notebook_week7

May 16, 2023

0.1 Assignment 7: Neural Networks using Keras and Tensorflow

Please see the associated document for questions

Name	Working Hours
Dimitrios Koutsakis	8
Bingcheng Chen	8

If you have problems with Keras and Tensorflow on your local installation please make sure they are updated. On Google Colab this notebook runs.

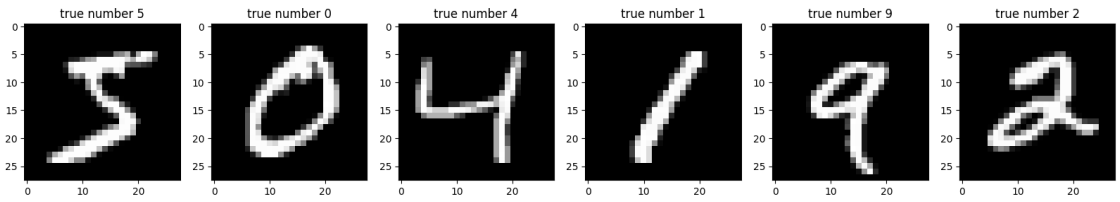
[ ]:
```
# pip install tensorflow
```

[ ]:
```
# imports
from __future__ import print_function
import keras
from keras import utils as np_utils
from keras.utils import to_categorical
import tensorflow
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Input, Dense, Flatten, Dropout, Activation,␣
↪BatchNormalization
from keras.layers import Conv2D, MaxPooling2D
from keras import backend as K
import tensorflow as tf
from matplotlib import pyplot as plt
from keras import layers
```

[ ]:
```
# Hyper-parameters data-loading and formatting
batch_size = 128
num_classes = 10
epochs = 10

img_rows, img_cols = 28, 28

# Load MNIST handwritten digit data
(x_train, lbl_train), (x_test, lbl_test) = mnist.load_data()
```

```
assert x_train.shape == (60000, 28, 28)
assert x_test.shape == (10000, 28, 28)
assert lbl_train.shape == (60000,)
assert lbl_test.shape == (10000,)

# 'channels_first' means that the color channels of an image tensor are the␣
 ↪first dimension in the tensor, followed by the spatial dimensions (height␣
 ↪and width). An image tensor would have shape (samples, channels, height,␣
 ↪width).
# 'channels_last' means that the color channels of an image tensor are the last␣
 ↪dimension in the tensor, after the spatial dimensions. In the channels_last␣
 ↪format, an image tensor would have shape (samples, height, width, channels).
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)
```

[ ]:
```
plt.figure(figsize=(20, 5))
# plot first few images in x_train dataset
for i in range(6):
    plt.subplot(1,6,i+1)
    # plot raw pixel data
    plt.imshow(x_train[i], cmap=plt.get_cmap('gray'))
    plt.title('true number {}'.format(lbl_train[i]))
# show the figure
plt.show()
```



1. Preprocessing

[ ]:
```
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')

x_train /= 255
x_test /= 255
```

```
y_train = keras.utils.to_categorical(lbl_train, num_classes)
y_test = keras.utils.to_categorical(lbl_test, num_classes)
```

Q1.1 Explain the data pre-processing highlighted in the notebook. Answer: - First, convert the x data type from integers to 32-bit floating point number which is commonly used when training a neural network. This is because float32 provides sufficient precision, and also many modern processors are optimized for float32 operations. - Second, since the pixel values for each image in the dataset are unsigned integers ranging from 0(black) to 255(white), the input features are scaled/normalised between 0.0 and 1.0. Rescaling can also help to prevent the model from being influenced too much by features with a large range of values. - Third, y dataset was converted to binary class matrix (ten output neurons). The reason we set this is because we want the model to make a prediction for each possible class. Each output neuron corresponds to a different class, and the value of that neuron represents the probability that the input image belongs to that class.

