Deep Learning Frameworks

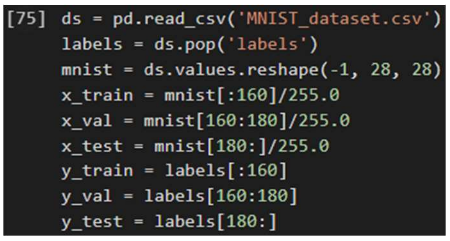
Assignment 03 – report by Frédéric Charon

Setting up the model

In this assignment we learn about deep learning frameworks by using either PyTorch or TensorFlow to create a classifier for the MNIST problem. The MNIST data set contains hand-written digits that are size-normalized and centered in a fixed-size image, the model should be able get these images and detect the digits they are representing. For the task I am choosing to use TensorFlow instead of PyTorch.

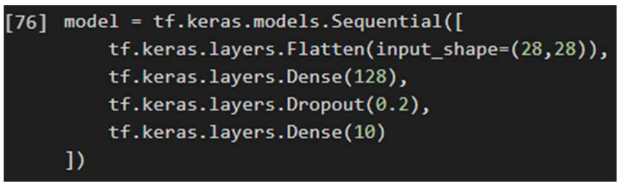
The first step is to upload the MNIST_dataset.csv since we only use a smaller dataset containing 200 images instead of the actual dataset which contains 60000 examples.

Next, we're going to do the pre-processing. We first pop out the labels of the images, containing the class of the digit, which are used as the output for our model. We reshape the values of our dataset to our images format, then we divide the examples into a training, a validation and a test dataset, where 80% of the examples are for training, 10% for validation and the last 10% for testing. The idea of these datasets is, that we use the training dataset to train the model and update its weights etc, where the validation and test are only used to get a prediction without including the learning of the model. The validation is used to optimize the model's hyperparameters – only after we optimized the model, we evaluate the test dataset to see the model's performance for data it hasn't seen before.



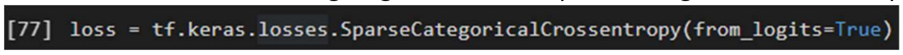
```
[75] ds = pd.read_csv('MNIST_dataset.csv')
     labels = ds.pop('labels')
     mnist = ds.values.reshape(-1, 28, 28)
     x_train = mnist[:160]/255.0
     x_val = mnist[160:180]/255.0
     x_test = mnist[180:]/255.0
     y_train = labels[:160]
     y_val = labels[160:180]
     y_test = labels[180:]
```

In our next step we start to define the model by implementing the different layers it will contain. We start with a Flatten layer, which flattens the input. The next layer is a Dense layer, it works like a perceptron where and we enter the number of nodes. Then we use a Dropout layer, which helps to prevent overfitting by randomly setting inputs to 0 and scaling up the other inputs (so the input sum doesn't change). The final layer is another Dense layer, this is our output layer, so we have to make sure the number of nodes equals the number of outputs we have. There are 10 different outputs (digits from 0 to 9), so we have to add 10 nodes to this layer. For each Dense layer we can also set an activation function.



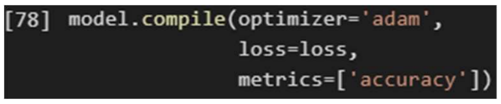
```
[76] model = tf.keras.models.Sequential([
         tf.keras.layers.Flatten(input_shape=(28,28)),
         tf.keras.layers.Dense(128),
         tf.keras.layers.Dropout(0.2),
         tf.keras.layers.Dense(10)
     ])
```

Now we define our loss function, we are going to use the SparseCategoricalCrossentropy() from TensorFlow.



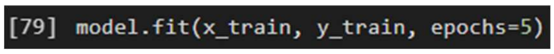
```
[77] loss = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
```

For compiling the model we use the .compile() function. Here we enter our optimizing function, the loss function we just defined and the metric 'accuracy' since that is the feature, we use to see the model's performance. For now, we use the adam function as optimizer.



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

Finally, we use the fit functioning by entering the training input, the training output and the number of epochs to let the model learn from the training data.



