Comparing CNN with ordinary logistic regression of the famous MNIST dog and cat data set

Aram Salihi[1], Martine Tan[2] & Andras Filip Plaian[2]
[1]Department of Informatics,University of Oslo, N-0316 Oslo, Norway
Department of Mathematics[2], University of Oslo, N-0316 Oslo, Norway

December 19, 2019

Abstract

In this numerical study we are going to compare two machine learning methods for image classification: Logistic Regression and Convolutional Neural Networks (CNN). We will use two image data sets. The first data set is the well known MNIST data with hand-written digits, consisting of 60000 images with a resolution of 28×28 pixels. The other data set consists of 5000 images of cats and dogs, with a chosen resolution of $150 \times 150 \times 3$ pixels. The goal of this study is to show that when the number of pixels and image complexity (background noise, unfocused objects, varying contrast levels, etc.) increases, the accuracy a simple machine learning method such as logistic regression is able to achieve decreases drastically, and that CNNs are much more suited to such tasks. On the MNIST data set, logistic regression was able to achieve an accuracy score of 92.56%, while our best CNN model achieved 98.76% accuracy. On the cats and dogs data, logistic regression barely outperformed random classification, with an accuracy of 53.07%, while our best CNN model achieved an 81.65% accuracy score. We conclude that CNN tremendously outperforms logistic regression in image classification, and that logistic regression is unsuitable for complex image data.

Contents

1 Introduction

In the past years the concept machine learning has become a hot topic and become a superstar. Different machine learning algorithms is being used in both academic/scientific community and as well in the industry. The idea behind machine learning is train a network with some arbitrary data in order to make network "smarter" so it can recognize pattern, images faces, etc.. which are related to the trained data used. In the context of image recognition algorithm are being widely used now such as in new modern f.ex Iphones, etc.. where you use your face to unlock your phone.

In this numerical study we are trying to compare an ordinary logistic regression with convolutional neural network on two different dataset (MNIST and cats and dogs).

1.1 The datasets

1.1.1 MNIST dataset

The MNIST dataset (Modified National Institute of Standards and Technology database) is one of the most common datasets used for demonstrating image classification, and consists of labelled pictures of hand written digits 0 to 9. It is accessible through many different sources - we obtained it through the keras.datasets package from tensorflow. This data set has 70000 images, of which

Figure 1: A sample of the MNIST data set pictures

60000 are reserved for training and 10000 are reserved for testing. The shape of each image is only 28×28 pixels. This makes this data set ideal for testing classification algorithms, because other image data sets typically have much larger images, which can result in a massive increase in computational time.

1.1.2 Cats and Dogs dataset

We use a reduced version of Kaggle's Dogs Vs. Cats data set with 5000 images, where 3750 are for training and 1250 are for testing. Both the training and test set have an equal amount of cat pictures and dog pictures, that is, the training set has 1875 pictures of cats and 1875 pictures of dogs, while the test set has 650 pictures of each class. The data set can be downloaded from

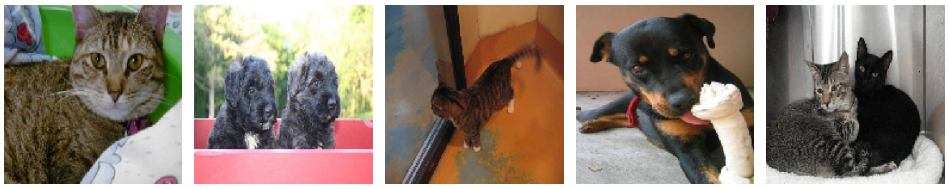


Figure 2: A sample of the cats and dogs data set pictures.

https://www.kaggle.com/c/dogs-vs-cats/data. The full-sized data set has 25000 images, but in the interest of keeping the computational time of our algorithms low, we only use 5000 images. Any machine learning model is only as good as the data it was trained upon, so using fewer images will of course reduce the performance and reliability of the model. However, as the goal of this project is not to break the kaggle record of 98.914 classification accuracy (see the leaderboard on the kaggle competition), but to compare the performance of logistic regression and convolutional neural networks, we prioritize computation speed.

The exact data which we used can be obtained by cloning our GitHub repository, and running the file "cats_and_dogs_generate_data.py". This program will select the images we used from the folder containing all the images, sort them into training and testing, and cats and dogs, and then store the images in .npy files. These .npy files are later loaded into the programs that perform logistic regression and make CNNs for this dataset. We will now go on and present the following theory for logistic regression and CNN, keep in mind, before we present CNN we would like to give a small introduction on classical neural network classifier. We will then present the results from this study.

2 Theory

2.1 Image classifications

2.2 Logistic Regression:

Logistic regression is a way of classifying an input to a correct output for a given problem for some arbitrary parameters. In other words we want to model the posterior probabilities of K classes as linear functions for a given input x and ensuring that the sum is one and remain in the interval $[0, 1]$. This form for regression is widely used when the problem has two outputs, for example true or false. For example, does the patience have diabetes given age, gender, height and weight? The answer is either yes or no. The mathematical model of this type of regression for K classes is given as

$$\log\left(\frac{\Pr(G=1|X=x)}{\Pr(G=K|X=x)}\right) = \beta_{10} + \beta_1^T x \tag{1}$$

$$\vdots$$

$$\log\left(\frac{\Pr(G=K-1|X=x)}{\Pr(G=K|X=x)}\right) = \beta_{(k-1)0} + \beta_{k-1}^T x \tag{2}$$

2.3 Cost function for logistic regression

Let $\boldsymbol{\beta} = [\beta_0, \beta_1, ..., \beta_p]$ be the vector of regression coefficients, and $X_i^* = [1, X_{i,1}, X_{i,2}, ..., X_{i,p}]$ the vector of predictor values for the i-th observation. For each pair of data points (y_i, X_i) we have the probability

$$p(y_i = 1|X_i^*\boldsymbol{\beta}) = \frac{\exp(\beta_0 + \beta_1 X_{i,1} + ... + \beta_p X_{i,p})}{1 + \exp(\beta_0 + \beta_1 X_{i,1} + ... + \beta_p X_{i,p})} = \frac{\exp(X_i^*\boldsymbol{\beta})}{1 + \exp(X_i^*\boldsymbol{\beta})}$$

and the corresponding probability

$$p(y_i = 0|X_i^*\boldsymbol{\beta}) = 1 - p(y_i = 1|X_i^*\boldsymbol{\beta}).$$

Then for our data set, $D = \{(y_i, X_i^*)\}_{i=1}^n$, we have the following probability of seeing the observed data

$$P(D|\boldsymbol{\beta}) = \prod_{i=1}^{n} \left[p(y_i = 1|X_i^*\boldsymbol{\beta})\right]^{y_i} \left[1 - p(y_i = 1|X_i^*\boldsymbol{\beta})\right]^{1-y_i}.$$

We want to maximize this probability. The log-likelihood function is then

$$
\begin{aligned}
l(\boldsymbol{\beta}) &= \sum_{i=1}^{n} \Big(y_i \log \Big[p(y_i = 1 | X_i^* \boldsymbol{\beta}) \Big] + (1 - y_i) \log \Big[1 - p(y_i = 1 | X_i^* \boldsymbol{\beta}) \Big] \Big) \\
&= \sum_{i=1}^{n} \Big(y_i \log \Big[\frac{\exp(X_i^* \boldsymbol{\beta})}{1 + \exp(X_i^* \boldsymbol{\beta})} \Big] + (1 - y_i) \log \Big[1 - \frac{\exp(X_i^* \boldsymbol{\beta})}{1 + \exp(X_i^* \boldsymbol{\beta})} \Big] \Big) \\
&= \sum_{i=1}^{n} \Big(y_i \Big(X_i^* \boldsymbol{\beta} - \log \big[1 + \exp(X_i^* \boldsymbol{\beta}) \big] \Big) + (1 - y_i) \Big(- \log \big[1 + \exp(X_i^* \boldsymbol{\beta}) \big] \Big) \Big) \\
&= \sum_{i=1}^{n} \Big(y_i X_i^* \boldsymbol{\beta} - \log \big[1 + \exp(X_i^* \boldsymbol{\beta}) \big] \Big)
\end{aligned}
$$

The cost function is just the negative log-likelihood, and thus we want to minimize this cost function

$$
C(\boldsymbol{\beta}) = - \sum_{i=1}^{n} \Big(y_i X_i^* \boldsymbol{\beta} - \log \big[1 + \exp(X_i^* \boldsymbol{\beta}) \big] \Big)
$$

. To find the optimal parameter $\boldsymbol{\beta}$ that minimizes the cost function, we will use stochastic gradient descent (SGD). For further detailed explanation of this topic, please see [Hastie et al., 2001].

2.4 Gradient/steepest descent (GD)

For a given continuous and differentialable multivariate function $F(\mathbf{x})$, there exists a set of solutions, where the function has global and local minima. The method is based on the idea that the gradient of $F(\mathbf{x})$ is where the function is decreasing the fastest in the negative direction of point $\mathbf{a}$. Thus it follows that

$$
\mathbf{a}_{n+1} = \mathbf{a}_n - \gamma \nabla F(\mathbf{a}_n) \tag{3}
$$

where $\gamma \in \mathbb{R}^m$ is a small number. With this in mind, we have that, $F(\mathbf{x}_0) \geq F(\mathbf{x}_1 \geq F(\mathbf{x}_2) \geq ...$, a decreasing monotonic sequence. The downside with this numerical method is its computational

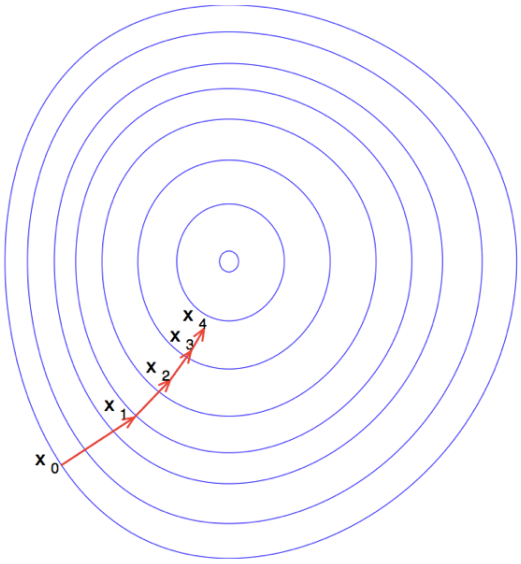


Figure 3: An illustration of how gradient descent works

complexity. In machine learning we want to minimize the gradient of the cost function by finding a solution set of $\boldsymbol{\beta}$ which achieve this. The data sets involved are often large, and thus the evaluation of the gradient can cost a lot of computational time. This decreases the efficiency of this method when evaluating large data sets, thus we want to use a variant of this method which is more optimized for such tasks. In the next section we will present the stochastic gradient descent which is an optimized version of gradient descent.

