AI Fundamentals - Assignment 3

This assignment requires you to use Tensorflow and Keras. Keras is a high-level Deep Learning API written in Python working as an interface to TensorFlow.

This assignment is divided in two parts. In the first part you will learn about Keras with the help of the example below and the Keras documentation. In the second part, you will practise training a Deep Learning model.

How to submit

Submit by uploading this notebook to Canvas. It should include **plots**, **results** and **code** showing how the results were genereated. Remember to name your file(s) appropriately. It is due on 11:59 of December 9, 2020.

Installation

Instructions can be found here:

- Tensorflow

Since Tensorflow 2.0, Keras is included in Tensorflow and will be automatically installed with Tensorflow. It can be accessed as `tensorflow.keras`

I recommend using `pip`. For Tensorflow is it sufficient to install the CPU version. The GPU version requires a good workstation with high-end Nvidia GPU(s), and it is not necessary for this tutorial.

If you're using a virtualenv:

```
pip3 install tensorflow
```

Add `sudo` for a systemwide installation (i.e. no `virtualenv`).

```
sudo pip3 install tensorflow
```

Make sure that you have `sklearn`, `matplotlib` and `numpy` installed, too.

Part 1 - understand a model

Optimizers

Loss is the penalty for a bad prediction. That is, loss is a number indicating how bad the model's prediction was on a single example. If the model's prediction is perfect, the loss is zero; otherwise, the loss is greater than zero. The goal of training a model is to find a set of weights and biases (i.e. parameters) that have, on average, a low loss across all examples. The term cost is used interchangably with loss. See the loss section in the Keras documentation for a list and descriptions of what is available.

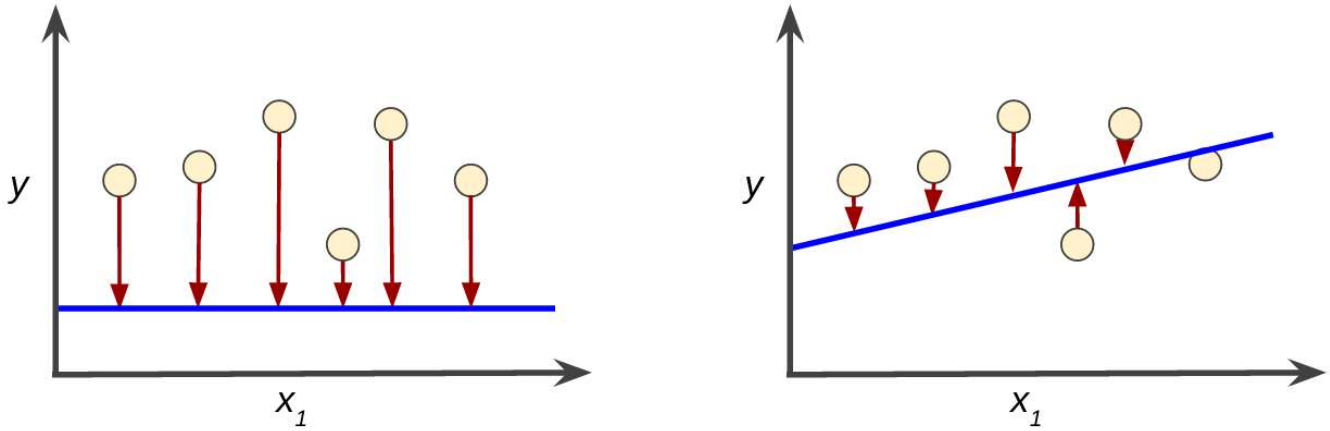


Figure 1. Left: high loss and right: low loss.

The optimizer is the algorithm used to minimize the loss/cost. Optimizers in neural networks work by finding the gradient/derivative of the loss with respect to the parameters (i.e. the weights). "Gradient" is the correct term since a we are looking at multi-dimensional systems (i.e. many parameters), however, the terms are often used interchangably. For those who didn't take multivariate calculus, just think of the gradient as a derivative. The derivative of the loss with respect to a parameters tells us how much the loss changes when we nudge a weight up or down. So, by knowing how a given parameter affects the loss the optimizer can change it so as to decrease the loss. The various optimizers differ in how they change the weights.

Mini-overview over popular optimizers

- **Stochastic Gradient Descent (SGD)**. This is the most basic and easy to understand optimizer. It updates the weights in the negative direction of the gradient by taking the average gradient of mini-batch of data (e.g. 20-1000 examples) in each step. Vanilla SGD only has one hyper-parameter, the learning rate.
- **Momentum**. This optimizer "gains speed" when the gradient has pointed in the same direction for several consecutive updates. That is, it has a momentum and want to keep moving in that direction. It gains momentum by accumulating an exponentially decaying moving average of past gradients. The step size depends on how large and aligned the sequence of gradients are. The most important hyper-parameter is alpha and common values are 0.5 and 0.9.
- **Nesterov Momentum**. This is a modification of the standard momentum optimizer.
- **AdaGrad**. This optimizer Ada-ptively sets the learning rate depending on the steepness/magnitude of the Grad-ients. This is done so that weights with big gradients get a smaller effective learning rate, and weights with small gradients will get a greater effective learning rate. The result is quicker progress in the more gently sloped directions of the weight space and a slowdown in stepp regions.
- **RMSProp**. This is modification of AdaGrad, where the accumulated gradient decays, that is, the influence of previous gradients gradually decreases.
- **Adam**. The name comes from "adaptive moments", and it is a combination of RMSProp and momentum. It has several hyper-parameters.

The above list just gives a quick overview of some of the most common. However, old optimizers are constantly improved and new are developed. SGD and momentum are most basic and easiest to understand and implement. They are still in use, but the more advanced optimizers tend to be better for practical use. Which one to use is generally an emperical question depending on both the data and the model.

For a more complete overview of optimization algorithms see this comparison, and to see what is available in Keras, see the optimizer section of the documentation.

See the images below for a comparison of optimizers in a 2D space (NAG: Nesterov accelerated gradient, Adadelta: an extension of AdaGrad).

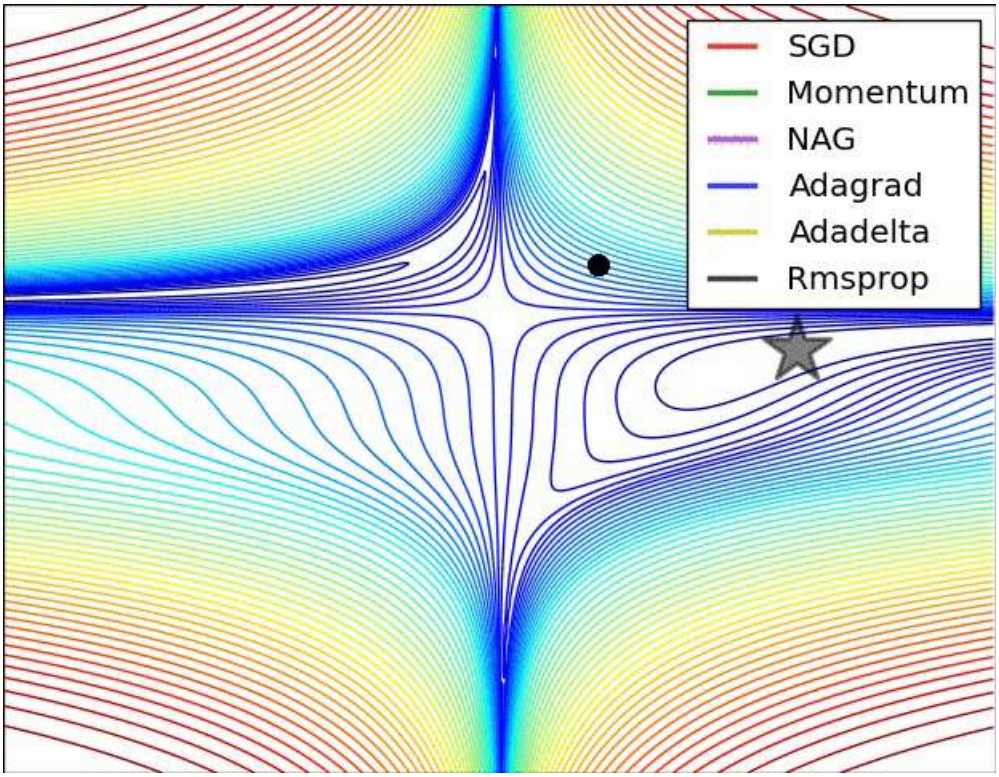


Figure 2. Comparison of six different optimizers.