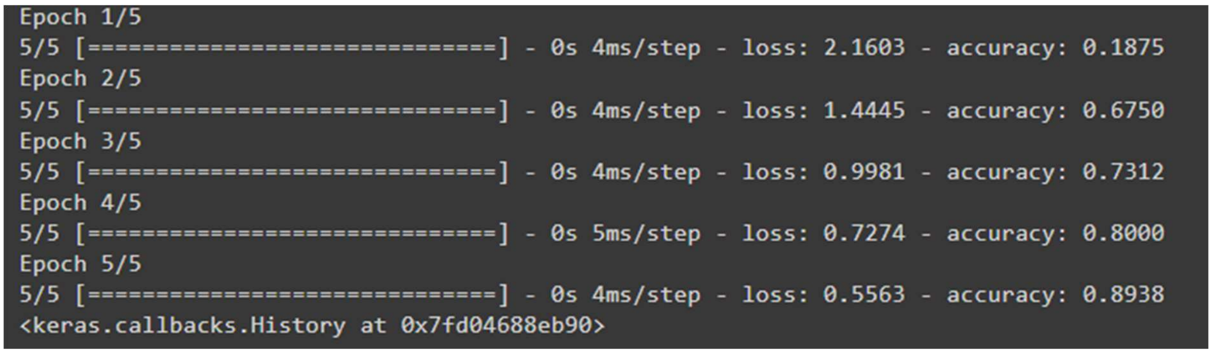
```
[79] model.fit(x_train, y_train, epochs=5)
```

Adjusting the model

With the settings we used above we get an output from the fit() method, showing us the resulting loss value and the accuracy after each epoch. By using these parameters

```
model = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(input_shape=(28,28)),
    tf.keras.layers.Dense(128),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(10)
])
model.compile(optimizer='adam',
              loss=loss,
              metrics=['accuracy'])
model.fit(x_train, y_train, epochs=5)
```

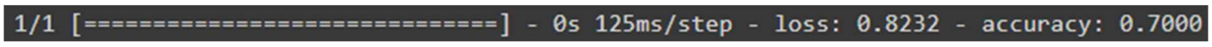
we got this output:



```
Epoch 1/5
5/5 [==============================] - 0s 4ms/step - loss: 2.1603 - accuracy: 0.1875
Epoch 2/5
5/5 [==============================] - 0s 4ms/step - loss: 1.4445 - accuracy: 0.6750
Epoch 3/5
5/5 [==============================] - 0s 4ms/step - loss: 0.9981 - accuracy: 0.7312
Epoch 4/5
5/5 [==============================] - 0s 5ms/step - loss: 0.7274 - accuracy: 0.8000
Epoch 5/5
5/5 [==============================] - 0s 4ms/step - loss: 0.5563 - accuracy: 0.8938
<keras.callbacks.History at 0x7fd04688eb90>
```

Then we compare the result with the results we get from evaluating the validation data

```
predictions = model(x_val).numpy()
model.evaluate(x_val, y_val)
```



```
1/1 [==============================] - 0s 125ms/step - loss: 0.8232 - accuracy: 0.7000
```

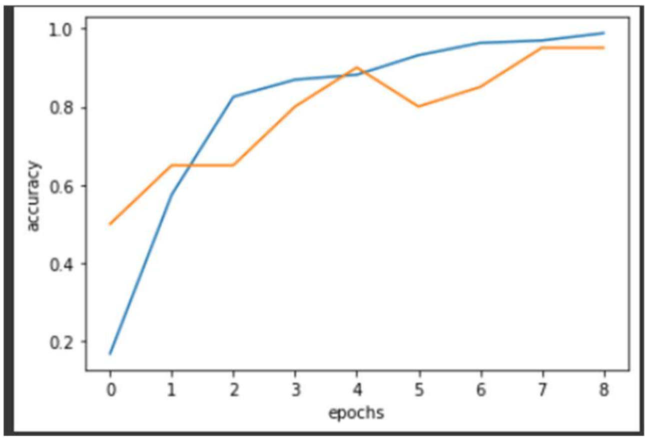
To get a good model we want our loss to be very low and the accuracy to be close to 1, so concerning this the result we got look pretty bad. Now we need to adjust the hyperparameters until we get a better result, these are the final settings I went with:

```
model = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(input_shape=(28,28)),
    tf.keras.layers.Dense(194),
    tf.keras.layers.Dense(100),
    tf.keras.layers.Dropout(0.4),
    tf.keras.layers.Dense(10)
])
model.compile(optimizer='adam',
              loss=loss,
              metrics=['accuracy'])
```

```
K.set_value(model.optimizer.learning_rate, 0.001)
model1 = model.fit(x_train, y_train, validation_data=(x_val, y_val), ep
ochs=9)
```

