group-12-notebook-week7-1

May 11, 2023

Group 12

Anakha Krishnavilasom Gopalakrishnan - 8 hours

Daniel Juster - 8 hours

#DAT405/DIT407 Introduction to Data Science and AI ## 2022-2023, Reading Period 4 ## Assignment 7

Neural Networks using Keras and Tensorflow. Please see the associated document for questions. If you have problems with Keras and Tensorflow on your local installation please make sure they are updated. On Google Colab this notebook runs. *kursiv text*

[ ]:
```
pip install tensorflow
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-
wheels/public/simple/
Requirement already satisfied: tensorflow in /usr/local/lib/python3.10/dist-
packages (2.12.0)
Requirement already satisfied: absl-py>=1.0.0 in /usr/local/lib/python3.10/dist-
packages (from tensorflow) (1.4.0)
Requirement already satisfied: astunparse>=1.6.0 in
/usr/local/lib/python3.10/dist-packages (from tensorflow) (1.6.3)
Requirement already satisfied: flatbuffers>=2.0 in
/usr/local/lib/python3.10/dist-packages (from tensorflow) (23.3.3)
Requirement already satisfied: gast<=0.4.0,>=0.2.1 in
/usr/local/lib/python3.10/dist-packages (from tensorflow) (0.4.0)
Requirement already satisfied: google-pasta>=0.1.1 in
/usr/local/lib/python3.10/dist-packages (from tensorflow) (0.2.0)
Requirement already satisfied: grpcio<2.0,>=1.24.3 in
/usr/local/lib/python3.10/dist-packages (from tensorflow) (1.54.0)
Requirement already satisfied: h5py>=2.9.0 in /usr/local/lib/python3.10/dist-
packages (from tensorflow) (3.8.0)
Requirement already satisfied: jax>=0.3.15 in /usr/local/lib/python3.10/dist-
packages (from tensorflow) (0.4.8)
Requirement already satisfied: keras<2.13,>=2.12.0 in
/usr/local/lib/python3.10/dist-packages (from tensorflow) (2.12.0)
Requirement already satisfied: libclang>=13.0.0 in
/usr/local/lib/python3.10/dist-packages (from tensorflow) (16.0.0)
Requirement already satisfied: numpy<1.24,>=1.22 in
```

```
/usr/local/lib/python3.10/dist-packages (from tensorflow) (1.22.4)
Requirement already satisfied: opt-einsum>=2.3.2 in
/usr/local/lib/python3.10/dist-packages (from tensorflow) (3.3.0)
Requirement already satisfied: packaging in /usr/local/lib/python3.10/dist-
packages (from tensorflow) (23.1)
Requirement already satisfied:
protobuf!=4.21.0,!=4.21.1,!=4.21.2,!=4.21.3,!=4.21.4,!=4.21.5,<5.0.0dev,>=3.20.3
in /usr/local/lib/python3.10/dist-packages (from tensorflow) (3.20.3)
Requirement already satisfied: setuptools in /usr/local/lib/python3.10/dist-
packages (from tensorflow) (67.7.2)
Requirement already satisfied: six>=1.12.0 in /usr/local/lib/python3.10/dist-
packages (from tensorflow) (1.16.0)
Requirement already satisfied: tensorboard<2.13,>=2.12 in
/usr/local/lib/python3.10/dist-packages (from tensorflow) (2.12.2)
Requirement already satisfied: tensorflow-estimator<2.13,>=2.12.0 in
/usr/local/lib/python3.10/dist-packages (from tensorflow) (2.12.0)
Requirement already satisfied: termcolor>=1.1.0 in
/usr/local/lib/python3.10/dist-packages (from tensorflow) (2.3.0)
Requirement already satisfied: typing-extensions>=3.6.6 in
/usr/local/lib/python3.10/dist-packages (from tensorflow) (4.5.0)
Requirement already satisfied: wrapt<1.15,>=1.11.0 in
/usr/local/lib/python3.10/dist-packages (from tensorflow) (1.14.1)
Requirement already satisfied: tensorflow-io-gcs-filesystem>=0.23.1 in
/usr/local/lib/python3.10/dist-packages (from tensorflow) (0.32.0)
Requirement already satisfied: wheel<1.0,>=0.23.0 in
/usr/local/lib/python3.10/dist-packages (from astunparse>=1.6.0->tensorflow)
(0.40.0)
Requirement already satisfied: ml-dtypes>=0.0.3 in
/usr/local/lib/python3.10/dist-packages (from jax>=0.3.15->tensorflow) (0.1.0)
Requirement already satisfied: scipy>=1.7 in /usr/local/lib/python3.10/dist-
packages (from jax>=0.3.15->tensorflow) (1.10.1)
Requirement already satisfied: google-auth<3,>=1.6.3 in
/usr/local/lib/python3.10/dist-packages (from
tensorboard<2.13,>=2.12->tensorflow) (2.17.3)
Requirement already satisfied: google-auth-oauthlib<1.1,>=0.5 in
/usr/local/lib/python3.10/dist-packages (from
tensorboard<2.13,>=2.12->tensorflow) (1.0.0)
Requirement already satisfied: markdown>=2.6.8 in
/usr/local/lib/python3.10/dist-packages (from
tensorboard<2.13,>=2.12->tensorflow) (3.4.3)
Requirement already satisfied: requests<3,>=2.21.0 in
/usr/local/lib/python3.10/dist-packages (from
tensorboard<2.13,>=2.12->tensorflow) (2.27.1)
Requirement already satisfied: tensorboard-data-server<0.8.0,>=0.7.0 in
/usr/local/lib/python3.10/dist-packages (from
tensorboard<2.13,>=2.12->tensorflow) (0.7.0)
Requirement already satisfied: tensorboard-plugin-wit>=1.6.0 in
/usr/local/lib/python3.10/dist-packages (from
```

```
tensorboard<2.13,>=2.12->tensorflow) (1.8.1)
Requirement already satisfied: werkzeug>=1.0.1 in
/usr/local/lib/python3.10/dist-packages (from
tensorboard<2.13,>=2.12->tensorflow) (2.3.0)
Requirement already satisfied: cachetools<6.0,>=2.0.0 in
/usr/local/lib/python3.10/dist-packages (from google-
auth<3,>=1.6.3->tensorboard<2.13,>=2.12->tensorflow) (5.3.0)
Requirement already satisfied: pyasn1-modules>=0.2.1 in
/usr/local/lib/python3.10/dist-packages (from google-
auth<3,>=1.6.3->tensorboard<2.13,>=2.12->tensorflow) (0.3.0)
Requirement already satisfied: rsa<5,>=3.1.4 in /usr/local/lib/python3.10/dist-
packages (from google-auth<3,>=1.6.3->tensorboard<2.13,>=2.12->tensorflow) (4.9)
Requirement already satisfied: requests-oauthlib>=0.7.0 in
/usr/local/lib/python3.10/dist-packages (from google-auth-
oauthlib<1.1,>=0.5->tensorboard<2.13,>=2.12->tensorflow) (1.3.1)
Requirement already satisfied: urllib3<1.27,>=1.21.1 in
/usr/local/lib/python3.10/dist-packages (from
requests<3,>=2.21.0->tensorboard<2.13,>=2.12->tensorflow) (1.26.15)
Requirement already satisfied: certifi>=2017.4.17 in
/usr/local/lib/python3.10/dist-packages (from
requests<3,>=2.21.0->tensorboard<2.13,>=2.12->tensorflow) (2022.12.7)
Requirement already satisfied: charset-normalizer~=2.0.0 in
/usr/local/lib/python3.10/dist-packages (from
requests<3,>=2.21.0->tensorboard<2.13,>=2.12->tensorflow) (2.0.12)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-
packages (from requests<3,>=2.21.0->tensorboard<2.13,>=2.12->tensorflow) (3.4)
Requirement already satisfied: MarkupSafe>=2.1.1 in
/usr/local/lib/python3.10/dist-packages (from
werkzeug>=1.0.1->tensorboard<2.13,>=2.12->tensorflow) (2.1.2)
Requirement already satisfied: pyasn1<0.6.0,>=0.4.6 in
/usr/local/lib/python3.10/dist-packages (from pyasn1-modules>=0.2.1->google-
auth<3,>=1.6.3->tensorboard<2.13,>=2.12->tensorflow) (0.5.0)
Requirement already satisfied: oauthlib>=3.0.0 in
/usr/local/lib/python3.10/dist-packages (from requests-oauthlib>=0.7.0->google-
auth-oauthlib<1.1,>=0.5->tensorboard<2.13,>=2.12->tensorflow) (3.2.2)
```