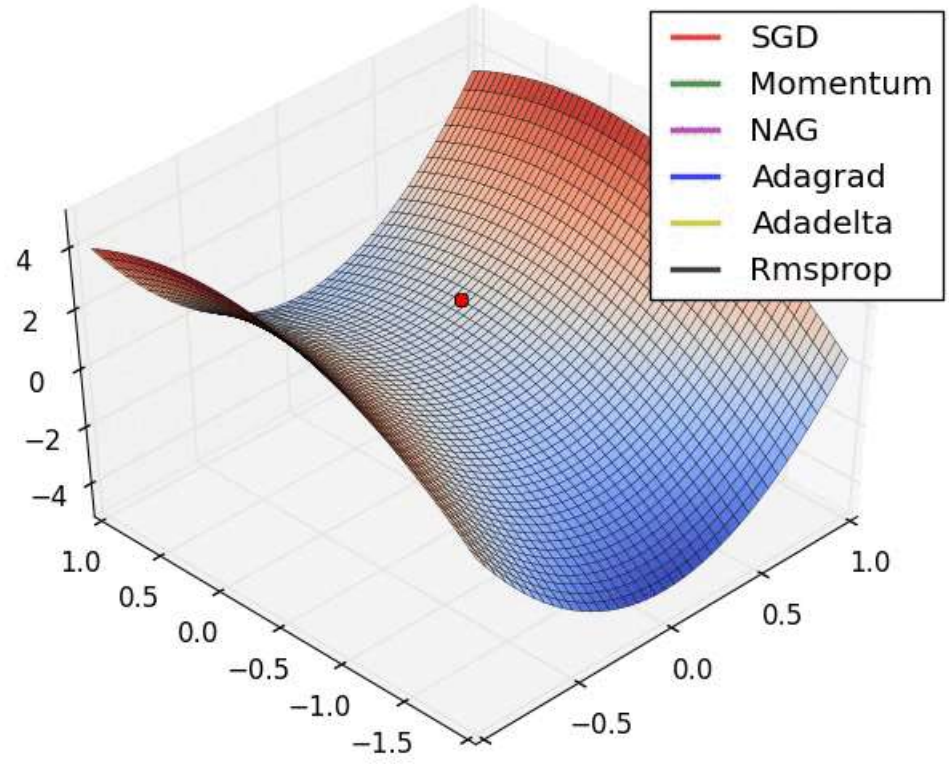


Figure 3. Comparison of six different optimizers at a saddle point.

```
# imports
import numpy as np
from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
# for the random seed
import tensorflow as tf
```

```
# set the random seeds to get reproducible results
np.random.seed(1)
tf.random.set_seed(2)

# Load data from https://www.openml.org/d/554
X, y = fetch_openml('mnist_784', version=1, return_X_y=True)
X, y = X[:1000], y[:1000]
X = X.reshape(X.shape[0], 28, 28, 1)
# Normalize
X = X / 255.
# number of unique classes
num_classes = len(np.unique(y))
y = y.astype(int)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=.2, random_state=1)

num_tot = y.shape[0]
num_train = y_train.shape[0]
num_test = y_test.shape[0]

y_oh = np.zeros((num_tot, num_classes))
y_oh[range(num_tot), y] = 1

y_oh_train = np.zeros((num_train, num_classes))
y_oh_train[range(num_train), y_train] = 1

y_oh_test = np.zeros((num_test, num_classes))
y_oh_test[range(num_test), y_test] = 1


type(X)

     numpy.ndarray


X.shape

     (1000, 28, 28, 1)


type(y)

     numpy.ndarray


y.shape

     (1000,)


ex_index = np.random.randint(y_train.shape[0], size=3)
ex_index

     array([ 37, 235,  72])


y_oh_ex = y_oh[ex_index]
y_oh_ex

     array([[1., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
            [0., 0., 0., 1., 0., 0., 0., 0., 0., 0.],
            [0., 1., 0., 0., 0., 0., 0., 0., 0., 0.]])


y_ex = y[ex_index]
y_ex
```

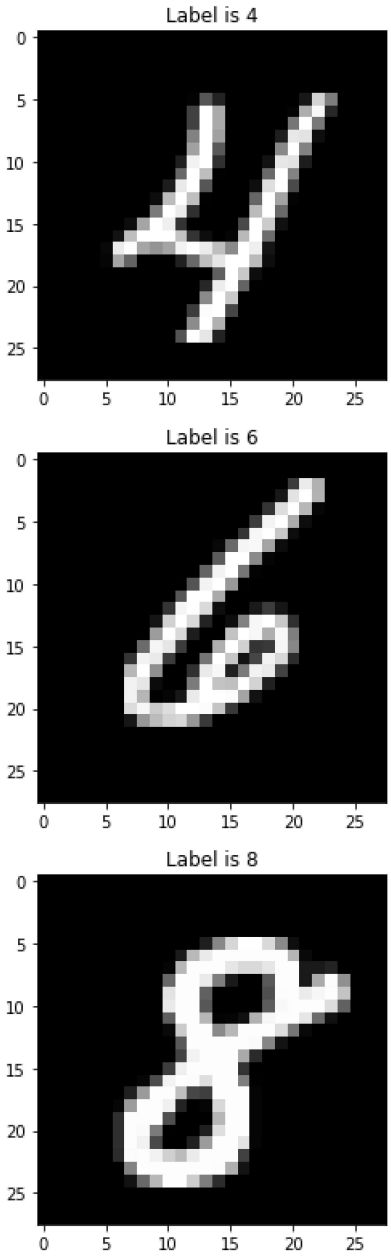
```
     array([0, 3, 1])
```

```
X_train[ex_index][0].shape
```

```
     (28, 28, 1)
```

```
X_ex = X_train[ex_index]
X_ex = X_ex.reshape(X_ex.shape[0],28,28)

for i in range(3):
  plt.title('Label is {label}'.format(label=y_train[ex_index][i]))
  fig = plt.figure
  plt.imshow(X_ex[i], cmap='gray')
  plt.show()
```



Question 1

The data set

Plot a three examples from the data set.

- What type of data are in the data set?

  ANS: Both X and y are numpy.ndarray but they have different shapes that X is in shape(1000,28,28,1) where y is in shape(1000,). That means X is a 4D array but y is a 1D array.

- What does the line `X = X.reshape(X.shape[0], 28, 28, 1)` do?

  ANS: The original dataset has 70000 rows and 784 features. Here we take the first 1000 rows as our dataset in this assignment. Before the reshape, X has a shape of (1000,784) which means 1000 rows and 784 columns. The reshape command here is to transform X dataset to a dataset that contains 1000 images(rows), each of size 28*28 pixels(28^2=784 columns). And since they are grayscale images, the "1" is an empty dimension to match the input shape of the neural network.

Look at how the encoding of the targets (i.e. `y`) is changed. E.g. the lines

```
y_oh = np.zeros((num_tot, num_classes))
y_oh[range(num_tot), y] = 1
```

Print out a few rows of `y` next to `y_oh`.

- What is the relationship between `y` and `y_oh`?

  ANS: y_oh is a 2D array whose length is the same as y, 1000. But y_oh has 10 columns representing y has 10 unique values. First we create a 2D array with all values are 0 in shape of (1000,10). Then we assign 1s to the 2D array according to the exact value of y. For example, if the 101th value of y is 6, then we fill the y_oh[100,5] with the number of 1. Therefore, we transformed the value of 6 to a expression of 0000010000. In this way, we made each entry of y become a binary series with length of 10. And we made a copy of y in catagorical variables instead of different integers.

- What is the type of encoding in `y_oh` called and why is it used?

  ANS: This is called one-hot encoding. This is used because we want to take the value of y into catagorical variables to make ML algorithms to do a better job in prediction. The one hot encoding allows can convert categorical data into numbers, and these numbers are not ordered value, they are the symbols of catagories. This is required for both input and output variables that are categorical. When a one hot encoding is used for the output variable, it may offer a more nuanced set of predictions than a single label, and therefore, we can estimate the MSE loss correspondingly.

- Plot three data examples in the same figure and set the correct label as title.
  - It should be possible to see what the data represent.

    ANS: Plots are shown as output above.

Question 2

The model

Below is some code for bulding and training a model with Keras.

- What type of network is implemented below? I.e. a normal MLP, RNN, CNN, Logistic Regression...?

  ANS : This is a Convolutional Neural Network (CNN), maybe a multi-layer CNN since there are two layers of ReLu.

- What does `Dropout()` do?

  ANS: The dropout function here allows us to ignore a certain proportion of units in the CNN network. By dropping a unit out, we temporarily ignore it from the network, along with all its incoming and outgoing (hidden and visible) connections. The number specified in the Dropout() function is the fraction of the input units to drop.

  Dropout function allow us to learn a fraction of the inputs in the network in each training iteration, instead of learning all the inputs together. Therefore, we can prevent the co-dependency between parameters developed during training, and

we can prevent our fitted model from overfitting.

- Which type of activation function is used for the hidden layers?

  ANS : ReLu function is used for the hidden layers.

- Which type of activation function is used for the output layer?

  ANS: Softmax function is used for the output layer.

- Why are two different activation functions used?

  ANS : Because ReLu and Softmax have different advantages and uses.

  ReLu function could train the network quickly because the gradient of ReLu is always high compared to other activation functions so that ReLu allows the network to converge very quickly. However, the Softmax function is totally different with ReLu. Softmax function is often used in the output layer because Softmax is able to normalizes the outputs for each catagory between 0 and 1, and divides by their sum, giving the probability of the input value being in a specific catagotry.

  To summarize, ReLu is used to train the model quickly while Softmax is to provide probability of a input belonging to a specific catagory.

- What optimizer is used in the model below?

  ANS : Stochastic gradient descent is used as the optimizer here.

- How often are the weights updated (i.e. after how many data examples)?

  ANS: We can see that we set the batch size to 32. And A batch size of 32 means that 32 samples from the training dataset will be used to estimate the error gradient before the model weights are updated.

  Therefore, the weights will be updated after every 32 data examples.

- What loss function is used?

  ANS : Categorical Crossentropy loss function is used since we have 10 different label catagories. This loss function allows us to compute the crossentropy loss between the labels and predictions.

- How many parameters (i.e. weights and biases, NOT hyper-parameters) does the model have?

  ANS : All Max-pooling functions do not require any parameters because it is a mathematical operation to find the maximum.

  The first Conv2D layer has 16 * (3 * 3+1) = 160 parameters. The second Conv2d layer has (3 * 3 * 16+1) * 32 = 4640 parameters. The third layer is a dense layer, from the output of model.summary() we can see that the third layer has (800+1) * 128 = 102528 parameters. And the last layer is a Softmax layer which has (128 * 10 +10) = 1290 parameters. Full architecture of the model is shown with the model.summary() command.