Which led to these outputs:

```
Epoch 1/9
5/5 [==============================] - 1s 50ms/step - loss: 2.3255 - accuracy: 0.1688 - val_loss: 1.6225 - val_accuracy: 0.5000
Epoch 2/9
5/5 [==============================] - 0s 13ms/step - loss: 1.3206 - accuracy: 0.5750 - val_loss: 1.0435 - val_accuracy: 0.6500
Epoch 3/9
5/5 [==============================] - 0s 18ms/step - loss: 0.7830 - accuracy: 0.8250 - val_loss: 0.8860 - val_accuracy: 0.6500
Epoch 4/9
5/5 [==============================] - 0s 15ms/step - loss: 0.5386 - accuracy: 0.8687 - val_loss: 0.6920 - val_accuracy: 0.8000
Epoch 5/9
5/5 [==============================] - 0s 15ms/step - loss: 0.4159 - accuracy: 0.8813 - val_loss: 0.5344 - val_accuracy: 0.9000
Epoch 6/9
5/5 [==============================] - 0s 16ms/step - loss: 0.2993 - accuracy: 0.9312 - val_loss: 0.5815 - val_accuracy: 0.8000
Epoch 7/9
5/5 [==============================] - 0s 13ms/step - loss: 0.2234 - accuracy: 0.9625 - val_loss: 0.5609 - val_accuracy: 0.8500
Epoch 8/9
5/5 [==============================] - 0s 15ms/step - loss: 0.1857 - accuracy: 0.9688 - val_loss: 0.4782 - val_accuracy: 0.9500
Epoch 9/9
5/5 [==============================] - 0s 15ms/step - loss: 0.1702 - accuracy: 0.9875 - val_loss: 0.4731 - val_accuracy: 0.9500
```



The blue line is the trainings accuracy and orange the validation accuracy

Making predictions:

Now we can make predictions on new images, for example on the test data:

```
predictions = model(x_test).numpy()
pred_test = predictions.argmax(axis=1)
```

out: `array([1, 8, 5, 0, 8, 4, 3, 5, 0, 1, 1, 1, 5, 4, 0, 3, 1, 6, 4, 2])`

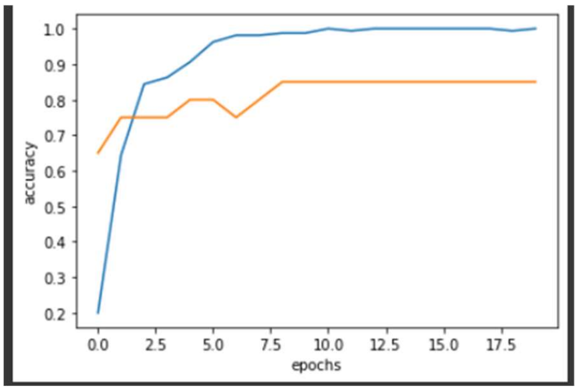
these are the predicted digits for the test data. We can also evaluate the test set to see our model's performance:

```
model.evaluate(x_test, y_test)
```

```
1/1 [==============================] - 0s 32ms/step - loss: 0.6469 - accuracy: 0.8000
```

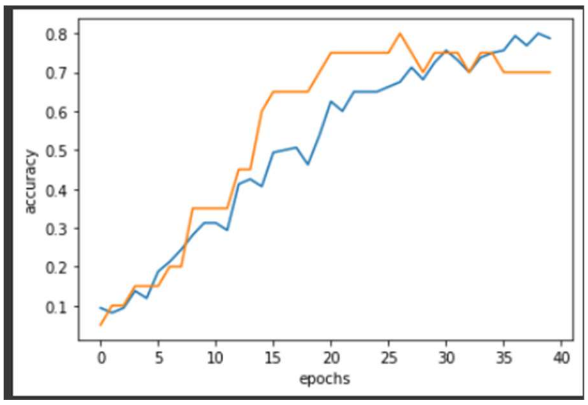
Overfit and underfit

There are two major cases we want to avoid in our model. The first one is overfitting, which means the model is trained too much to just represent the trainings data, so its output shows great result when concerning this data, but new data like validation or testing data are showing bad results. This can for example be shown by adding too many epochs at the learning progress.



In this example we used the same code as above and let it run for 20 epochs instead of 9, as we can see the accuracy is stagnating at 1 which is a sign of overfitting.

The other case is underfitting – here the accuracy takes way too long for reaching a high value. This happens when for example the learning rate is too low, even with using many epochs it takes way too long for the accuracy to go as high as we want it to be.