[ ]:
```python
# imports
from __future__ import print_function
import keras
from keras import utils as np_utils
import tensorflow
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten
from keras.layers import Conv2D, MaxPooling2D
from keras import backend as K
import tensorflow as tf
```

```
from matplotlib import pyplot as plt
```

[ ]:
```
# Hyper-parameters data-loading and formatting

batch_size = 128
num_classes = 10
epochs = 10
img_rows, img_cols = 28, 28

(x_train, lbl_train), (x_test, lbl_test) = mnist.load_data()

if K.image_data_format() == 'channels_first':
    x_train = x_train.reshape(x_train.shape[0], 1, img_rows, img_cols)
    x_test = x_test.reshape(x_test.shape[0], 1, img_rows, img_cols)
    input_shape = (1, img_rows, img_cols)
else:
    x_train = x_train.reshape(x_train.shape[0], img_rows, img_cols, 1)
    x_test = x_test.reshape(x_test.shape[0], img_rows, img_cols, 1)
    input_shape = (img_rows, img_cols, 1)
```

Preprocessing

[ ]:
```
x_train = x_train.astype('float32') # Convert training data type to float32 so,␣
 ↪in the next step, we can use "/".
x_test = x_test.astype('float32')   # Convert test data type to float32 so, in␣
 ↪the next step, we can use "/".

x_train /= 255 # 0-255 is a lot of versions of grey, so make it "boolean" by␣
 ↪dividing with 255. Now it is white or black only.
x_test /= 255  # 0-255 is a lot of versions of grey, so make it "boolean" by␣
 ↪dividing with 255. Now it is white or black only.

# "to_categorical" creates a matrix with only zeros and one 1 that are unique.␣
 ↪These are called "one-hot encoding".
# num_classes is already set to 10 (0-9), so we will get unique 10 matrixes.
# We do this for lbl_train and lbl_test
y_train = keras.utils.np_utils.to_categorical(lbl_train, num_classes)
y_test = keras.utils.np_utils.to_categorical(lbl_test, num_classes)
```

[ ]:
```
## Define model ##
model = Sequential()

model.add(Flatten())
model.add(Dense(64, activation = 'relu'))
model.add(Dense(64, activation = 'relu'))
model.add(Dense(num_classes, activation='softmax'))
```