  Therefore, this model has 160 + 4640 + 102528 + 1290 = 108618 parameters.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, Flatten
from tensorflow.keras.layers import Conv2D, MaxPooling2D
from tensorflow.keras.optimizers import SGD


model = Sequential()

model.add(Conv2D(16, (3, 3), activation='relu', input_shape=(28, 28, 1)))
# Max pooling
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.))
```

```
model.add(Conv2D(32, (3, 3), activation='relu'))
# Max pooling
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())

model.add(Dense(128, activation='relu'))
model.add(Dropout(0.))

model.add(Dense(10, activation='softmax'))

sgd = SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)

# Compile the model
model.compile(loss='categorical_crossentropy', optimizer=sgd)

# Train the model
model.fit(X_train, y_oh_train, batch_size=32, epochs=60)

# Evaluate performance
test_loss = model.evaluate(X_test, y_oh_test, batch_size=32)

predictions = model.predict(X_test, batch_size=32)
predictions = np.argmax(predictions, axis=1) # change encoding again
print('Accuracy:', (predictions == y_test).sum() / predictions.shape[0])
```

```
Epoch 1/60
25/25 [==============================] - 1s 19ms/step - loss: 2.2322
Epoch 2/60
25/25 [==============================] - 0s 19ms/step - loss: 1.3094
Epoch 3/60
25/25 [==============================] - 0s 20ms/step - loss: 0.5476
Epoch 4/60
25/25 [==============================] - 0s 20ms/step - loss: 0.3458
Epoch 5/60
25/25 [==============================] - 0s 20ms/step - loss: 0.2481
Epoch 6/60
25/25 [==============================] - 0s 19ms/step - loss: 0.1618
Epoch 7/60
25/25 [==============================] - 1s 20ms/step - loss: 0.1702
Epoch 8/60
25/25 [==============================] - 0s 19ms/step - loss: 0.1203
Epoch 9/60
25/25 [==============================] - 0s 19ms/step - loss: 0.0991
Epoch 10/60
25/25 [==============================] - 1s 21ms/step - loss: 0.0544
Epoch 11/60
25/25 [==============================] - 1s 24ms/step - loss: 0.0416
Epoch 12/60
25/25 [==============================] - 1s 20ms/step - loss: 0.0405
Epoch 13/60
25/25 [==============================] - 1s 20ms/step - loss: 0.0346
Epoch 14/60
25/25 [==============================] - 0s 19ms/step - loss: 0.0197
Epoch 15/60
25/25 [==============================] - 0s 20ms/step - loss: 0.0155
Epoch 16/60
25/25 [==============================] - 0s 19ms/step - loss: 0.0179
Epoch 17/60
25/25 [==============================] - 0s 19ms/step - loss: 0.0119
Epoch 18/60
25/25 [==============================] - 0s 19ms/step - loss: 0.0071
Epoch 19/60
25/25 [==============================] - 0s 19ms/step - loss: 0.0077
Epoch 20/60
25/25 [==============================] - 1s 20ms/step - loss: 0.0040
Epoch 21/60
25/25 [==============================] - 1s 22ms/step - loss: 0.0042
Epoch 22/60
25/25 [==============================] - 1s 20ms/step - loss: 0.0032
```

```
Epoch 23/60
25/25 [==============================] - 0s 20ms/step - loss: 0.0035
Epoch 24/60
25/25 [==============================] - 1s 21ms/step - loss: 0.0023
Epoch 25/60
25/25 [==============================] - 1s 23ms/step - loss: 0.0022
Epoch 26/60
25/25 [==============================] - 1s 22ms/step - loss: 0.0022
Epoch 27/60
25/25 [==============================] - 1s 24ms/step - loss: 0.0021
Epoch 28/60
25/25 [==============================] - 0s 19ms/step - loss: 0.0019
Epoch 29/60
25/25 [==============================] - 0s 20ms/step - loss: 0.0019
Epoch 30/60
```