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

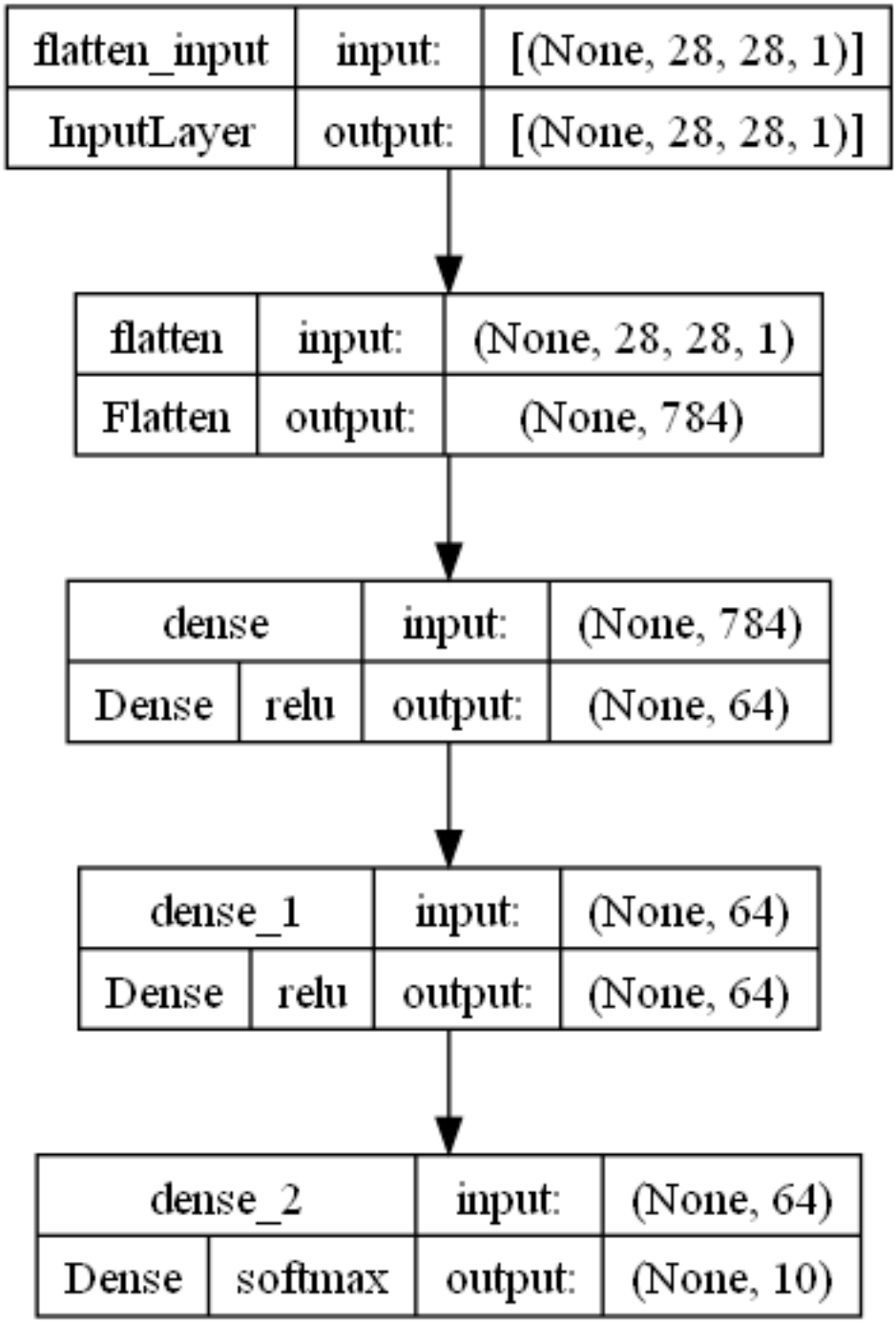
[ ]:
```
## Define model ##
model = Sequential()

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



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 [==============================] - 2s 3ms/step - loss: 0.4555 -
accuracy: 0.8698 - val_loss: 0.2520 - val_accuracy: 0.9260
Epoch 2/10
469/469 [==============================] - 1s 2ms/step - loss: 0.2256 -
accuracy: 0.9344 - val_loss: 0.2005 - val_accuracy: 0.9390
Epoch 3/10
469/469 [==============================] - 1s 2ms/step - loss: 0.1736 -
accuracy: 0.9491 - val_loss: 0.1509 - val_accuracy: 0.9549
Epoch 4/10
```

```
469/469 [==============================] - 1s 2ms/step - loss: 0.1444 -
accuracy: 0.9585 - val_loss: 0.1373 - val_accuracy: 0.9595
Epoch 5/10
469/469 [==============================] - 1s 2ms/step - loss: 0.1227 -
accuracy: 0.9641 - val_loss: 0.1315 - val_accuracy: 0.9600
Epoch 6/10
469/469 [==============================] - 1s 2ms/step - loss: 0.1085 -
accuracy: 0.9682 - val_loss: 0.1050 - val_accuracy: 0.9682
Epoch 7/10
469/469 [==============================] - 1s 2ms/step - loss: 0.0953 -
accuracy: 0.9716 - val_loss: 0.1077 - val_accuracy: 0.9656
Epoch 8/10
469/469 [==============================] - 1s 2ms/step - loss: 0.0862 -
accuracy: 0.9744 - val_loss: 0.1001 - val_accuracy: 0.9701
Epoch 9/10
469/469 [==============================] - 1s 2ms/step - loss: 0.0779 -
accuracy: 0.9775 - val_loss: 0.0990 - val_accuracy: 0.9703
Epoch 10/10
469/469 [==============================] - 1s 2ms/step - loss: 0.0712 -
accuracy: 0.9796 - val_loss: 0.0888 - val_accuracy: 0.9720
Test loss: 0.08882012963294983, Test accuracy 0.972000002861023
```

[ ]:
```
# Converts a Keras model to dot format and save to a file.
keras.utils.plot_model(model, show_shapes=True, show_layer_activations=True)
```

[ ]:



[ ]:
```
# get a summary of the model
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
```

```
=================================================================
 flatten (Flatten)           (None, 784)               0

 dense (Dense)               (None, 64)                50240

 dense_1 (Dense)             (None, 64)                4160

 dense_2 (Dense)             (None, 10)                650

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

Q2.1 How many layers does the network in the notebook have? How many neurons does each layer have? What activation functions and why are these appropriate for this application? What is the total number of parameters for the network? Why do the input and output layers have the dimensions they have? Answer:

According to the plot and summary of the model above, we can conclude:

- There are 4 layers in the network, the input layer has 784 neurons, the two hidden layers both have 64 neurons and the output layer has 10 neurons.
- The two hidden layers use activation function 'relu (rectified linear unit)', $f(x) = max(0, x)$, the reason we use this activation function is because it is computationally efficient, effective at reducing the impact of the vanishing gradient problem, and able to learn complex nonlinear relationships in the input data.
- The output layer uses 'softmax' activation function, because 'Softmax' function converts a vector of values to a probability distribution which means the elements of the output vector are in range (0, 1) and sum to 1. In this case, this is what we want to output - the value of each neuron represents the confidence that the input image belongs to the corresponding number.
- The total number of parameters for this network is 55050.
- The input layer has the dimension $784 = 28 * 28$, this is because neural network models require the input to be a one-dimensional vector, thus 28x28 input image was flattened into a one-dimensional vector of length 784.
- The output layer has the dimension 10, because the handwritten digits range from 0 to 9. Therefore, we use 10 output neurons, one for each possible number.

Q2.2 What loss function is used to train the network? What is the functional form (a mathematical expression) of the loss function? and how should we interpret it? Why is it appropriate for the problem at hand? Answer:

The loss function used to train the network is 'categorical_crossentropy' function. It is commonly used when there are multiple label classes and one_hot representation output. In this case, since we have 10 classes, it is appropriate to use this loss function. The mathematical expression of this function is as follows:

$Loss = -\frac{1}{N}\sum_{i=1}^{N}\sum_{c=1}^{C}\mathbf{1}_{y_i \in C_c} log p_{model}[y_i \in C_c]$

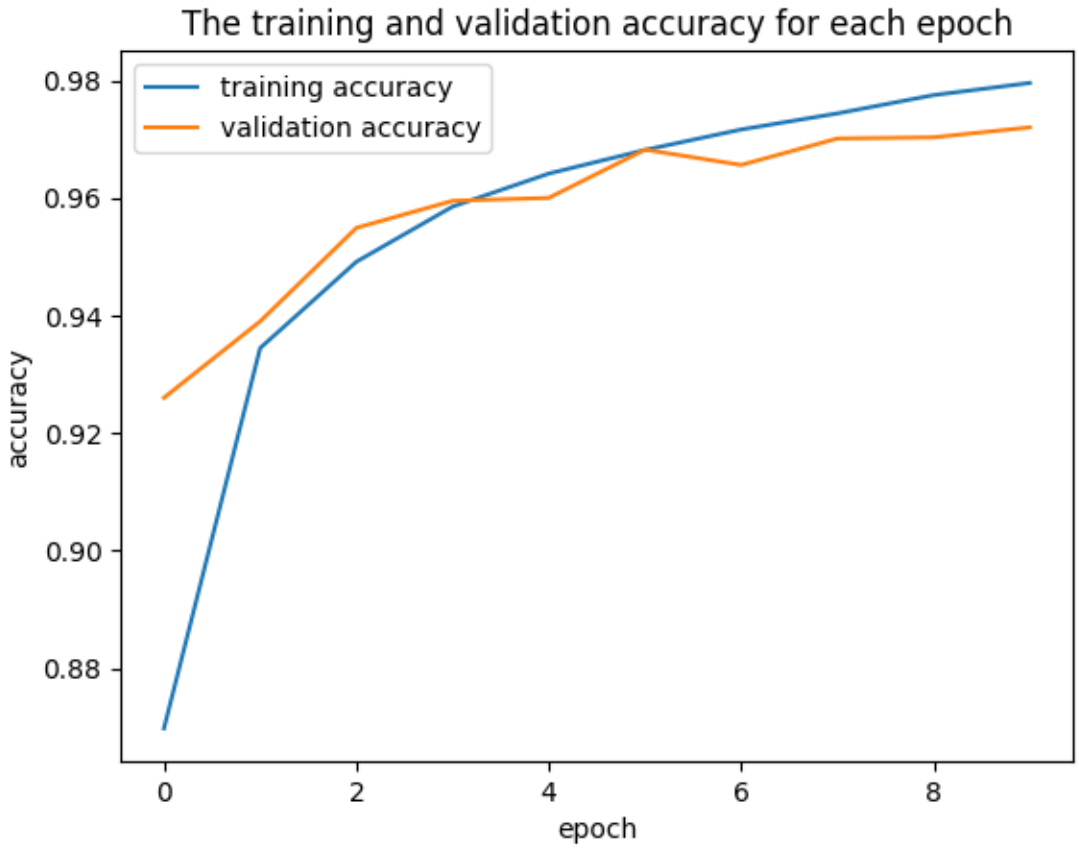
where N is the number of samples in the dataset, C is the number of classes(10), $\mathbf{1}_{y_i \in C_c}$ is the indicator function that takes value 1 if the i-th instance belongs to class c, and 0 otherwise. The $p_{model}[y_i \in C_c]$ is the probability predicted by the model for the 'i'th observation to belong to the 'c'th category.

The loss function computes the average negative log-likelihood of the predicted probability distribution over all instances in the dataset. It measures the difference between the true label distribution and the predicted label distribution and tries to minimize it during the training process.

A lower value of the loss function indicates that the model is making more accurate predictions. Conversely, a higher value of the loss function indicates that the model is making less accurate predictions.

Q2.3 Train the network for 10 epochs and plot the training and validation accuracy for each epoch.

```
[ ]: plt.plot(fit_info.history['accuracy'])
plt.plot(fit_info.history['val_accuracy'])
plt.title('The training and validation accuracy for each epoch')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['training accuracy', 'validation accuracy'], loc='upper left')
plt.show()
```



Q2.4 Update the model to implement a three-layer neural network where the hidden layers have 500 and 300 hidden units respectively. Train for 40 epochs. What is the best validation accuracy you can achieve? Geoff Hinton (a co-pioneer of Deep learning) claimed this network could reach a validation accuracy of 0.9847 (http://yann.lecun.com/exdb/mnist/) using weight decay (L2 regularization of weights(kernels): https://keras.io/api/layers/regularizers/). 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. Plot the final validation accuracy with standard deviation (computed from the replicates) as a function of the regularization factor. How close do you get to Hintons result? – 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)

[ ]:
```
import numpy as np

epochs = 40
L2 = np.linspace(0.000001, 0.001, 5)

accuracy_matrix = np.zeros([3,5],dtype=np.float32)

for l2 in L2:
    print('L2={}'.format(l2))

    for i in range(3):
        model = Sequential()
        model.add(Flatten())
        model.add(Dense(500, activation = 'relu', kernel_regularizer=keras.
 ↪regularizers.L2(l2 = l2)))
        model.add(Dense(300, activation = 'relu',kernel_regularizer=keras.
 ↪regularizers.L2(l2 = l2)))
        model.add(Dense(num_classes, activation='softmax'))

