maximinus II(24), Nerva(20), Aurelian(20), Delmatius(4), and Gallienus(20). The coin depicting Delmatius is very worn down. Even though the coin is classified correctly, it relies on many distributed features while coins in better condition seem to rely on more specific features.

5.3 Effects of Coin Grade

Using the validation set given by Prof. Arandjelović we were able to analyse the prediction accuracy based on the coin grade. We can see an average increase of over 10% top-1 and top-3 accuracy if the quality changes from fine to very fine. The differences between very fine and extremely fine do not seem to make a difference.

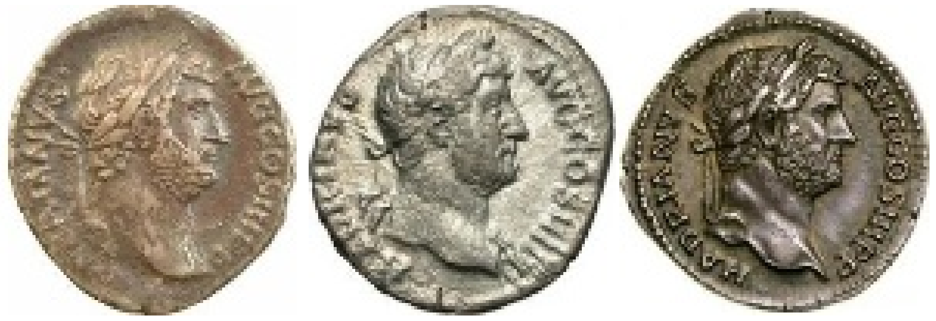


Fig. 5.6 All three images are the same coin type depicting Hadrian. From left to right the grades are fine (F), very fine (V), and extremely fine (E).

Table 5.1 The mean accuracy on the validation set over five runs.

Set	Fine	Very Fine	Extremely Fine
top-1 unbalanced	71.35% (±2.26)	87.59% (±1.78)	87.42% (±0.94)
top-1 uniform	69.71% (±2.11)	83.97% (±4.19)	84.93% (±2.72)
top-3 unbalanced	85.40% (±2.98)	97.24% (±0.00)	97.35% (±0.94)
top-3 uniform	84.12% (±2.26)	96.55% (±1.26)	95.70% (±1.27)

5.4 Face Similarity

Some classes share very distinctive features. This makes them very difficult to classify. With our CNN we can extract the second most probable class in order to group classes together. In table 5.2 we present a small subset of similar classes with the two classes which are most probable apart from the correct class prediction.

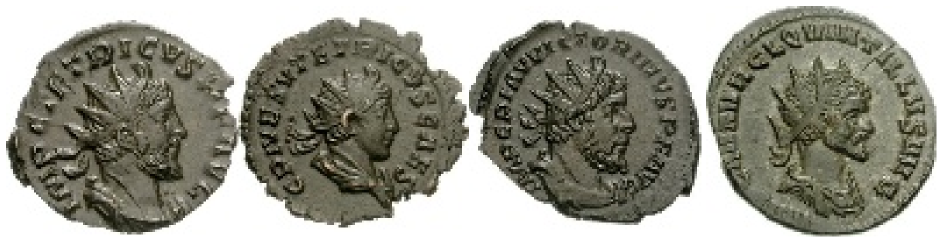


Fig. 5.7 Some coin classes share strong similarities. The emperors depicted from the left to the right is Tetricus I, Tetricus II, Victorinus, and Quintillus.

Table 5.2 The most probable alternative class predictions. This incomplete list presents some of the most similar classes according to the prediction of our network. The percentage is the proportion to be the second best choice of all the remaining 82 classes.