```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 26, 26, 16)        160
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 13, 13, 16)        0
_________________________________________________________________
dropout (Dropout)            (None, 13, 13, 16)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 11, 11, 32)        4640
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 5, 5, 32)          0
_________________________________________________________________
flatten (Flatten)            (None, 800)               0
_________________________________________________________________
dense (Dense)                (None, 128)               102528
_________________________________________________________________
dropout_1 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_1 (Dense)              (None, 10)                1290
=================================================================
Total params: 108,618
Trainable params: 108,618
Non-trainable params: 0
_________________________________________________________________
```

Part 2 - train a model

A model's performance depends on many factors apart from the model architecture (e.g. type and number of layers) and the dataset. Here you will get to explore some of the factors that affect model performance. Much of the skill in training deep learning models lies in quickly finding good values/options for these choises.

In order to observe the learning process it is best to compare the training set loss with the loss on the test set. How to visualize these variables with Keras is described under Training history visualization in the documentation.

You will explore the effect of 1) optimizer, 2) training duration, and 3) dropout (see the question above).

When training, an **epoch** is one pass through the full training set.

Question 3

- **Vizualize the training**. Use the model above to observe the training process. Train it for 150 epochs and then plot both "loss" and "val_loss" (i.e. loss on the valiadtion set, here the terms "validation set" and "test set" are used interchangably, but this is not always true). What is the optimal number of epochs for minimizing the test set loss?

    - Remember to first reset the weights (`model.reset_states()`), otherwise the training just continues from where it was stopped earlier.

- ANS: Below are the codes I run to visualize the loss and val_loss. Since the model.rest_state() does not work and model.compile() does not work either, I re-set the entire model each time. In the following code I train the model for 150 epochs. And from the plot in the output, we can see that the val_loss is approximately decreasing in the first 20 epochs and then the val_loss is steady increasing after 25 ~ 30 epochs.

  Therefore, I would say the at least 30 epochs are required for the best performance(lowest val_loss). However I would

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, Flatten
from tensorflow.keras.layers import Conv2D, MaxPooling2D
from tensorflow.keras.optimizers import SGD

model = Sequential()

model.add(Conv2D(16, (3, 3), activation='relu', input_shape=(28, 28, 1)))
# Max pooling
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.))

model.add(Conv2D(32, (3, 3), activation='relu'))
# Max pooling
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())

model.add(Dense(128, activation='relu'))
model.add(Dropout(0.))

model.add(Dense(10, activation='softmax'))

sgd = SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)

# Compile the model
model.compile(loss='categorical_crossentropy', optimizer=sgd)

# Train the model
history = model.fit(X_train, y_oh_train, batch_size=32, epochs=150, validation_data=(X_test, y_oh_test))

# Evaluate performance
test_loss = model.evaluate(X_test, y_oh_test, batch_size=32)

predictions = model.predict(X_test, batch_size=32)
predictions = np.argmax(predictions, axis=1) # change encoding again
print('Accuracy:', (predictions == y_test).sum() / predictions.shape[0])

# Plot learning
fig = plt.figure(figsize=(12,4))
ax0 = fig.add_subplot(121)
ax0.plot(history.history['val_loss'], label='validation loss')
ax0.plot(history.history['loss'], label='training loss')
ax0.set_ylabel('Loss')
ax0.set_xlabel('Epoch')
ax0.legend()
```

```
Epoch 1/150
25/25 [==============================] - 1s 30ms/step - loss: 2.2317 - val_loss: 1.7812
Epoch 2/150
25/25 [==============================] - 1s 25ms/step - loss: 1.2515 - val_loss: 0.7918
Epoch 3/150
25/25 [==============================] - 1s 26ms/step - loss: 0.5384 - val_loss: 0.4853
Epoch 4/150
25/25 [==============================] - 1s 34ms/step - loss: 0.3447 - val_loss: 0.3674
Epoch 5/150
25/25 [==============================] - 1s 26ms/step - loss: 0.2615 - val_loss: 0.4285
Epoch 6/150
25/25 [==============================] - 1s 22ms/step - loss: 0.2047 - val_loss: 0.3760
Epoch 7/150
25/25 [==============================] - 1s 22ms/step - loss: 0.1651 - val_loss: 0.3550
Epoch 8/150
25/25 [==============================] - 1s 26ms/step - loss: 0.1485 - val_loss: 0.3529
Epoch 9/150
25/25 [==============================] - 1s 26ms/step - loss: 0.0979 - val_loss: 0.3083
Epoch 10/150
25/25 [==============================] - 1s 25ms/step - loss: 0.0642 - val_loss: 0.3670
Epoch 11/150
25/25 [==============================] - 1s 24ms/step - loss: 0.0480 - val_loss: 0.3779
Epoch 12/150
25/25 [==============================] - 1s 22ms/step - loss: 0.0588 - val_loss: 0.3347
Epoch 13/150
25/25 [==============================] - 1s 22ms/step - loss: 0.0394 - val_loss: 0.3355
Epoch 14/150
25/25 [==============================] - 1s 23ms/step - loss: 0.0254 - val_loss: 0.3773
Epoch 15/150
25/25 [==============================] - 1s 21ms/step - loss: 0.0252 - val_loss: 0.3683
Epoch 16/150
25/25 [==============================] - 1s 24ms/step - loss: 0.0289 - val_loss: 0.3750
Epoch 17/150
25/25 [==============================] - 1s 26ms/step - loss: 0.0172 - val_loss: 0.3083
Epoch 18/150
25/25 [==============================] - 1s 26ms/step - loss: 0.0071 - val_loss: 0.3438
Epoch 19/150
25/25 [==============================] - 1s 25ms/step - loss: 0.0071 - val_loss: 0.3501
Epoch 20/150
25/25 [==============================] - 1s 23ms/step - loss: 0.0047 - val_loss: 0.3726
Epoch 21/150
25/25 [==============================] - 1s 24ms/step - loss: 0.0037 - val_loss: 0.3642
Epoch 22/150
25/25 [==============================] - 1s 25ms/step - loss: 0.0030 - val_loss: 0.3717
Epoch 23/150
25/25 [==============================] - 1s 23ms/step - loss: 0.0029 - val_loss: 0.3664
Epoch 24/150
25/25 [==============================] - 1s 22ms/step - loss: 0.0021 - val_loss: 0.3806
Epoch 25/150
25/25 [==============================] - 1s 23ms/step - loss: 0.0021 - val_loss: 0.3783
Epoch 26/150
25/25 [==============================] - 1s 23ms/step - loss: 0.0021 - val_loss: 0.3848
Epoch 27/150
25/25 [==============================] - 1s 26ms/step - loss: 0.0019 - val_loss: 0.3901
Epoch 28/150
25/25 [==============================] - 1s 25ms/step - loss: 0.0017 - val_loss: 0.3885
Epoch 29/150
25/25 [==============================] - 1s 22ms/step - loss: 0.0017 - val_loss: 0.3952
Epoch 30/150
25/25 [==============================] - 1s 22ms/step - loss: 0.0014 - val_loss: 0.3944
Epoch 31/150
25/25 [==============================] - 1s 23ms/step - loss: 0.0015 - val_loss: 0.3976
Epoch 32/150
25/25 [==============================] - 1s 27ms/step - loss: 0.0014 - val_loss: 0.4037
Epoch 33/150
25/25 [==============================] - 1s 25ms/step - loss: 0.0012 - val_loss: 0.4005
Epoch 34/150
25/25 [==============================] - 1s 22ms/step - loss: 0.0012 - val_loss: 0.4063
Epoch 35/150
25/25 [==============================] - 1s 22ms/step - loss: 0.0012 - val_loss: 0.4084
Epoch 36/150
25/25 [==============================] - 1s 22ms/step - loss: 9.8031e-04 - val_loss: 0.4090
```

```
25/25 [==============================] - 1s 22ms/step - loss: 9.8051e-04 - val_loss: 0.4090
Epoch 37/150
25/25 [==============================] - 1s 22ms/step - loss: 0.0011 - val_loss: 0.4141
Epoch 38/150
25/25 [==============================] - 1s 22ms/step - loss: 0.0011 - val_loss: 0.4146
Epoch 39/150
25/25 [==============================] - 1s 26ms/step - loss: 0.0011 - val_loss: 0.4188
Epoch 40/150
25/25 [==============================] - 1s 24ms/step - loss: 9.5357e-04 - val_loss: 0.4164
Epoch 41/150
25/25 [==============================] - 1s 23ms/step - loss: 9.2746e-04 - val_loss: 0.4199
Epoch 42/150
25/25 [==============================] - 1s 22ms/step - loss: 8.8658e-04 - val_loss: 0.4209
Epoch 43/150
25/25 [==============================] - 1s 22ms/step - loss: 8.1601e-04 - val_loss: 0.4244
Epoch 44/150
25/25 [==============================] - 1s 23ms/step - loss: 6.9085e-04 - val_loss: 0.4255
Epoch 45/150
25/25 [==============================] - 1s 24ms/step - loss: 6.8848e-04 - val_loss: 0.4255
Epoch 46/150
25/25 [==============================] - 1s 22ms/step - loss: 8.0820e-04 - val_loss: 0.4287
Epoch 47/150
25/25 [==============================] - 1s 24ms/step - loss: 6.6913e-04 - val_loss: 0.4314
Epoch 48/150
25/25 [==============================] - 1s 34ms/step - loss: 5.9091e-04 - val_loss: 0.4321
Epoch 49/150
25/25 [==============================] - 1s 23ms/step - loss: 7.0046e-04 - val_loss: 0.4317
Epoch 50/150
25/25 [==============================] - 1s 22ms/step - loss: 6.2281e-04 - val_loss: 0.4357
Epoch 51/150
25/25 [==============================] - 1s 22ms/step - loss: 5.8914e-04 - val_loss: 0.4352
Epoch 52/150
25/25 [==============================] - 1s 23ms/step - loss: 5.2691e-04 - val_loss: 0.4361
Epoch 53/150
25/25 [==============================] - 1s 22ms/step - loss: 5.2206e-04 - val_loss: 0.4375
Epoch 54/150
25/25 [==============================] - 1s 24ms/step - loss: 5.3382e-04 - val_loss: 0.4402
Epoch 55/150
25/25 [==============================] - 1s 26ms/step - loss: 6.5266e-04 - val_loss: 0.4399
Epoch 56/150
25/25 [==============================] - 1s 25ms/step - loss: 5.4927e-04 - val_loss: 0.4431
Epoch 57/150
25/25 [==============================] - 1s 25ms/step - loss: 4.6083e-04 - val_loss: 0.4435
Epoch 58/150
25/25 [==============================] - 1s 26ms/step - loss: 5.6716e-04 - val_loss: 0.4439
Epoch 59/150
25/25 [==============================] - 1s 26ms/step - loss: 5.6154e-04 - val_loss: 0.4471
Epoch 60/150
25/25 [==============================] - 1s 26ms/step - loss: 5.3198e-04 - val_loss: 0.4460
Epoch 61/150
25/25 [==============================] - 1s 26ms/step - loss: 4.9546e-04 - val_loss: 0.4498
Epoch 62/150
25/25 [==============================] - 1s 26ms/step - loss: 4.6048e-04 - val_loss: 0.4499
Epoch 63/150
25/25 [==============================] - 1s 26ms/step - loss: 4.8227e-04 - val_loss: 0.4479
Epoch 64/150
25/25 [==============================] - 1s 22ms/step - loss: 4.4597e-04 - val_loss: 0.4511
Epoch 65/150
25/25 [==============================] - 1s 23ms/step - loss: 4.1749e-04 - val_loss: 0.4519
Epoch 66/150
25/25 [==============================] - 1s 25ms/step - loss: 3.9417e-04 - val_loss: 0.4527
Epoch 67/150
25/25 [==============================] - 1s 25ms/step - loss: 3.8451e-04 - val_loss: 0.4530
Epoch 68/150
25/25 [==============================] - 1s 22ms/step - loss: 3.7447e-04 - val_loss: 0.4557
Epoch 69/150
25/25 [==============================] - 1s 24ms/step - loss: 3.9265e-04 - val_loss: 0.4561
Epoch 70/150
25/25 [==============================] - 1s 26ms/step - loss: 3.7356e-04 - val_loss: 0.4568
Epoch 71/150
25/25 [==============================] - 1s 26ms/step - loss: 3.6184e-04 - val_loss: 0.4573
Epoch 72/150
25/25 [==============================] - 1s 26ms/step - loss: 4.8039e-04 - val_loss: 0.4587
```