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

        fit_info = model.fit(x_train, y_train,
           batch_size=batch_size,
           epochs=epochs,
           verbose=0,
           validation_data=(x_test, y_test)
           )

        score = model.evaluate(x_test, y_test, verbose=0)
```

```
        accuracy_matrix[i][np.where(L2 == l2)[0][0]] = score[1]

        print('Network No.{}: Accuracy Score = {}'.format(i, score[1]))

    print()
```

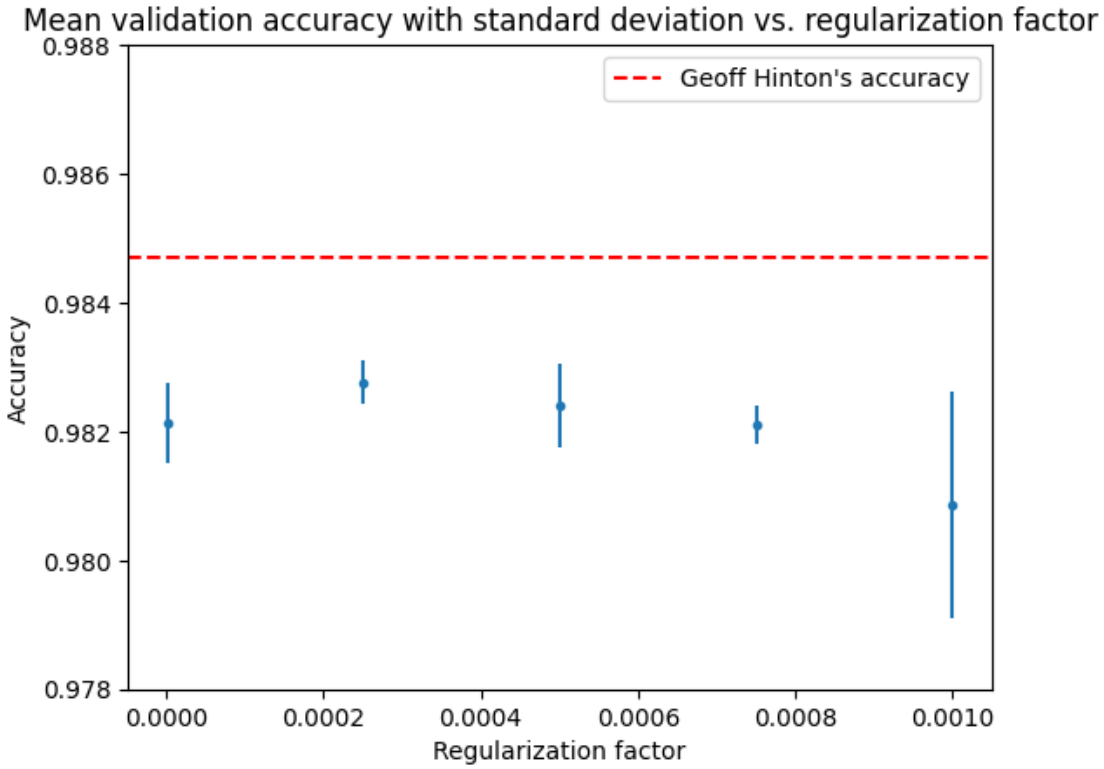
```
L2=1e-06
Network No.0: Accuracy Score = 0.9814000129699707
Network No.1: Accuracy Score = 0.9829000234603882
Network No.2: Accuracy Score = 0.9821000099182129

L2=0.00025075000000000005
Network No.0: Accuracy Score = 0.9822999835014343
Network No.1: Accuracy Score = 0.9830999970436096
Network No.2: Accuracy Score = 0.9829000234603882

L2=0.0005005000000000001
Network No.0: Accuracy Score = 0.9832000136375427
Network No.1: Accuracy Score = 0.9815999865531921
Network No.2: Accuracy Score = 0.9824000000953674

L2=0.0007502500000000002
Network No.0: Accuracy Score = 0.9819999933242798
Network No.1: Accuracy Score = 0.9818000197410583
Network No.2: Accuracy Score = 0.9825000166893005

L2=0.001
Network No.0: Accuracy Score = 0.982699990272522
Network No.1: Accuracy Score = 0.9814000129699707
Network No.2: Accuracy Score = 0.9785000085830688
```

[ ]:
```
accuracy_matrix
```

[ ]:
```
array([[0.9814, 0.9823, 0.9832, 0.982 , 0.9827],
       [0.9829, 0.9831, 0.9816, 0.9818, 0.9814],
       [0.9821, 0.9829, 0.9824, 0.9825, 0.9785]], dtype=float32)
```

[ ]:
```
accuracy_mean = np.mean(accuracy_matrix, axis=0)
accuracy_std = np.std(accuracy_matrix, axis=0)

plt.errorbar(L2, accuracy_mean, accuracy_std, linestyle='None', marker='.')
plt.axhline(y=0.9847, color = 'r', linestyle='--', label = "Geoff Hinton's␣
 ↪accuracy")
plt.title('Mean validation accuracy with standard deviation vs. regularization␣
 ↪factor', fontsize=12)
plt.ylabel('Accuracy')
```

```
plt.xlabel('Regularization factor')
plt.ylim(0.978,0.988)
plt.legend()

plt.show()
```



[ ]:
```
Best_accuracy = np.max(accuracy_matrix)
print(f'The best accuracy we can get is {Best_accuracy}.')
rate = Best_accuracy/(0.9847)*100
print(f"The best accuracy we can get is {rate:.2f}% close to Geoff Hinton's␣
  ↪accuracy.")
```

```
The best accuracy we can get is 0.983299970626831.
The best accuracy we can get is 99.86% close to Geoff Hinton's accuracy.
```

Answer:

The accuracy is slightly smaller than Geoff Hinton's accuracy. This may have happened because we set different values of learning rate, batch size and epochs.

3. Convolutional layers

Q3.1 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.

Attempt: Model 1 - 99.12%

[ ]:
```
def build_model():
    # Initialize a sequential model object
    model = Sequential()

    # Add a 2D convolutional layer with 20 filters, a kernel size of (5,5),␣
↪ReLU activation.
    # The input shape of the layer is (28, 28, 1), corresponding to the size of␣
↪images 28x28.
    # Add a max pooling layer with a pool size of (2,2) to downsample the␣
↪feature maps.
    model.add(Conv2D(20,(5,5),activation='relu', input_shape=(28, 28, 1)))
    model.add(MaxPooling2D((2,2)))

