Final Project Report

Vemund .V Brynjulfsen

Introduction

I'm coding a CNN from scratch.

I will try to only use pure python as much as possible (not use any in-built libraries).

This means I will code convolutional layers to process the images, and a neural network to classify the images.

I will implement code for training a CNN model, which means I will implement forward- and back propagation for each layer.

Dataset

I will train the model on the MNIST dataset.

The dataset consists of handwritten digits. The digits range from 0-9.

The model will be trained to try to classify which digit is being displayed on the images.

I got this dataset from canvas on the page Course → Files → Data_Features

I will use the files for Digit_training.zip and Digit_testing.zip.

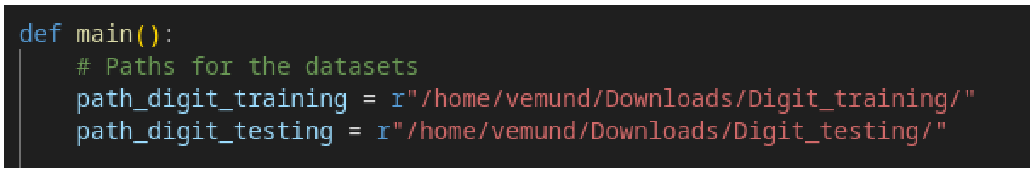
There are two fields in the main.py file where the paths for the images can be specified.

The path specified should be at the folder contining the subfolders for the digits.

For example if the path is **some_folder/some_other_folder/Digit_testing/0**

Then the specified path for the variable in main.py should be **some_folder/some_other_folder/Digit_testing/**

Example:



```
def main():
    # Paths for the datasets
    path_digit_training = r"/home/vemund/Downloads/Digit_training/"
    path_digit_testing = r"/home/vemund/Downloads/Digit_testing/"
```

Motivation

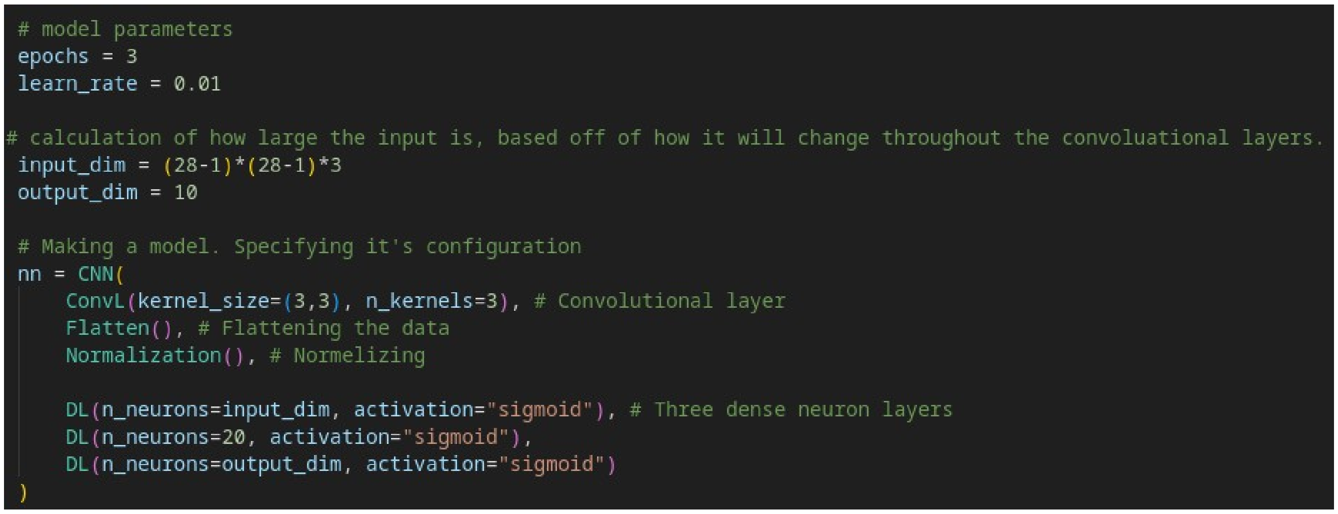
The motivation for the project topic is to get a deeper understanding of how a CNN works.

By coding the CNN from scratch, will will be able to fully edit the code such that I can see how the data is handled and classified. I can for example look at how images change throughout the layers, how the kernels and neurons change throughout the training, etc.

Analysis of results

I've coded a framework for making a CNN. A CNN can be configured when initializing it.