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

fit_info = model.fit(x_train, y_train,
           batch_size=batch_size,
           epochs= epochs,
           verbose=1,
           validation_data=(x_test, y_test))
score = model.evaluate(x_test, y_test, verbose=0)
print('Test loss: {}, Test accuracy {}'.format(score[0], score[1]))
```

```
Epoch 1/10
469/469 [==============================] - 7s 12ms/step - loss: 0.4714 -
accuracy: 0.8671 - val_loss: 0.2524 - val_accuracy: 0.9278
Epoch 2/10
469/469 [==============================] - 5s 10ms/step - loss: 0.2231 -
accuracy: 0.9359 - val_loss: 0.1903 - val_accuracy: 0.9434
Epoch 3/10
469/469 [==============================] - 5s 10ms/step - loss: 0.1722 -
accuracy: 0.9496 - val_loss: 0.1555 - val_accuracy: 0.9525
Epoch 4/10
469/469 [==============================] - 6s 12ms/step - loss: 0.1434 -
accuracy: 0.9577 - val_loss: 0.1352 - val_accuracy: 0.9582
Epoch 5/10
469/469 [==============================] - 4s 9ms/step - loss: 0.1225 -
accuracy: 0.9639 - val_loss: 0.1166 - val_accuracy: 0.9656
Epoch 6/10
469/469 [==============================] - 5s 11ms/step - loss: 0.1073 -
accuracy: 0.9686 - val_loss: 0.1065 - val_accuracy: 0.9682
Epoch 7/10
469/469 [==============================] - 5s 11ms/step - loss: 0.0946 -
accuracy: 0.9716 - val_loss: 0.1219 - val_accuracy: 0.9639
Epoch 8/10
469/469 [==============================] - 3s 6ms/step - loss: 0.0854 -
accuracy: 0.9748 - val_loss: 0.0974 - val_accuracy: 0.9687
Epoch 9/10
469/469 [==============================] - 2s 5ms/step - loss: 0.0771 -
accuracy: 0.9771 - val_loss: 0.0985 - val_accuracy: 0.9695
Epoch 10/10
469/469 [==============================] - 3s 5ms/step - loss: 0.0705 -
accuracy: 0.9793 - val_loss: 0.0837 - val_accuracy: 0.9746
Test loss: 0.0837351605296135, Test accuracy 0.9746000170707703
```

0.1 1 Pre-processing

1.1. Explain the data pre-processing highlighted in the notebook

1.1 Answer:

See comments above...

0.2 2 Network model, training, and changing hyper-parameters

2.1.1 How many layers does the network in the notebook have?

2.1.2 How many neurons does each layer have?

2.1.3 What activation functions?

2.1.4 and why are these appropriate for this application?

2.1.5 What is the total number of parameters for the network?

2.1.6 Why do the input and output layers have the dimensions they have?

Answers:

See comments ad text below.

2.1.1 How many layers does the network in the notebook have?

We know for sure that we have 3 types layers: Input, hidden and output. We can create e.g 2 hidden layers with the first one taking a input shape of 28x28 = 784 elements and connecting them all together with Dense. Sequential is good for image classification

2.1.2 How many neurons does each layer have?

The batchsize is set to 128, so we use 64 neurons for each hiden layer.

2.1.3 What activation functions?

2.1.4 and why are these appropriate for this application?

We use relu for the hidden layers to active when input is "1",
3and softmax for the output layer to activate probability that altogether sums up to 1.

2.1.5 What is the total number of parameters for the network?

Using the model.summary(), we get more information, including the total numbers of parameters (weights and bias) = 55050

2.1.6 Why do the input and output layers have the dimensions they have?

The input dimension is 28x28 as this is the number of pixels, and the output is 10, equal to number of digits.

[ ]:
```
model = Sequential()
model.add(Flatten(input_shape=(28,28)))
model.add(Dense(64, activation='relu')) # Hidden layer 1
model.add(Dense(64, activation='relu'))                     # Hidden layer 2
model.add(Dense(10, activation='softmax'))                  # Output layer

# print the summary of the model
model.summary()
```

```
Model: "sequential_15"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten_15 (Flatten)        (None, 784)               0