2.5 Stochastic gradient descent (SGD)

A variant of the famous gradient descent/steepest descent (GD) is the stochastic gradient decent (SGD). The main difference between steepest/gradient descent and stochastic gradient is that we divide the matrix $C(\boldsymbol{\beta})$ into mini batches of size M, where the number of vectors are n/M. We then randomly pick out one row vector from our matrix and evaluate the gradient of it. To do so we write the cost function as row vectors,

$$C(\boldsymbol{\beta}) = \sum_{i=1}^{N} \mathbf{c}(x_i, \beta_i) \tag{4}$$

. We then define a batch of vectors as B_k, where k is picked at random, thus we want to sum over the i'th index which chooses a random picked vector from B_k

$$\sum_{i \in B_k} \mathbf{c}(x_i, \beta_i) \tag{5}$$

SGD will then look like

$$\beta_{j+1} = \beta_j + \eta \nabla \sum_{i \in B_k} \mathbf{c}(x_i, \beta_i) \tag{6}$$

where η is the learning rate. To apply this gradient method, we need the derivative of the cost function. Let $\boldsymbol{p}$ be the vector containing the probabilities $p(y_i = 1 | X_i^*, \boldsymbol{\beta})$. Then we have the derivative

$$\begin{aligned}
\frac{\partial C(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}} &= -\sum_{i=1}^{n} \Big[y_i X_i^* - \frac{\exp(X_i^* \boldsymbol{\beta})}{1 + \exp(X_i^* \boldsymbol{\beta})}) X_i^* \Big] \\
&= -\sum_{i=1}^{n} X_i^* \Big(y_i - \frac{\exp(X_i^* \boldsymbol{\beta})}{1 + \exp(X_i^* \boldsymbol{\beta})} \Big) \\
&= -\sum_{i=1}^{n} X_i^* (y_i - p_i) \\
&= -X^T (\boldsymbol{y} - \boldsymbol{p})
\end{aligned}$$

We split the data into mini batches of size M, such that there are n/M mini batches B_k for $k = 1, 2, ..., n/M$. Each stochastic gradient descent step is then

$$\beta_{j+1} = \beta_j - \gamma_j \sum_{i \in B_k} -X_i^* (y_i - p_i)$$

where the k is picked at random, with equal probability, from the interval [1, n/M].

2.6 Feed Forward Neural Network, single layer perceptron

The most simple method for deep learning is the so called Feed Forward Neural Network. We will now consider three layers. One for input,one for hidden and one for output respectively.

- Input layer: Consider i inputs nodes, $i \in \{1, ..., N\}$, where each of these nodes takes one value from a data set $\mathbf{x}_i$. This node will then pass the input information to the next node (in our case the hidden node).
- Hidden layer: We only have one layer consisting of some nodes which evaluates the information from all the input nodes. This is simply done by summing over all of the input values. The quantity which is calculated is called the activation

$$z_i^1 = \sum_{j=1}^{N} w_{ij} x_i + b_i$$

, where w_{ij} is the weight to fit the data and b_j is the bias. The task of these nodes is then to activate a so called activation function $f(z_i^1)$ in order to get the processed data a_i^1.

$$a_i^1 = f^1\left(z_i^1\right) = f^1\left(\sum_{j=1}^{N} w_{ij}^1 x_i + b_i^1\right) \tag{7}$$

The activation function depends of what type neural network this is, in our case this is a classification neural network. For this classification network, the function is a sigmoid function (to avoid overflow, we have used scipy.special.expit)

$$f(z_l^1) = \frac{e^{z_l^1}}{e^{z_l^1} + 1} \tag{8}$$

For regression network, it is enough just to pass the data through the network, i.e to reproduce ordinary linear regression, but this is not enough in most cases. This will be presented in detail in section (**??**). When all of this is done, the processed data is then sent to another layer for further evaluation, in this case the output layer.

- Output layer (down below):

2.6.0.1 Classification

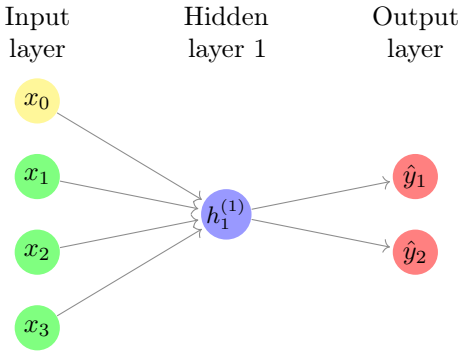
When the data is evaluated by the hidden layer, we must check the probability of which class the evaluated data point belongs to. This is given by the soft max function. Consider the a_i^1 from the hidden layer, the soft max function is

$$\sigma(a_i^1) = \frac{\exp\left(a_i^1\right)}{\sum_{i=1}^{G-1} \exp(a_i^1)} \tag{9}$$

where G is the number of categories which depends on the problem.

This can be illustrated as a figure shown down below.

2.6.0.2 Classification SLP



2.7 Multilayer perceptron FFNN

Instead of having one input, hidden and output layer. We now want to create a multilayer system with L layers. One layer for input, with N nodes, $L-2$ hidden layers with M_l nodes, and one ouput layer with 2 nodes. The idea behind this is that each neuron from the input sends the input to another neuron in the hidden layer.

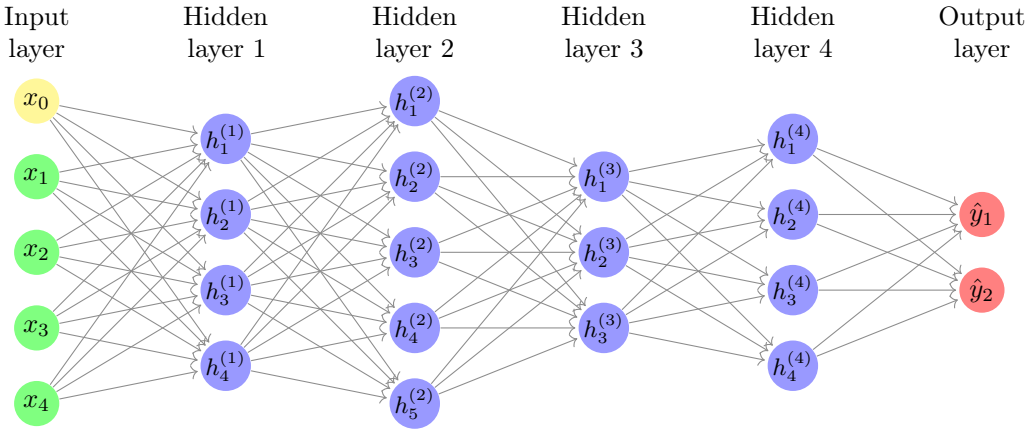
When the information comes the hidden layer, the information is evaluated, processed and sent to the next node in another hidden layer. We will denote the processes information as, $a_i^l = f^l(z_i^l)$ where $l \in \{2, ..., l-1\}$ layers, and $i \in \{1, ..., M_l\}$ nodes. Where the activation is mathematically defined as

$$a_i^l = f^l(z_i^l) = f^l\left(\sum_j^{M_{l-1}} w_{ij}^l a_j^{l-1} + b_i^l\right) \tag{10}$$

Keep in mind that this can be formulated as a matrix multiplication

$$\mathbf{a}^l = f^l(\mathbf{z}^l) = f^l\left(W^l \mathbf{a}^{l-1} + \mathbf{b}^l\right). \tag{11}$$

When the output of all the nodes in the hidden layers are computed, the values of the subsequent layer can be calculated and so forth until the output is obtained. In this layer the, we have the softmax as previous mentioned, which normalizes and sets up a probability distribution. We then back propagate in order to achieve a best fit of the weights W and the bias b. An example of mlp FFNN is illustrated down below.



This is neural network consisting of one input (7 nodes), four hidden and one output layer (2 nodes).

2.8 Backpropagation in MLP FFNN

The goal of any supervised learning is to find a function or weights which maps the best set of inputs to their correct output. In a neural network, we use backpropagation to achieve this. The idea is: when we have proceeded through each layer, and arrived at the output layer at layer L, we then wish to minimize the error by calculating the gradient to the cost/loss function $C(W)$ at layer L. From this we want to propagate backwards and compute all error and use stochastic gradient descent to update the weights W^l and bias $\mathbf{b}^l$ until an optimal solution is found. As explained, before we apply stochastic gradient descent, we need to compute the gradient of the output at layer L, consider the following equation for the error at the output level

$$\delta_j^L = f'(z_j^L) \odot \frac{\partial C}{\partial a_j^L} \qquad \boldsymbol{\delta}^L = f'(\mathbf{z}^L) \odot \frac{\partial C}{\partial \mathbf{a}^L} \tag{12}$$

This holds for any general cost function C, and activation function f. The function C can represent a loss function for binary classification or regression problem. We then compute the error for each layer $L-1 \to 2$ as follows

$$\delta_j^l = \sum_k \delta_k^{l+1} w^{l+1} kj f'(z_j^l) \qquad \boldsymbol{\delta}^l = \boldsymbol{\delta}^{l+1}(W^{l+1})^T f'(\mathbf{z}^l) \tag{13}$$

After computing the errors, we update the weight W^l and bias $\mathbf{b}^l$ for each layer $l = L-1 \to 2$ by using stochastic gradient descent as explained in section (2.5). The algorithm in matrix notation is given as

$$W^{l+1} = W^l - \eta(\mathbf{a}^{l-1})^T \boldsymbol{\delta}^l \tag{14}$$

$$b^{l+1} = b^l - \eta \boldsymbol{\delta}^l \tag{15}$$

2.8.1 Backpropagation for binary classification problem

For a binary classification problem, we are looking at a function cost function which is the same as logistic regression.

$$C(\boldsymbol{\beta}) = -\sum_{i=1}^{n} \Big(y_i X_i^* \boldsymbol{\beta} - \log\left[1 + \exp(X_i^* \boldsymbol{\beta})\right] \Big)$$

For a neural network this can be changed to

$$C(W) = -\sum_{i=1}^{n} (t_i \log\big(a_i^L\big) + (1-t_i)\log\big(1-a_i^L\big)) \tag{16}$$

Where t_i is the so called target variable, and as before a_i^L is the activation in layer L. We want to look at the following expression

$$\mathbf{z}^l = (W^l)^T \mathbf{a}^{l-1} + \mathbf{b}^l,$$

where $a_j^{l-1} = f(z_j^{l-1})$. In the case of classification, we choose the activation function f to be the sigmoid function (8). From this, the gradient of the cost function differentiated with respect to the activation a_i^L at layer L is

$$\frac{\partial C(W)}{\partial a_i^L} = \frac{a_i^L - t_i}{a_i^L(1-a_i^L)}$$

We can now use the previous expression to find a new expression for the gradient of the cost function with respect to w_{ij},

$$\frac{\partial C(W)}{\partial w_i j^L} = (a_i^L - t_i) a_j^{L-1} \tag{17}$$

and the derivative with respect to the bias is

$$\frac{\partial C(W)}{\partial b_j^L} = \delta_j^L. \tag{18}$$

The error generated in the L'th layer is given by

$$\delta_j^L = t_j - y_j^L \to \boldsymbol{\delta}^{\boldsymbol{L}} = \mathbf{t} - \mathbf{y} \tag{19}$$

For more detailed explaination about back propagation, please see [David E. Rumelhart, 1986]

2.9 Convolutional Neural Network (CNN)

In these following sections we are trying to generalize neural network presented in previous section. We are now trying to build the necessary theory in order to understand build a basic CNN network. For a in depth explanation and theory see [Goodfellow et al., 2016], for a deeper explaination and derivation of backpropagation in CNN see [Bouvrie, 2006]. It is worth mentioning that we are using the latest version of tensorflow from `https://www.tensorflow.org/`, and the theory is just simple introduction to the traditional convolutional neural network in order to give the read an idea of this network works.

