Assignment 1 Report

Janik Lobsiger, 20685234

October 3, 2019

Contents

1 MNIST

In the following we use the training and testset from `http://yann.lecun.com/exdb/mnist/`. In all experiments we use a batchsize of 64 and train our models for 20 epochs.

The code was written in Spyder (Anaconda) in Windows 10.

1.1 K Nearest Neighbors (KNN)

Code: *knn.py*

Description: To solve this task we train a *KNeighborsClassifier* model from the *sci-kit* library. We use the Manhattan distance as our distance metric, since this corresponds to the SAD. By finding the nearest neighbor, we can achieve an accuracy of 96.31%. We also train the same model by considering k = 2, ... , 10 instead of just 1 neighbor. For this we can directly use scikit's *score* method which predicts the majority label of the k nearest neighbors of each test input. The results are shown in figure 1. As you can see, varying k between 1 and 10 doesn't have a large influence on the prediction accuracy.

Sources:

- `https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html#sklearn.neighbors.KNeighborsClassifier.predict`

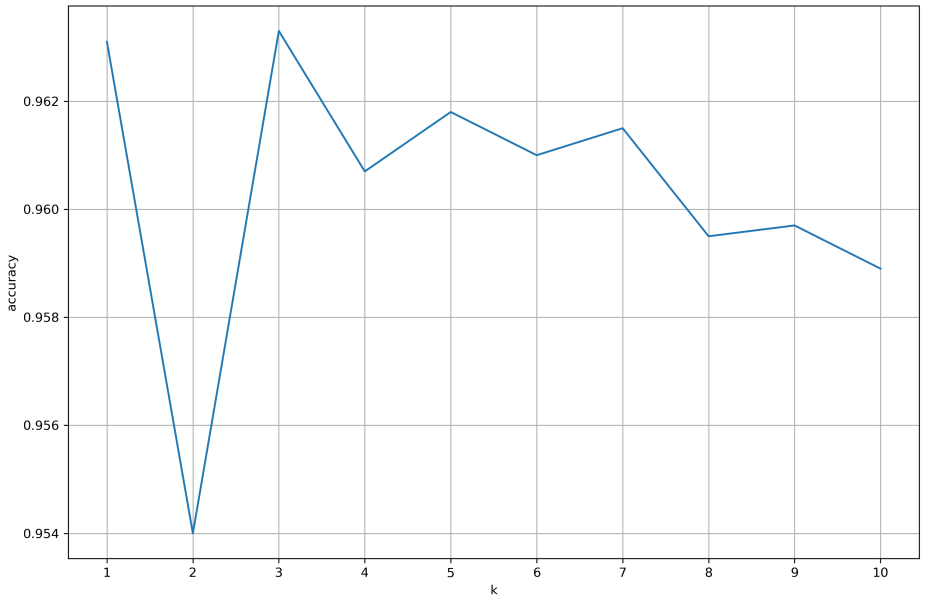

Figure 1: Accuracy of KNN for k = 1, ..., 10

1.2 Multilayer Perceptron (MLP)

Code: *mlp.py*

Description: We use the *Keras* library with *Tensorflow* backend. We implement the network as outlined in the exercise with two hidden layers with varying number of nodes and *ReLU* activation function. For the output layer we use the *softmax* activation to get a 10-dimensional probability vector. Finally we use the *Crossentropy* loss and *Adam* for optimization. Table 1

shows how the size of the model varies if we change the number of nodes in the hidden layers. Figure 2 shows the corresponding accuracies on the testset. Using 4 nodes per hidden layer already results in an accuracy of 85.55%. This can further be improved to 97.31% with 64 nodes. Going from 64 to 256 nodes increases the model size by a factor of almost 5 while only giving marginal improvements in accuracy.

#nodes	**#parameters**
4	3,210
8	6,442
16	13,002
32	26,506
64	55,050
128	118,282
256	269,322