 dense_45 (Dense)            (None, 64)                50240

 dense_46 (Dense)            (None, 64)                4160

 dense_47 (Dense)            (None, 10)                650

=================================================================
Total params: 55,050
Trainable params: 55,050
Non-trainable params: 0
_________________________________________________________________
```

2.2.1 What loss function is used to train the network?

2.2.1 Answer:

Using softmax on output layer indicates that we should use the default loss-function: categorical_crossentropy. It is designed for one-hot encoding.

2.2.2 What is the functional form (a mathematical expression) of the loss function?

Ref. https://gombru.github.io/2018/05/23/cross_entropy_loss/

2.2.3 and how should we interpret it?

2.2.3 Answer:

In our case, Si = the softmax function = p i range from 1-10 ti is the one hot encoded value (0 or 1)

So, for a 3, we get

losscse = - (0*log(p1) + 0*log(p2) + 1*log(p3) + 0*log(p4) +... 0*log(p10))

2.2.4 Why is it appropriate for the problem at hand?

2.2.4 Answer:

This will have a bigger impact on bigger errors so the learning will improve.

2.3. Train the network for 10 epochs and plot the training and validation accuracy for each epoch.

[ ]:
```python
model.compile(optimizer='sgd',
              loss='binary_crossentropy',
              metrics=['accuracy'])
```

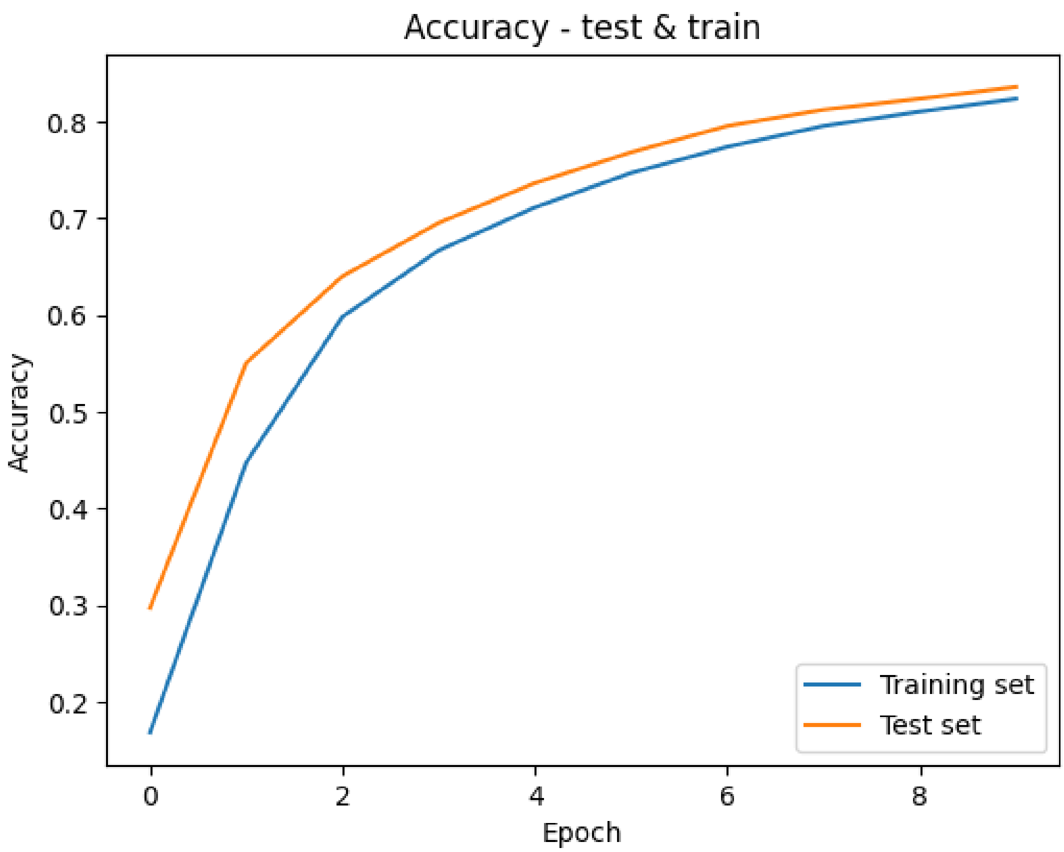
[ ]:
```
batch_size = 128
epochs = 10


history = model.fit(x_train, y_train,
          batch_size = batch_size,
          epochs = epochs,
          verbose = 1, # yes, print progress bar. This takes time....
          validation_data=(x_test, y_test))
```

```
Epoch 1/10
469/469 [==============================] - 4s 6ms/step - loss: 0.4141 -
accuracy: 0.1683 - val_loss: 0.3190 - val_accuracy: 0.2971
Epoch 2/10
469/469 [==============================] - 5s 10ms/step - loss: 0.3066 -
accuracy: 0.4475 - val_loss: 0.2927 - val_accuracy: 0.5506
Epoch 3/10
469/469 [==============================] - 6s 12ms/step - loss: 0.2804 -
accuracy: 0.5983 - val_loss: 0.2657 - val_accuracy: 0.6402
Epoch 4/10
469/469 [==============================] - 5s 10ms/step - loss: 0.2539 -
accuracy: 0.6669 - val_loss: 0.2389 - val_accuracy: 0.6954
Epoch 5/10
469/469 [==============================] - 4s 9ms/step - loss: 0.2280 -
accuracy: 0.7114 - val_loss: 0.2134 - val_accuracy: 0.7366
Epoch 6/10
469/469 [==============================] - 6s 12ms/step - loss: 0.2038 -
accuracy: 0.7471 - val_loss: 0.1904 - val_accuracy: 0.7684
Epoch 7/10
469/469 [==============================] - 4s 9ms/step - loss: 0.1828 -
accuracy: 0.7740 - val_loss: 0.1712 - val_accuracy: 0.7956
Epoch 8/10
469/469 [==============================] - 3s 7ms/step - loss: 0.1658 -
accuracy: 0.7956 - val_loss: 0.1561 - val_accuracy: 0.8125
Epoch 9/10
469/469 [==============================] - 4s 9ms/step - loss: 0.1524 -
accuracy: 0.8105 - val_loss: 0.1441 - val_accuracy: 0.8238
Epoch 10/10
469/469 [==============================] - 3s 6ms/step - loss: 0.1418 -
accuracy: 0.8237 - val_loss: 0.1345 - val_accuracy: 0.8360
```

[ ]:
```
print("Accuracy:",model.evaluate(x_test, y_test)[1])
```

```
313/313 [==============================] - 1s 2ms/step - loss: 0.1345 -
accuracy: 0.8360
Accuracy: 0.8360000252723694
```

```
[ ]: plt.plot(history.history['accuracy'])
     plt.plot(history.history['val_accuracy'])
     plt.title('Accuracy - test & train')
     plt.ylabel('Accuracy')
     plt.xlabel('Epoch')
     plt.legend(['Training set', 'Test set'], loc='lower right')
     plt.show()
```



2.4 Aswers

See commets ad text below.

