DAT405 Assignment 7 – Group 95

Martin Blom - (8.5 hrs)
Jakob Windt - (8.5 hrs)

March 8, 2023

Question 1

0.1 1.1

The data contains 60,000 images of hand drawn numbers between 0-9. Firstly, the data gets reshaped from an image to an 2D-array representing individual pixels. Then the values get converted to ”float32” which gives us their respective grey-scale number. This number is anywhere between 0-255. If we divide the number by 255, the result ends up being a value between 0-1 that can be later used to train a model. Lastly, the results get categorized into 10 separate classes, which represent the numbers between 0-9.

Question 2

0.2 2.1

There are a total of 4 layers in the network; 3 Dense layers and 1 Flatten layer. The flatten layer doesn't contain any neurons, 2 of the Dense layers have 64 neurons each, and the last Dense layer only has 10 neurons. The first 2 Dense layers use *relu* as their activation functions. Relu which stands for *The Rectified Linear Unit* returns 0 if the input is negative, and returns the input if it's something else. This can be described as y = max(0,x). The reason behind using these activation functions is to ensure all the data is positive to avoid problems. The last Dense layer uses the *softmax* function, which converts a vector of values to a probability distribution between 0-1. This is because we want to keep the data values between 0-1 for fitting and evaluating the data later on.

The total amount of parameter in the network is 55,050 which was determined by using the **model.Summary** function from the Keras library.

The Flatten input layer converts our data from (Batch count, (28*28)) -¿ (Batch count, 1). This means that we go from 28*28 dimensions to 1 dimension. The last Dense output layer converts from (Batch count, 64) -¿ (Batch count, 10) which means we get left with 10 dimensions that represent the numbers between 0-9.

0.3 2.2

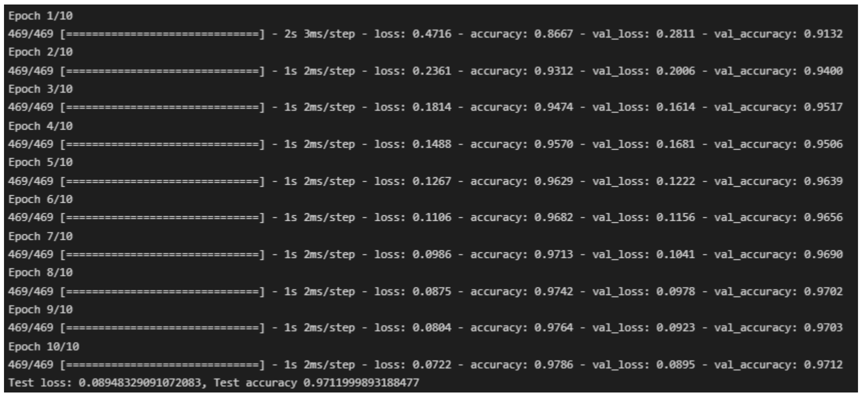
The loss function used in the code is called ”Categorical Cross Entropy”. This function multiplies the

$$f(s)_i = \frac{e^{s_i}}{\sum_j^C e^{s_j}} \quad CE = -\sum_i^C t_i log(f(s)_i)$$

Figure 1: Mathematical formula for loss

value 1 from the correct answer vector with the natural logged computer predicted probability answer which results in the loss probability. The reason why its used is because its designed for when there are more than 2 classes/categorizes of data.

0.4 2.3



```
Epoch 1/10
469/469 [==============================] - 2s 3ms/step - loss: 0.4716 - accuracy: 0.8667 - val_loss: 0.2811 - val_accuracy: 0.9132
Epoch 2/10
469/469 [==============================] - 1s 2ms/step - loss: 0.2361 - accuracy: 0.9312 - val_loss: 0.2006 - val_accuracy: 0.9400
Epoch 3/10
469/469 [==============================] - 1s 2ms/step - loss: 0.1814 - accuracy: 0.9474 - val_loss: 0.1614 - val_accuracy: 0.9517
Epoch 4/10
469/469 [==============================] - 1s 2ms/step - loss: 0.1488 - accuracy: 0.9570 - val_loss: 0.1681 - val_accuracy: 0.9506
Epoch 5/10
469/469 [==============================] - 1s 2ms/step - loss: 0.1267 - accuracy: 0.9629 - val_loss: 0.1222 - val_accuracy: 0.9639
Epoch 6/10
469/469 [==============================] - 1s 2ms/step - loss: 0.1106 - accuracy: 0.9682 - val_loss: 0.1156 - val_accuracy: 0.9656
Epoch 7/10
469/469 [==============================] - 1s 2ms/step - loss: 0.0986 - accuracy: 0.9713 - val_loss: 0.1041 - val_accuracy: 0.9690
Epoch 8/10
469/469 [==============================] - 1s 2ms/step - loss: 0.0875 - accuracy: 0.9742 - val_loss: 0.0978 - val_accuracy: 0.9702
Epoch 9/10
469/469 [==============================] - 1s 2ms/step - loss: 0.0804 - accuracy: 0.9764 - val_loss: 0.0923 - val_accuracy: 0.9703
Epoch 10/10
469/469 [==============================] - 1s 2ms/step - loss: 0.0722 - accuracy: 0.9786 - val_loss: 0.0895 - val_accuracy: 0.9712
Test loss: 0.08948329091072083, Test accuracy 0.9711999893188477
```