2.9.1 Motivation

As we know for each layer in NN (neural network) we have to perform a matrix multiplication of the activation's $\mathbf{a}^l$ with the new weights W^l. Imagine having an image of size $M \times N \times 3$, the required operation needed are $\mathcal{O}(M \times N \times 3)$, for a high pixelated picture with three channels (RGB), the number of required operation are immense. For a $280 \times 280 \times 3$ image, we require 235200 operations. Hence the computational complexity is quadratic, $\mathcal{O}(n^2)$. Thus for large n, we wish to develop a structure which does not include matrix multiplication.

Another important scenario is shifting the pixels in the image from one position (x_1, y_1) to another position (x_2, y_2). This geometric operation is called translation. In an ordinary neural network, when this geometric operation is applied, the network interpret this image as a new image, thus we can expect a different result from the output layer. We wish to remove this type of behaviour, since a translated image of cat is still a image of a cat, thus we expect our network to classify it as a cat and not as something else. Therefore, a network which focuses on important pixels of the object we want to classify and not the total picture is required. In the following section we will develop a structure and a operator which handles these problems in a smart and efficient way.

2.9.2 Idea:

We want to develop a structure which includes the following properties:

2.9.2.1 Sparse interaction

Also referred as sparse weight. This is accomplished by making the kernel w smaller than the input (i.e an image). In this context, kernel is simply a feature map or a so called feature detector. Meaning, we can detect important features from the image and focus on those pixels rather than focus on the entire image. Example and a simple description of kernel/filters are presented in section (2.9.7). This will result in less stored parameters, which means less memory taken and the efficiency of the program increases. As previously mentioned, if we have an image of size $m \times n \times 3$ (three color channels), the required operation per layer is $m \times n \times \times 3$. Imagine that we cant detect the most important features with the image and use that for classification, meaning instead of having an image of size $m \times n \times 3$ we can reduce it to $k \times n \times 3$ where $k << m$. The required operation is thus $\mathcal{O}(k \times n \times 3)$.

2.9.2.2 Parameter sharing

In an ordinary neural network, the weights/parameters are used only once, meaning we have to use new parameters and never revisit them again. Thus, we need to redo the matrix multiplication l times, where l is the total number of hidden layers. Instead of doing this, we wish to use the same parameter more than once, we want to create a system which uses the same kernel in all position of the input source for a given layer. When this method is employed, we will achieve a more effective network since we can reduce the total number of free parameters. Consider a kernel w and image I with dimension $k \times n$ and $m \times n \times 3$. For each layer l the image has proceeded through, we will apply the kernel w with each position in the image I, in order to achieve a new result S with

trained weights. Meaning the very same kernel w is applied several times. Mathematically we will describe this process as

$$S = I * w \tag{20}$$

As we can see this new operator $*$ is not a matrix multiplication operator but an another type of linear operator which will employ parameter sharing and hence reduce the number of operation. This operator will be presented and developed in section (2.9.3). All in all, instead of learning a set of different k parameters for each neuron for every location l in our network, we only want to learn one set k parameters for every neuron in a specific layer. Thus we can manage to reduce total parameters $k \cdot l$ with a factor of l. To further optimize this process we want to introduce the concept **pooling**. This operation will be introduced and described later (section (2.9.9)).

2.9.2.3 Equivariant representation

The cause of parameter sharing fixes the problem with translation of the input I. When a function $f, g \in \mathbb{R}$ is equivariant to another function, we have the property that $f(g(x)) = g(f(x))$. Meaning if we want to translate an image (moving one pixel at position (x_1, y_1) to another position (x_2, y_2)) before the convolution we get the same result as convoluting the image and then translate the image. Another example is increasing the brightness of an image before or after the convolution operation. Consider a function h which increases the brightness and an input I. The result is

$$I' = h(I)$$

where I' has increased brightness. Then consider if we apply convolution on image I before the brightness function h is applied, the result will be the exact same if we apply convolution on I'. This can be interpret as we have a 2-D map which describes a time series of how input I changes with time, if we do translation before the convolution or vice versa, the same result will occur, but in different time in this 2-D map.

2.9.3 Convolution operator $*$

Consider two continous real or complex functions $I(\tau) \in \mathbb{R}$ and $w(\tau)$. We are now interested to express a new function $s(t)$, when taking the integral as the function $w(\tau)$ is shifted over function $I(t)$, thus we can interpret this as the function "blends with another function. More formally I and w are two function objects defined in $\mathbb{C}^n$ where both

$$\lim_{t\to\infty} |I(t)| = 0 \quad \lim_{t\to\infty} |w(t)| = 0$$

the convolved feature $s(t)$ is then defined to be the integral in the integral limit $[0, t]$ (but also $(-\infty, \infty)$

$$(I * w)(t) = \int I(\tau) w(t - \tau)\, \mathrm{d}\tau = s(t) \tag{21}$$

Where the asterisk sign $*$ is representing convolution. Having discrete points, this integral can be expressed as

$$s_n = (I * w)_n = \sum_{m=-\infty}^{\infty} I_m w_{n-m} \tag{22}$$

Keep in mind if the functions I and w lies in a finite set X, the sum can be expressed as

$$s_n = (I * w)_n = \sum_{m=-M}^{M} I_m w_{n-m} \tag{23}$$

Where the set $X \in \{-M, ..., M\}$. Another closely related operator is the so called cross correlation operator. Since these are quite often used among each other in the CNN literature we will also present the mathematical definition of this in the next section.

2.9.4 Cross Correlation operator $\star$

Consider two continuous real or complex valued functions $I(\tau) \in \mathbb{R}$ and $w(\tau)$. We are now interested to express a new function $s(t)$. From the definition of convolution in (21), and defining the conjugate of $I(t)$ as $\overline{I(t)}$, from this we can write the convolution as

$$(\overline{I(-t)} * w(t)) = \int \overline{I(-\tau)} w(t-\tau)\,\mathrm{d}\tau = (I \star w)(t) \tag{24}$$

This is equivalent to letting $\tau' = -\tau$ and $\mathrm{d}\tau' = -\mathrm{d}\tau$ be dummy variable, we can express the cross correlation as

$$(I \star w)(t) = \int \overline{I(\tau')} w(t+\tau')\,\mathrm{d}\tau' \equiv \int \overline{I(\tau)} w(t+\tau')\,\mathrm{d}\tau \tag{25}$$

If we consider our functions to be discrete functions, thus not continous, the definition of the cross correlation can simply be rewritten to

$$(I \star w)(t) = \sum_{m=-M}^{M} \overline{I_m} w_{n+m} \tag{26}$$

Note that one of the main differences are the positive sign instead of the negative sign in the function argument and indexing.

2.9.5 Discrete Fourier transformation (DFT)

Consider a complex sequence $\mathbf{x}_k$ with N of both complex and real valued numbers. The Fourier transformation transform this sequence into a another sequence $\mathbf{y}_k$ of complex numbers. The discrete transformation is defined to be

$$\mathbf{y}_k = \sum_{n=0}^{N-1} \mathbf{x}_k e^{\frac{2\pi i n}{N} k} \tag{27}$$

This transformation is quite often denoted with the symbol $\mathcal{F}$. Thus, we can write that the fourier transformation of a discrete function f is

$$\mathcal{F}\{f_k\} = g_k \tag{28}$$

2.9.6 Cross Correlation and Convolution Theorem

Consider two complex valued functions f and g which lies in the complex set. If the fourier transform of the function f and g exist, thus $\mathcal{F}\{f_k\}$ and $\mathcal{F}\{g_k\}$. Then cross correlation and convolution are identical, if only if the function f and g are even functions. Thus

$$I \star w \equiv I * w \tag{29}$$

2.9.7 Filter/ kernel

Kernel is also called a feature detector. This object simply detects important features for an image. There several different kernel which has different task. Consider l number of convolutional layer, each layer has their own kernel which has a specific task. For example, traditionally animal and human recognition starts with a kernel which detects edges.

sobel edge detector

Consider an input image $\mathbf{V}$, for sake of simplicity we will assume that the image is a grayscale image, thus the dimension is $n \times m$. For edge detecting on a image $\mathbf{V}$ the sobel operator for detecting are defined to be

$$\nabla \mathbf{F} = \begin{pmatrix} G_x \\ G_y \end{pmatrix} = \begin{pmatrix} \frac{\partial \mathbf{f}}{\partial x} \\ \frac{\partial \mathbf{f}}{\partial y} \end{pmatrix}$$

Where the sobel operator $\hat{G_x}$ and $\hat{G_y}$ are defined to be

$$\hat{G_x} = \begin{pmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{pmatrix} \quad \hat{G_x} = \begin{pmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & 0 & +1 \end{pmatrix}$$

For a in depth explanation of various intensity, edge, etc.. detection kernels see [**?**].

