FYS-STK4155

# Image recognition with CNN and support vector machines

PROJECT 3

Ingrid Utseth
ingridut@fys.uio.no
Betina Høyer Wester
betiahw@student.matnat.uio.no
Polina Dobrovolskaia
polinad@mail.uio.no
Henrik Løland Gjestang
henriklg@student.matnat.uio.no
Mona Heggen
monahe@student.matnat.uio.no

Dato: December 17, 2018

## Abstract

In this project we study and attempt to make an comparison between our own NumPy-based CNN, Keras CNN, TensorFlow CNN and support vector machine model of MNIST database. We find that to make a computationally fast CNN is quite hard. However, prediction accuracy for NumPy-based CNN, Keras CNN, TensorFlow CNN and support vector machine models are 95.3%, 98.9%, 99.0% and 96.9%, respectively. Due to computational difficulties of NumPy-based CNN we had to resort to a smaller database of digits, 8 x 8 database, to obtain such a good accuracy.

Another aspects that we could emphasize in case of NumPy-based, TensorFlow and Keras CNNs is that employed structure of CNN model and the training has an immense impact on the final prediction accuracy.

# Contents

## 1 Introduction

Convolutional Neural Networks (CNNs) are a category of neural networks which has proven to be very efficient in image recognition and classification. CNN have a wide application in image and video recognition due to its two main characteristics; feature extraction and classification. The CNN method was first published by Yann LeCun [1], yet his work went unnoticed for fourteen years before it was proven to be of a great value. The publicly accepted implementation of the CNN was first seen during *ImageNet Computer Vision competition* where this method managed to achieve an accuracy of 84.7% when classifying millions of images from a thousand classes [2]. Today, the CNN is widely used and has by far surpassed human level of performance in classifying images [3], [4]. There are four most common building blocks in a Convolutional Neural Network,

- Convolution
- Non Linearity (ReLU)
- Max or average pooling
- Classification (Fully Connected Layer)

In this study we will endeavor upon creating CNN with our own handcrafted code (from now referred as NumPy-based), support vector machines, Keras and TensorFlow libraries. With this we would like to investigate if it is possible to achieve reasonable self-made CNN model for recognition of handwritten digits. We begin our outline by specifying the characteristics of different building blocks employed in a CNN set up. We will then explain the theory behind the CNN model together with the interpretation of used data. In the next sections we will explain our approach and methods of setting up Convolutional Neural Network.

# 2 Theory

## 2.1 The data set: MNIST

The MNIST database is a repository of handwritten digits that consists of sixty thousand training images and ten thousand test image sets. These images are size-normalized and centered in a fixed-size $28 \times 28$ pixels image. Some example digits are shown in figure 1.

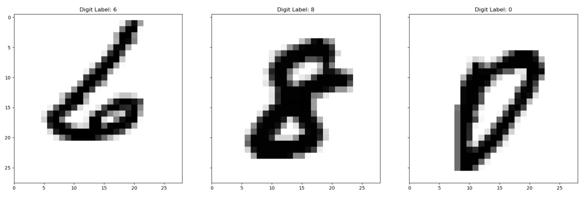


**Figure 1:** Example images from the MNIST data set (size $28 \times 28$). [5]

Since our NumPy-based CNN proved to be fairly slow, we tested this network on smaller data set with handwritten digits. The size of the images in this data set is $8 \times 8$ pixels and it consists of 1797 samples, three samples are shown in figure 2.

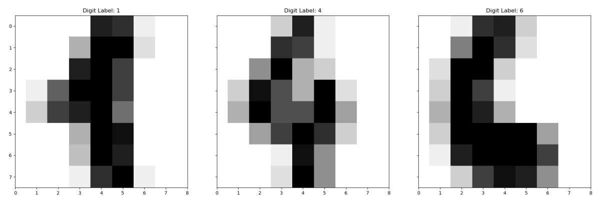


**Figure 2:** Example images from the smaller handwritten data set (size $8 \times 8$).

## 2.2 Support Vector Machines

Support vector machines belong to one of the great methods for performing regression, outlier detection, linear classification, and of course non-linear classification. In this project we will cover only one of support vector machine usages, namely, classification.

The idea behind the classification is to separate the classes in a given space using hyperplanes [6]. Given a classification problem with p numbers of features and c classes, the goal is to find c-1 hyperplanes of dimension p1 to separate the classes,

$$b + w_1 x_1 + w_2 x_2 + ... + w_p x_p = 0 \tag{1}$$

In vector form:

$$w^t x + b = 0 \tag{2}$$

We can find parameter b and vector w by the following ways,

- Defining a cost function which contains a set of miss-classified points and try to minimize this function
- Use gradient descent to solve the following equations

  $$b \leftarrow b + \eta \frac{\partial C}{\partial b} w \leftarrow c + \eta \frac{\partial C}{\partial w} \tag{3}$$

  where $\eta$ is the learning rate.
- find the largest value M defined by

  $$\frac{1}{||w||} y_i (w^T x_i + b \leq M \forall i = 1, 2, ..., n) \tag{4}$$

We want the hyperplane with the largest distance to each class if possible to ensure that future samples are classified correctly.

## 2.3 Convolutional Neural Network

### Convolution

The first step in a convolutional neural network is to extract features from the input image. This is done to preserve the relationship between pixels by learning image features using filters, or *kernels*. As a result, the network learn filters that activate when it detects some specific patterns or features. The operation can best be explained with the example in figure 3.