    # Add a second 2D convolutional layer with 40 filters, a kernel size of␣
↪(5,5).
    # The input shape of the layer is (20, 12, 12), corresponding to the output␣
↪shape of the previous layer.
    model.add(Conv2D(40,(5,5),activation='relu', input_shape=(20, 12, 12)))
    model.add(MaxPooling2D((2,2)))

    # Flatten the output of the previous layer to feed it to a fully connected␣
↪layer.
    model.add(Flatten())

    # Add a full connected layer with 100 units.
    model.add(Dense(100, activation = 'sigmoid'))

    # Add the output layer
    model.add(Dense(num_classes, activation='softmax'))

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

new_model = build_model()
fit_info = new_model.fit(x_train, y_train,
    batch_size=128,
```

```
    epochs=60,
    verbose=1,
    validation_data=(x_test, y_test))
score = new_model.evaluate(x_test, y_test, verbose=0)
print('Test loss: {}, Test accuracy {}'.format(score[0], score[1]))
```

```
Epoch 1/60
469/469 [==============================] - 11s 22ms/step - loss: 0.6304 -
accuracy: 0.8192 - val_loss: 0.1781 - val_accuracy: 0.9531
Epoch 2/60
469/469 [==============================] - 10s 22ms/step - loss: 0.1493 -
accuracy: 0.9586 - val_loss: 0.1154 - val_accuracy: 0.9679
Epoch 3/60
469/469 [==============================] - 10s 22ms/step - loss: 0.1017 -
accuracy: 0.9718 - val_loss: 0.0807 - val_accuracy: 0.9775
Epoch 4/60
469/469 [==============================] - 10s 22ms/step - loss: 0.0804 -
accuracy: 0.9773 - val_loss: 0.0656 - val_accuracy: 0.9806
Epoch 5/60
469/469 [==============================] - 10s 22ms/step - loss: 0.0682 -
accuracy: 0.9802 - val_loss: 0.0588 - val_accuracy: 0.9820
Epoch 6/60
469/469 [==============================] - 10s 22ms/step - loss: 0.0593 -
accuracy: 0.9834 - val_loss: 0.0515 - val_accuracy: 0.9847
Epoch 7/60
469/469 [==============================] - 10s 22ms/step - loss: 0.0526 -
accuracy: 0.9849 - val_loss: 0.0573 - val_accuracy: 0.9810
Epoch 8/60
469/469 [==============================] - 11s 22ms/step - loss: 0.0474 -
accuracy: 0.9868 - val_loss: 0.0442 - val_accuracy: 0.9874
Epoch 9/60
469/469 [==============================] - 11s 23ms/step - loss: 0.0434 -
accuracy: 0.9876 - val_loss: 0.0403 - val_accuracy: 0.9879
Epoch 10/60
469/469 [==============================] - 11s 23ms/step - loss: 0.0395 -
accuracy: 0.9890 - val_loss: 0.0387 - val_accuracy: 0.9884
Epoch 11/60
469/469 [==============================] - 11s 23ms/step - loss: 0.0365 -
accuracy: 0.9898 - val_loss: 0.0383 - val_accuracy: 0.9877
Epoch 12/60
469/469 [==============================] - 10s 22ms/step - loss: 0.0336 -
accuracy: 0.9905 - val_loss: 0.0373 - val_accuracy: 0.9884
Epoch 13/60
469/469 [==============================] - 10s 22ms/step - loss: 0.0316 -
accuracy: 0.9913 - val_loss: 0.0352 - val_accuracy: 0.9894
Epoch 14/60
469/469 [==============================] - 11s 23ms/step - loss: 0.0295 -
accuracy: 0.9919 - val_loss: 0.0355 - val_accuracy: 0.9887
```

```
Epoch 15/60
469/469 [==============================] - 11s 23ms/step - loss: 0.0273 -
accuracy: 0.9931 - val_loss: 0.0332 - val_accuracy: 0.9891
Epoch 16/60
469/469 [==============================] - 11s 23ms/step - loss: 0.0258 -
accuracy: 0.9930 - val_loss: 0.0356 - val_accuracy: 0.9887
Epoch 17/60
469/469 [==============================] - 10s 22ms/step - loss: 0.0237 -
accuracy: 0.9938 - val_loss: 0.0331 - val_accuracy: 0.9898
Epoch 18/60
469/469 [==============================] - 13s 27ms/step - loss: 0.0228 -
accuracy: 0.9940 - val_loss: 0.0323 - val_accuracy: 0.9902
Epoch 19/60
469/469 [==============================] - 13s 27ms/step - loss: 0.0213 -
accuracy: 0.9943 - val_loss: 0.0324 - val_accuracy: 0.9900
Epoch 20/60
469/469 [==============================] - 12s 25ms/step - loss: 0.0199 -
accuracy: 0.9952 - val_loss: 0.0304 - val_accuracy: 0.9907
Epoch 21/60
469/469 [==============================] - 11s 24ms/step - loss: 0.0192 -
accuracy: 0.9951 - val_loss: 0.0316 - val_accuracy: 0.9893
Epoch 22/60
469/469 [==============================] - 11s 23ms/step - loss: 0.0178 -
accuracy: 0.9957 - val_loss: 0.0293 - val_accuracy: 0.9904
Epoch 23/60
469/469 [==============================] - 10s 21ms/step - loss: 0.0166 -
accuracy: 0.9963 - val_loss: 0.0307 - val_accuracy: 0.9904
Epoch 24/60
469/469 [==============================] - 10s 21ms/step - loss: 0.0154 -
accuracy: 0.9967 - val_loss: 0.0294 - val_accuracy: 0.9908
Epoch 25/60
469/469 [==============================] - 10s 21ms/step - loss: 0.0148 -
accuracy: 0.9967 - val_loss: 0.0293 - val_accuracy: 0.9905
Epoch 26/60
469/469 [==============================] - 10s 22ms/step - loss: 0.0140 -
accuracy: 0.9970 - val_loss: 0.0283 - val_accuracy: 0.9915
Epoch 27/60
469/469 [==============================] - 10s 21ms/step - loss: 0.0131 -
accuracy: 0.9974 - val_loss: 0.0295 - val_accuracy: 0.9903
Epoch 28/60
469/469 [==============================] - 10s 21ms/step - loss: 0.0123 -
accuracy: 0.9976 - val_loss: 0.0315 - val_accuracy: 0.9897
Epoch 29/60
469/469 [==============================] - 10s 22ms/step - loss: 0.0119 -
accuracy: 0.9977 - val_loss: 0.0307 - val_accuracy: 0.9902
Epoch 30/60
469/469 [==============================] - 12s 25ms/step - loss: 0.0108 -
accuracy: 0.9981 - val_loss: 0.0279 - val_accuracy: 0.9910
```