2.9.8 Convolutional layer in CNN

Using the mathematical background presented in the previous section, we will now try to develop a operator for a more generalized neural network. Consider an input image as a tensor $\mathbf{V}$ with dimension $(n_{inputs} \times n_{width} \times n_{height} \times 3)$. Further consider a filter/kernel as a volume of learnable neurons denoted as $\mathbf{K}$. This filter has a width, height and the same number of depth as the image, thus $M \times M \times 3$. For sake of simplicity and for visualization, We want to divide the image and the kernel into three pieces, each representing a color in RGB, thus one for red, blue and green. As the example down below We then have 6 matrices in total (3 input and 3 kernel matrices for each

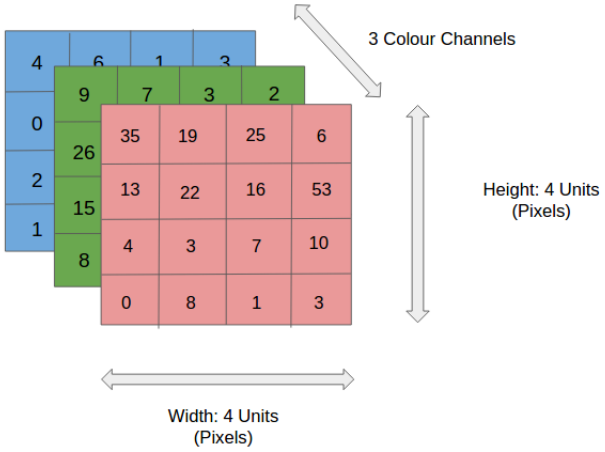


Figure 4: An image tensor $\mathbf{V}$ divided into three matrices each representing a color, red, green and blue.

image sent into our network) in each convolutional layer. We then place the center of the kernels on one of the pixels in the divided color images, and let it slide one pixel along the color images and perform a dot product. This spatial step length is called **stride** and is a hyper parameter, we will denote this parameter as s. The purpose of this parameter is to create a wider field of view. If We have an image where the same pattern occurs several times after each pixel jump, we can simply increase the stride to avoid unnecessary evaluation, but in this theory we will just an arbitrary stride s. The choice of s in this article is discussed later in the section method (referer her).

Recall the dot product, this product is simply convolution of the kernel and the image. Using

expression (23) with proper indexing and for the matrices that are divided into color matrices, we can formulate the convolution for the three channel as

$$\mathbf{Z} = \mathbf{V} * \mathbf{K} = \text{conv}(\mathbf{V}, \mathbf{K}, s) = \sum_{i=-M}^{M} \sum_{j=-M}^{M} V_{ij} K_{p-i+s,q-j+s} \tag{30}$$

Mathematically this expression tells us that kernel $\mathbf{K}$ is flipped, notice the indexes. Quite often in CNN cross correlation are used instead of convolution. This simply because of the theorem from (2.9.6). Since there exist a Fourier transform and assuming the kernel and input image are even, correlation and convolution are simply identical. Thus the result will be the same. Quite often in CNN, both cross correlation and convolution are used. This is due to the kernel are trained such that weights inside the kernel are suited best for the operation. For a general traditional CNN the 2D cross correlation with stride are defined to be

$$\mathbf{Z} = \mathbf{V} \star \mathbf{K} = \text{cross}(\mathbf{V}, \mathbf{K}, s) = \sum_{i=-M}^{M} \sum_{j=-M}^{M} V_{ij} K_{p+i+s,q+j+s} \tag{31}$$

Keep in mind that we need to evaluate every pixel in our image. If the kernel does not fit (meaning, some of the entries of the kernel are outside of the image), we can risk that the output volume can get a lower spatial dimension compared to the input volume.

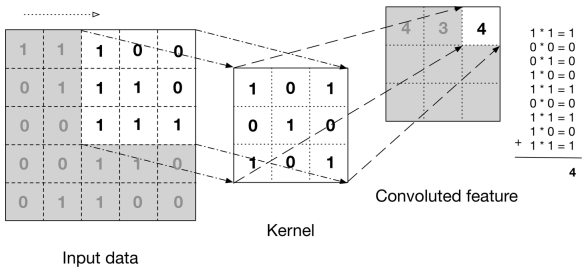


Figure 5: This diagram illustrates that the convolved feature has a lower spatial dimension compared to the input. This problem arises when the kernel cannot place it own center on all of the pixels in the image

We can solve this by using padding the input volume. It is common to use **zero padding**, thus we add new rows and columns of zeros which are necessary in order to fit the entire kernel on our image. In order to find the optimal padding, denote stride as S, spatial filter as F, input spatial

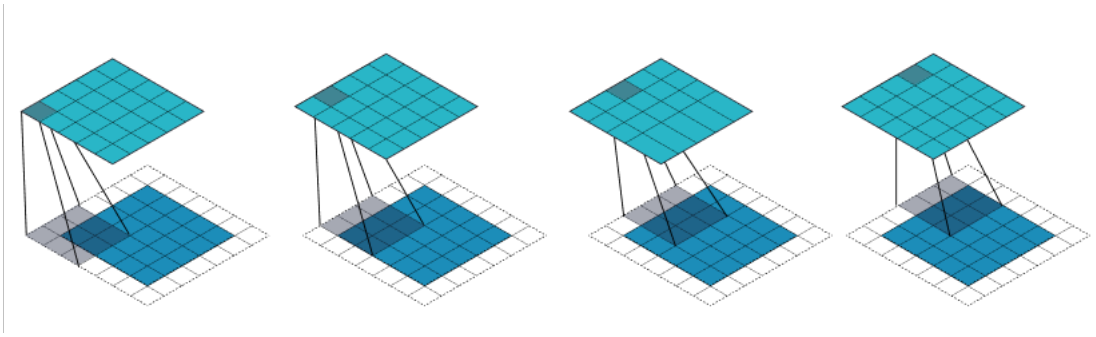


Figure 6: In this illustration we can see that we have added extra rows and columns in order to fit the kernel on our image. Thus, the resulting convolved feature map will have the same dimension as the input image

size N^i, and output spatial size as N^{i+1}, consider the following

$$N^{i+1} = \frac{N^i + 2P - F}{S} \tag{32}$$

In order to have same dimension of output and input image, we must force $N^{i+1} = N^i$ and have stride, $S = 1$. Thus, this can be expressed as

$$P = \frac{F - 1}{2} \tag{33}$$

Notice that the same kernel has been used across the image, meaning we have used the same weights on different locations. From previously it has been discussed that one of the key ideas in CNN is to employ a **weight/parameter** sharing in each convolutional layer in order to reduce computation and number of parameter in order to increase the efficiency of the network, by using the same kernel across the image. When the convolved image is sent to another layer we will use a new kernel, this process will then repeat until we reach the end, but before we discuss and present what happens in the end, we must introduce more crucial concepts and operations in order to have fully behaved and well work Network.

When each pixel has been convolved and a new feature map with same dimension as input image has been created, then this map is given to hidden neurons. Each of this neurons are then activated by using a linear rectifier, the most common rectifier function in CNN is ReLu and LeakyReLu. We will use the ReLu function as our activation function for our hidden neurons.

2.9.8.1 ReLu

For a given input signal x, the rectifier function $f(x)$ is defined to be as the positive part of its argument, thus

$$f(x) = \max\{0, x\} = x^+$$

2.9.8.2 Number of hidden neurons in the kernel volume

If each element in the kernel represent a neuron, the number of hidden neurons inside this kernel is simply defined to be N^{i+1} as we have seen, thus we can compute expression (32) in order to know how many hidden neurons we want in each volume. The image $\mathbf{V}$ is now evaluated, we then wish to send the convolved and activated map to next convolutional layer, but before we do this we most downsample this activation map by using the concept pooling. This concept is introduced in the next section.

2.9.9 Pooling

In order to non-linearly downsample a activation map, we simply use a so called pooling method. The idea is that the exact location of a feature is less important than a rough estimate of the relative location of other feature. What this results is that the spatial dimension is reduced, forcing translation invariance, number of parameters, a way of controlling overfitting and the amount of computation is also reduced. In many CNN architectures the pooling map sizes are by default 2×2, but larger sizes can also be used, but this may lead to even more low resolution convolved feature, which is not always good. There are two commonly used pooling method such as **max pooling** and **average pooling**. In this numerical study we will only consider a pooling size of 2×2

Max pooling

Consider a pooling of size 2×2. Meaning we will look at small of portion of our convolved map $\mathbf{Z}$ and take the maximum value, store the value and its location for backpropgation. Thus the

operator can mathematically be defined as: Let $\mathbf{A}$ be a small cluster of convolved map $\mathbf{Z}$, $\mathbf{A} \in \mathbf{Z}$, with dimension 2×2. Letting the pool filter slide j times with one spatial jump along the map, the resulting values are stored into a new matrix $\mathbf{Q}$

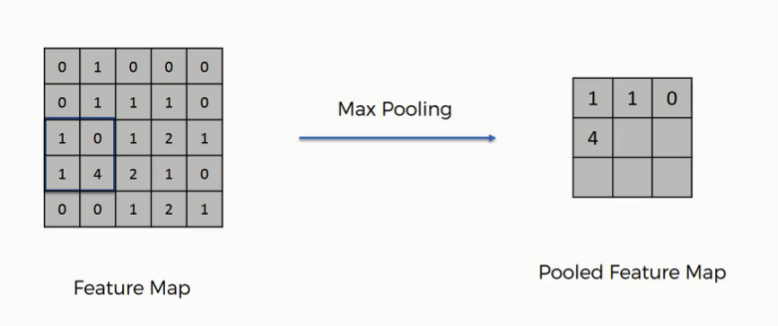


Figure 7: An illustration showing how 2×2 pooling filter. Here we a feature activated map $\mathbf{Z}$ is max pooled into another map $\mathbf{Q}$

Average pooling

Consider a pooling of size 2×2. Consider a small cluster $\mathbf{A}$ from the convolved map Z, the average pooling is simply the sum of the rows in $\mathbf{A}$ divided by the length of the pooling size. Thus, for a pooling filter size of 2×2, the average can be expressed as

$$\text{average} = \frac{a_{10} + a_{20} + a_{12} + a_{11}}{4}$$

These value are then stored into a new matrix $\mathbf{Q}$.

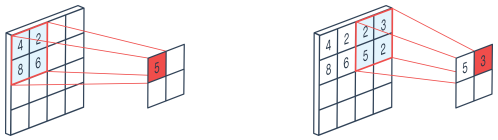


Figure 8: Here we take the average of the sum of the rows in the blue marking, the calculated value is then stored in to a new matrix $\mathbf{Q}$

2.10 The structure of a CNN network

Now as we have defined how the convolutional layer work, we wish to set up a well functioning CNN network. Consider a general $n \times m \times 3$ image, this image is then sent to the first convolution layer. After being convolved and pooled into a new data $\mathbf{Q}$, the new result can either be sent to a new convolutional layer for further convolutions or be flattend out and given a tiny bit of information to each neuron in a traditional fully connected neural network or so called fully connected **dense layer**, and activate them with a linear rectifier function, ReLu. From here they are sent to the output layer where the softmax function is applied. Consider a activated neuron with activation a^i from the dense layer, the soft max function is

$$\sigma(a_i^1) = \frac{\exp(a_i^1)}{\sum_{i=1}^{G-1} \exp(a_i^1)} \tag{34}$$

where G is the number of categories which depends on the problem. In order to minimize the for example cross-entropy function (what type of loss function is used depends heavily of the

problem, for a binary classification problem the cross-entropy is the most common to use), we must apply backpropgation. In this numerical article we will not present the theoretical framework for backpropagation in CNN, for a detailed derivation see [Bouvrie, 2006]. A diagram of how visually a CNN network looks is given down below

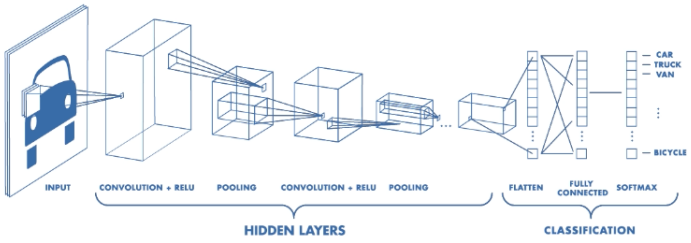


Figure 9: As we can see a CNN network may consist of several convolution layer but always ends with to two fully connected dense layer before the information is sent to the output layer where the softmax function is used.