Target Class	2nd best guess	
Severus II	Claudius II Gothicus (54.12%)	Constantius I Chlorus (36.47%)
Julia Maesa	Galeria Valeria (74.19%)	Philip II (22.58%)
Arcadius	Honorius (59.15%)	Gallienus (24.65%)
Lucius Verus	Aelius (62.32%)	Tetricus I (21.74%)
Maximinus I	Trajan Decius (66.67%)	Philip I (11.11%)
Tetricus II	Tetricus I (67.44%)	Victorinus (11.63%)
Trajan	Tiberius (71.97%)	Caracalla (08.28%)
Victorinus	Tetricus I (62.34%)	Postumus (24.68%)
Honorius	Arcadius (73.24%)	Aelia Flaccilla (09.86%)
Constantine II	Constantius II (44.62%)	Licinius II (17.94%)
Commodus	Marcus Aurelius (32.34%)	Macrinus (20.96%)

5.5 Limitations

After a thorough evaluation of the networks ability on our own dataset, we came to the conclusion that some aspects of the data have room for improvement. For a lot of classes in our dataset, the number of samples was just below 100. During our experiments, we have found a few incorrect samples which have no emperor depicted or the possibility of a wrong classification. Furthermore, training the network is very computationally expensive and might take several hours without the use of GPUs.

5.6 Future Work

Even though the network was trained extensively on the training data, it was never overfitted. This is a sign of a good network architecture. However, the network fails to converge even more but some of the misclassification is very close to the correct class in similarity. In future work, it might interesting to add occlusion to the data augmentation step together with more data and maybe even more coin classes. This might lead to more reliable features. The fact that the network doesn't overfit is a sign of a lack of data as it would help the network to generalise better.

The field of Deep Learning is currently moving at a fast pace. Already there new ideas similar to CNN which might achieve a better generalisation on the training data while using fewer parameters. It would be interesting to compare the effectiveness of different NN approaches on this particular type of data. We also think it would be time for a big public data set with a human expert accuracy to enable a better comparison among researchers. With these promising results, we would love to see other tasks based on ancient coins such as the automatic description of the coin sides using coded English text or the automatic caption generation based on the inscription of the coin.

5.7 Conclusion

The result of this work shows that a good CNN is able to achieve very good and maybe even state-of-the-art results in ancient Roman imperial coin classification based only on the emperor depicted on the obverse of the coin. We have introduced recent advancements in the training of CNNs and successfully applied them on our own and very challenging dataset. It is noted that, in contrast to the common opinion, a big and modern CNN architecture does not seem to struggle with strong class imbalances. We believe CNNs or a similar technology will be at the core of any future professional coin classification application.

References

[1] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Kaiser, L., Kudlur, M., Levenberg, J., Man, D., Monga, R., Moore, S., Murray, D., Shlens, J., Steiner, B., Sutskever, I., Tucker, P., Vanhoucke, V., Vasudevan, V., Vinyals, O., Warden, P., Wicke, M., Yu, Y., and Zheng, X. (2015). TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. *None*, page 19.

[2] Arandjelovic, O. (2010). Automatic attribution of ancient Roman imperial coins. *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*, pages 1728–1734.

[3] Bergstra, J. and Yoshua Bengio, U. (2012). Random Search for Hyper-Parameter Optimization. *Journal of Machine Learning Research*, 13:281–305.

[4] Bottou, L. (1998). Online Algorithms and Stochastic Approximations. In *Online Learning and Neural Networks*, pages 1–34.

[5] Cybenko, G. (1989). Correction: Approximation by Superpositions of a Sigmoidal Function. *Mathematics of Control, Signals, and Systems*, 2:303–314.

[6] Fukushima, K. (1980). Neocognitron : A self-organizing neural network model for a mechanism of pattern recognition\runaffected by shift in position. *Biological Cybernetics*, 36:193–202.

[7] He, K., Zhang, X., Ren, S., and Sun, J. (2015). Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. *arXiv preprint*, pages 1–11.

[8] Hinton, G. E., Srivastava, N., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. R. (2012). Improving neural networks by preventing co-adaptation of feature detectors. *ArXiv e-prints*, pages 1–18.

[9] Hochreiter, S. (1991). *Untersuchungen zu dynamischen neuronalen Netzen*. PhD thesis, Technische Universität München.

[10] Ioffe, S. and Szegedy, C. (2015). Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *arXiv*.

[Kim and Pavlovic Rutgers] Kim, J. and Pavlovic Rutgers, V. Discovering Characteristic Landmarks on Ancient Coins using Convolutional Networks.

[12] Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). ImageNet Classification with Deep Convolutional Neural Networks. *Advances In Neural Information Processing Systems*, pages 1–9.

[13] LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient Based Learning Applied to Document Recognition. *Proceedings of the IEEE*, 86(11):2278–2324.

[14] McCulloch, W. S. and Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*, 5(4):115–133.