```
Epoch 31/60
469/469 [==============================] - 11s 24ms/step - loss: 0.0106 -
accuracy: 0.9981 - val_loss: 0.0281 - val_accuracy: 0.9911
Epoch 32/60
469/469 [==============================] - 13s 27ms/step - loss: 0.0100 -
accuracy: 0.9984 - val_loss: 0.0288 - val_accuracy: 0.9904
Epoch 33/60
469/469 [==============================] - 13s 27ms/step - loss: 0.0095 -
accuracy: 0.9985 - val_loss: 0.0275 - val_accuracy: 0.9914
Epoch 34/60
469/469 [==============================] - 13s 28ms/step - loss: 0.0091 -
accuracy: 0.9985 - val_loss: 0.0274 - val_accuracy: 0.9911
Epoch 35/60
469/469 [==============================] - 14s 29ms/step - loss: 0.0086 -
accuracy: 0.9988 - val_loss: 0.0289 - val_accuracy: 0.9905
Epoch 36/60
469/469 [==============================] - 13s 29ms/step - loss: 0.0082 -
accuracy: 0.9988 - val_loss: 0.0277 - val_accuracy: 0.9908
Epoch 37/60
469/469 [==============================] - 15s 32ms/step - loss: 0.0078 -
accuracy: 0.9989 - val_loss: 0.0279 - val_accuracy: 0.9911
Epoch 38/60
469/469 [==============================] - 13s 28ms/step - loss: 0.0074 -
accuracy: 0.9990 - val_loss: 0.0275 - val_accuracy: 0.9913
Epoch 39/60
469/469 [==============================] - 13s 27ms/step - loss: 0.0071 -
accuracy: 0.9991 - val_loss: 0.0273 - val_accuracy: 0.9914
Epoch 40/60
469/469 [==============================] - 13s 27ms/step - loss: 0.0068 -
accuracy: 0.9991 - val_loss: 0.0278 - val_accuracy: 0.9912
Epoch 41/60
469/469 [==============================] - 12s 26ms/step - loss: 0.0065 -
accuracy: 0.9991 - val_loss: 0.0273 - val_accuracy: 0.9908
Epoch 42/60
469/469 [==============================] - 12s 26ms/step - loss: 0.0063 -
accuracy: 0.9992 - val_loss: 0.0272 - val_accuracy: 0.9912
Epoch 43/60
469/469 [==============================] - 13s 28ms/step - loss: 0.0059 -
accuracy: 0.9994 - val_loss: 0.0273 - val_accuracy: 0.9912
Epoch 44/60
469/469 [==============================] - 13s 27ms/step - loss: 0.0057 -
accuracy: 0.9994 - val_loss: 0.0281 - val_accuracy: 0.9908
Epoch 45/60
469/469 [==============================] - 12s 26ms/step - loss: 0.0056 -
accuracy: 0.9992 - val_loss: 0.0270 - val_accuracy: 0.9914
Epoch 46/60
469/469 [==============================] - 12s 26ms/step - loss: 0.0053 -
accuracy: 0.9995 - val_loss: 0.0269 - val_accuracy: 0.9912
```

```
Epoch 47/60
469/469 [==============================] - 13s 27ms/step - loss: 0.0051 -
accuracy: 0.9994 - val_loss: 0.0269 - val_accuracy: 0.9906
Epoch 48/60
469/469 [==============================] - 12s 26ms/step - loss: 0.0049 -
accuracy: 0.9995 - val_loss: 0.0276 - val_accuracy: 0.9908
Epoch 49/60
469/469 [==============================] - 13s 28ms/step - loss: 0.0048 -
accuracy: 0.9995 - val_loss: 0.0273 - val_accuracy: 0.9914
Epoch 50/60
469/469 [==============================] - 12s 26ms/step - loss: 0.0046 -
accuracy: 0.9995 - val_loss: 0.0265 - val_accuracy: 0.9912
Epoch 51/60
469/469 [==============================] - 13s 27ms/step - loss: 0.0044 -
accuracy: 0.9995 - val_loss: 0.0274 - val_accuracy: 0.9911
Epoch 52/60
469/469 [==============================] - 12s 26ms/step - loss: 0.0043 -
accuracy: 0.9995 - val_loss: 0.0267 - val_accuracy: 0.9916
Epoch 53/60
469/469 [==============================] - 12s 26ms/step - loss: 0.0041 -
accuracy: 0.9997 - val_loss: 0.0269 - val_accuracy: 0.9923
Epoch 54/60
469/469 [==============================] - 12s 25ms/step - loss: 0.0040 -
accuracy: 0.9996 - val_loss: 0.0262 - val_accuracy: 0.9912
Epoch 55/60
469/469 [==============================] - 14s 29ms/step - loss: 0.0039 -
accuracy: 0.9997 - val_loss: 0.0264 - val_accuracy: 0.9915
Epoch 56/60
469/469 [==============================] - 13s 28ms/step - loss: 0.0037 -
accuracy: 0.9997 - val_loss: 0.0269 - val_accuracy: 0.9912
Epoch 57/60
469/469 [==============================] - 12s 25ms/step - loss: 0.0036 -
accuracy: 0.9997 - val_loss: 0.0265 - val_accuracy: 0.9917
Epoch 58/60
469/469 [==============================] - 12s 25ms/step - loss: 0.0035 -
accuracy: 0.9997 - val_loss: 0.0258 - val_accuracy: 0.9916
Epoch 59/60
469/469 [==============================] - 11s 23ms/step - loss: 0.0034 -
accuracy: 0.9997 - val_loss: 0.0258 - val_accuracy: 0.9918
Epoch 60/60
469/469 [==============================] - 11s 23ms/step - loss: 0.0033 -
accuracy: 0.9998 - val_loss: 0.0263 - val_accuracy: 0.9913
Test loss: 0.0262740571051836, Test accuracy 0.9912999868392944
```

Attempt: Model 2 - 99.50% This model is inspired by Jay Gupta's LeNet-5 v2.0 convolutional neural network model, LeNet-5 v2.0 link, our model is based on LeNet-5 v2.0.

[ ]:
```
from keras.preprocessing.image import ImageDataGenerator
from keras.callbacks import ReduceLROnPlateau