Table 1: Number of parameters of the MLP by # nodes in the hidden layers

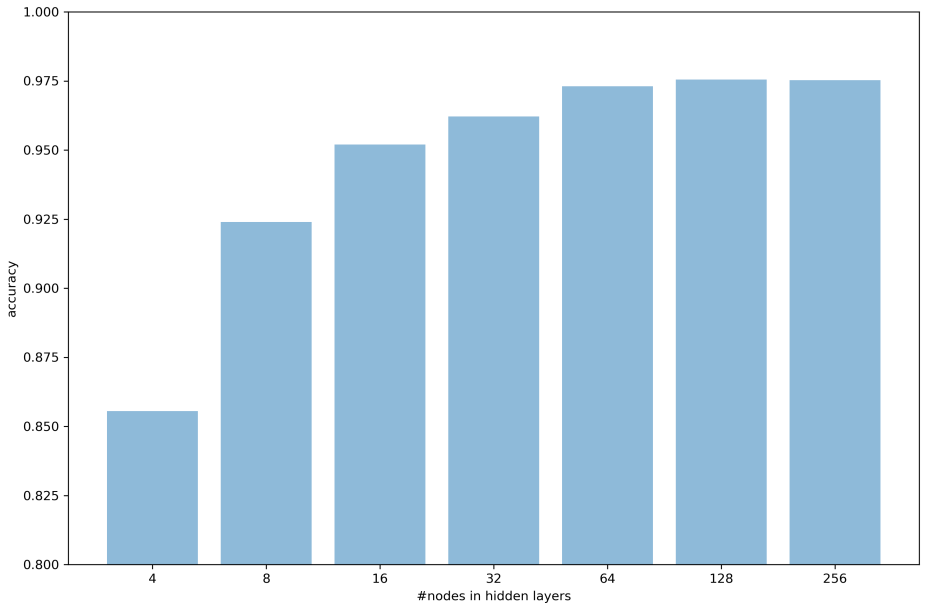

Figure 2: Accuracy of MLP for different number of nodes in the hidden layers

1.3 Convolutional Neural Networks (CNN)

Code: *cnn.py*

Description: For this exercise we simply implement the LeNet-5 architecture from the provided tutorial. (Note: I only looked at the architecture, not the Keras code, which has a bug in layer C5). Optimization details are similar to the previous task. With only 44,426 parameters, this LeNet-5 implementation can further improve the accuracy over the MLP to 98.49%.

Sources:

- `https://engmrk.com/lenet-5-a-classic-cnn-architecture/`

1.4 Context Aggregation Networks (CAN)

Code: *can.py*

Description: We again use Keras to implement the architecture given in the exercise. Additionally we insert a 10-node fully-connected layer with softmax activation in the end of the architecture. This layer doesn't really add complexity (since it is only 110 parameters in size), but is required to map the CAN output to a probability vector. Without it my model was not able to make any meaningful predictions. For the *Leaky ReLU* layers we use a slope of $\alpha = 0.3$. With the proposed 32 feature channels in each layer we reach an accuracy of 99.03%, which further improves upon the results of the CNN with even less parameters (31,064). We run additional tests with 16 and 64 feature channels. The results are summarized in table 2 and 3.

#Feature channels	**#parameters**
16	8,680
32	31,064
64	117,304

Table 2: Number of parameters of the CAN by #Feature channels

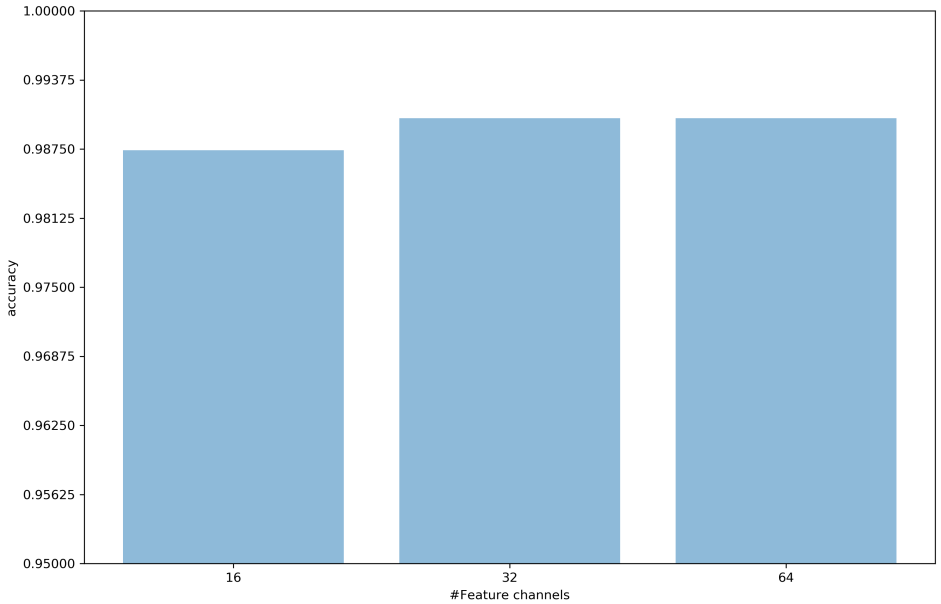

Figure 3: Accuracy of the CAN by #Feature channels