25/25 [==============================] - 1s 26ms/step - loss: 4.8059e-04 - val_loss: 0.4587
Epoch 73/150
25/25 [==============================] - 1s 26ms/step - loss: 4.1381e-04 - val_loss: 0.4598
Epoch 74/150
25/25 [==============================] - 1s 22ms/step - loss: 3.7948e-04 - val_loss: 0.4619
Epoch 75/150
25/25 [==============================] - 1s 23ms/step - loss: 3.3218e-04 - val_loss: 0.4594
Epoch 76/150
25/25 [==============================] - 1s 23ms/step - loss: 3.2240e-04 - val_loss: 0.4613
Epoch 77/150
25/25 [==============================] - 1s 24ms/step - loss: 3.5311e-04 - val_loss: 0.4634
Epoch 78/150
25/25 [==============================] - 1s 26ms/step - loss: 2.9094e-04 - val_loss: 0.4646
Epoch 79/150
25/25 [==============================] - 1s 26ms/step - loss: 3.6342e-04 - val_loss: 0.4635
Epoch 80/150
25/25 [==============================] - 1s 25ms/step - loss: 3.2788e-04 - val_loss: 0.4652
Epoch 81/150
25/25 [==============================] - 1s 25ms/step - loss: 2.7355e-04 - val_loss: 0.4653
Epoch 82/150
25/25 [==============================] - 1s 27ms/step - loss: 3.6827e-04 - val_loss: 0.4663
Epoch 83/150
25/25 [==============================] - 1s 26ms/step - loss: 3.1238e-04 - val_loss: 0.4688
Epoch 84/150
25/25 [==============================] - 1s 23ms/step - loss: 3.4938e-04 - val_loss: 0.4676
Epoch 85/150
25/25 [==============================] - 1s 22ms/step - loss: 3.6110e-04 - val_loss: 0.4689
Epoch 86/150
25/25 [==============================] - 1s 22ms/step - loss: 3.0831e-04 - val_loss: 0.4697
Epoch 87/150
25/25 [==============================] - 1s 25ms/step - loss: 2.8078e-04 - val_loss: 0.4702
Epoch 88/150
25/25 [==============================] - 1s 25ms/step - loss: 2.8954e-04 - val_loss: 0.4702
Epoch 89/150
25/25 [==============================] - 1s 23ms/step - loss: 2.9907e-04 - val_loss: 0.4713
Epoch 90/150
25/25 [==============================] - 1s 23ms/step - loss: 2.7487e-04 - val_loss: 0.4726
Epoch 91/150
25/25 [==============================] - 1s 22ms/step - loss: 2.5759e-04 - val_loss: 0.4732
Epoch 92/150
25/25 [==============================] - 1s 31ms/step - loss: 2.9935e-04 - val_loss: 0.4733
Epoch 93/150
25/25 [==============================] - 1s 26ms/step - loss: 2.7350e-04 - val_loss: 0.4748
Epoch 94/150
25/25 [==============================] - 1s 29ms/step - loss: 2.5524e-04 - val_loss: 0.4748
Epoch 95/150
25/25 [==============================] - 1s 27ms/step - loss: 2.8040e-04 - val_loss: 0.4757
Epoch 96/150
25/25 [==============================] - 1s 26ms/step - loss: 2.2801e-04 - val_loss: 0.4775
Epoch 97/150
25/25 [==============================] - 1s 24ms/step - loss: 2.6851e-04 - val_loss: 0.4760
Epoch 98/150
25/25 [==============================] - 1s 24ms/step - loss: 2.3550e-04 - val_loss: 0.4780
Epoch 99/150
25/25 [==============================] - 1s 25ms/step - loss: 2.3886e-04 - val_loss: 0.4783
Epoch 100/150
25/25 [==============================] - 1s 25ms/step - loss: 2.2607e-04 - val_loss: 0.4796
Epoch 101/150
25/25 [==============================] - 1s 26ms/step - loss: 2.3051e-04 - val_loss: 0.4787
Epoch 102/150
25/25 [==============================] - 1s 25ms/step - loss: 2.8846e-04 - val_loss: 0.4800
Epoch 103/150
25/25 [==============================] - 1s 24ms/step - loss: 2.6176e-04 - val_loss: 0.4809
Epoch 104/150
25/25 [==============================] - 1s 23ms/step - loss: 2.7580e-04 - val_loss: 0.4814
Epoch 105/150
25/25 [==============================] - 1s 22ms/step - loss: 2.5323e-04 - val_loss: 0.4818
Epoch 106/150
25/25 [==============================] - 1s 24ms/step - loss: 1.9882e-04 - val_loss: 0.4838
Epoch 107/150
25/25 [==============================] - 1s 22ms/step - loss: 2.4111e-04 - val_loss: 0.4832
Epoch 108/150
25/25 [==============================] - 1s 23ms/step - loss: 2.5817e-04 - val_loss: 0.4833

```
25/25 [==============================] - 1s 23ms/step - loss: 2.9817e-04 - val_loss: 0.4859
Epoch 109/150
25/25 [==============================] - 1s 23ms/step - loss: 2.2261e-04 - val_loss: 0.4832
Epoch 110/150
25/25 [==============================] - 1s 22ms/step - loss: 2.1806e-04 - val_loss: 0.4857
Epoch 111/150
25/25 [==============================] - 1s 23ms/step - loss: 2.5266e-04 - val_loss: 0.4848
Epoch 112/150
25/25 [==============================] - 1s 23ms/step - loss: 2.1069e-04 - val_loss: 0.4859
Epoch 113/150
25/25 [==============================] - 1s 23ms/step - loss: 1.8586e-04 - val_loss: 0.4865
Epoch 114/150
25/25 [==============================] - 1s 22ms/step - loss: 2.1460e-04 - val_loss: 0.4869
Epoch 115/150
25/25 [==============================] - 1s 23ms/step - loss: 1.8935e-04 - val_loss: 0.4874
Epoch 116/150
25/25 [==============================] - 1s 23ms/step - loss: 2.2871e-04 - val_loss: 0.4880
Epoch 117/150
25/25 [==============================] - 1s 24ms/step - loss: 1.8353e-04 - val_loss: 0.4886
Epoch 118/150
25/25 [==============================] - 1s 26ms/step - loss: 2.2334e-04 - val_loss: 0.4891
Epoch 119/150
25/25 [==============================] - 1s 25ms/step - loss: 1.8152e-04 - val_loss: 0.4897
Epoch 120/150
25/25 [==============================] - 1s 26ms/step - loss: 1.7862e-04 - val_loss: 0.4898
Epoch 121/150
25/25 [==============================] - 1s 23ms/step - loss: 1.7904e-04 - val_loss: 0.4907
Epoch 122/150
25/25 [==============================] - 1s 24ms/step - loss: 1.9358e-04 - val_loss: 0.4913
Epoch 123/150
25/25 [==============================] - 1s 26ms/step - loss: 2.0359e-04 - val_loss: 0.4904
Epoch 124/150
25/25 [==============================] - 1s 26ms/step - loss: 2.2004e-04 - val_loss: 0.4924
Epoch 125/150
25/25 [==============================] - 1s 26ms/step - loss: 1.6767e-04 - val_loss: 0.4934
Epoch 126/150
25/25 [==============================] - 1s 26ms/step - loss: 2.0147e-04 - val_loss: 0.4930
Epoch 127/150
25/25 [==============================] - 1s 25ms/step - loss: 1.6473e-04 - val_loss: 0.4927
Epoch 128/150
25/25 [==============================] - 1s 27ms/step - loss: 1.7369e-04 - val_loss: 0.4932
Epoch 129/150
25/25 [==============================] - 1s 22ms/step - loss: 1.8280e-04 - val_loss: 0.4940
Epoch 130/150
25/25 [==============================] - 1s 23ms/step - loss: 1.6621e-04 - val_loss: 0.4943
Epoch 131/150
25/25 [==============================] - 1s 24ms/step - loss: 1.6256e-04 - val_loss: 0.4955
Epoch 132/150
25/25 [==============================] - 1s 22ms/step - loss: 1.8444e-04 - val_loss: 0.4957
Epoch 133/150
25/25 [==============================] - 1s 23ms/step - loss: 1.6103e-04 - val_loss: 0.4966
Epoch 134/150
25/25 [==============================] - 1s 23ms/step - loss: 1.7540e-04 - val_loss: 0.4958
Epoch 135/150
25/25 [==============================] - 1s 23ms/step - loss: 1.7207e-04 - val_loss: 0.4970
Epoch 136/150
25/25 [==============================] - 1s 33ms/step - loss: 1.6454e-04 - val_loss: 0.4971
Epoch 137/150
25/25 [==============================] - 1s 24ms/step - loss: 1.6876e-04 - val_loss: 0.4977
Epoch 138/150
25/25 [==============================] - 1s 23ms/step - loss: 1.5186e-04 - val_loss: 0.4981
Epoch 139/150
25/25 [==============================] - 1s 23ms/step - loss: 1.6753e-04 - val_loss: 0.4986
Epoch 140/150
25/25 [==============================] - 1s 23ms/step - loss: 1.5816e-04 - val_loss: 0.4990
Epoch 141/150
25/25 [==============================] - 1s 27ms/step - loss: 1.9293e-04 - val_loss: 0.5002
Epoch 142/150
25/25 [==============================] - 1s 26ms/step - loss: 1.5653e-04 - val_loss: 0.5005
Epoch 143/150
25/25 [==============================] - 1s 26ms/step - loss: 1.5908e-04 - val_loss: 0.5006
Epoch 144/150
25/25 [==============================] - 1s 23ms/step - loss: 1.7403e-04 - val_loss: 0.5010
```

```
25/25 [==============================] - 1s 23ms/step - loss: 1.7405e-04 - val_loss: 0.5010
Epoch 145/150
25/25 [==============================] - 1s 23ms/step - loss: 1.4901e-04 - val_loss: 0.5019
Epoch 146/150
25/25 [==============================] - 1s 23ms/step - loss: 1.4301e-04 - val_loss: 0.5011
Epoch 147/150
25/25 [==============================] - 1s 23ms/step - loss: 1.6392e-04 - val_loss: 0.5015
Epoch 148/150
25/25 [==============================] - 1s 22ms/step - loss: 1.4850e-04 - val_loss: 0.5033
Epoch 149/150
25/25 [==============================] - 1s 24ms/step - loss: 1.3602e-04 - val_loss: 0.5026
Epoch 150/150
25/25 [==============================] - 1s 26ms/step - loss: 1.5112e-04 - val_loss: 0.5038
7/7 [==============================] - 0s 7ms/step - loss: 0.5038
Accuracy: 0.925
<matplotlib.legend.Legend at 0x7f6c1c31f278>
```

- **Optimizer**. Select three different optimizers and for each find the close-to-optimal hyper-parameter(s). In your answer, include a) your three choises, b) best hyper-parameters for each of the three optimizers and, c) the code that produced the results.
    - *NOTE* that how long the training takes varies with optimizer. I.e., make sure that the model is trained for long enough to reach optimal performance.

      ANS : Here I used Adam, Nadam(NAG) and Adadelta as the three optimizers I chose. Below are the codes to use Grid Search to find the optimal hyper-parameters. All Grid Search are done based on 50 epochs.

      Best hyper-parameters for Adam: beta_1 = 0.5, beta_2 = 0.8, learning_rate = 0.01. The final accuracy score is 0.931233

      Best hyper-parameters for Nadam(NAG): beta_1 = 0.9, beta_2 = 0.8, learning_rate = 1. The final accuracy score is 0.925000.