[15] Rosenblatt, F. (1958). The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386–408.

[16] Simonyan, K. and Zisserman, A. (2014). Very Deep Convolutional Networks for Large-Scale Image Recognition. *ArXiv*, pages 1–14.

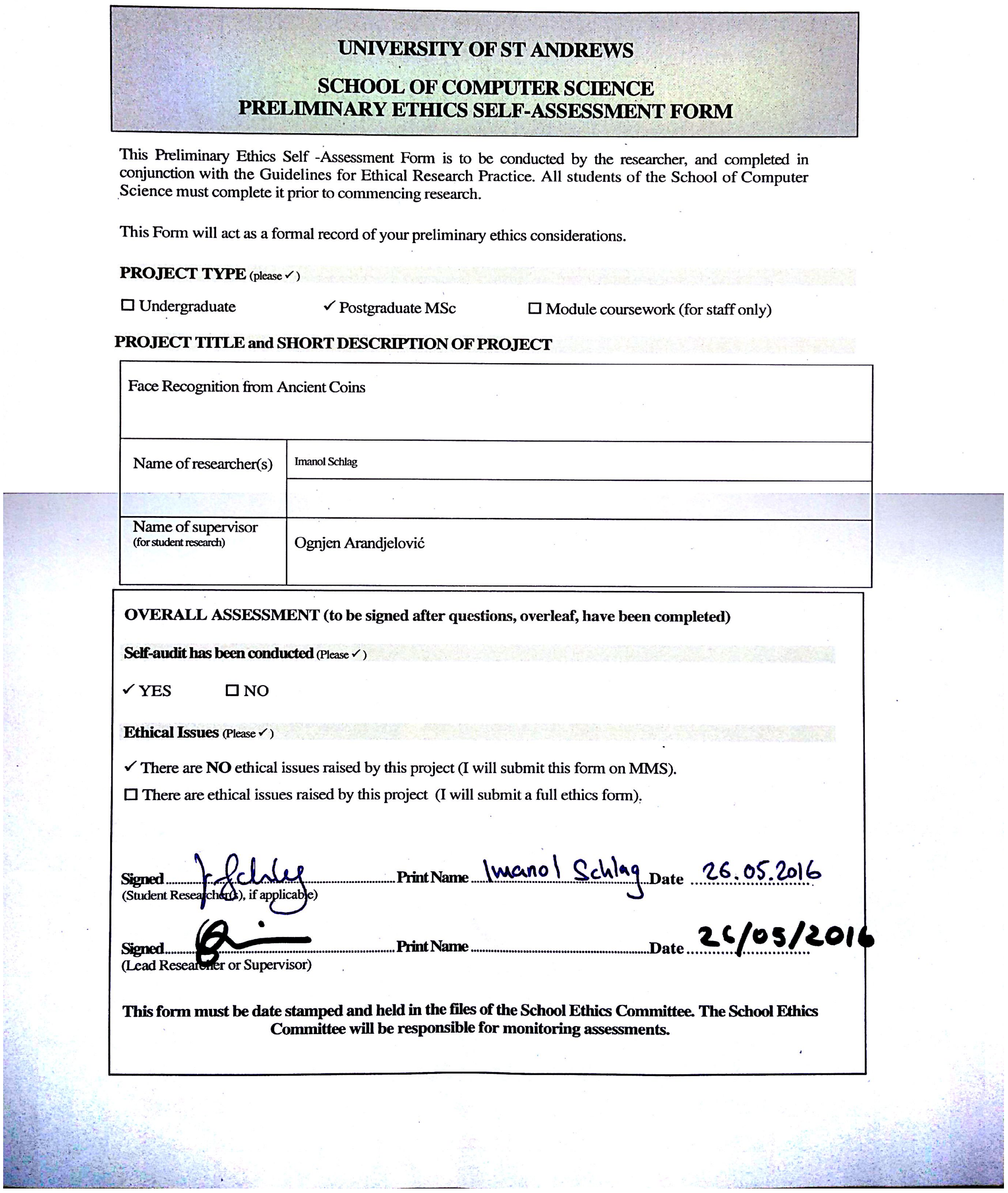
[17] Srivastava, N., Hinton, G. E., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout : A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research (JMLR)*, 15:1929–1958.