# Create a data augmentation stage with rotations, zooms and image shift
datagen = ImageDataGenerator(
        rotation_range = 10, # Rotate images randomly by up to 10 degrees
        zoom_range = 0.1, # Zoom in or out by up to 10%
        width_shift_range = 0.1,  # Shift the image horizontally by up to 10%
        height_shift_range = 0.1)  # Shift the image vertically by up to 10%

datagen.fit(x_train)
```

[ ]:
```
def build_model():

    model = Sequential()

    # 1. Add 2 2D convolutional layer with 32 filters, each with a 5x5 kernel, and ReLU activation function to the model.
    # 2. Add a batch normalization layer, it works by normalizing the inputs to a layer, i.e., transforming them to have zero mean and unit variance.
    # This has the effect of stabilizing the distribution of inputs, reducing the effects of internal covariate shift, and allowing the use of higher learning rates.
    # 3. Add a max pooling layer with a pool size of (2,2) to downsample the feature maps.
    model.add(Conv2D(32,(5,5),activation='relu', input_shape=(28, 28, 1)))
    model.add(Conv2D(32,(5,5),activation='relu', input_shape=(20, 12, 12)))
    model.add(BatchNormalization())
    model.add(MaxPooling2D((2,2)))

    model.add(Conv2D(64,(3,3),activation='relu'))
    model.add(Conv2D(64,(3,3),activation='relu'))
    model.add(BatchNormalization())
    model.add(MaxPooling2D((2,2)))

    # Flatten the output of the previous layer to feed it to a fully connected layer.
    model.add(Flatten())

    model.add(Dense(256, activation = 'relu'))
    model.add(BatchNormalization())
    model.add(Dropout(0.25))

    model.add(Dense(128, activation = 'relu'))
    model.add(BatchNormalization())
    model.add(Dropout(0.25))
```

```
    model.add(Dense(84, activation = 'relu', kernel_regularizer=keras.
 ↪regularizers.L2(l2 = 0.1)))
    model.add(BatchNormalization())
    model.add(Dropout(0.25))


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

    model.compile(loss=keras.losses.categorical_crossentropy,
                  optimizer='adam',
                  metrics=['accuracy'])
    return model
```

[ ]:
```
new_model = build_model()

# Define a learning rate scheduler to reduce the learning rate when the␣
 ↪validation loss plateaus
variable_learning_rate = ReduceLROnPlateau(monitor='val_loss', factor = 0.2,␣
 ↪patience = 2)

fit_info = new_model.fit(x_train, y_train,
    batch_size=64,
    epochs=30,
    callbacks = [variable_learning_rate], # using the learning rate scheduler␣
 ↪defined above as a callback to adjust the learning rate
    verbose=1,
    validation_data=(x_test, y_test))
score = new_model.evaluate(x_test, y_test, verbose=0)
print('Test loss: {}, Test accuracy {}'.format(score[0], score[1]))
```

```
Epoch 1/30
938/938 [==============================] - 51s 53ms/step - loss: 0.9662 -
accuracy: 0.9374 - val_loss: 0.0945 - val_accuracy: 0.9854 - lr: 0.0010
Epoch 2/30
938/938 [==============================] - 50s 53ms/step - loss: 0.1207 -
accuracy: 0.9787 - val_loss: 0.1111 - val_accuracy: 0.9838 - lr: 0.0010
Epoch 3/30
938/938 [==============================] - 53s 56ms/step - loss: 0.1034 -
accuracy: 0.9827 - val_loss: 0.1153 - val_accuracy: 0.9793 - lr: 0.0010
Epoch 4/30
938/938 [==============================] - 49s 53ms/step - loss: 0.0486 -
accuracy: 0.9916 - val_loss: 0.0336 - val_accuracy: 0.9935 - lr: 2.0000e-04
Epoch 5/30
938/938 [==============================] - 49s 53ms/step - loss: 0.0363 -
accuracy: 0.9932 - val_loss: 0.0330 - val_accuracy: 0.9937 - lr: 2.0000e-04
Epoch 6/30
938/938 [==============================] - 50s 53ms/step - loss: 0.0303 -
```