      Best hyper-parameters for Adadelta: rho = 1, learning_rate = 0.1. The final accuracy score is 0.191218.

```
# Use scikit-learn to grid search the hyperparameter for Adam
import numpy
from tensorflow.keras.optimizers import Nadam
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.optimizers import Adadelta
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, Flatten
from tensorflow.keras.layers import Conv2D, MaxPooling2D
from sklearn.model_selection import GridSearchCV
from keras.models import Sequential
from keras.layers import Dense
from keras.wrappers.scikit_learn import KerasClassifier

Adam = Adam(learning_rate=0.001,beta_1=0.9,beta_2=0.999,epsilon=1e-07,amsgrad=False)
Adadelta = Adadelta(learning_rate=0.001, rho=0.95, epsilon=1e-07)
NAG = Nadam(learning_rate=0.001, beta_1=0.9, beta_2=0.999, epsilon=1e-07)

# Function to create model, required for KerasClassifier
def create_model(learning_rate=0.001, beta_1=0.9, beta_2=0.999):
  # create model
  model = Sequential()
  model.add(Conv2D(16, (3, 3), activation='relu', input_shape=(28, 28, 1)))
  # Max pooling
  model.add(MaxPooling2D(pool_size=(2, 2)))
```

```
  model.add(MaxPooling2D(pool_size=(2, 2)))
  model.add(Dropout(0.))
  model.add(Conv2D(32, (3, 3), activation='relu'))
  # Max pooling
  model.add(MaxPooling2D(pool_size=(2, 2)))
  model.add(Flatten())
  model.add(Dense(128, activation='relu'))
  model.add(Dropout(0.))
  model.add(Dense(10, activation='softmax'))


  # Compile the model
  model.compile(loss='categorical_crossentropy', optimizer=Adam, metrics=["accuracy"])

  return model
# fix random seed for reproducibility
seed = 1
numpy.random.seed(seed)

# create model
model = KerasClassifier(build_fn=create_model, verbose=0)
# define the grid search parameters
space = dict()
space['learning_rate'] = [0.001, 0.01, 0.1,1]
space['beta_1'] = [0.5,0.7,0.9]
space['beta_2'] = [0.8,0.9,0.999]
space['batch_size'] = [32]
space['epochs']=[50]


grid = GridSearchCV(estimator=model, param_grid=space, n_jobs=-1, cv=3)
grid_result = grid.fit(X_train, y_oh_train)

# summarize results
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print("%f (%f) with: %r" % (mean, stdev, param))
```

```
Best: 0.931233 using {'batch_size': 32, 'beta_1': 0.5, 'beta_2': 0.8, 'epochs': 50, 'learning_rate': 0.01}
0.913736 (0.015977) with: {'batch_size': 32, 'beta_1': 0.5, 'beta_2': 0.8, 'epochs': 50, 'learning_rate': 0.001
0.931233 (0.014550) with: {'batch_size': 32, 'beta_1': 0.5, 'beta_2': 0.8, 'epochs': 50, 'learning_rate': 0.01}
0.917472 (0.020201) with: {'batch_size': 32, 'beta_1': 0.5, 'beta_2': 0.8, 'epochs': 50, 'learning_rate': 0.1}
0.911243 (0.015731) with: {'batch_size': 32, 'beta_1': 0.5, 'beta_2': 0.8, 'epochs': 50, 'learning_rate': 1}
0.919992 (0.026391) with: {'batch_size': 32, 'beta_1': 0.5, 'beta_2': 0.9, 'epochs': 50, 'learning_rate': 0.001
0.914984 (0.015187) with: {'batch_size': 32, 'beta_1': 0.5, 'beta_2': 0.9, 'epochs': 50, 'learning_rate': 0.01}
0.912483 (0.019511) with: {'batch_size': 32, 'beta_1': 0.5, 'beta_2': 0.9, 'epochs': 50, 'learning_rate': 0.1}
0.923728 (0.019765) with: {'batch_size': 32, 'beta_1': 0.5, 'beta_2': 0.9, 'epochs': 50, 'learning_rate': 1}
0.916223 (0.020499) with: {'batch_size': 32, 'beta_1': 0.5, 'beta_2': 0.999, 'epochs': 50, 'learning_rate': 0.0
0.914975 (0.015563) with: {'batch_size': 32, 'beta_1': 0.5, 'beta_2': 0.999, 'epochs': 50, 'learning_rate': 0.0
0.922494 (0.015726) with: {'batch_size': 32, 'beta_1': 0.5, 'beta_2': 0.999, 'epochs': 50, 'learning_rate': 0.1
0.914993 (0.004793) with: {'batch_size': 32, 'beta_1': 0.5, 'beta_2': 0.999, 'epochs': 50, 'learning_rate': 1}
0.914989 (0.009935) with: {'batch_size': 32, 'beta_1': 0.7, 'beta_2': 0.8, 'epochs': 50, 'learning_rate': 0.001
0.920001 (0.010729) with: {'batch_size': 32, 'beta_1': 0.7, 'beta_2': 0.8, 'epochs': 50, 'learning_rate': 0.01}
0.912492 (0.008912) with: {'batch_size': 32, 'beta_1': 0.7, 'beta_2': 0.8, 'epochs': 50, 'learning_rate': 0.1}
0.913726 (0.014189) with: {'batch_size': 32, 'beta_1': 0.7, 'beta_2': 0.8, 'epochs': 50, 'learning_rate': 1}
0.908737 (0.014215) with: {'batch_size': 32, 'beta_1': 0.7, 'beta_2': 0.9, 'epochs': 50, 'learning_rate': 0.001
0.904992 (0.017435) with: {'batch_size': 32, 'beta_1': 0.7, 'beta_2': 0.9, 'epochs': 50, 'learning_rate': 0.01}
0.914989 (0.009935) with: {'batch_size': 32, 'beta_1': 0.7, 'beta_2': 0.9, 'epochs': 50, 'learning_rate': 0.1}
0.904987 (0.014218) with: {'batch_size': 32, 'beta_1': 0.7, 'beta_2': 0.9, 'epochs': 50, 'learning_rate': 1}
0.917486 (0.021468) with: {'batch_size': 32, 'beta_1': 0.7, 'beta_2': 0.999, 'epochs': 50, 'learning_rate': 0.0
```

```
0.914993 (0.007143) with: {'batch_size': 32, 'beta_1': 0.7, 'beta_2': 0.999, 'epochs': 50, 'learning_rate': 0.0
0.913736 (0.013433) with: {'batch_size': 32, 'beta_1': 0.7, 'beta_2': 0.999, 'epochs': 50, 'learning_rate': 0.1
0.909986 (0.011156) with: {'batch_size': 32, 'beta_1': 0.7, 'beta_2': 0.999, 'epochs': 50, 'learning_rate': 1}
0.921254 (0.005232) with: {'batch_size': 32, 'beta_1': 0.9, 'beta_2': 0.8, 'epochs': 50, 'learning_rate': 0.001
0.928745 (0.005364) with: {'batch_size': 32, 'beta_1': 0.9, 'beta_2': 0.8, 'epochs': 50, 'learning_rate': 0.01}
0.924990 (0.010667) with: {'batch_size': 32, 'beta_1': 0.9, 'beta_2': 0.8, 'epochs': 50, 'learning_rate': 0.1}
0.916242 (0.008909) with: {'batch_size': 32, 'beta_1': 0.9, 'beta_2': 0.8, 'epochs': 50, 'learning_rate': 1}
0.916237 (0.011680) with: {'batch_size': 32, 'beta_1': 0.9, 'beta_2': 0.9, 'epochs': 50, 'learning_rate': 0.001
0.911234 (0.023224) with: {'batch_size': 32, 'beta_1': 0.9, 'beta_2': 0.9, 'epochs': 50, 'learning_rate': 0.01}
0.913745 (0.003197) with: {'batch_size': 32, 'beta_1': 0.9, 'beta_2': 0.9, 'epochs': 50, 'learning_rate': 0.1}
0.909995 (0.020075) with: {'batch_size': 32, 'beta_1': 0.9, 'beta_2': 0.9, 'epochs': 50, 'learning_rate': 1}
0.909976 (0.023991) with: {'batch_size': 32, 'beta_1': 0.9, 'beta_2': 0.999, 'epochs': 50, 'learning_rate': 0.0
0.923733 (0.019501) with: {'batch_size': 32, 'beta_1': 0.9, 'beta_2': 0.999, 'epochs': 50, 'learning_rate': 0.0
0.919987 (0.009463) with: {'batch_size': 32, 'beta_1': 0.9, 'beta_2': 0.999, 'epochs': 50, 'learning_rate': 0.1
0.918748 (0.016851) with: {'batch_size': 32, 'beta_1': 0.9, 'beta_2': 0.999, 'epochs': 50, 'learning_rate': 1}
```

```
# Use scikit-learn to grid search the hyperparameter of Nadam (NAG)
import numpy
from tensorflow.keras.optimizers import Nadam
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.optimizers import Adadelta
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, Flatten
from tensorflow.keras.layers import Conv2D, MaxPooling2D
from sklearn.model_selection import GridSearchCV
from keras.models import Sequential
from keras.layers import Dense
from keras.wrappers.scikit_learn import KerasClassifier

Adam = Adam(learning_rate=0.001,beta_1=0.9,beta_2=0.999,epsilon=1e-07,amsgrad=False)
Adadelta = Adadelta(learning_rate=0.001, rho=0.95, epsilon=1e-07)
NAG = Nadam(learning_rate=0.001, beta_1=0.9, beta_2=0.999, epsilon=1e-07)

# Function to create model, required for KerasClassifier
def create_model(learning_rate=0.001, beta_1=0.9, beta_2=0.999):
  # create model
  model = Sequential()
  model.add(Conv2D(16, (3, 3), activation='relu', input_shape=(28, 28, 1)))
  # Max pooling
  model.add(MaxPooling2D(pool_size=(2, 2)))
  model.add(Dropout(0.))
  model.add(Conv2D(32, (3, 3), activation='relu'))
  # Max pooling
  model.add(MaxPooling2D(pool_size=(2, 2)))
  model.add(Flatten())
  model.add(Dense(128, activation='relu'))
  model.add(Dropout(0.))
  model.add(Dense(10, activation='softmax'))


  # Compile the model
  model.compile(loss='categorical_crossentropy', optimizer=NAG, metrics=["accuracy"])

  return model
# fix random seed for reproducibility
seed = 2
numpy.random.seed(seed)

# create model
model = KerasClassifier(build_fn=create_model, verbose=0)
# define the grid search parameters
space = dict()
space['learning_rate'] = [0.001, 0.01, 0.1,1]
space['beta_1'] = [0.5,0.7,0.9]
space['beta_2'] = [0.8,0.9,0.999]