The structure of the CNN network used in this numerical study is described in section of methods.

3 Method

3.1 CNN structure when using Tensorflow 2.0

As mentioned in the begining of the CNN theory, we are using the latest tensorflow.

3.1.1 tensorflow.keras.sequential()

This is a simple stack layer class which does CNN analysis. Before giving the image, we need to initialize a CNN object. In order to run this super class we need to add layers to a object created from sequential class. We have used the following functions:

3.1.1.1 tensorflow.keras.layers.Conv2D()

This function take an input image and send it a convolution layer which perform one convolution operation without pooling. The function Conv2D has the following input:

- **Conv2D** the following argument uses these parameters
  - **filter** Number of neurons we want to activate, this is an integer. This function filters out important features we want to investigate, this happens under convolution operation when the filter is sliding over the input image
  - **kernel size** An integer or list/tuples. This defines the height and the width of the 2D convolution window.
  - **Padding** This argument has only two possible parameters. "same" and "valid".
  - actiavtion Which activation function the neurons use when they get the input. Relu, Leakyrelu, sigmoid,etc.. If no actiaction is specified, the default is $a(x) = x$.
  - **Input shape** we have to specify the shape of the input.
- **MaxPooling2D** the following argument uses these parameters

  - **pool_size** The size of the pooling matrix. 2×2 is the default.
  - **strides** If no given, default is 1. Number of jump the pooling windows does from one place to another.

- **Flatten()**
  - No input is given. This functions flattens out the tensor into a vector.

- **Dense** the following argument uses these parameters:
  - **Units** Number of neurons, this is an integer.
  - **activation** Which activation function the neuron uses, if none parameter is given, the default is $a(x) = 0$.

A basic CNN network would like (the order cannot be changed if we want a traditional CNN network).

- define a object called f.ex "**model = sequential()**" and make an instance to the sequential class.
- perform a convolution operation by setting **model.Conv2D** with custom defined arguments and parameters (see above).
- Perform a pooling by using **model.Pooling2D**
- Flatten out the tensor by using **model.flatten()**
- Send the activated information to a fully connected hidden neural layer by using **model.Dense**
- perform another **model.Dense**
- fit the model by giving a train tensor, this done by writing history = model.fit(x_train, y_train, batch_size = batch_size, epochs = epochs, validation_data = (x_test, y_test))
- finally give a test tensor to check your result: results = model.evaluate(x_test, y_test).

Keep in mind, in order to achieve best accuracy, one has to do some grid and parameter search test to see how many convolution layer are needed, how to feed the data, etc.. In the result section we are going to present few of the hyper parameters which gives the best result, but notice there can be other approaches too since the best accuracy gained on the cats dogs data are 98.9%.

3.2 Accuracy, Mean Squared Error, and the R2s-score

We need a way to measure the performance of the trained models. For classification, this means a measure of how accurate the predictions given by the model are, for the given test data. This is checked by looking at so called accuracy score. Consider the number of samples, $n_{samples}$, the predicted variable $\hat{y}_i$, and the corresponding "true" value y_i. Then the accuracy is defined as

$$accuracy(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{n_{samples}} \sum_{i=1}^{n_{samples}} 1(\hat{y}_i = y_i) \tag{35}$$

Where $\hat{y}_i$ is the predicted value and y_i is the corresponding true value.

For regression, the performance is measured by how close the predicted values are to the true values in the test set. We do this by the mean squared error (MSE),

$$MSE(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{n_{samples}} \sum_{i=1}^{n_{samples}} (\hat{y}_i - y_i)^2. \tag{36}$$

Additionally, we use the R2-score, which is a measure of the degree to which the variance in the target variables is predictable by the variance in the independent predictor variables. The R2-score is defined as

$$R2(\mathbf{y}, \hat{\mathbf{y}}) = 1 - \frac{\sum_{i=1}^{n_{samples}} (y_i - \hat{y}_i)^2}{\sum_{i=1}^{n_{samples}} (y_i - \bar{y})^2}, \tag{37}$$

where $\bar{y} = \frac{1}{n_{samples}} \sum_{i=1}^{n_{samples}} y_i$

3.3 Confusion Matrix

Another way of measuring the performance of our machine learning classification problem is by calculating the Confusion Matrix. It is a table of four different combinations of predicted and actual values.

	Positive	Negative
Positive	True Positive	False Positive
Negative	False Negative	True Negative

- True Positive(TP): We predicted a cat and its true.
- True Negative(TN): We predicted a dog(negative of cat) and it is true.
- False Positive(FP): We predicted a cat, but it is actually a dog.
- False Negative(FN): We predicted a dog, but it is actually a cat.

It is useful for measuring recall and precision among others.

$$\text{Recall} = \frac{TP}{TP + FN} \tag{38}$$

Out of all the positive classes, how much we predicted correctly. It should be high as possible.

$$\text{Precision} = \frac{TP}{TP + FP} \tag{39}$$

Out of all the positive classes we have predicted correctly, how many are actually positive.

4 Results

4.1 Classification of the MNIST numbers using Logistic Regression

We have used SciKit learn's LogisticRegression class. In addition to the data preprocessing described in section **??**, each 28×28 pixel image was flattened to one vector of length 784. In other words, the MNIST data consists of $n = 60000$ observations (images), with 784 predictors (the pixels). Each observation has a corresponding label, indicating which digit $0, 1, 2, ..., 9$ the hand-drawn image is depicting.

SciKit learn's LogisticRegression class can apply different algorithms to solve the optimization problem. We chose to use the *lbfgs* algorithm, as it supports multinomial loss and proved to be most efficient. Lastly, it is worth noting that the Logistic Regression class adds a constant to the decision function by default. In other words, the class automatically adds an intercept-variable to the model, along with the other predictors. This means we do not need to include a vector of 1's in the first column of the so-called design matrix, which holds the predictor data (i.e. the image).

The results we obtained with Logistic Regression are summarized in the table below.

	Train	Test
Acc	0.9393	0.9256
Loss	2.0982	2.5697

The computation time of creating a Logistic Regression instance, fitting the model to the training data, and make predictions on both the test data and the training data was 119s.

An accuracy of 92.56% on the test data seems good. The confusion matrix in figure 10 gives us a more complete picture of the performance of the logistic regression model. We chose to show percentages instead of image count in the confusion matrix, as it gives a better sense of the scale of the number of correctly and incorrectly predicted images. The confusion matrix shows that

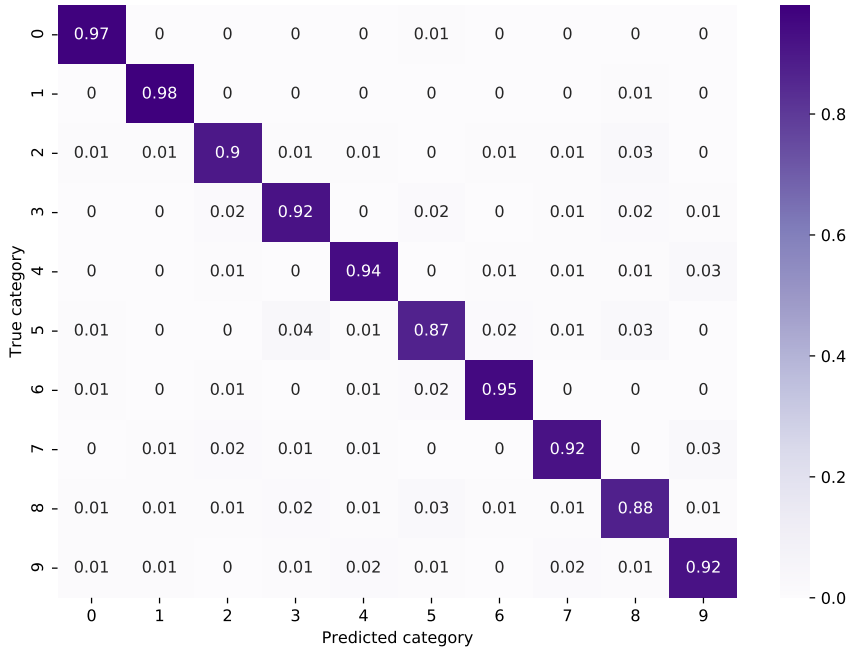


Figure 10: Confusion matrix for the predictions made on the MNIST test data using logistic regression

the digits 5 and 8 were more often misclassified than the other numbers. Meanwhile, 1 and 0 were seemingly the "easiest" for the model to classify. It is interesting to note that 5 was mostly missclassified as 3 or 8, while 8 was missclassified at least once as one of all the other digits, but more often as either 5 or 3. Figure 11 shows 10 of the missclassified images.