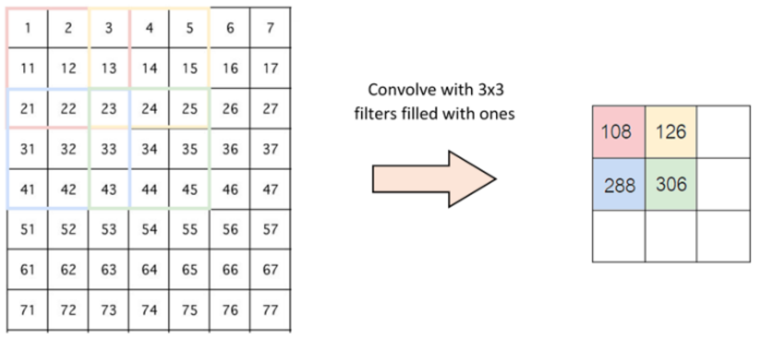


**Figure 3:** Kernel size of 3 × 3 and stride of 2 pixels. [7]

Stride is the number of pixels the filter jumps for each computation.

The convolution of $f$ and $g$ is written as $f * g$, and is defined as the integral of the product of the two functions after one (usually the filter) is reversed and shifted.

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t - \tau)d\tau \tag{5}$$

### Non Linearity (ReLU)

Rectified Linear unit function, known as simply ReLU, is an activation function represented by equation (6). It sets all negative numbers to zero, by discarding them from the activation map entirely. In this way, ReLU increases the nonlinear properties of the decision function and thus of the overall network without affecting the receptive fields of the convolution layer.[8]

$$ReLU(x) = max(0, x) \tag{6}$$

### Max pooling

Pooling layers are applied to reduce the number of parameters when the images are considerably large. Spatial pooling, or merely downsampling, reduces the dimensionality of each image but it keeps the important information. The most used downsampling is max pooling. It extracts the largest element from the rectified feature map and thus reduces computational complexity of the algorithm.

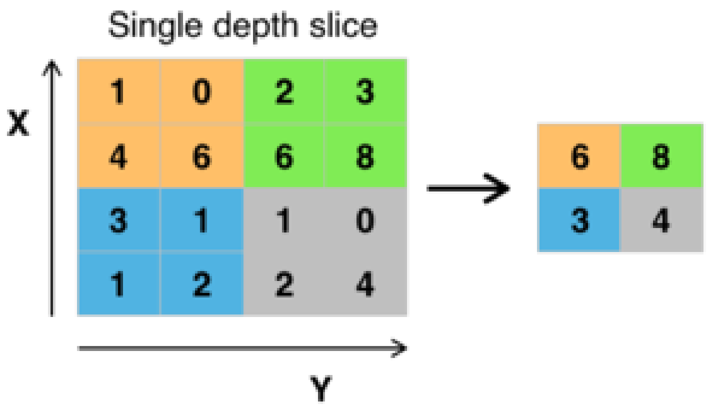


**Figure 4:** Max pooling with a $2 \times 2$ filter and stride of 2. [7]

### Classification

Classification is the last building block of a convolutional neural network. It is here the high-level reasoning is done.

### Feed Forward

In the feed forward algorithm input image will be processed through all the layers in the neural network. The first layer will be a convolution layer, containing $K$ filters $F_i^1$, $i = 1, ..., K$, of size $k \times k$ and a bias $b^1$. The image will be convoluted with each filter, and the bias is added.

$$\hat{z}_i^l = I * \hat{F}_i^l + b^l, \tag{7}$$

where $*$ (asterisk) is the convolution operator equation 5. The final output of each convolutional layer $l$ is $a^l$,

$$\hat{a}_i^l = f(z_i^l), \tag{8}$$

where $f$ represents the ReLU activation function. After going through the convolution layer, the next layer could be a pooling layer, which will reduce the dimensions (number of parameters) of the model. [9] The pooling layer could either use a max pooling function, which takes the maximum value of each pooling block of a predetermined size. An example of max pooling is shown in figure 4.

Another option is the use average pooling, which simply uses the average value of each pooling block.

Before getting our final output $\hat{y}$, we need to collect the outputs from all the filters, which will be an input to a fully connected layer. We simply take the column vectors of each input $a_i^{l-1}$ and concatenate the vectors into a column vector of length $N^{l-1}\dot{n}^{l-1}\dot{n}^{l-1}$ (where $N^{l-1}$ and $n^{l-1}$ represents respectively the number of matrices and the size of each matrix in the input).

After the vectorization, we can add a simply fully connected layer which uses the softmax activation function to classify the input image, much like a neural network would. The softmax function is an accepted standard probability function for a multiclass classifier. [10] The total sum of the probabilities will always add up to 1 when using softmax.