Figure 2: Print of accuracy summary for all epochs

0.5 2.4

Using 40 epochs and changing the units in the 2 hidden Dense layers to 500 and 300 respectively we end up with a validation accuracy of 0.9809.

The most probable reason we don’t end up with the same result as Hinton is that there might be hardware limitations. We also don’t know how many layers/neurons/epochs Hinton used in his model. When running our code for the first time it crashed for some unknown reason. Our guess is that either some cache or the RAM was overloaded, which caused VSC to crash.

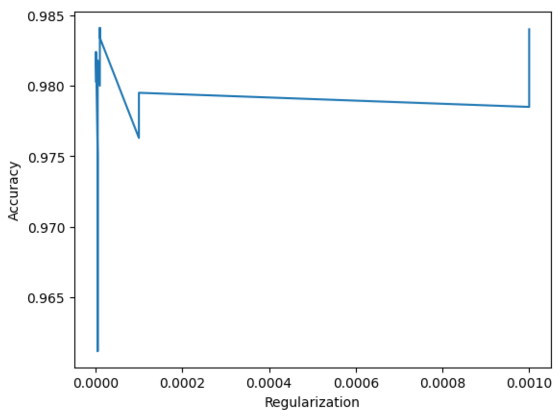


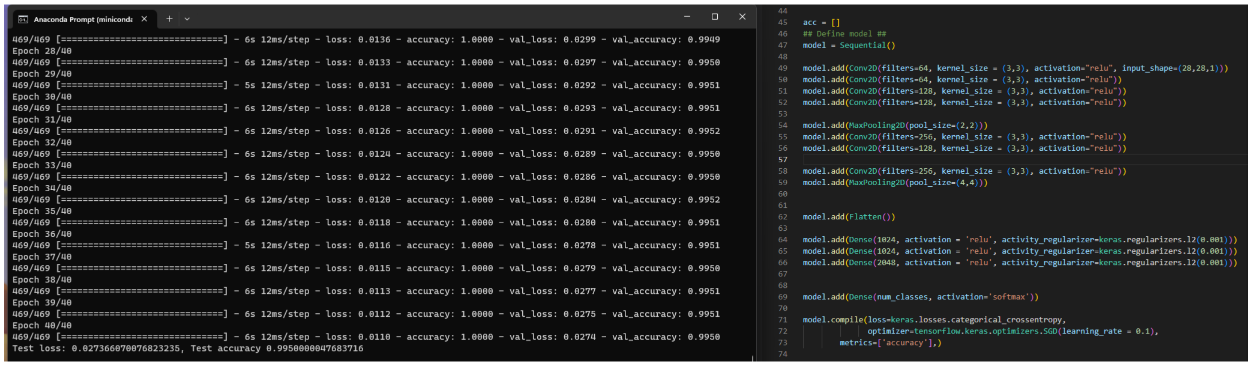
Figure 3: Accuracy vs Regularization factor plot

Question 3

0.6 3.1

We ended up getting 99.5% as our best validated accuracy. This was achieved by a lot of testing that was made possible by changing tensorflow from CPU to GPU calculations. This gave substantial improvements on training time, for ex. our final model takes 24 minutes per epoch in google co-labs but only 5 seconds on our machine configured to use the GPU.