```
accuracy: 0.9945 - val_loss: 0.0343 - val_accuracy: 0.9936 - lr: 2.0000e-04
Epoch 7/30
938/938 [==============================] - 52s 55ms/step - loss: 0.0291 -
accuracy: 0.9950 - val_loss: 0.0285 - val_accuracy: 0.9952 - lr: 2.0000e-04
Epoch 8/30
938/938 [==============================] - 49s 53ms/step - loss: 0.0252 -
accuracy: 0.9956 - val_loss: 0.0307 - val_accuracy: 0.9936 - lr: 2.0000e-04
Epoch 9/30
938/938 [==============================] - 51s 54ms/step - loss: 0.0238 -
accuracy: 0.9959 - val_loss: 0.0369 - val_accuracy: 0.9935 - lr: 2.0000e-04
Epoch 10/30
938/938 [==============================] - 51s 54ms/step - loss: 0.0151 -
accuracy: 0.9977 - val_loss: 0.0219 - val_accuracy: 0.9947 - lr: 4.0000e-05
Epoch 11/30
938/938 [==============================] - 50s 54ms/step - loss: 0.0105 -
accuracy: 0.9982 - val_loss: 0.0221 - val_accuracy: 0.9943 - lr: 4.0000e-05
Epoch 12/30
938/938 [==============================] - 51s 54ms/step - loss: 0.0090 -
accuracy: 0.9985 - val_loss: 0.0217 - val_accuracy: 0.9949 - lr: 4.0000e-05
Epoch 13/30
938/938 [==============================] - 51s 54ms/step - loss: 0.0083 -
accuracy: 0.9985 - val_loss: 0.0216 - val_accuracy: 0.9947 - lr: 4.0000e-05
Epoch 14/30
938/938 [==============================] - 51s 55ms/step - loss: 0.0077 -
accuracy: 0.9987 - val_loss: 0.0205 - val_accuracy: 0.9943 - lr: 4.0000e-05
Epoch 15/30
938/938 [==============================] - 52s 56ms/step - loss: 0.0073 -
accuracy: 0.9988 - val_loss: 0.0206 - val_accuracy: 0.9951 - lr: 4.0000e-05
Epoch 16/30
938/938 [==============================] - 50s 54ms/step - loss: 0.0062 -
accuracy: 0.9991 - val_loss: 0.0200 - val_accuracy: 0.9947 - lr: 4.0000e-05
Epoch 17/30
938/938 [==============================] - 51s 55ms/step - loss: 0.0065 -
accuracy: 0.9990 - val_loss: 0.0198 - val_accuracy: 0.9953 - lr: 4.0000e-05
Epoch 18/30
938/938 [==============================] - 52s 55ms/step - loss: 0.0057 -
accuracy: 0.9993 - val_loss: 0.0217 - val_accuracy: 0.9951 - lr: 4.0000e-05
Epoch 19/30
938/938 [==============================] - 55s 58ms/step - loss: 0.0059 -
accuracy: 0.9992 - val_loss: 0.0214 - val_accuracy: 0.9948 - lr: 4.0000e-05
Epoch 20/30
938/938 [==============================] - 67s 71ms/step - loss: 0.0049 -
accuracy: 0.9995 - val_loss: 0.0213 - val_accuracy: 0.9947 - lr: 8.0000e-06
Epoch 21/30
938/938 [==============================] - 65s 69ms/step - loss: 0.0044 -
accuracy: 0.9995 - val_loss: 0.0211 - val_accuracy: 0.9949 - lr: 8.0000e-06
Epoch 22/30
938/938 [==============================] - 60s 64ms/step - loss: 0.0039 -
```

```
accuracy: 0.9997 - val_loss: 0.0208 - val_accuracy: 0.9949 - lr: 1.6000e-06
Epoch 23/30
938/938 [==============================] - 50s 54ms/step - loss: 0.0040 -
accuracy: 0.9996 - val_loss: 0.0208 - val_accuracy: 0.9951 - lr: 1.6000e-06
Epoch 24/30
938/938 [==============================] - 62s 67ms/step - loss: 0.0036 -
accuracy: 0.9996 - val_loss: 0.0206 - val_accuracy: 0.9951 - lr: 3.2000e-07
Epoch 25/30
938/938 [==============================] - 60s 64ms/step - loss: 0.0036 -
accuracy: 0.9997 - val_loss: 0.0204 - val_accuracy: 0.9950 - lr: 3.2000e-07
Epoch 26/30
938/938 [==============================] - 51s 55ms/step - loss: 0.0039 -
accuracy: 0.9995 - val_loss: 0.0205 - val_accuracy: 0.9950 - lr: 6.4000e-08
Epoch 27/30
938/938 [==============================] - 51s 55ms/step - loss: 0.0036 -
accuracy: 0.9997 - val_loss: 0.0205 - val_accuracy: 0.9951 - lr: 6.4000e-08
Epoch 28/30
938/938 [==============================] - 51s 55ms/step - loss: 0.0037 -
accuracy: 0.9997 - val_loss: 0.0206 - val_accuracy: 0.9952 - lr: 1.2800e-08
Epoch 29/30
938/938 [==============================] - 51s 55ms/step - loss: 0.0036 -
accuracy: 0.9996 - val_loss: 0.0204 - val_accuracy: 0.9950 - lr: 1.2800e-08
Epoch 30/30
938/938 [==============================] - 52s 55ms/step - loss: 0.0037 -
accuracy: 0.9996 - val_loss: 0.0205 - val_accuracy: 0.9950 - lr: 2.5600e-09
Test loss: 0.02048979140818119, Test accuracy 0.9950000047683716
```

Q3.2 Discuss the differences and potential benefits of using convolutional layers over fully connected ones for the application? Answer:

Differences

1. In fully connected neural networks, each neuron in one layer is connected to every neuron in the next layer, resulting in a dense matrix of weights that needs to be learned during training. This can be computationally expensive and prone to overfitting.
2. In CNNs, each neuron in a layer is connected only to a small region of the input data, known as the local receptive field. This enables the network to learn local patterns, such as edges and textures, and reduces the number of parameters that need to be learned. This can result in faster training times or the ability to construct deeper networks with the same computational resources.

Potential benefits

1. Convolutional layers require fewer parameters than fully connected layers, which makes them more efficient and easier to train.
2. Convolutional layers are translation-invariant, which means they can recognize patterns regardless of their position in the image.
3. Convolutional layers are particularly effective in capturing spatial information and local patterns in images.