Here's an example of a possible configuration:



```
 # model parameters
 epochs = 3
 learn_rate = 0.01

# calculation of how large the input is, based off of how it will change throughout the convoluational layers.
 input_dim = (28-1)*(28-1)*3
 output_dim = 10

 # Making a model. Specifying it's configuration
 nn = CNN(
     ConvL(kernel_size=(3,3), n_kernels=3), # Convolutional layer
     Flatten(), # Flattening the data
     Normalization(), # Normelizing

     DL(n_neurons=input_dim, activation="sigmoid"), # Three dense neuron layers
     DL(n_neurons=20, activation="sigmoid"),
     DL(n_neurons=output_dim, activation="sigmoid")
 )
```

The results of the CNN will somewhat vary, depending on:

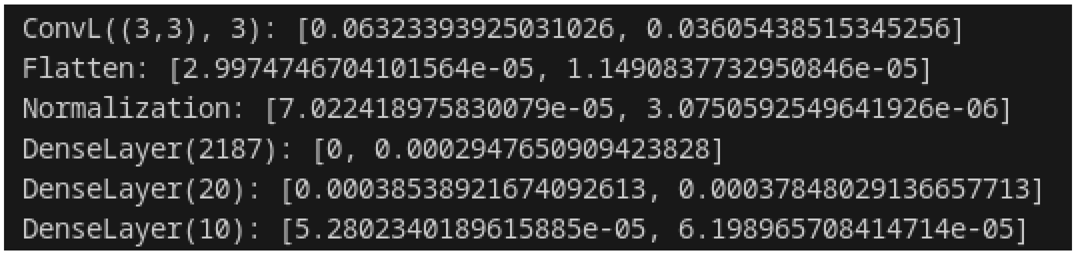
- the configuration of the CNN.
- values initialized at random (for example weight values initialized to be -0.5 – 0.5).

For the results, I'll use the configuration above.

Running time

I will analyze the running time of the CNN. The running time will depend on the configuration of the CNN. Larger kernels, more neurons, and more layer will increase the running time, as there are more computations that needs to be done.

I'm using the above configuration.



```
ConvL((3,3), 3): [0.06323393925031026, 0.036054385153452256]
Flatten: [2.9974746704101564e-05, 1.1490837732950846e-05]
Normalization: [7.0224189758300079e-05, 3.0750592549641926e-06]
DenseLayer(2187): [0, 0.0002947650909423828]
DenseLayer(20): [0.00038538921674092613, 0.00037848029136657713]
DenseLayer(10): [5.2802340189615885e-05, 6.198965708414714e-05]
```

This image shows the average runtime for forward- and back propagation in each layer.

The first number in the list is for forward propagation, and the last number is for backward propagation.

We can see that the constitutional layer has the largest running time.

The 2nd dense layer has the 2nd largest runtime.

The first layer has no running time for the forward propagation, because the only thing being done is to input the already-exiting values from the convolutionl layers into the neurons.

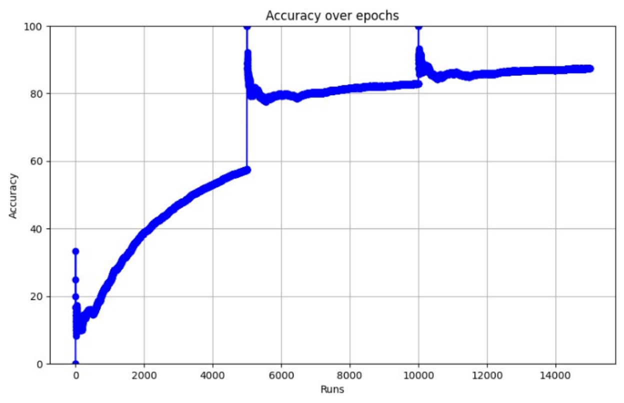
But the running times for the other dense layers rather are meant for the amount of time taken to update the weights between the dense layers.

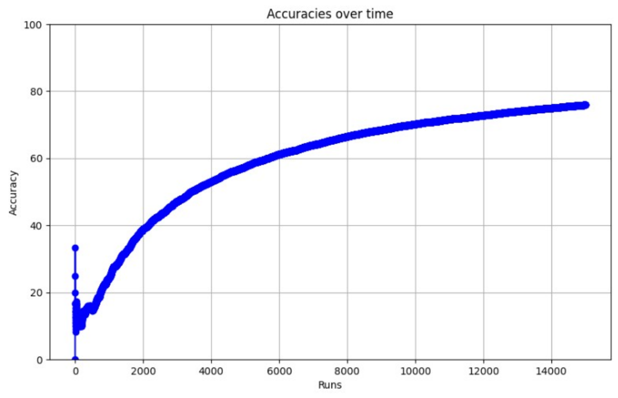
The times for the dense layers show in large part time for matrix multiplication between layers.

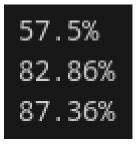
Accuracies

Here are the accuracies:

Accuracy over time:







```
57.5%
82.86%
87.36%
```

The image of the prints show the accuracy for each epoch.

The first image shows the accuracy over all runs.

The 2nd image shows the accuracy for each epoch. The first image's ending accuracy is dragged down as the inaccuracies of the other epochs will drag the overall accuracy score down.

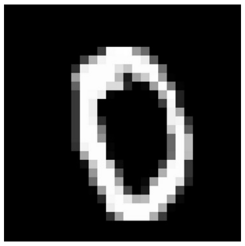
In the 2nd image this effect doesn't occur.