[18] Stinchcombe, M. and White, H. (1989). Universal approximation using feedforward networks with non-sigmoid hidden layer activation functions.

[19] Zhou, B., Khosla, A., Lapedriza, A., Oliva, A., and Torralba, A. (2014). Object Detectors Emerge in Deep Scene CNNs. *Arxiv*, page 12.

Appendix A

Ethical Clearance



UNIVERSITY OF ST ANDREWS

SCHOOL OF COMPUTER SCIENCE
PRELIMINARY ETHICS SELF-ASSESSMENT FORM

This Preliminary Ethics Self -Assessment Form is to be conducted by the researcher, and completed in conjunction with the Guidelines for Ethical Research Practice. All students of the School of Computer Science must complete it prior to commencing research.

This Form will act as a formal record of your preliminary ethics considerations.

PROJECT TYPE (please ✓)

☐ Undergraduate ✓ Postgraduate MSc ☐ Module coursework (for staff only)

PROJECT TITLE and SHORT DESCRIPTION OF PROJECT

Face Recognition from Ancient Coins	
Name of researcher(s)	Imanol Schlag
Name of supervisor (for student research)	Ognjen Arandjelović

OVERALL ASSESSMENT (to be signed after questions, overleaf, have been completed)

Self-audit has been conducted (Please ✓)

✓ YES ☐ NO

Ethical Issues (Please ✓)

✓ There are **NO** ethical issues raised by this project (I will submit this form on MMS).

☐ There are ethical issues raised by this project (I will submit a full ethics form).

Signed [illegible] **Print Name** Imanol Schlag **Date** 26.05.2016
(Student Researcher(s), if applicable)

Signed [illegible] **Print Name** **Date** 26/05/2016
(Lead Researcher or Supervisor)

This form must be date stamped and held in the files of the School Ethics Committee. The School Ethics Committee will be responsible for monitoring assessments.

Appendix B

List of Emperors

label, encoding	top-1 accuracy		
	train set	test set	validation set
victorinus, 0	100.00%	93.33%	
claudius, 1	100.00%	93.33%	62.50%
constans, 2	100.00%	70.00%	
otacilia_severa, 3	100.00%	90.00%	
faustina_ii_jr., 4	100.00%	76.67%	
delmatius, 5	100.00%	40.00%	
diocletian, 6	99.66%	73.33%	
vitellius, 7	100.00%	83.33%	83.33%
vespasian, 8	100.00%	73.33%	68.57%
julia_domna, 9	100.00%	100.00%	94.44%
augustus, 10	100.00%	90.00%	96.15%
marcus_aurelius, 11	100.00%	93.33%	81.82%
julia_maesa, 12	100.00%	83.33%	91.67%
helena, 13	100.00%	86.67%	
nero, 14	100.00%	96.67%	100.00%
elagabalus, 15	100.00%	83.33%	75.00%
caligula, 16	100.00%	80.00%	60.00%
vetranio, 17	98.08%	76.67%	
constantine_i, 18	99.96%	93.33%	
probus, 19	100.00%	90.00%	
domitian, 20	100.00%	93.33%	100.00%
maximinus_ii, 21	100.00%	66.67%	
gratian, 22	100.00%	56.67%	

label, encoding	top-1 accuracy		
	train	test	valid.
julia_mamaea, 23	100.00%	93.33%	100.00%
geta, 24	100.00%	40.00%	72.73%
galerius, 25	99.62%	56.67%	
florian, 26	100.00%	80.00%	
constantius_i_chlorus, 27	100.00%	60.00%	
constantius_ii, 28	100.00%	86.67%	
gordian_iii, 29	100.00%	100.00%	92.86%
maximianus, 30	100.00%	63.33%	
severus_alexander, 31	100.00%	83.33%	
tacitus, 32	100.00%	83.33%	
constantius_gallus, 33	100.00%	80.00%	
theodosius_i, 34	100.00%	36.67%	
macrinus, 35	100.00%	90.00%	80.00%
constantine_ii, 36	99.78%	66.67%	
crispus, 37	99.84%	80.00%	
salonina, 38	100.00%	70.00%	
philip_ii, 39	100.00%	70.00%	
septimius_severus, 40	100.00%	100.00%	88.00%
clodius_albinus, 41	100.00%	83.33%	80.00%
hadrian, 42	100.00%	100.00%	88.46%
aurelian, 43	100.00%	83.33%	
philip_i, 44	100.00%	96.67%	
claudius_ii_gothicus, 45	100.00%	80.00%	
valerian_i, 46	100.00%	83.33%	
licinius_ii, 47	100.00%	53.33%	
postumus, 48	100.00%	83.33%	
valentinian_ii, 49	100.00%	53.33%	
trajan, 50	100.00%	86.67%	68.42%
antoninus_pius, 51	100.00%	86.67%	95.00%
gallienus, 52	100.00%	76.67%	
licinius_i, 53	100.00%	73.33%	
galba, 54	100.00%	93.33%	92.86%
nerva, 55	100.00%	93.33%	83.33%