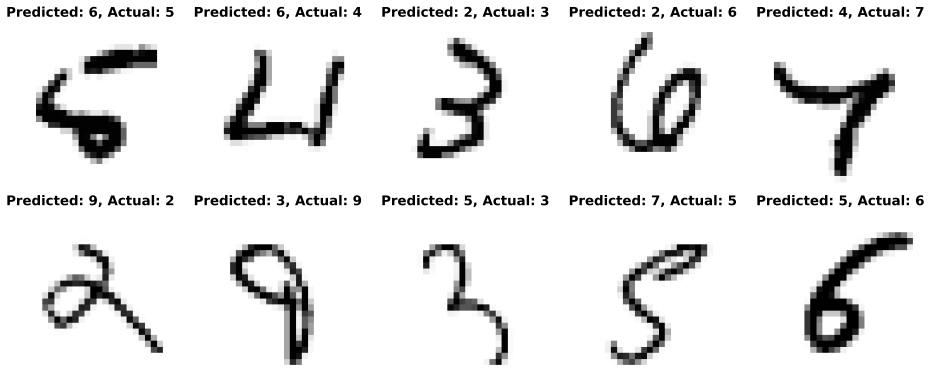


Figure 11: A sample of the MNIST images that were missclassified by logistic regression

4.2 Classification of the MNIST numbers using CNN

4.2.1 Constructing a simple CNN

We begin by constructing a CNN model with some random choice of parameters. Later, we will investigate how the choice of batch size and the number of epochs affects the model performance, as well as the form of the CNN itself. A summary of the model used can be seen in figure 12. We

Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 28, 28, 28)	56
max_pooling2d (MaxPooling2D)	(None, 14, 14, 28)	0
conv2d_1 (Conv2D)	(None, 14, 14, 32)	928
max_pooling2d_1 (MaxPooling2	(None, 7, 7, 32)	0
conv2d_2 (Conv2D)	(None, 7, 7, 64)	2112
max_pooling2d_2 (MaxPooling2	(None, 3, 3, 64)	0
flatten (Flatten)	(None, 576)	0
dense (Dense)	(None, 512)	295424
dense_1 (Dense)	(None, 10)	5130

Total params: 303,650
Trainable params: 303,650
Non-trainable params: 0

Figure 12

used a batch size of 128 and 30 epochs. This model obtained an 0.8533 accuracy score, and a 0.3505 loss. Figure 13 shows how the the accuracy and loss improves for each epoch as the model is trained. We want to investigate how we can improve the performance of CNN.

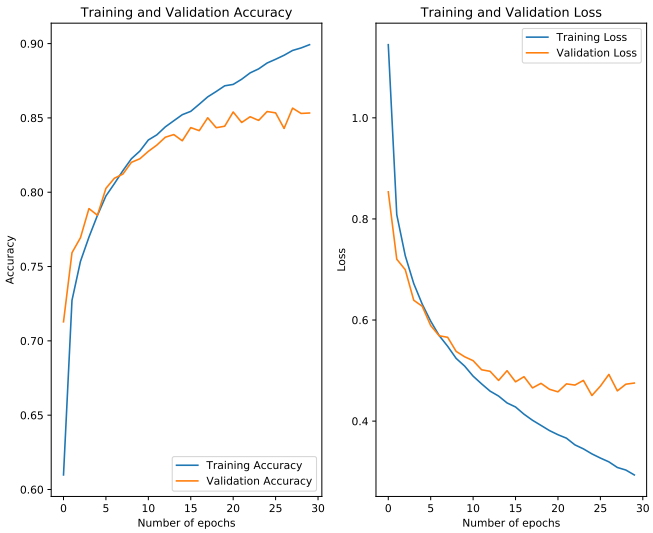


Figure 13

4.2.2 Choice of batch size and epochs

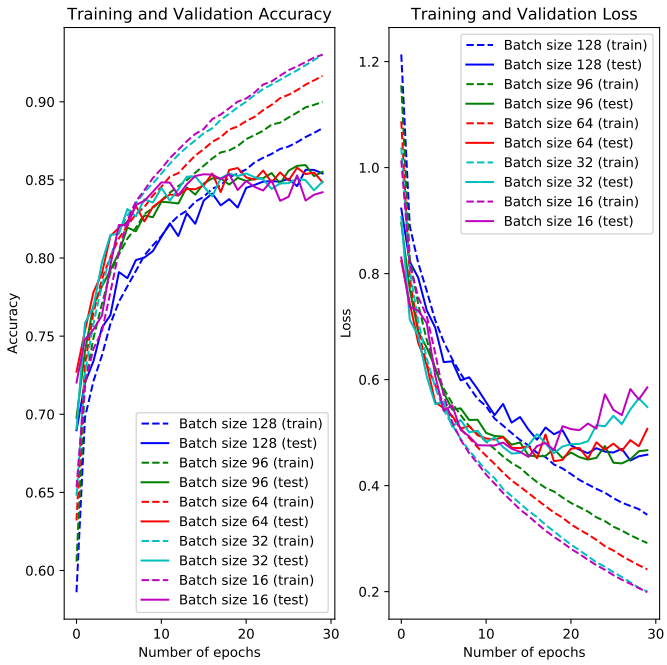


Figure 14: Accuracy score and loss of a CNN trained with varying batch sizes

To improve the previous model, we take a look at the effect of the choice of batch size and epochs. The batch size affects the optimizer, which in our case is the "adam" optimizer. From

figure 14, we see that a smaller batch size makes the accuracy score increase faster, and the loss decrease faster. In other words, a smaller batch size requires fewer epochs before the accuracy score and loss converges. We see that while the scores on the training set keep improving with the number of epochs, the scores on the test set reach a peak accuracy score of about 0.85, and a loss of about 0.48. It is worth to note that for the smaller batch sizes, the loss begins increasing again after a certain number of epochs, which is a sign of overfitting.

4.2.3 The number of convolution layers in the CNN

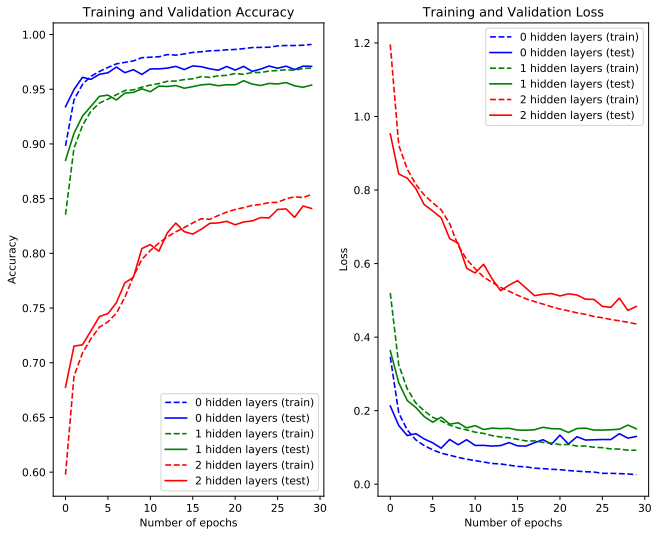


Figure 15: Performance of CNNs with increased number of hidden convolutional (and pooling) layers

How many convolution layers are necessary? We compared the performance of CNNs with an increasing number of pairs of hidden convolutional and pooling layers. Each model used a batch size of 64 and 30 epochs. The input layer in each model consisted of a convolution layer with 32 neurons, followed by a max-pooling layer. After adding the hidden convolution and pooling layers (0, 1, and 2 pairs respectively, in the models we compared), the network was concluded with a flattening layer, a dense layer with 64 neurons, and the output layer with 10 neurons (one for each digit category) and the softmax function as activation. Figure 15 shows that the simplest model with no more convolution layers than the input layer, performed significantly better than the models with added convolution layers. This suggests that our original model which obtained ac accuracy of 0.8533 is too deep, and a simpler model is sufficient to classify this dataset.

4.2.4 The final CNN model used to classify the MNIST numbers

Based on the previous sections, we now create a new CNN model with fewer layers, and a lower batch size and number of epochs. The model is was initialized with the code below, and a summary of the model can be seen in the table following the code.

```
model = tf.keras.Sequential([
        Conv2D(28, kernel_size = (3,3), padding='same', activation='relu',
            input_shape=image_shape),
        MaxPooling2D(pool_size=(2,2)),
        Flatten(),
```

```
        Dense(128, activation='relu'),
        Dense(10, activation='softmax')
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 28, 28, 28)        280
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 14, 14, 28)        0
_________________________________________________________________
flatten (Flatten)            (None, 5488)              0
_________________________________________________________________
dense (Dense)                (None, 128)               702592
_________________________________________________________________
dense_1 (Dense)              (None, 10)                1290
=================================================================
Total params: 704,162
Trainable params: 704,162
Non-trainable params: 0
```

This model obtained an accuracy of 0.9876, and had a loss of 0.0325. Figure 16 shows the model performance for each epoch, and the confusion matrix in figure 17 shows the percentages of of correctly and incorrectly classified images.

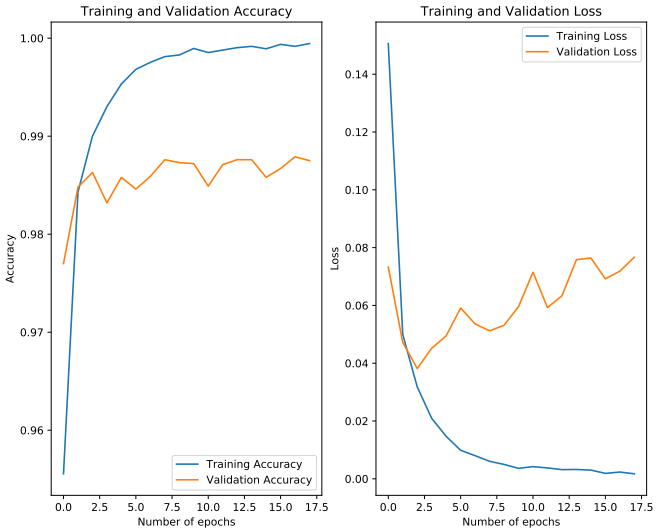


Figure 16: The accuracy score and loss of the revised CNN model

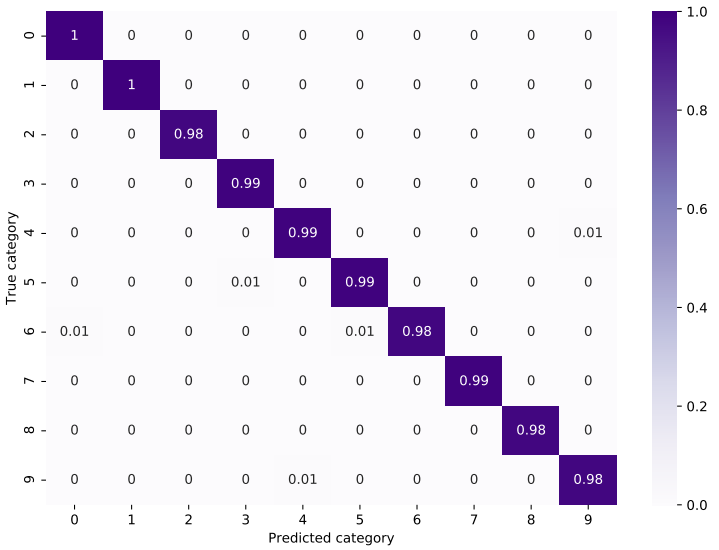


Figure 17: The confusion matrix for the predictions on the MNIST test data using the revised CNN model.