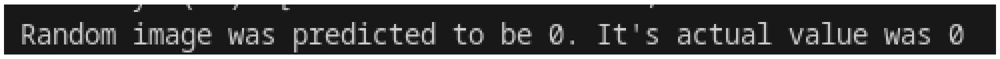
The accuracies starts low. But as the model adjusts it's weights during training, the model's accuracy increases.

The accuracy eventually stagnates as the there is not much more optimization of the weights that are possible.

There are also methods to predict a single image, and to test the model on testing data.

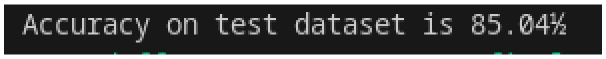
If use the model to predict a single image from the testing dataset, it manages to label the image correctly.





```
Random image was predicted to be 0. It's actual value was 0
```

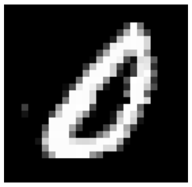
Accuracy on the test dataset:

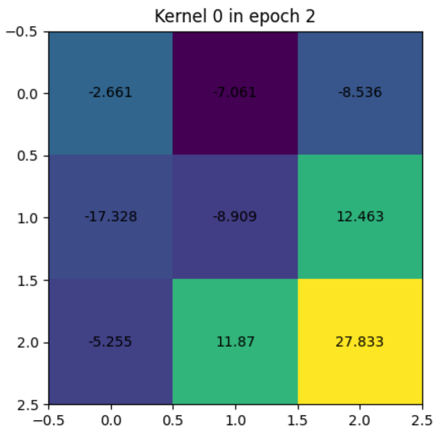


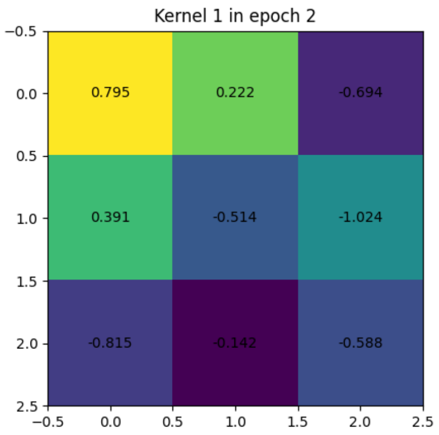
```
Accuracy on test dataset is 85.04%
```

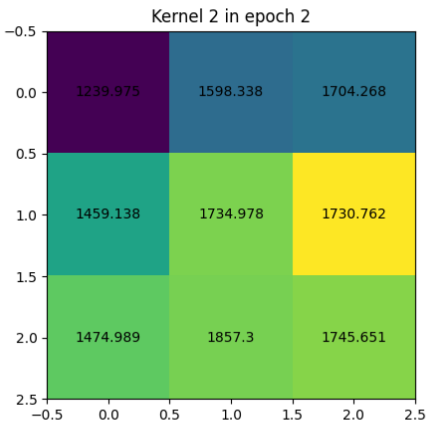
Kernels

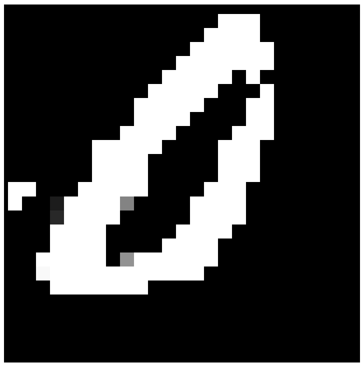
Original image:

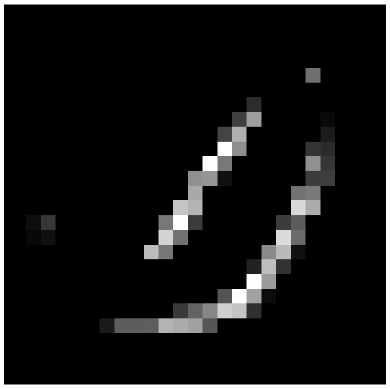


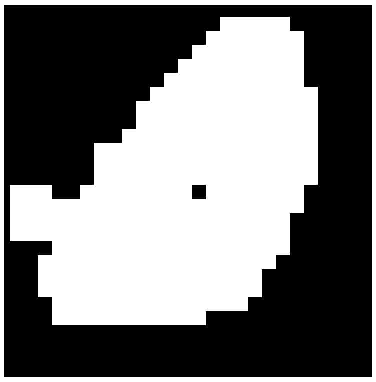












Here we can see the kernels in the convolutional layer. “Epoch 2” is really the last epoch 3, but the index is used in the title (and indexes start at 0).
The kernels are applied over the images to draw out features of the images, so that the neural network can better classify them.
We can see how different kernels captures different features of the images.

Conclusion
The CNN has a good performance. It’s able train in order to classify the hand written digits.
It’s interesting to see how the weights and values change as the model is being trained, which leads to the model’s accuracy increasing.

The CNN API can be extended to have more functionalities.
It can for example have more possible plots and graphs that can be made from it.
It can be extended to have different types of layers, such as a pooling layer.
Currently it has these possible layers: Dense layer, flattening layer, normalization layer, convolutional layer.
It can be extended to have more possible activation functions, such as relu. Currently it only has the sigmoid activation function.