We decided to research on effective sequential models for processing images. We found that using a layer called "MaxPooling2D" gave very good improvements as anticipated. The layer works by taking a "part" of an image at a time and using the maximum value to replace that "part". This is effectively increasing the contrast between parts of the image where its all white and parts with black in it.



```
469/469 [==============================] - 6s 12ms/step - loss: 0.0136 - accuracy: 1.0000 - val_loss: 0.0299 - val_accuracy: 0.9949
Epoch 28/40
469/469 [==============================] - 6s 12ms/step - loss: 0.0133 - accuracy: 1.0000 - val_loss: 0.0297 - val_accuracy: 0.9950
Epoch 29/40
469/469 [==============================] - 5s 12ms/step - loss: 0.0131 - accuracy: 1.0000 - val_loss: 0.0292 - val_accuracy: 0.9951
Epoch 30/40
469/469 [==============================] - 6s 12ms/step - loss: 0.0128 - accuracy: 1.0000 - val_loss: 0.0293 - val_accuracy: 0.9951
Epoch 31/40
469/469 [==============================] - 6s 12ms/step - loss: 0.0126 - accuracy: 1.0000 - val_loss: 0.0291 - val_accuracy: 0.9952
Epoch 32/40
469/469 [==============================] - 6s 12ms/step - loss: 0.0124 - accuracy: 1.0000 - val_loss: 0.0289 - val_accuracy: 0.9950
Epoch 33/40
469/469 [==============================] - 6s 12ms/step - loss: 0.0122 - accuracy: 1.0000 - val_loss: 0.0286 - val_accuracy: 0.9950
Epoch 34/40
469/469 [==============================] - 6s 12ms/step - loss: 0.0120 - accuracy: 1.0000 - val_loss: 0.0284 - val_accuracy: 0.9952
Epoch 35/40
469/469 [==============================] - 6s 12ms/step - loss: 0.0118 - accuracy: 1.0000 - val_loss: 0.0280 - val_accuracy: 0.9951
Epoch 36/40
469/469 [==============================] - 5s 12ms/step - loss: 0.0116 - accuracy: 1.0000 - val_loss: 0.0278 - val_accuracy: 0.9951
Epoch 37/40
469/469 [==============================] - 6s 12ms/step - loss: 0.0115 - accuracy: 1.0000 - val_loss: 0.0279 - val_accuracy: 0.9950
Epoch 38/40
469/469 [==============================] - 6s 12ms/step - loss: 0.0113 - accuracy: 1.0000 - val_loss: 0.0277 - val_accuracy: 0.9951
Epoch 39/40
469/469 [==============================] - 6s 12ms/step - loss: 0.0112 - accuracy: 1.0000 - val_loss: 0.0275 - val_accuracy: 0.9951
Epoch 40/40
469/469 [==============================] - 6s 12ms/step - loss: 0.0110 - accuracy: 1.0000 - val_loss: 0.0274 - val_accuracy: 0.9950
Test loss: 0.027366070076823235, Test accuracy 0.9950000047683716
```

```python

acc = []
## Define model ##
model = Sequential()

model.add(Conv2D(filters=64, kernel_size = (3,3), activation="relu", input_shape=(28,28,1)))
model.add(Conv2D(filters=64, kernel_size = (3,3), activation="relu"))
model.add(Conv2D(filters=128, kernel_size = (3,3), activation="relu"))
model.add(Conv2D(filters=128, kernel_size = (3,3), activation="relu"))

model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Conv2D(filters=256, kernel_size = (3,3), activation="relu"))
model.add(Conv2D(filters=128, kernel_size = (3,3), activation="relu"))

model.add(Conv2D(filters=256, kernel_size = (3,3), activation="relu"))
model.add(MaxPooling2D(pool_size=(4,4)))


model.add(Flatten())

model.add(Dense(1024, activation = 'relu', activity_regularizer=keras.regularizers.l2(0.001)))
model.add(Dense(1024, activation = 'relu', activity_regularizer=keras.regularizers.l2(0.001)))
model.add(Dense(2048, activation = 'relu', activity_regularizer=keras.regularizers.l2(0.001)))


model.add(Dense(num_classes, activation='softmax'))

model.compile(loss=keras.losses.categorical_crossentropy,
                optimizer=tensorflow.keras.optimizers.SGD(learning_rate = 0.1),
        metrics=['accuracy'],)

```

Figure 4: Result and model design

0.7 3.2

The biggest difference between convolutional layers vs. dense layers are how they are connected. Convolutional layers use fewer parameters by forcing input values to share the parameters, while dense layers use linear operation meaning every output is calculated using every input. Convolutional layers also have smaller number of weights per layer, making them better with high dimensional inputs such as images.

References

[1] "Understanding Categorical Cross-Entropy Loss, Binary Cross-Entropy Loss, Softmax Loss, Logistic Loss, Focal Loss and All Those Confusing Names." Understanding Categorical Cross-Entropy Loss, Binary Cross-Entropy Loss, Softmax Loss, Logistic Loss, Focal Loss and All Those Confusing Names, 23 May 2018, https://gombru.github.io/2018/05/23/cross_entropy_loss/.