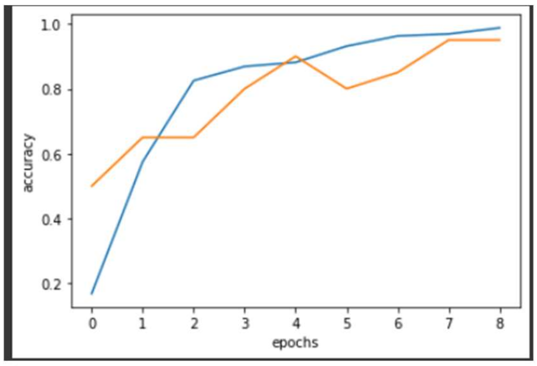


This is an example for underfitting, I reduced the learning rate from 0.001 to 0.00005. As we can see, even after a total of 40 epochs the training accuracy (blue line) is just reaching 80%.

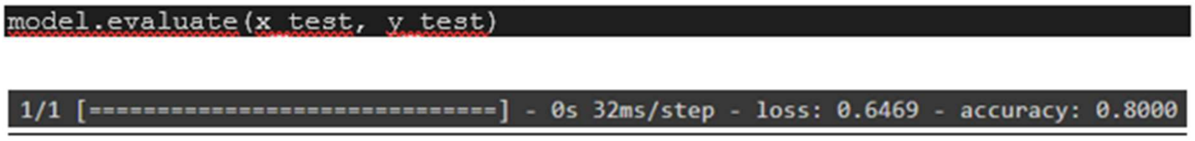
Comparing different optimizer functions

As we learned in the lecture there are different optimizer functions we can use. We will replace the optimizer used for our model to see the differences in the performance.

The first implementation we showed earlier is using the ADAM optimization. These where the result we got:



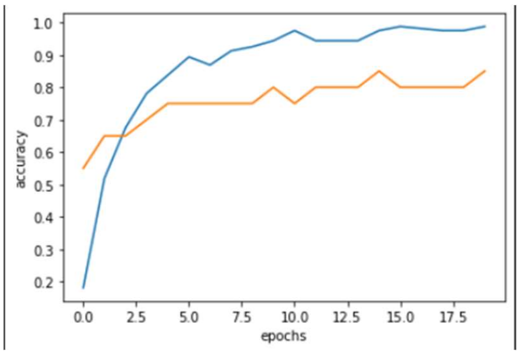
The learning curve looks good, since we adapted the program to have a relatively high performance using ADAM.



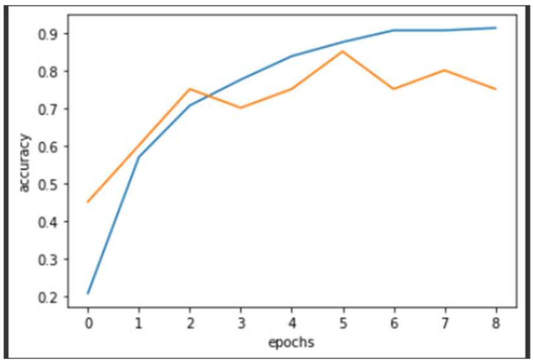
```
model.evaluate(x_test, y_test)
```

```
1/1 [==============================] - 0s 32ms/step - loss: 0.6469 - accuracy: 0.8000
```

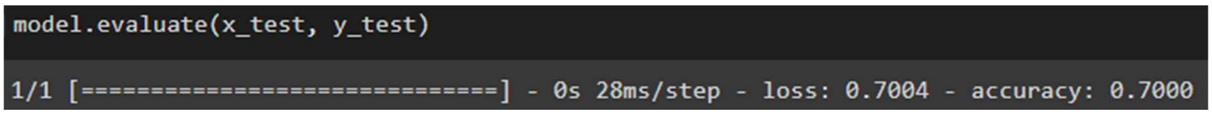
When we replace the ADAM functions with the SGD function, we get this result:



The SGD function it is very reliant on a good learning rate to work properly, so I moved it up 0.05. I also set the epochs up to 20.



This example is using the same hyperparameters as above and uses the SGD function as well, the difference is that we’re adding the momentum parameter. I set it to 0.5 and as we see we get a smoother learning curve than before.



```
model.evaluate(x_test, y_test)
```

```
1/1 [==============================] - 0s 28ms/step - loss: 0.7004 - accuracy: 0.7000
```