label, encoding	top-1 accuracy		
	train	test	valid.
valentinian_i, 56	100.00%	80.00%	
faustina_i_sr., 57	100.00%	96.67%	
honorius, 58	100.00%	33.33%	
julian_ii, 59	100.00%	86.67%	
tetricus_ii, 60	100.00%	90.00%	
quintillus, 61	100.00%	50.00%	
commodus, 62	100.00%	80.00%	70.59%
otho, 63	100.00%	93.33%	92.31%
magnentius, 64	100.00%	66.67%	
valens, 65	100.00%	83.33%	
jovian, 66	100.00%	70.00%	
arcadius, 67	100.00%	56.67%	
titus, 68	100.00%	80.00%	75.00%
carausius, 69	100.00%	76.67%	
aelia_flaccilla, 70	100.00%	83.33%	
severus_ii, 71	100.00%	50.00%	
galeria_valeria, 72	100.00%	90.00%	
carinus, 73	100.00%	40.00%	
maxentius, 74	100.00%	90.00%	
fausta, 75	100.00%	90.00%	
tiberius, 76	100.00%	76.67%	66.67%
tetricus_i, 77	100.00%	96.67%	
trajan_decius, 78	100.00%	80.00%	
caracalla, 79	100.00%	100.00%	80.95%
lucius_verus, 80	100.00%	86.67%	80.00%
maximinus_i, 81	100.00%	93.33%	
aelius, 82	100.00%	60.00%	63.64%

Appendix C

Emperor Similarities

class	most similar	second most similar
aelia_flaccilla	arcadius(23.33%)	honorius(20.00%)
aelius	lucius_verus(56.67%)	augustus(16.67%)
antoninus_pius	nerva(30.00%)	marcus_aurelius(26.67%)
arcadius	honorius(23.33%)	valentinian_ii(16.67%)
augustus	tiberius(43.33%)	faustina_i_sr.(10.00%)
aurelian	probus(26.67%)	tacitus(23.33%)
caligula	claudius(40.00%)	tiberius(20.00%)
caracalla	septimius_severus(26.67%)	trajan(26.67%)
carausius	carinus(26.67%)	aurelian(16.67%)
carinus	aurelian(20.00%)	tacitus(20.00%)
claudius	caligula(32.26%)	nero(25.81%)
claudius_ii_gothicus	gallienus(43.33%)	quintillus(33.33%)
clodius_albinus	septimius_severus(53.33%)	antoninus_pius(20.00%)
commodus	marcus_aurelius(43.33%)	lucius_verus(16.67%)
constans	constantius_ii(53.33%)	delmatius(16.67%)
constantine_i	vetranio(40.00%)	crispus(10.00%)
constantine_ii	constantius_ii(33.33%)	crispus(23.33%)
constantius_gallus	constantius_ii(30.00%)	julian_ii(20.00%)
constantius_i_chlorus	galerius(40.00%)	diocletian(26.67%)
constantius_ii	constans(26.67%)	constantine_ii(20.00%)
crispus	constantine_i_(the_great)(30.00%)	constans(20.00%)
delmatius	constans(34.48%)	constantine_i(13.79%)
diocletian	constantius_i_chlorus(33.33%)	maximianus(26.67%)
domitian	nerva(26.67%)	titus(13.33%)
elagabalus	caracalla(40.00%)	severus_alexander(36.67%)