```
[ ]: # [2.4. Update the model to implement a three-layer neural network
     # where the hidden layers have 500 and 300 hidden units respectively.
     # -------------------------------------------------------------------

     model = Sequential()
     model.add(Flatten(input_shape=(28,28)))
     model.add(Dense(500, activation='relu')) # Hidden layer 1
     model.add(Dense(300, activation='relu'))                    # Hidden layer 2
     model.add(Dense(10, activation='softmax'))                  # Output layer
```

```
# print the summary of the model
model.summary()
```

```
Model: "sequential_16"

_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten_16 (Flatten)        (None, 784)               0

 dense_48 (Dense)            (None, 500)               392500

 dense_49 (Dense)            (None, 300)               150300

 dense_50 (Dense)            (None, 10)                3010

=================================================================
Total params: 545,810
Trainable params: 545,810
Non-trainable params: 0
_________________________________________________________________
```

[ ]:
```
# 2.4 Train for 40 epochs.
#-----------------------



epochs = 40

model.compile(optimizer='sgd',
              loss='binary_crossentropy',
              metrics=['accuracy'])

history = model.fit(x_train, y_train,
          batch_size= batch_size,
          epochs=epochs,
          verbose=1, # yes, print progress bar. This takes time....
          validation_data=(x_test, y_test))

print("Accuracy:",model.evaluate(x_test, y_test)[1])
```

```
Epoch 1/40
469/469 [==============================] - 9s 18ms/step - loss: 0.3654 -
accuracy: 0.3306 - val_loss: 0.2954 - val_accuracy: 0.5842
Epoch 2/40
469/469 [==============================] - 7s 14ms/step - loss: 0.2758 -
accuracy: 0.6663 - val_loss: 0.2531 - val_accuracy: 0.7340
Epoch 3/40
469/469 [==============================] - 8s 17ms/step - loss: 0.2357 -
```

```
accuracy: 0.7476 - val_loss: 0.2142 - val_accuracy: 0.7812
Epoch 4/40
469/469 [==============================] - 6s 14ms/step - loss: 0.2008 -
accuracy: 0.7835 - val_loss: 0.1829 - val_accuracy: 0.8056
Epoch 5/40
469/469 [==============================] - 8s 17ms/step - loss: 0.1742 -
accuracy: 0.8057 - val_loss: 0.1601 - val_accuracy: 0.8203
Epoch 6/40
469/469 [==============================] - 7s 14ms/step - loss: 0.1550 -
accuracy: 0.8208 - val_loss: 0.1438 - val_accuracy: 0.8363
Epoch 7/40
469/469 [==============================] - 8s 17ms/step - loss: 0.1409 -
accuracy: 0.8329 - val_loss: 0.1315 - val_accuracy: 0.8477
Epoch 8/40
469/469 [==============================] - 6s 13ms/step - loss: 0.1302 -
accuracy: 0.8432 - val_loss: 0.1220 - val_accuracy: 0.8569
Epoch 9/40
469/469 [==============================] - 7s 15ms/step - loss: 0.1217 -
accuracy: 0.8511 - val_loss: 0.1144 - val_accuracy: 0.8657
Epoch 10/40
469/469 [==============================] - 7s 15ms/step - loss: 0.1147 -
accuracy: 0.8576 - val_loss: 0.1080 - val_accuracy: 0.8706
Epoch 11/40
469/469 [==============================] - 7s 15ms/step - loss: 0.1089 -
accuracy: 0.8641 - val_loss: 0.1026 - val_accuracy: 0.8762
Epoch 12/40
469/469 [==============================] - 8s 16ms/step - loss: 0.1039 -
accuracy: 0.8693 - val_loss: 0.0980 - val_accuracy: 0.8787
Epoch 13/40
469/469 [==============================] - 6s 14ms/step - loss: 0.0997 -
accuracy: 0.8733 - val_loss: 0.0940 - val_accuracy: 0.8825
Epoch 14/40
469/469 [==============================] - 7s 15ms/step - loss: 0.0959 -
accuracy: 0.8775 - val_loss: 0.0906 - val_accuracy: 0.8855
Epoch 15/40
469/469 [==============================] - 6s 13ms/step - loss: 0.0926 -
accuracy: 0.8810 - val_loss: 0.0875 - val_accuracy: 0.8886
Epoch 16/40
469/469 [==============================] - 7s 15ms/step - loss: 0.0897 -
accuracy: 0.8836 - val_loss: 0.0847 - val_accuracy: 0.8916
Epoch 17/40
469/469 [==============================] - 8s 16ms/step - loss: 0.0871 -
accuracy: 0.8862 - val_loss: 0.0823 - val_accuracy: 0.8930
Epoch 18/40
469/469 [==============================] - 7s 14ms/step - loss: 0.0848 -
accuracy: 0.8882 - val_loss: 0.0801 - val_accuracy: 0.8953
Epoch 19/40
469/469 [==============================] - 7s 15ms/step - loss: 0.0827 -
```