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \quad \text{for } j = 1, ..., K. \tag{9}$$

For our loss function, we use the cross-entropy error function for a multiclass output.

$$C(\hat{y}) = -\sum_{i=1}^{N} t_i log(y_i) \tag{10}$$

### Backpropagation

Starting from the final (fully-connected) layer $L$, we need to compute derivative of the loss function (function 10) with regards to the activation function in order to update the weights. Computing the gradient of the loss function yields

$$\frac{\partial C}{\partial y_i} = -\frac{t_i}{y_i} \tag{11}$$

We also require the gradient of the output of the final layer $y_i$ with regards to the input $z_k^L$ of the activation function (equation 9)

$$\frac{\partial y_i}{\partial z_k^L} = \begin{cases} y_i(1 - y_i), & i = k \\ -y_i y_k, & i \neq k \end{cases} \tag{12}$$

Now with regards to $z_i^L$:

$$\begin{aligned} \frac{\partial C}{\partial z_i^L} &= \sum_k^N \frac{\partial C}{\partial y_k} \frac{\partial y_k}{\partial z_i^L} \\ &= \frac{\partial C}{\partial y_i} \frac{\partial y_i}{\partial z_i^L} - \sum_k^N \frac{\partial C}{\partial y_k} \frac{\partial y_k}{\partial z_i^L} \\ &= -t_i(1 - y_i) + \sum_{k \neq i} t_k y_i \\ &= y_i - t_i \end{aligned} \tag{13}$$

And finally with regards to the weights

$$\frac{\partial C}{\partial w_i j^L} = (y_i - t_i) a_j^{L-1}, \tag{14}$$

where $\hat{a}_j^{L-1}$ is the vectorized output from the previous layer. From here, we will propagate the error through the layers. The error with regards to the input $a_i^L$ to the fully connected layer is simply

$$\delta^{L-1} = \frac{\partial C}{\partial a_i^L} = \sum_i^N (y_i - t_i) w_{ji}^L \tag{15}$$

Thus the error is propagated backwards through each layer. If max pooling was used in a pooling layer, the error will only be propagated to the input that had the highest value in the forward pass. The other values will be set to zero. If average pooling was used, the error is averaged in the backwards pass. [9]

Here, $a^l$ is the output of a convolutional layer $l$. Since a convolutional layer is always preceded and followed by a activation layer, the input to layer $l$ is $a^{l-1} = \sigma(z^l)$. Now consider the error with regards to $z^l$

$$\begin{aligned} \delta_{ij}^l &= \frac{\partial C}{\partial z_{ij}^l} \\ &= \sum_i^{\prime} \sum_j^{\prime} \frac{\partial C}{\partial z_{i'j'}^{l+1}} \frac{\partial z_{i'j'}}{\partial z_{ij}^l} \\ &= \sum_{i'} \sum_{j'} \delta_{i'j'}^{l+1} \frac{\partial(\hat{W}\sigma(z^l) + b^{l+1})}{\partial z_{ij}^l} \\ &= \delta^{l+1} * ROT180(w^{l+1}) \sigma'(z^l) \end{aligned} \tag{16}$$

Having found the error, the gradient of the cost function with regards to the weights is

$$\frac{\partial C}{\partial w_{ij}^l} = \delta_{ij}^l * \sigma ROT180(z_{ij}^{l-1}) \tag{17}$$

# 3 Method

## 3.1 Numpy based CNN

There are several python libraries available to easily set up, train and test a convolutional neural network. Later in this report we will use the Keras and TensorFlow libraries. To test our understanding of the subject we also built a CNN from scratch using NumPy functions. The full source code can be found in our GitHub.

Our CNN has classes for three types of layers; convolution layer, max pool layer and fully connected layer. For the convolution layer, one defines number of filters in the layer n_filter and kernel size of each filter kernel_size. One can also define an eta-value that controls the rate of change of the filter weights. ReLU was used as an activation function in the convolution layer.

For the max pool layer, one can define the stride. In the final fully connected layer, we need to define n_categories (number of classes) and n_images (number of input images), which are used to define the size of the weight matrix. Softmax was the activation function in the fully connected layer.

Each class has feed forward and backpropagation functions. A model class can be used to add layers, and perform the feed forward and backpropagation operations in the right order. The weights in the filters and in the fully connected layer were randomly initialized.

## 3.2 CNN with Keras

In this part we will explain and discuss how we have build up a CNN by use of Keras library and its MNIST data set of handwritten digits from 0 to 9. The source code we have created is based on two sources [11] and [12].

```
from Keras.data sets import mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

In this way we import and split our data into training and test data. Where the training data set now consists of 60 000 training images, along with a test set of 10 000 images.

x_train, x_test contain uint8 arrays of grayscale image data with shape (number samples, 28, 28). While y_train, y_test contain uint8 arrays of digit labels integers in range 0 to 9 with shape of (number samples,).

```
X_train = X_train.reshape(60000,28,28,1)
X_test = X_test.reshape(10000,28,28,1)
```

Here we reshape our training and test data with the following parameters: number of images, size, while the last digit indicates greyscale images.

In the next step we categorize our y values, converting simple int values in to array of zeroes and ones. By the use of Keras utils class to_categorical we make each y value an array of 0 and 1 which implies which label that specific digit is.

```
from keras.utils import to_categorical
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)
```

```
y_train[0]
#array([0.,0.,1.,0.,,0.,,0.,,0.,,0.,,0.], dtype=float32)
```

In the above code passage setting 1 in third position of y_train[0] array we define representation of digit 2. In the similar way, will all digits in y arrays can be converted to arrays.

Next, we build up the model we use Sequential layer structure. The Sequential model is a linear stack of layers. Then we will add layers by the use of .add() function.