```

```
space['beta_2'] = [0.8,0.9,0.999]
space['batch_size'] = [32]
space['epochs']=[50]


grid = GridSearchCV(estimator=model, param_grid=space, n_jobs=-1, cv=3)
grid_result = grid.fit(X_train, y_oh_train)
# summarize results
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print("%f (%f) with: %r" % (mean, stdev, param))
```

```
Best: 0.925000 using {'batch_size': 32, 'beta_1': 0.9, 'beta_2': 0.8, 'epochs': 50, 'learning_rate': 1}
0.921250 (0.015291) with: {'batch_size': 32, 'beta_1': 0.5, 'beta_2': 0.8, 'epochs': 50, 'learning_rate': 0.001
0.907489 (0.023800) with: {'batch_size': 32, 'beta_1': 0.5, 'beta_2': 0.8, 'epochs': 50, 'learning_rate': 0.01}
0.914998 (0.004702) with: {'batch_size': 32, 'beta_1': 0.5, 'beta_2': 0.8, 'epochs': 50, 'learning_rate': 0.1}
0.921240 (0.010671) with: {'batch_size': 32, 'beta_1': 0.5, 'beta_2': 0.8, 'epochs': 50, 'learning_rate': 1}
0.916251 (0.019891) with: {'batch_size': 32, 'beta_1': 0.5, 'beta_2': 0.9, 'epochs': 50, 'learning_rate': 0.001
0.920006 (0.006997) with: {'batch_size': 32, 'beta_1': 0.5, 'beta_2': 0.9, 'epochs': 50, 'learning_rate': 0.01}
0.914984 (0.009000) with: {'batch_size': 32, 'beta_1': 0.5, 'beta_2': 0.9, 'epochs': 50, 'learning_rate': 0.1}
0.911243 (0.018717) with: {'batch_size': 32, 'beta_1': 0.5, 'beta_2': 0.9, 'epochs': 50, 'learning_rate': 1}
0.921240 (0.022089) with: {'batch_size': 32, 'beta_1': 0.5, 'beta_2': 0.999, 'epochs': 50, 'learning_rate': 0.0
0.909990 (0.019146) with: {'batch_size': 32, 'beta_1': 0.5, 'beta_2': 0.999, 'epochs': 50, 'learning_rate': 0.0
0.909990 (0.016239) with: {'batch_size': 32, 'beta_1': 0.5, 'beta_2': 0.999, 'epochs': 50, 'learning_rate': 0.1
0.919997 (0.015409) with: {'batch_size': 32, 'beta_1': 0.5, 'beta_2': 0.999, 'epochs': 50, 'learning_rate': 1}
0.918739 (0.015151) with: {'batch_size': 32, 'beta_1': 0.7, 'beta_2': 0.8, 'epochs': 50, 'learning_rate': 0.001
0.922498 (0.010752) with: {'batch_size': 32, 'beta_1': 0.7, 'beta_2': 0.8, 'epochs': 50, 'learning_rate': 0.01}
0.923742 (0.011634) with: {'batch_size': 32, 'beta_1': 0.7, 'beta_2': 0.8, 'epochs': 50, 'learning_rate': 0.1}
0.912501 (0.019891) with: {'batch_size': 32, 'beta_1': 0.7, 'beta_2': 0.8, 'epochs': 50, 'learning_rate': 1}
0.913750 (0.012233) with: {'batch_size': 32, 'beta_1': 0.7, 'beta_2': 0.9, 'epochs': 50, 'learning_rate': 0.001
0.915003 (0.006327) with: {'batch_size': 32, 'beta_1': 0.7, 'beta_2': 0.9, 'epochs': 50, 'learning_rate': 0.01}
0.907498 (0.001854) with: {'batch_size': 32, 'beta_1': 0.7, 'beta_2': 0.9, 'epochs': 50, 'learning_rate': 0.1}
0.912492 (0.011642) with: {'batch_size': 32, 'beta_1': 0.7, 'beta_2': 0.9, 'epochs': 50, 'learning_rate': 1}
0.916237 (0.016913) with: {'batch_size': 32, 'beta_1': 0.7, 'beta_2': 0.999, 'epochs': 50, 'learning_rate': 0.0
0.916233 (0.014568) with: {'batch_size': 32, 'beta_1': 0.7, 'beta_2': 0.999, 'epochs': 50, 'learning_rate': 0.0
0.911253 (0.015375) with: {'batch_size': 32, 'beta_1': 0.7, 'beta_2': 0.999, 'epochs': 50, 'learning_rate': 0.1
0.916233 (0.016942) with: {'batch_size': 32, 'beta_1': 0.7, 'beta_2': 0.999, 'epochs': 50, 'learning_rate': 1}
0.912483 (0.012500) with: {'batch_size': 32, 'beta_1': 0.9, 'beta_2': 0.8, 'epochs': 50, 'learning_rate': 0.001
0.911248 (0.019905) with: {'batch_size': 32, 'beta_1': 0.9, 'beta_2': 0.8, 'epochs': 50, 'learning_rate': 0.01}
0.916233 (0.014568) with: {'batch_size': 32, 'beta_1': 0.9, 'beta_2': 0.8, 'epochs': 50, 'learning_rate': 0.1}
0.925000 (0.018349) with: {'batch_size': 32, 'beta_1': 0.9, 'beta_2': 0.8, 'epochs': 50, 'learning_rate': 1}
0.919978 (0.012899) with: {'batch_size': 32, 'beta_1': 0.9, 'beta_2': 0.9, 'epochs': 50, 'learning_rate': 0.001
0.913745 (0.017052) with: {'batch_size': 32, 'beta_1': 0.9, 'beta_2': 0.9, 'epochs': 50, 'learning_rate': 0.01}
0.916233 (0.016942) with: {'batch_size': 32, 'beta_1': 0.9, 'beta_2': 0.9, 'epochs': 50, 'learning_rate': 0.1}
0.912487 (0.022596) with: {'batch_size': 32, 'beta_1': 0.9, 'beta_2': 0.9, 'epochs': 50, 'learning_rate': 1}
0.909990 (0.010681) with: {'batch_size': 32, 'beta_1': 0.9, 'beta_2': 0.999, 'epochs': 50, 'learning_rate': 0.0
0.922489 (0.012436) with: {'batch_size': 32, 'beta_1': 0.9, 'beta_2': 0.999, 'epochs': 50, 'learning_rate': 0.0
0.916247 (0.012382) with: {'batch_size': 32, 'beta_1': 0.9, 'beta_2': 0.999, 'epochs': 50, 'learning_rate': 0.1
0.922484 (0.020431) with: {'batch_size': 32, 'beta_1': 0.9, 'beta_2': 0.999, 'epochs': 50, 'learning_rate': 1}
```

```
# Use scikit-learn to grid search the hyperparameter of Adadelta
import numpy
from tensorflow.keras.optimizers import Nadam
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.optimizers import Adadelta
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, Flatten
from tensorflow.keras.layers import Conv2D, MaxPooling2D
from sklearn.model_selection import GridSearchCV
from keras.models import Sequential
```

```
from keras.layers import Dense
from keras.wrappers.scikit_learn import KerasClassifier

Adam = Adam(learning_rate=0.001,beta_1=0.9,beta_2=0.999,epsilon=1e-07,amsgrad=False)
Adadelta = Adadelta(learning_rate=0.001, rho=0.95, epsilon=1e-07)
NAG = Nadam(learning_rate=0.001, beta_1=0.9, beta_2=0.999, epsilon=1e-07)

# Function to create model, required for KerasClassifier
def create_model(learning_rate=0.001, rho=0.95):
  # create model
  model = Sequential()
  model.add(Conv2D(16, (3, 3), activation='relu', input_shape=(28, 28, 1)))
  # Max pooling
  model.add(MaxPooling2D(pool_size=(2, 2)))
  model.add(Dropout(0.))
  model.add(Conv2D(32, (3, 3), activation='relu'))
  # Max pooling
  model.add(MaxPooling2D(pool_size=(2, 2)))
  model.add(Flatten())
  model.add(Dense(128, activation='relu'))
  model.add(Dropout(0.))
  model.add(Dense(10, activation='softmax'))


  # Compile the model
  model.compile(loss='categorical_crossentropy', optimizer=Adadelta, metrics=["accuracy"])

  return model
# fix random seed for reproducibility
seed = 3
numpy.random.seed(seed)

# create model
model = KerasClassifier(build_fn=create_model, verbose=0)
# define the grid search parameters
space = dict()
space['learning_rate'] = [0.001, 0.01, 0.1,1]
space['rho'] = [0.001, 0.01, 0.1,1]
space['batch_size'] = [32]
space['epochs']=[50]


grid = GridSearchCV(estimator=model, param_grid=space, n_jobs=-1, cv=3)
grid_result = grid.fit(X_train, y_oh_train)
# summarize results
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print("%f (%f) with: %r" % (mean, stdev, param))
```

```
Best: 0.191218 using {'batch_size': 32, 'epochs': 50, 'learning_rate': 0.1, 'rho': 1}
0.142547 (0.026802) with: {'batch_size': 32, 'epochs': 50, 'learning_rate': 0.001, 'rho': 0.001}
0.166291 (0.024093) with: {'batch_size': 32, 'epochs': 50, 'learning_rate': 0.001, 'rho': 0.01}
0.122483 (0.024832) with: {'batch_size': 32, 'epochs': 50, 'learning_rate': 0.001, 'rho': 0.1}
0.145063 (0.036177) with: {'batch_size': 32, 'epochs': 50, 'learning_rate': 0.001, 'rho': 1}
0.133682 (0.042832) with: {'batch_size': 32, 'epochs': 50, 'learning_rate': 0.01, 'rho': 0.001}
0.162527 (0.035582) with: {'batch_size': 32, 'epochs': 50, 'learning_rate': 0.01, 'rho': 0.01}
0.148752 (0.007731) with: {'batch_size': 32, 'epochs': 50, 'learning_rate': 0.01, 'rho': 0.1}
0.152568 (0.039080) with: {'batch_size': 32, 'epochs': 50, 'learning_rate': 0.01, 'rho': 1}
```

```
0.111238 (0.011469) with: {'batch_size': 32, 'epochs': 50, 'learning_rate': 0.1, 'rho': 0.001}
0.167432 (0.055441) with: {'batch_size': 32, 'epochs': 50, 'learning_rate': 0.1, 'rho': 0.01}
0.159913 (0.057994) with: {'batch_size': 32, 'epochs': 50, 'learning_rate': 0.1, 'rho': 0.1}
0.191218 (0.018310) with: {'batch_size': 32, 'epochs': 50, 'learning_rate': 0.1, 'rho': 1}
0.163715 (0.043172) with: {'batch_size': 32, 'epochs': 50, 'learning_rate': 1, 'rho': 0.001}
0.139980 (0.025549) with: {'batch_size': 32, 'epochs': 50, 'learning_rate': 1, 'rho': 0.01}
0.093774 (0.023995) with: {'batch_size': 32, 'epochs': 50, 'learning_rate': 1, 'rho': 0.1}
0.158716 (0.058294) with: {'batch_size': 32, 'epochs': 50, 'learning_rate': 1, 'rho': 1}
```