```
accuracy: 0.8902 - val_loss: 0.0781 - val_accuracy: 0.8973
Epoch 20/40
469/469 [==============================] - 7s 14ms/step - loss: 0.0808 -
accuracy: 0.8923 - val_loss: 0.0764 - val_accuracy: 0.8995
Epoch 21/40
469/469 [==============================] - 7s 15ms/step - loss: 0.0790 -
accuracy: 0.8941 - val_loss: 0.0747 - val_accuracy: 0.9014
Epoch 22/40
469/469 [==============================] - 6s 14ms/step - loss: 0.0774 -
accuracy: 0.8954 - val_loss: 0.0732 - val_accuracy: 0.9022
Epoch 23/40
469/469 [==============================] - 7s 15ms/step - loss: 0.0760 -
accuracy: 0.8969 - val_loss: 0.0719 - val_accuracy: 0.9044
Epoch 24/40
469/469 [==============================] - 7s 15ms/step - loss: 0.0746 -
accuracy: 0.8988 - val_loss: 0.0706 - val_accuracy: 0.9064
Epoch 25/40
469/469 [==============================] - 7s 15ms/step - loss: 0.0733 -
accuracy: 0.9002 - val_loss: 0.0694 - val_accuracy: 0.9070
Epoch 26/40
469/469 [==============================] - 7s 15ms/step - loss: 0.0721 -
accuracy: 0.9013 - val_loss: 0.0683 - val_accuracy: 0.9089
Epoch 27/40
469/469 [==============================] - 7s 15ms/step - loss: 0.0710 -
accuracy: 0.9026 - val_loss: 0.0672 - val_accuracy: 0.9098
Epoch 28/40
469/469 [==============================] - 8s 16ms/step - loss: 0.0700 -
accuracy: 0.9039 - val_loss: 0.0662 - val_accuracy: 0.9109
Epoch 29/40
469/469 [==============================] - 7s 15ms/step - loss: 0.0690 -
accuracy: 0.9047 - val_loss: 0.0654 - val_accuracy: 0.9119
Epoch 30/40
469/469 [==============================] - 7s 14ms/step - loss: 0.0680 -
accuracy: 0.9057 - val_loss: 0.0645 - val_accuracy: 0.9126
Epoch 31/40
469/469 [==============================] - 8s 16ms/step - loss: 0.0672 -
accuracy: 0.9070 - val_loss: 0.0637 - val_accuracy: 0.9136
Epoch 32/40
469/469 [==============================] - 7s 14ms/step - loss: 0.0663 -
accuracy: 0.9078 - val_loss: 0.0629 - val_accuracy: 0.9145
Epoch 33/40
469/469 [==============================] - 8s 17ms/step - loss: 0.0655 -
accuracy: 0.9086 - val_loss: 0.0621 - val_accuracy: 0.9156
Epoch 34/40
469/469 [==============================] - 7s 14ms/step - loss: 0.0647 -
accuracy: 0.9097 - val_loss: 0.0614 - val_accuracy: 0.9155
Epoch 35/40
469/469 [==============================] - 8s 17ms/step - loss: 0.0640 -
```

```
accuracy: 0.9104 - val_loss: 0.0608 - val_accuracy: 0.9169
Epoch 36/40
469/469 [==============================] - 7s 16ms/step - loss: 0.0633 -
accuracy: 0.9112 - val_loss: 0.0601 - val_accuracy: 0.9171
Epoch 37/40
469/469 [==============================] - 7s 14ms/step - loss: 0.0626 -
accuracy: 0.9117 - val_loss: 0.0595 - val_accuracy: 0.9185
Epoch 38/40
469/469 [==============================] - 8s 16ms/step - loss: 0.0619 -
accuracy: 0.9127 - val_loss: 0.0589 - val_accuracy: 0.9185
Epoch 39/40
469/469 [==============================] - 7s 14ms/step - loss: 0.0613 -
accuracy: 0.9135 - val_loss: 0.0583 - val_accuracy: 0.9195
Epoch 40/40
469/469 [==============================] - 8s 16ms/step - loss: 0.0607 -
accuracy: 0.9143 - val_loss: 0.0578 - val_accuracy: 0.9205
313/313 [==============================] - 1s 5ms/step - loss: 0.0578 -
accuracy: 0.9205
Accuracy: 0.9204999804496765
```

2.4 What is the best validation accuracy you can achieve?

The best is 0.9205

[ ]:
```
# 2.4 Implement weight decay on hidden units and train and select 5␣
↪regularization factors from 0.000001 to 0.001.
# Train 3 replicates networks for each regularization factor.

from keras import regularizers
import numpy as np

# Set some numbers
batch_size=32
epochs = 10 # We elaborated with smaler numbers since the calculation took a␣
↪long time. We used 1 this time.
reg_factors = np.linspace(0.000001, 0.001, 5)
replicas = 3