```
from keras.models import Sequential
from keras.layers import Dense, Conv2D, Flatten

model = Sequential()

model.add(Conv2D(100, kernel_size=3, activation='relu', input_shape=(28,28,1)))
model.add(Conv2D(50, kernel_size=3, activation='relu'))
model.add(Flatten())
model.add(Dense(10, activation='softmax'))
```

As our first layer we set Conv2D, which is 2D convolution layer. This layer creates a convolution kernel. The first parameter indicates number of neurons in this layer, which we set to 100 neurons. The kernel_size specifies the height and width of the filter matrix of our convolution. Here its is set to be a single integer, 3. For the activation function we use ReLU, Rectified Linear Unit. Next, the last parameter in the Conv2D function, states the size of the input image [11].

The second layer is as well a Conv2D, all parameter are set to be the same as in the previous layer, except number of neurons. Due to our experimental reason, we set it to be half the value of the number of neurons in the first layer. The number of neurons in the two layers is something we will try to adjust to see the impact on the resulting accuracy.

The next layer we add is to transform our output from previous layers to the output layer. This is done by Flatten().

Last layer, the Dense layer, is set up with ten nodes since we are classifying ten digits, and with activation function softmax. The output of the softmax function is probability distribution in the ten classes. It tells us the probability that any of the classes are more probable than other.

```
model.compile(optimizer='adam', loss='categorical_crossentropy', ..
        metrics=['accuracy'])
```

The next building step is to compile our model. We use adam as our optimizer, which will control the learning rate during training. A smaller learning rate tunes more accurately weights and biases. However, the time to produce a good estimate for these will be consume more time. For the loss we choose categorical_crossentropy. And as before, we wish to study accuracy of our model. That is something we define in the metric parameter.

```
model.fit(X_train, y_train, validation_data=(X_test, y_test), epochs=10)
```

To train and validate our model we use fit() function. The input parameters are our training data, test data and the number of epochs. The epochs define the number of times our model will cycle through the data. This number is also something we need to examine and adjust. Increasing this number to a certain degree will improve the accuracy of the model. The batch_size is not set and thus has a default value of 32; which we decided to keep unchanged. We start off with setting initial number of epochs to 10.

```
model.predict()
```

Later to make predictions with our model, we can use predict() function. The input is a image matrix with handwritten digit.

## 3.3 Create CNN with TensorFlow and TensorBoard

In this part we look at labeling MNIST by using a CNN with TensorFlow. We also look at how TensorBoard can be used to visualize graphs of the CNN model and how each layer works and connects to the other. Application of the TensorBoard gives better understanding and pin point the main essence of the computational flow in the network.

To get the right data set format for TensorFlow we use one_hot encoding. This function encodes the labels into a set of values, where the actual class is high (1) and the other is low (0). This is necessary when we are going to use next_batch.

This CNN is built up like the other CNN we have investigated. The CNN is using two 2D convolutional layers and maxpooling $2 \times 2$ kernel between. ReLU is also used as the activation function in this part. Then there is a fully connected layer and a readout layer.

TensorBoard is a suite of web applications for inspecting and understanding your TensorFlow runs and graphs. TensorBoard currently supports five visualizations: scalars, images, audio, histograms, and graphs. The computations you will use in TensorFlow for things such as training a massive deep neural network, can be fairly complex and confusing, TensorBoard will make this a lot easier to understand, debug, and optimize your TensorFlow programs. [13]

TensorBoard creates model graph and visualize the outputs and results over time. You can organize the complicated flow by creating scopes to group operations belonging together. We use name_scope() to better organize the model graph in TensorBoard.

## 3.4 Support Vector Machines

For the implementation of of support vector machines with Gaussian decision boundaries we will utilize scikit-learn library. We do this by converting the image from a $28 \times 28$ array to a $784 \times 1$ array and let each pixel be considered as a feature. We need to choose two parameters: $C$ and $\gamma$. $C$ is the penalty parameter of the error term. A high $C$ will try to classify all training data correctly, while a smaller $C$ will result in a smoother decision boundary. $\gamma$ defines how much influence a single training sample has. We choose these parameters by looking at a smaller set of the training data and testing with different values for $C$ and $\gamma$,

```
# ----- Find best values for c and gamma ------------------
from sklearn.model_selection import GridSearchCV
gamma_values = np.logspace(-3, 0, 4) #array([0.001, 0.01 , 0.1  , 1.   ])
c_values = np.logspace(-1, 1, 3)     #array([ 0.1,  1. , 10. ])
parameters = {'kernel':['rbf'], 'C':c_values, 'gamma': gamma_values}
svm_clsf = svm.SVC()
grid_clsf = GridSearchCV(estimator=svm_clsf,param_grid=parameters,...
                                        n_jobs=1, verbose=2)
grid_clsf.fit(X_train, y_train)
sorted(grid_clsf.cv_results_.keys())
classifier = grid_clsf.best_estimator_
params = grid_clsf.best_params_
scores = grid_clsf.cv_results_['mean_test_score'].reshape(len(c_values),..
                                        len(gamma_values))
print(scores)
```

Then we would classify the entire data set in the following way,

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(images/255, ...
                        targets, test_size=0.4, train_size = 0.1)
param_C = 10
param_gamma = 0.01
classifier = svm.SVC(C=param_C, gamma=param_gamma)
classifier.fit(X_train, y_train)
```

## 4 Results and discussion

### 4.1 Convolutional Neural Network

#### Numpy based CNN

Our NumPy-based CNN was set up with two convolutional layers with 10 and 2 filters respectively of size $2 \times 2$. The activation function in each convolution layer was simply ReLU. In between each convolution layer, we added a max pooling layer with $stride = 2$. The final layer is a fully connected layer with the softmax activation function. The initial weights and filter variables are all random.

```
model = CNN(X_train[0:100, :], Y_train[0:100, :])
model.add_conv_layer(n_filters=10, kernel_size=[2, 2])
model.add_maxpool_layer(stride=2)
model.add_conv_layer(n_filters=2, kernel_size=[2, 2])
model.add_maxpool_layer(stride=2)
model.add_fullyconnected_layer()
```

Since our NumPy CNN is fairly slow, with many loops and variables, we decided to utilize a simpler data set than the MNIST data set. Instead, we used a similar data set with handwritten numbers of size $8 \times 8$. Using the train_test_split function from the sklearn python model, we split the data set into training and test data and processed the target data to work with softmax output.