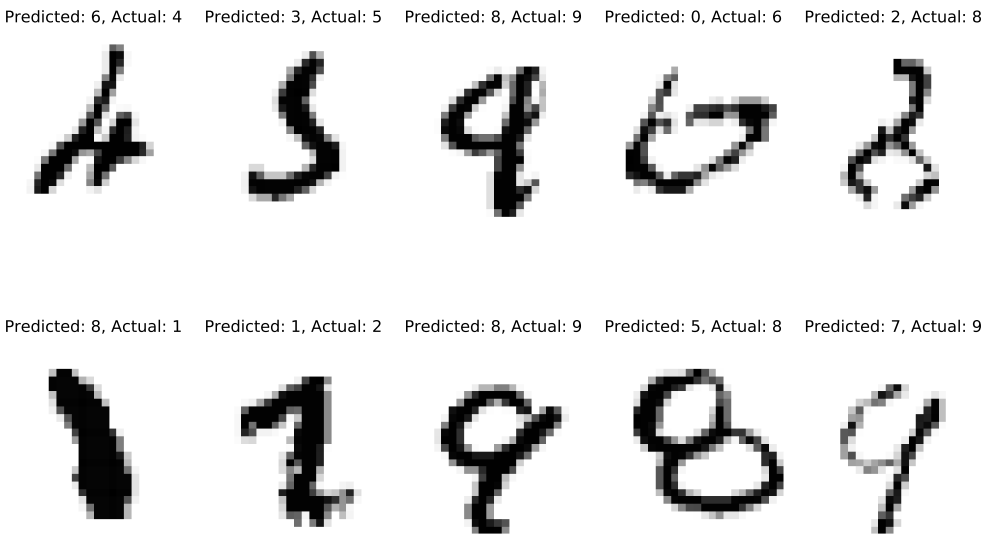


Figure 18: A sample of the images that were misclassified by the CNN

4.3 Classification of Cats and Dogs using Logistic Regression

We will once again use SciKit learn's LogisticRegression class. In addition to the data preprocessing described in section **??**, we will need to flatten the images into vectors, because LogisticRegression does not handle the x-inputs as matrices. Each image is of the shape $150 \times 150 \times 3$, where the last 3 index comes from the fact that the images are RGB format, and there are thus three layers in each image. We will first convert the images to grayscale, so they are 150 *times*150 pixels, and then flatten the images to $150 \times 150 = 2250$ vectors. Thus the cats and dogs data consists of $n = 5000$ observations (images), with 22500 predictors (the pixels). Each observation has a corresponding

label, indicating whether the image depicts a cat or a dog.

The results we obtained are summarized in the table below. The computation time of creating the LogisticRegression instance, fitting the model to the training data, and making predictions on both the test and training data was 173s.

	Train	Test
Acc	1.0000	0.5307
Loss	0.0000	16.2102

An accuracy of 53.07% on the test set is disappointing - it is barely above random classification.

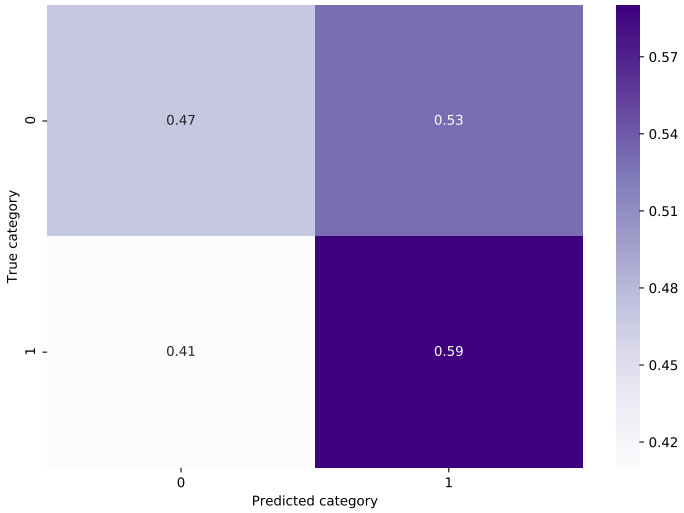


Figure 19: Confusion matrix for the test images, classified with logistic regression.

4.4 Classification of Cats and Dogs using CNNs

4.4.1 Fitting a simple CNN to the cats and dogs data

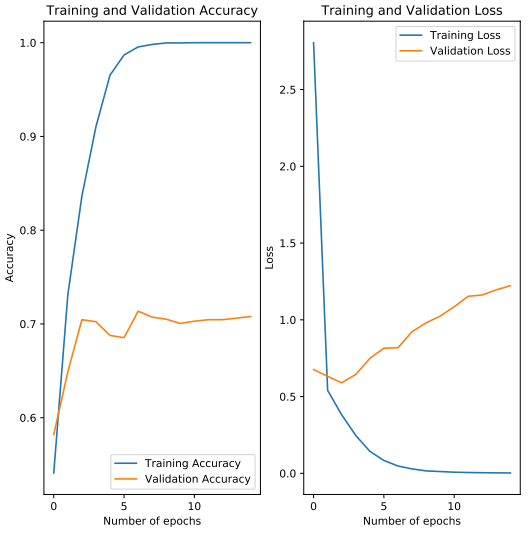
We once again begin by constructing a simple CNN model with some choice parameters. This time, our starting model is very simple, and consists of one convolution layer with 32 neurons as the input layer, a max pooling layer, a flatting layer to flat out the images, a dense layer with 128 neurons, and lastly an output layer with 1 neuron and the Sigmoid function as activation function. Note that for this dataset, we use the Sigmoid function and not the softmax function, because the classification is binary - there are only two classes the images can be classified to. A summary of the model can be seen in figure 20. Figure 21a shows the results for 15 epochs. The minimum loss was 0.6403, and the corresponding accuracy 0.7080. The confusion matrix can be seen in figure 21b.

Model: "sequential"

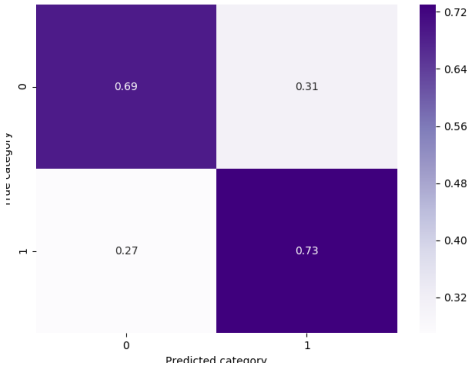
Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 150, 150, 32)	896
max_pooling2d (MaxPooling2D)	(None, 75, 75, 32)	0
flatten (Flatten)	(None, 180000)	0
dense (Dense)	(None, 128)	23040128
dense_1 (Dense)	(None, 1)	129

Total params: 23,041,153
Trainable params: 23,041,153
Non-trainable params: 0

Figure 20



(a) Acuuracy score and loss on the cats and dogs data with a simple CNN



(b) The confusion matrix of the test data evaluated with the final model

4.4.2 Choosing the number of convolution layers in the CNN

We once again are faced with the problem of deciding the best structure of our network. Theoretically, there are infinitely many ways one can define the network structure. It seems a good idea to start with deciding how many pairs of convolutional and max-pooling layers the network should have. We do this by keeping the other parameters (batch size, the number of epochs, and the number of nodes in each layer) constant, and then evaluating the performance of models with an increasing number of hidden convolutional and pooling layers. The network starts with an input convolutional layer with 32 nodes and kernel size $(3, 3)$, followed by a max-pooling layer. From there on, $l = 0, ..., 5$ pairs of "hidden" convolutional layers and max-pooling layers are added to the model, with the same number of neurons and kernel size as the input layer. All networks

are concluded with a flattening-layer to flatten the images, a normal layer with 128 neurons, and finally the output layer with only one neuron. The activation function is as before ReLu, except for the output layer which uses the Sigmoid-function. From figure 22, we see that the networks

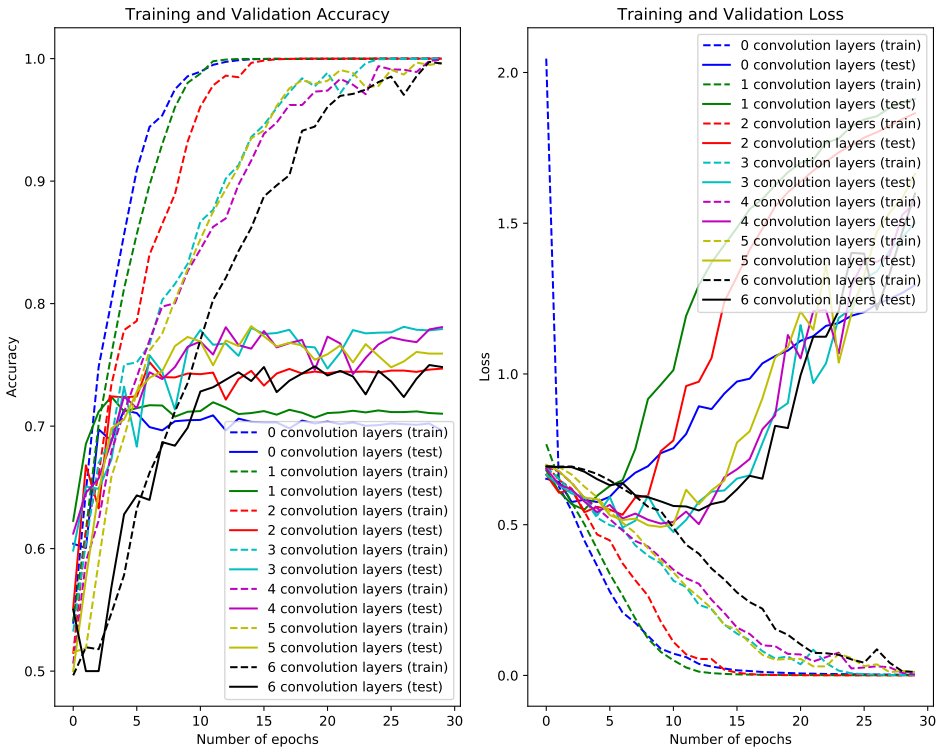


Figure 22: The accuracy score and loss of CNN models with an inclearing number of pairs of convolution and max pooling layers.

with 3, 4, and 5 convolution layers a better accuracy score. The highest accuracy obtained on the test images was 0.7815 with 5 hidden convolution layers, however we see that the deeper networks require more iterations (epochs) before the training accuracy converges to 1. The minimum test loss of 0.4757 was obtained with 3 hidden convolution layers. For loss, we also see that the deeper networks require more iterations before converging to 0. (The minimum train loss was 0.0002069, for the CNN with 2 hidden convolution layers). Note that in this case, using 7 convolution layers is not possible, as the image cannot be "compressed" any further. Some more tests for CNNs with 3, 4 and 5 convolution layers yielded mixed answers as to which performs better. This indicates that there is not much difference in performance between CNNs with 4 or 5 hidden convolution layers, and that there is a certain variability in the results. For simplicity, we choose to work with CNNs with 5 hidden pairs of convolution and max-pooling layers from now on, because that is the model that achieved the highest accuracy score in our initial test.