accuracies = {}
for l2_factor in reg_factors:
    regularizer = regularizers.l2(l2_factor)
    print('Regularization factor:', l2_factor)
    accuracies[l2_factor] = []
    for replica in range(replicas):
        print ({replica})
        model = Sequential()
        model.add(Flatten(input_shape=(28,28)))
```

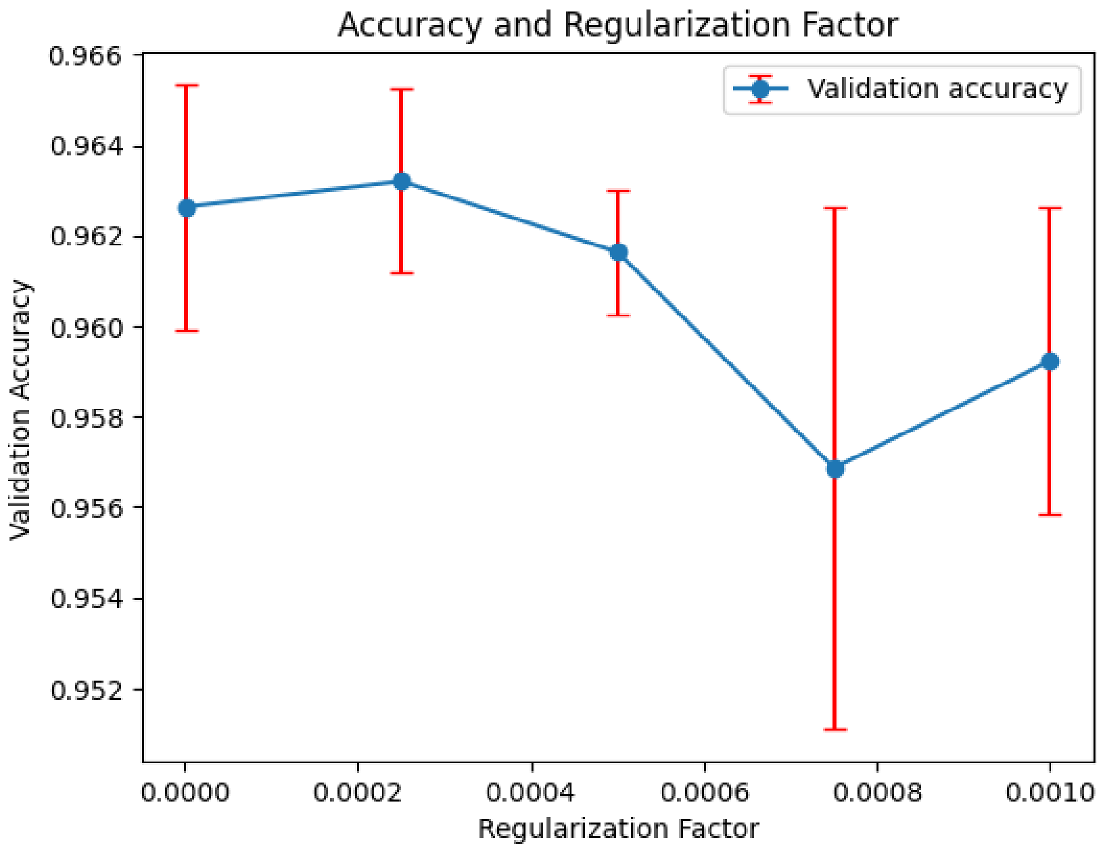
```
        model.add(Dense(500, activation='relu',␣
↪kernel_regularizer=regularizer)) # Hidden layer 1
        model.add(Dense(300, activation='relu'))                    # Hidden␣
↪layer 2
        model.add(Dense(10, activation='softmax'))                  # Output␣
↪layer

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

        fit = model.fit(x_train, y_train,
                        batch_size=batch_size,
                        epochs=epochs,
                        verbose=1, # yes, print progress bar. This takes time...
↪.
                        validation_data=(x_test, y_test))

        val_accuracy = fit.history['val_accuracy'][-1]  # take the last␣
↪validation accuracy
        accuracies[l2_factor].append(val_accuracy)
```

```
Regularization factor: 1e-06
{0}
1875/1875 [==============================] - 19s 10ms/step - loss: 0.2472 -
accuracy: 0.9264 - val_loss: 0.1306 - val_accuracy: 0.9595
{1}
1875/1875 [==============================] - 16s 8ms/step - loss: 0.2484 -
accuracy: 0.9269 - val_loss: 0.1200 - val_accuracy: 0.9623
{2}
1875/1875 [==============================] - 16s 8ms/step - loss: 0.2470 -
accuracy: 0.9259 - val_loss: 0.1173 - val_accuracy: 0.9661
Regularization factor: 0.00025075000000000005
{0}
1875/1875 [==============================] - 16s 8ms/step - loss: 0.3973 -
accuracy: 0.9260 - val_loss: 0.2649 - val_accuracy: 0.9637
{1}
1875/1875 [==============================] - 17s 9ms/step - loss: 0.3937 -
accuracy: 0.9265 - val_loss: 0.2676 - val_accuracy: 0.9605
{2}
1875/1875 [==============================] - 17s 9ms/step - loss: 0.3945 -
accuracy: 0.9256 - val_loss: 0.2501 - val_accuracy: 0.9654
Regularization factor: 0.0005005000000000001
{0}
1875/1875 [==============================] - 18s 9ms/step - loss: 0.5163 -
accuracy: 0.9258 - val_loss: 0.3589 - val_accuracy: 0.9612
```

```
{1}
1875/1875 [==============================] - 15s 8ms/step - loss: 0.5149 -
accuracy: 0.9260 - val_loss: 0.3556 - val_accuracy: 0.9602
{2}
1875/1875 [==============================] - 15s 8ms/step - loss: 0.5158 -
accuracy: 0.9265 - val_loss: 0.3548 - val_accuracy: 0.9635
Regularization factor: 0.0007502500000000002
{0}
1875/1875 [==============================] - 16s 8ms/step - loss: 0.6207 -
accuracy: 0.9273 - val_loss: 0.4189 - val_accuracy: 0.9626
{1}
1875/1875 [==============================] - 16s 8ms/step - loss: 0.6219 -
accuracy: 0.9250 - val_loss: 0.4262 - val_accuracy: 0.9590
{2}
1875/1875 [==============================] - 16s 8ms/step - loss: 0.6182 -
accuracy: 0.9265 - val_loss: 0.4499 - val_accuracy: 0.9490
Regularization factor: 0.001
{0}
1875/1875 [==============================] - 16s 8ms/step - loss: 0.7059 -
accuracy: 0.9276 - val_loss: 0.4672 - val_accuracy: 0.9574
{1}
1875/1875 [==============================] - 16s 8ms/step - loss: 0.7053 -
accuracy: 0.9274 - val_loss: 0.4589 - val_accuracy: 0.9563
{2}
1875/1875 [==============================] - 15s 8ms/step - loss: 0.7090 -
accuracy: 0.9239 - val_loss: 0.4495 - val_accuracy: 0.9640
```

[ ]:

```python
# Plot-time
import matplotlib.pyplot as plt

# calculate mean and std of validation accuracies for each regularization factor
mean_acc = [np.mean(accs) for accs in accuracies.values()]
std_acc = [np.std(accs) for accs in accuracies.values()]

# Use error-bars
plt.errorbar(reg_factors, mean_acc, yerr=std_acc, label="Validation accuracy",
 fmt='-o', ecolor='r', capsize=4)

plt.title("Accuracy and Regularization Factor")
plt.xlabel("Regularization Factor")
plt.ylabel("Validation Accuracy")
plt.legend()
plt.show()
```



[ ]:
```python
# 2.4 How close do you get to Hintons result?

# Round values in mean_acc to 4 decimals and pick the biggest(max). Calculate %
max_acc = max(round(acc, 4) for acc in mean_acc)
hinton_max = 0.9847
compared_proc = 100* (hinton_max-max_acc)/hinton_max


print(f'Max accuracy: {max_acc}')
print(f'Max accuracy according to Hinton: {hinton_max}')
print(f'Accuracy compared to Hinton: {hinton_max-max_acc:.4f}')
print(f'Accuracy compared to Hinton %: {compared_proc:.2f}')
```

```
Max accuracy: 0.9632
Max accuracy according to Hinton: 0.9847
Accuracy compared to Hinton: 0.0215
Accuracy compared to Hinton %: 2.18
```

2.4 Final reflection

If you do not get the same results, what factors may influence this? (hint: What information is *not given by Hinton on the MNIST database that may influence Model training).*

Answer:

The result is not the same.

We do know all details regarding Hintons setup, so we can only speculate: - Number of hidden layers might be different. More might give better accuracy, but also might be overfitting only. - Increasing the batch and number of hidden units per hidden layer might give us a better accuracy - Hinton might have used a different activation function. Perhaps relu/softmax isn't the best. - Same with optimization algorithm. Is our selected SGD the best? - More epochs might have an impact.

##3 Convolutional layers

3.1. Design a model that makes use of at least one convolutional layer – how performant a model can you get? – According to the MNIST database it should be possible reach to 99% accuracy on the validation data. If you choose to use any layers apart from the convolutional layers and layers that you used in previous questions, you must describe what they do. If you do not reach 99% accuracy, report your best performance, and explain your attempts and thought process.

3.2. Discuss the differences and potential benefits of using convolutional layers over fully connected ones for the application?

[ ]:
```
# 3.1. Design a model that makes use of at least one convolutional layer

batch_size = 32
epochs = 40 # We elaborated with smaler numbers since the calculation took a␣
 ↪long time. We used 4 this time.

model = Sequential()

# Adding more layers, using Conv2D ad maxpooling2D
# For Conv2D we test with 32 filters and 3x3 kernel size
# For maxpooling2D we downsize to a pool = 2x2. This will reduce the␣
 ↪computational time
# 2 of each
model.add(Conv2D(32,kernel_size=(3,3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2,2),strides=2))
model.add(Conv2D(32,kernel_size=(3,3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2,2),strides=2))

# Now, continue with same layers as earlier.
model.add(Flatten(input_shape=(28,28)))
model.add(Dense(64, activation='relu'))                        # Hidden layer 1
model.add(Dense(64, activation='relu'))                         # Hidden layer 2
model.add(Dense(10, activation='softmax'))                        # Output layer
```

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

fit = model.fit(x_train, y_train,
                batch_size=batch_size,
                epochs=epochs,
                verbose=1, # yes, print progress bar. This takes time....
                validation_data=(x_test, y_test))

val_accuracy = fit.history['val_accuracy'][-1]  # take the last validation␣
 ↪accuracy
accuracies[l2_factor].append(val_accuracy)
```

```
Epoch 1/4
1875/1875 [==============================] - 71s 37ms/step - loss: 0.2087 -
accuracy: 0.9335 - val_loss: 0.0594 - val_accuracy: 0.9810
Epoch 2/4
1875/1875 [==============================] - 66s 35ms/step - loss: 0.0553 -
accuracy: 0.9832 - val_loss: 0.0479 - val_accuracy: 0.9846
Epoch 3/4
1875/1875 [==============================] - 79s 42ms/step - loss: 0.0390 -
accuracy: 0.9876 - val_loss: 0.0424 - val_accuracy: 0.9856
Epoch 4/4
1875/1875 [==============================] - 67s 36ms/step - loss: 0.0285 -
accuracy: 0.9907 - val_loss: 0.0258 - val_accuracy: 0.9926
```

[ ]:
```
print(f'Max accuracy: {val_accuracy:.4f}')
```

```
Max accuracy: 0.9926
```

3.2. Discuss the differences and potential benefits of using convolutional layers over fully connected ones for the application?

3.2 Answer:

Convolutional layers are designed to work with images so it makes sense that the accuracy would be better. We are using 3x3 pixels as filter on the images which is a lot bigger than 28x28. This will simplify and remove noise, so the risk for overfitting should not be a as big as for fully connected layers.