We trained the model using batch gradient descent with 20 rounds and batch size $= 10$. After 20 rounds, we got an accuracy of 0.8861, which certainly proved that our NumPy CNN worked. The confusion matrix (figure 9) shows that our network got particularly good at classifying 0 and 6 correctly. However, it did sometimes confuse 4 and 7, and 5 and 3.

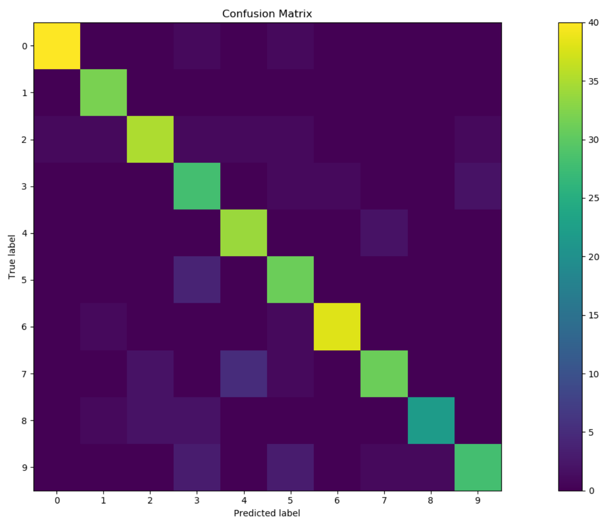


**Figure 5:** Confusion matrix from NumPy-based CNN with 20 rounds.

Increasing the training rounds to 100 increased the accuracy to 0.9528. The confusion matrix in figure 6 shows however, that the model still struggles with correctly classifying certain numbers. It miss-classifies numbers as 3 particularly often; 10 samples are incorrectly classified as 3.

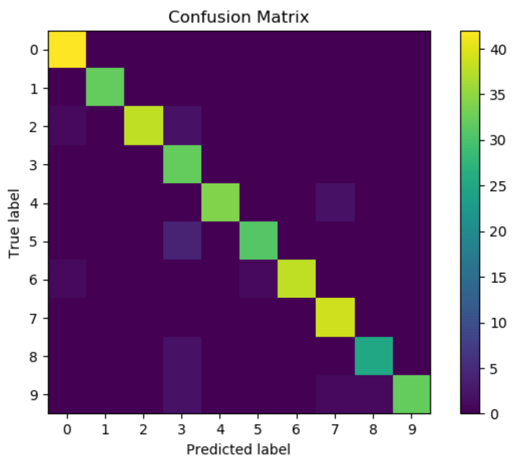


**Figure 6:** Confusion matrix from NumPy CNN with 100 rounds

There are certainly more we could do to improve the result. We could investigate how various numbers of filters and their size affect the result, replace the max pool layer with average pool or try a different way of initiating the random variables. There are better and likely more efficient ways of doing the gradient descent. We use a simple, fixed learning rate, but an adaptive learning rate would probably give us faster convergence. There are also ways to improve the computation speed, for instance rewriting the program using Cython. However, implementing and running the program with these suggested changes is too difficult and time consuming for this project. We have shown that a very simple and basic CNN can be written using NumPy, and it does work. However, since fast and precise python modules for CNN classification already exist, there is little point in spending time writing your own python module for this task (apart from a purely academic interest in the inner workings of a CNN). Therefore, we move on to classification with the Keras and TensorFlow python modules.

### Convolutional neural network with Keras

Table 1 illustrates the model accuracy as we vary number of nodes and epochs in the model training of our Keras model set up in section 3.2. From the table we that increasing the number of nodes in a layer does not necessarily produces the best accuracy of our model which has been also stated in many of our sources. In the same way, we saw that increasing number of epoch, did not result in a better accuracy. However the computation time increased drastically, an expected consequence of dealing with big data.

With 10 epochs for different combinations of nodes the computational time grew, while the accuracy dropped.

| | nodes layer 1 | nodes layer 2 | epochs | accuracy |
|---|---|---|---|---|
| Test 1 | 100 | 50 | 2 | 0.4098 |
| Test 2 | 100 | 50 | 3 | 0.8086 |
| Test 3 | 100 | 50 | 10 | 0.7905 |
| Test 4 | 50 | 25 | 2 | 0.0974 |
| Test 5 | 50 | 25 | 3 | 0.9892 |
| Test 6 | 50 | 25 | 4 | 0.9719 |
| Test 7 | 50 | 25 | 10 | 0.2110 |
| Test 8 | 55 | 30 | 2 | 0.2000 |
| Test 9 | 55 | 30 | 3 | 0.7041 |
| Test 10 | 55 | 30 | 10 | 0.3050 |
| Test 11 | 60 | 32 | 2 | 0.7217 |
| Test 12 | 60 | 32 | 3 | 0.9121 |
| Test 13 | 60 | 32 | 4 | 0.9111 |
| Test 14 | 60 | 32 | 10 | 0.2297 |

**Table 1:** Test with different parameter for Keras convolutional network. Number of nodes in first and second layer and number of epochs are varied.