class	most similar	second most similar
fausta	marcus_aurelius(23.33%)	faustina_i_sr.(23.33%)
faustina_i_sr.	faustina_ii_jr.(53.33%)	julia_domna(16.67%)
faustina_ii_jr.	faustina_i_sr.(76.67%)	theodosius_i(06.67%)
florian	tacitus(33.33%)	aurelian(26.67%)
galba	vespasian(40.00%)	caligula(20.00%)
galeria_valeria	helena(63.33%)	julia_maesa(23.33%)
galerius	constantius_i_chlorus(36.67%)	maximinus_ii(23.33%)
gallienus	valerian_i(36.67%)	claudius_ii_gothicus(23.33%)
geta	caracalla(60.00%)	septimius_severus(06.67%)
gordian_iii	trajan_decius(56.67%)	otacilia_severa(13.33%)
gratian	theodosius_i(16.67%)	delmatius(16.67%)
hadrian	otho(20.00%)	trajan(20.00%)
helena	galeria_valeria(40.00%)	otho(16.67%)
honorius	arcadius(50.00%)	theodosius_i(13.33%)
jovian	valens(30.00%)	valentinian_i(20.00%)
julia_domna	aelia_flaccilla(36.67%)	julia_maesa(26.67%)
julia_maesa	julia_mamaea(30.00%)	galeria_valeria(23.33%)
julia_mamaea	otacilia_severa(36.67%)	julia_maesa(30.00%)
julian_ii	jovian(26.67%)	diocletian(13.33%)
licinius_i	vetranio(26.67%)	severus_ii(20.00%)
licinius_ii	constantine_ii(50.00%)	crispus(16.67%)
lucius_verus	marcus_aurelius(33.33%)	aelius(23.33%)
macrinus	postumus(20.00%)	commodus(16.67%)
magnentius	valentinian_i(26.67%)	constantine_i(26.67%)
marcus_aurelius	commodus(46.67%)	victorinus(20.00%)
maxentius	constantine_i(33.33%)	claudius_ii_gothicus(26.67%)
maximianus	diocletian(30.00%)	galerius(26.67%)
maximinus_i	trajan_decius(43.33%)	philip_i(23.33%)
maximinus_ii	constantine_i(26.67%)	licinius_i(23.33%)
nero	constantius_gallus(30.00%)	aelius(16.67%)
nerva	claudius(53.33%)	domitian(20.00%)
otacilia_severa	salonina(33.33%)	constantius_gallus(33.33%)
otho	vitellius(53.33%)	nero(30.00%)
philip_i	philip_ii(40.00%)	trajan_decius(33.33%)
philip_ii	philip_i(43.33%)	gordian_iii(26.67%)
postumus	victorinus(60.00%)	carausius(13.33%)

class	most similar	second most similar
probus	diocletian(23.33%)	aurelian(16.67%)
quintillus	claudius_ii_gothicus(63.33%)	caligula(10.00%)
salonina	otacilia_severa(20.00%)	galeria_valeria(13.33%)
septimius_severus	clodius_albinus(30.00%)	postumus(23.33%)
severus_alexander	elagabalus(30.00%)	vitellius(20.00%)
severus_ii	constantius_i_chlorus(53.33%)	claudius_ii_gothicus(10.00%)
tacitus	florian(43.33%)	aurelian(20.00%)
tetricus_i	victorinus(60.00%)	arcadius(06.67%)
tetricus_ii	tetricus_i(63.33%)	gallienus(20.00%)
theodosius_i	constantius_ii(20.00%)	arcadius(20.00%)
tiberius	augustus(46.67%)	claudius(16.67%)
titus	domitian(40.00%)	vespasian(40.00%)
trajan	tiberius(53.33%)	hadrian(13.33%)
trajan_decius	gordian_iii(33.33%)	philip_i(16.67%)
valens	constans(23.33%)	jovian(23.33%)
valentinian_i	jovian(40.00%)	gratian(23.33%)
valentinian_ii	gratian(50.00%)	arcadius(06.67%)
valerian_i	gallienus(40.00%)	philip_i(16.67%)
vespasian	titus(43.33%)	vitellius(20.00%)
vetranio	licinius_i(33.33%)	constantine_i(26.67%)
victorinus	tetricus_i(63.33%)	postumus(26.67%)
vitellius	otho(43.33%)	vespasian(13.33%)