4.4.3 Choice of batch size and number of epochs

In the previous section, we managed to improve the accuracy score on the test set from around 70.80% to 78.15%. We now want to see of we can further improve the accuracy score on the test set by choosing a more suitable batch size than 64, which is what we started with. We did a similar test as the ones done before, where we construct CNNs of the same structure, but which use different batch sizes, and compare their performance. The structure of the model is the same as the model which performed the best in the previous section, that is, one with 5 pairs of convolution and max-pooling layers, a flattening layer, a dense layer with 128 neurons, and an output layer

with 1 neuron and the Sigmoid function as activation function. In figure 23 we see that the batch

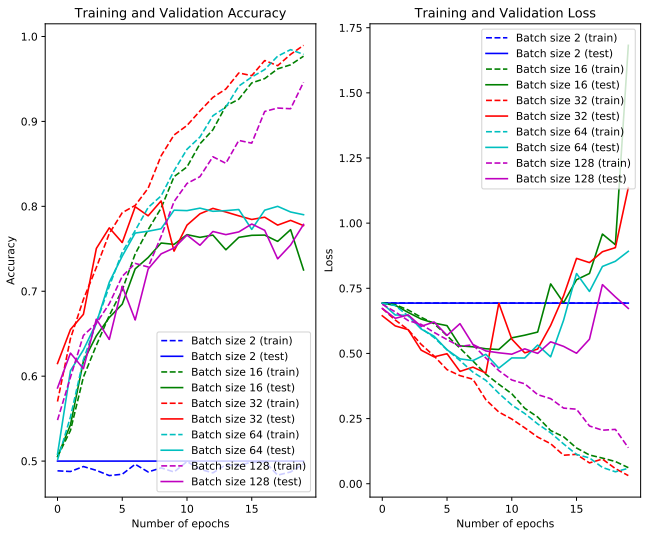


Figure 23: Comparison of the performance of CNNs with different batch sizes

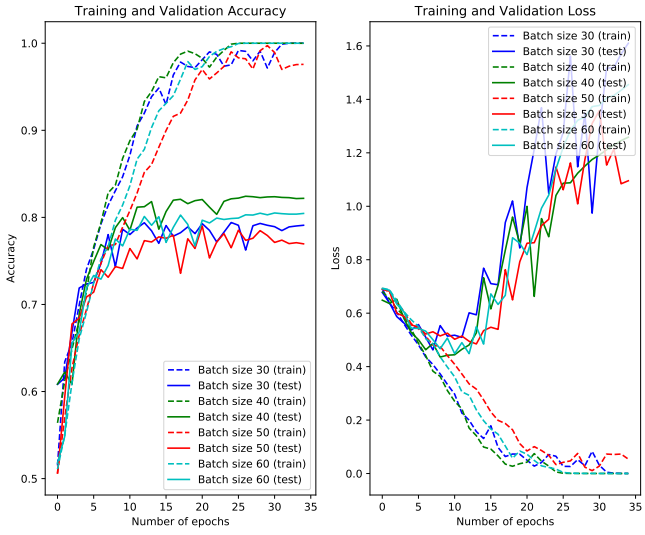
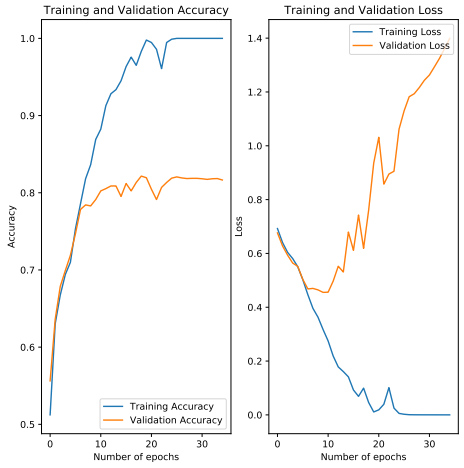


Figure 24: A comparison of a narrower range of batch sizes

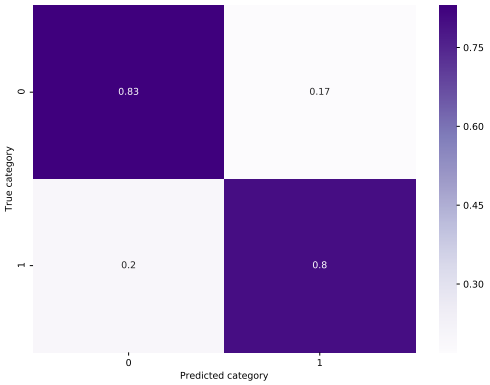
sizes that yield the highest accuracy scores and the lowest loss on the test images is 32 and 64. We then performed the same test again, for batch sizes between 30 and 60, to further investigate the performance of CNNs with batch sizes in this range. Figure 24 shows that the CNN with batch size 40 performed best. It obtained an accuracy score of 0.824267 on the test set, and a loss of 0.435749.

4.4.4 The final CNN model for classifying the cats and dogs data

We have finally landed on the final CNN model, with batch size 40, and 5 pairs of convolution and max pooling layers. We create and train this model one last time, to summarize out results. The



(a) The accuracy score and loss for each epoch during the training of the model



(b) The confusion matrix of the trained model

CNN model obtained an accuracy score of 0.816533 on the test images. This is a little lower than the accuracy score we obtained from the same model in the test in the previous section, which indicates that the results vary somewhat. Figure 18 shows some of the images that our network misclassified. It is interesting to note that unlike the misclassified MNIST images, these images are easy for the human eye to classify correctly.

Figure 26: A sample of the misclassified images. The label "1" stands for cat, while "0" is dog.

5 Discussion

5.1 Existing research

The MNIST data set has become one of the standard data set to benchmark different Machine Learning algorithms. For example the scientific paper [Cireşan, 2012] trains several deep neural networks and analyses their performance. The authors achieved a classification error rate of 0.23% .

The Dogs vs. Cats is also quite famous for being a standard data set in Machine Learning. In the following report [Bang Liu,] the authors explore CNN to learn features of images and trained Backpropagation(BP) Neural Networks and SVMs for classification. Their result had an accuracy of 92.1 % with CNN and 94 % using Support Vector Machines (SVM).

5.2 The results of state of the art methods

The website https://paperswithcode.com/sota/image-classification-on-mnist ranks the Machine Learning methods according to percentage error on the MNIST data set. The best algorithm has superhuman classification rate of less than 0.20 % , while pure CNN is ranked as number 54 with an error rate of 1.2%.

At https://www.kaggle.com/c/dogs-vs-cats/leaderboard one can observe that the best current accuracy score on the cats and dogs dataset is 0.98914. Unfortunately, we are not able to know which Machine Learning method they have used.

6 Conclusion

The CNN algorithm clearly outperforms Logistic regression on classification of both data sets.

- MNIST data: The highest accuracy score we managed to obtain with Logistic Regression was 92.56%, while CNN obtained 98.76% with potential of even further improvement.
- Cats and dogs data: With logistic regression we obtained an accuracy score of 53.07%, while our final CNN model obtained 81.65% with potential for further improvement.

Regarding computation time, there is no doubt that the Logistic method is the fastest one, using 2 to 10 minutes to train the model, while the CNN used 20 to 30 minutes every time. This makes perfect sense due to the complex structure of CNNs compared with logistic regression. Furthermore, our CNN model has a much greater potential for improvement than the logistic regression model. SciKit Learns's LogisticRegression class is already quite optimized, and while one could potentially improve its performance by tuning the parameters, it is limited by the underlying theory of logistic regression. On the other hand, there are several ways in which we could further improve the performance of our CNN models. We discuss some of these methods in section the next section on future work.

It is clear that logistic regression is not an optimal method for image classification. There are many instances where this simple model is suitable, but it is unable to capture the complexity present in image data. On the MNIST data, the performance was surprisingly decent, which is most likely because the images themselves are simple. By simple, we mean that the images have clear outlines of the numbers, high contrast, and no background noise. In comparison, the cats and dogs data contain images with varying items in the background and even in front of the animals, a wide variety of positions of the animals (standing, sitting, running, etc.), and the images also have drastically different contrast levels. Additionally, for logistic regression the spatial information of the images is lost because the images have to be flattened before fitting the model. It comes as no surprise that logistic regression has a hard time finding patterns in such data, and that CNNs are able to glean much more information from the images.

7 Future work

7.1 Increased data set and image augmentation

In Machine Learning it is known that the more data you have, the better trained your model will be. In fact, the ML model is only as good as the data upon which it was trained. To increase our accuracy scores on classification of the cats and dogs data, we could use the full data set, which consists of 25000 images instead of only the 5000 images that we used in this study. We can also further increase the amount of available training images by augmenting our original images. DataImageGenerator from Keras can increase the data set and provide more variations in the data as well, thanks to image augmentation techniques. Image augmentations techniques are methods of artificially increasing the variations of images in our data-set by using horizontal/vertical flips, rotations, variations in brightness of images, horizontal/vertical shifts etc. So by implementing Data Image Generator one can expect to get a more accurate algorithm.

We chose to not use image augmentation in this study, to instead focus on the performance of each machine learning method on the raw data. However, using the DataImageGenerator from Keras, we could almost certainly improve the performance of our CNN on the cats and dogs data. Furthermore, an interesting idea for augmenting the MNIST image data is to implement a rotation of the numbers 6 and 9. When rotated 180° these numbers will look quite similar. Would for example logistic regression still characterise a rotated 6 as the number 6?

7.1.1 Include normal Neural Networks in the comparison

One machine learning method for classification which we have not discussed is Neural Networks (NN). The classic Neural Networks do not use convolution like CNN, but are able to classify problems of higher complexity than logistic regression. In fact, one can view Neural Networks as a generalisation of logistic regression, as a one-layer neural network is equivalent to logistic regression. We would expect Neural Networks to

References

[Bang Liu,] Bang Liu, K. Z. Image classification for dogs and cats. https://sites.ualberta.ca/~bang3/files/DogCat_report.pdf.

[Bouvrie, 2006] Bouvrie, J. (2006). *Notes on Convolutional Neural Networks*. http://cogprints.org/5869/1/cnn_tutorial.pdf.

[Cireşan, 2012] Cireşan, Ueli Meier, J. S. (2012). Multi-coloumn deep neural networks for image classification. *arXiv:1202.2745*.

[David E. Rumelhart, 1986] David E. Rumelhart, G. E. H. . R. J. W. (1986). *Learning representations by back-propagating errors*, volume 1.

[Goodfellow et al., 2016] Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press. http://www.deeplearningbook.org.

[Hastie et al., 2001] Hastie, T., Tibshirani, R., and Friedman, J. (2001). *The Elements of Statistical Learning*. Springer Series in Statistics. Springer New York Inc., New York, NY, USA.