For CNN recognition model with Keras the gave the best representation of the MNIST data set when we had two Conv2D layers with 50 and 25 nodes respectively, ran the training with 3 epochs. The best and quickest obtained accuracy was 0.9892.

### CNN with TensorFlow and TensorBoard

Each training using CPU took approximately 30 minutes for 1000 steps and an hour for 2000 steps. Our observation, shown in table 2 is that TensorFlow code is best when using learning rate = 0.001, Adam optimizer [14] and ReLU. It gets approximately 99.0% accuracy for the best test batch.

| | $\gamma = 0.0001$ | $\gamma = 0.001$ | $\gamma = 0.01$ | $\gamma = 0.1$ |
|---|---|---|---|---|
| steps = 1000 | 96.3% | 98.5% | 96.3% | 10.3% |
| steps = 2000 | 97.5% | 99.0% | 11.4% | 11.3% |

**Table 2:** Test accuracy: Grid search of learning rate, $\gamma$, and number of steps

We also looked at some of the available tools in the TensorFlow toolbox, specifically TensorBoard. It gives a frictionless way to visualize and inspect the model. By using scopes we can organize the graph and then get a quite good overview of the code and how the neural network is built up. The graph can be optimized and organized as the user require. TensorBoard can visualize scalars, images, graphs, distributions and histograms. Out of the box TensorBoard can be used both by TensorFlow and Keras. We organized the TensorFlow code by using scopes and the graph gives a good overview of the results and how the neural network is built up. In figure 7 and 8 are some results of the outputs from TensorBoard ($\gamma = 0.001$, steps = 1000). Here is the output figures from TensorBoard of loss function, train accuracy, and the model graph.

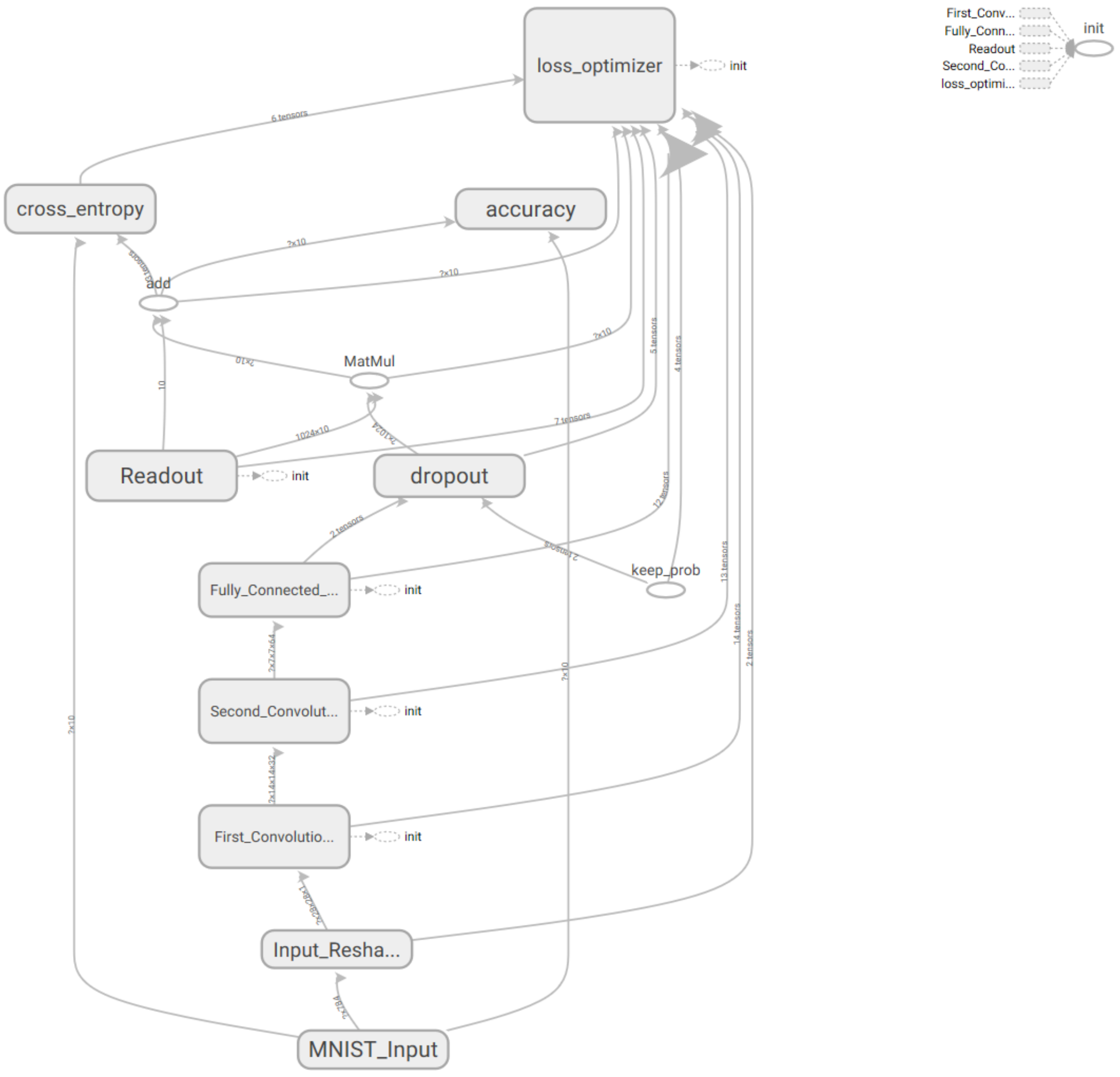
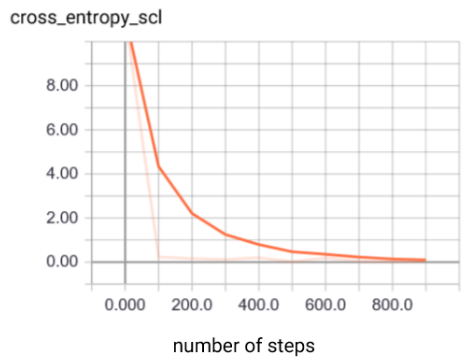


**Figure 7:** The output graph from TensorBoard graph



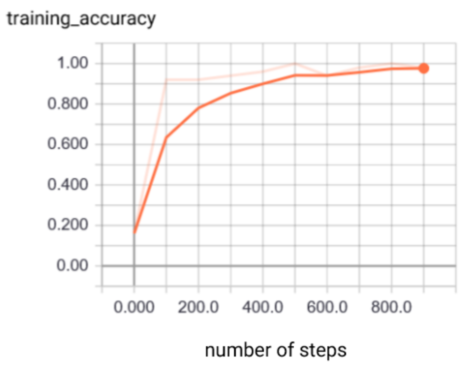


**Figure 8:** The output figures from TensorBoard for the loss function (left) and training accuracy over time/number of steps (right)

As we can see the figures in this section indicates that the network is working as it should, the training accuracy is going up and the loss is less when the steps increases.

The accuracy of the test data with only four layers, 0.001 as learning rate and 2000 steps is 99.0%. This is promising results for this particular problem. It might be possible to get this higher using more layers and fine-tuning. The problem could be a overfitted model that only recognizes the training data. A negative side for the TensorFlow method is that it uses approximately 30 to 60 minutes for each training. But on the positive side you get a good overview of the CNN using TensorBoard.

## 4.2 Support Vector Machine

After testing with different values for C and $\theta$, we get the accuracy in Table 3.

| | $\gamma = 0.001$ | $\gamma = 0.01$ | $\gamma = 0.1$ | $\gamma = 1$ |
|---|---|---|---|---|
| C = 0.1 | 0.708 | 0.901 | 0.214 | 0.118 |
| C = 1 | 0.898 | 0.945 | 0.827 | 0.1177 |
| C = 10 | 0.928 | 0.9534 | 0.827 | 0.1177 |

**Table 3:** Grid search of $\gamma$ and C values.

We therefore choose $\gamma = 0.01$ and $C = 10$. With these parameters, we get an accuracy of 0.96096 using 40% of the data set for training and 10% for testing. The confusion matrix is shown in figure 9.Using the whole set for training and testing (80% for training and 20% for testing) we get an accuracy of 0.969.

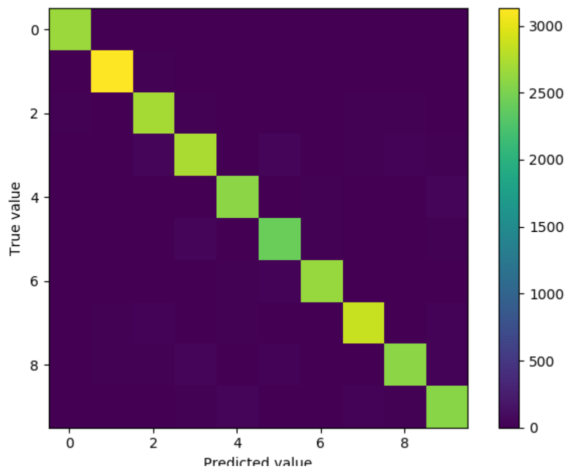


**Figure 9:** Confusion matrix from SVM with 40% of the data set for training and 10% for testing.

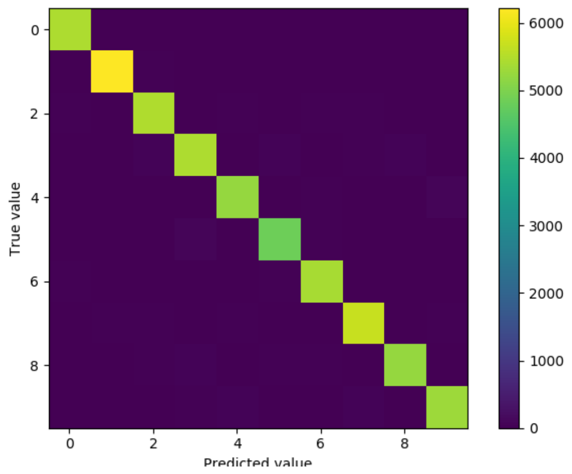


**Figure 10:** Confusion matrix from SVM with 80% of the data set for training and 20% for testing.

## 5 Conclusions and perspectives

In this project we have strived to apply our practical tools obtained through out the course to study convolutional neural networks with a MNIST data set. With that, we have compared models from NumPy-based, Keras and TensorFlow CNN models. As well, we have made a model with support vector machines to model the MNIST data. The training computational time and accuracy were two model traits that we focused in this analysis. The NumPy-based CNN resulted in an accuracy of 95.3% at its best; however the computational time was a protracted process, which resulted in that we had to use an alternative and simplified data set than the MNIST database. Keras on the other hand, with its 3 training epochs, and a running time less than a minute on a simple student computer gave a prediction accuracy of 98.9%. Likewise, the leading prediction accuracy for CPU-based TensorFlow using 2000 steps with learning rate of 0.001 was 99.0%. Support vector machines, as well, provided us with an excellent prediction accuracy of 96.9%.

Despite satisfactory accuracy, we had to make some omissions and alternations. Among the most prominent differences which make it difficult to make exact comparison between the models are the type, sequence and size of the layers in a model. As well the computational time versus accuracy.

The databases used in this project (MNIST and simplified 8 x 8 digit database) are fairly pellucid to classify due to its binary representation of digits. Remarkably, the main variant from one digit class to another is the pixel positions. All our models produced good prediction accuracy, yet the compiling time was variable from one model to another. In spite of the fact, for analysis of digit database we would propose support vector machines method. It has simple implementation, runs swiftly and produces high prediction accuracy.

For the further investigations constructing a more optimal NumPy-based CNN model would be both challenging and compelling. Then, a next step could involve studying digit data sets with more noise and examine the accuracy our models.

Yet, another study could involve investigation if our models could work as effectively on other data sets as for instance, CIFAR10 [15]. This data set consists of sixty thousand 32 x 32 colour images in ten classes and have more variance like background and pixel intensity.

Furthermore, image recognition studies would be interesting to perform with transferred learning network. Two examples could be inception-V3 classifier or YOLO [16]. For very complex data set and problems transfer learning can be a good fit to fine tune the model for the specific problem.

## 6 Python code

All of our code can be found on our GitHub repository: `https://github.com/henriklg/Project3_FYS-STK4155`

We would like to thank professor Morten Hjorth-Jensen for his expert advice and encouragement throughtout this difficult project, as well as Kristine Baluka Hein, Bendik Samseth and Øyvind Sigmundson Schøyen for their brilliance in the lab.

# Bibliography

[1] Y. Lecun, L. Bottou, Y. Bengio and P. Haffner, "Gradient-based learning applied to document recognition," in Proceedings of the IEEE, vol. 86, no. 11, pp. 2278-2324 (Nov. 1998) `http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=726791&isnumber=15641`

[2] Alex Krizhevsky, et al. “ImageNet Classification with Deep Convolutional Neural Networks.” Communications of the ACM, vol. 60, no. 6, 2017, pp. 84–90., doi:10.1145/3065386. `https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf`

[3] R. Wu, S. Yan, Y. Shan, Q. Dang, G. Sun, "Deep Image: Scaling up Image Recognition", `https://arxiv.org/pdf/1501.02876v3.pdf`

[4] L. Chen, S. Wang, W. Fan, J. Sun, S. Naoi. "Beyond Human Recognition: A CNN-Based Framework for Handwritten Character Recognition", `https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=7486592`

[5] Yann LeCun, et al. Courant Institute, NYU, "THE MNIST DATABASE of handwritten digits". `http://yann.lecun.com/exdb/mnist/`

[6] M. Hjorth-Jensen, "Data Analysis and Machine Learning: Support Vector Machines". `https://compphysics.github.io/MachineLearning/doc/pub/svm/pdf/svm-minted.pdf`

[7] Raghav Prabhu, "Understanding of Convolutional Neural Network (CNN)—Deep Learning" (March 2. 2018) `https://medium.com/@RaghavPrabhu/understanding-of-convolutional-neural-network-cnn-deep-learning-99760835f148`

[8] Sagar Sharma, "Activation Functions: Neural Networks" (Sep 6, 2017) `https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6`

[9] A. Karpathy, "CS231n Convolutional Neural Networks for Visual Recognition". `http://cs231n.github.io/convolutional-networks/`

[10] Peter Sadowski, "Notes on Backpropagation". University of California Irvine. `https://www.ics.uci.edu/~pjsadows/notes.pdf`

[11] Eijaz Allibhai, "Building a Convolutional Neural Network (CNN) in Keras". `https://towardsdatascience.com/building-a-convolutional-neural-network-cnn-in-keras-329fbbadc5f5`

[12] Aditya Sharma, "Convolutional Neural Networks in Python with Keras" `https://www.datacamp.com/community/tutorials/convolutional-neural-networks-python`

[13] LearningTensorFlow.com, "Visualisation with TensorBoard" `https://www.learningtensorflow.com/Visualisation/`

[14] Diederik P. Kingma, Jimmy Lei Ba, "ADAM: A method for stochastic optimization", `https://arxiv.org/pdf/1412.6980.pdf`

[15] Alex Krizhevsky, "Learning Multiple Layers of Features from Tiny Images", `https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf`

[16] Joseph Redmon, Ali Farhadi, "YOLOv3: An Incremental Improvement" University of Washington, `https://arxiv.org/pdf/1804.02767.pdf`

[17] Grzegorz Gwardys, "Convolutional Neural Networks backpropagation: from intuition to derivation" (April 22. 2016). `https://grzegorzgwardys.wordpress.com/2016/04/22/8/`