- **Dropout**. Use the best optimizer and do hyper-parameter seach and find the best value for `Dropout()`.

ANS: From the above output we know that the Adam optimizer has the highest accuracy score of 0.931233 with beta_1 = 0.5, beta_2 = 0.8, learning_rate = 0.01. I do a Grid Search below to search for the optimal dropout rate.

The best value for the dropout rate is 0.4 with the improved accuracy score of 0.934997 while keeping beta_1 = 0.5, beta_2 = 0.8 and learning_rate = 0.01 unchanged.

```
# Use scikit-learn to grid search the dropout rate
import numpy
from sklearn.model_selection import GridSearchCV
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
from keras.wrappers.scikit_learn import KerasClassifier
from keras.constraints import maxnorm

# Function to create model, required for KerasClassifier
def create_model(learning_rate=0.01, beta_1=0.5, beta_2=0.8, dropout_rate=0.):
  # create model
  model = Sequential()
  model.add(Conv2D(16, (3, 3), activation='relu', input_shape=(28, 28, 1)))
  # Max pooling
  model.add(MaxPooling2D(pool_size=(2, 2)))
  model.add(Dropout(0.))
  model.add(Conv2D(32, (3, 3), activation='relu'))
  # Max pooling
  model.add(MaxPooling2D(pool_size=(2, 2)))
  model.add(Flatten())
  model.add(Dense(128, activation='relu'))
  model.add(Dropout(0.))
  model.add(Dense(10, activation='softmax'))


  # Compile the model
  model.compile(loss='categorical_crossentropy', optimizer=Adam, metrics=["accuracy"])

  return model

# fix random seed for reproducibility
seed = 4
numpy.random.seed(seed)

# create model
model = KerasClassifier(build_fn=create_model, verbose=0)

# define the grid search parameters
space = {}
space['dropout_rate'] =  [0, 0.2, 0.4, 0.6, 0.8]
space['learning_rate'] = [0.01]
space['beta_1'] = [0.5]
space['beta_2'] = [0.8]
space['batch_size'] = [32]
space['epochs']=[50]
```

```
grid = GridSearchCV(estimator=model, param_grid=space, n_jobs=-1, cv=3)
grid_result = grid.fit(X_train, y_oh_train)
# summarize results
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print("%f (%f) with: %r" % (mean, stdev, param))
```

```
Best: 0.934997 using {'batch_size': 32, 'beta_1': 0.5, 'beta_2': 0.8, 'dropout_rate': 0.4, 'epochs': 50, 'learn
0.929989 (0.009918) with: {'batch_size': 32, 'beta_1': 0.5, 'beta_2': 0.8, 'dropout_rate': 0, 'epochs': 50, 'le
0.927501 (0.013782) with: {'batch_size': 32, 'beta_1': 0.5, 'beta_2': 0.8, 'dropout_rate': 0.2, 'epochs': 50, '
0.934997 (0.009366) with: {'batch_size': 32, 'beta_1': 0.5, 'beta_2': 0.8, 'dropout_rate': 0.4, 'epochs': 50, '
0.922484 (0.015179) with: {'batch_size': 32, 'beta_1': 0.5, 'beta_2': 0.8, 'dropout_rate': 0.6, 'epochs': 50, '
0.919992 (0.014499) with: {'batch_size': 32, 'beta_1': 0.5, 'beta_2': 0.8, 'dropout_rate': 0.8, 'epochs': 50, '
```

- **Best model**. Combine the what you learned from the above three questions to build the best model. How much better is it than the worst and average models?

  ANS : As we see from above, the best model is the one with Adam optimizer: beta_1 = 0.5, beta_2 = 0.8 and learning_rate = 0.01 and dropout rate = 0.4. This model has the best accuracy score of 0.934997.

  The worst one is the model with Adadelta whose accuracy score is 0.191218. The average of these models are (0.931233 + 0.925000 + 0.191218 + 0.934997)/4 = 0.74561125.

  Therefore, the best model is about 75% better than the worst one and 20% better than the averages.

- **Results on the test set**. When doing this search for good model configuration/hyper-parameter values, the data set was split into *two* parts: a training set and a test set (the term "validation" was used interchangably wiht "test"). For your final model, is the performance (i.e. accuracy) on the test set representative for the performance one would expect on a previously unseen data set (drawn from the same distribution)? Why?

  ANS:

  In the following code, I modified the model and plot the loss and the val_loss again with the hyperparameters of the best model where beta_1 = 0.5, beta_2 = 0.8 and learning_rate = 0.01 and dropout rate = 0.4. As we can see from the output below that the model accuracy on the test set is improved from 0.93 to 0.96.

  However, this final best model may not work as well as expected on a previous unseen dataset, even the dataset is drawn from the same distribution. In previous questions, the hyperparameters are tuned with the training set data and the model are trained with training set. The "best" hyperparameters found by Grid Search are based on the model performance/accuracy on training set. Then, we selected the best final model solely based on the output of Grid Search. This means those "best hyperparameters" and the "best model" are the best for training set and they are not necesscary the best for the test/validation set. We should evaluate the model on the test/validation set with the tuned hyperparameters again each time whenever we did a Grid Search. And then, we should select the best model based on the best performance/accuracy on the test/validation set instead of on the training set.

  From the loss and val_loss graph below we can see that the val_loss is very unstable while the loss of train set is steadly decreasing.

  Therefore, in this case, the "best final model" has the risk of overfitting on the training set and it may not be significantly representative for the performance on a strange dataset.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, Flatten
from tensorflow.keras.layers import Conv2D, MaxPooling2D
from tensorflow.keras.optimizers import Adam
```

```
model = Sequential()

model.add(Conv2D(16, (3, 3), activation='relu', input_shape=(28, 28, 1)))
# Max pooling
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.4))

model.add(Conv2D(32, (3, 3), activation='relu'))
# Max pooling
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())

model.add(Dense(128, activation='relu'))
model.add(Dropout(0.4))

model.add(Dense(10, activation='softmax'))

Adam = Adam(learning_rate=0.01,beta_1=0.5,beta_2=0.8,epsilon=1e-07,amsgrad=False)

# Compile the model
model.compile(loss='categorical_crossentropy', optimizer=Adam)

# Train the model
history = model.fit(X_train, y_oh_train, batch_size=32, epochs=50, validation_data=(X_test, y_oh_test))

# Evaluate performance
test_loss = model.evaluate(X_test, y_oh_test, batch_size=32)

predictions = model.predict(X_test, batch_size=32)
predictions = np.argmax(predictions, axis=1) # change encoding again
print('Accuracy:', (predictions == y_test).sum() / predictions.shape[0])

# Plot learning
fig = plt.figure(figsize=(12,4))
ax0 = fig.add_subplot(121)
ax0.plot(history.history['val_loss'], label='validation loss')
ax0.plot(history.history['loss'], label='training loss')
ax0.set_ylabel('Loss')
ax0.set_xlabel('Epoch')
ax0.legend()
```

```
Epoch 1/50
25/25 [==============================] - 1s 29ms/step - loss: 1.9620 - val_loss: 0.5396
Epoch 2/50
25/25 [==============================] - 1s 23ms/step - loss: 0.5469 - val_loss: 0.4167
Epoch 3/50
25/25 [==============================] - 1s 22ms/step - loss: 0.3733 - val_loss: 0.3332
Epoch 4/50
25/25 [==============================] - 1s 23ms/step - loss: 0.3256 - val_loss: 0.2690
Epoch 5/50
25/25 [==============================] - 1s 22ms/step - loss: 0.2423 - val_loss: 0.2224
Epoch 6/50
25/25 [==============================] - 1s 23ms/step - loss: 0.1714 - val_loss: 0.2703
Epoch 7/50
25/25 [==============================] - 1s 22ms/step - loss: 0.1625 - val_loss: 0.2806
Epoch 8/50
25/25 [==============================] - 1s 22ms/step - loss: 0.1436 - val_loss: 0.3186
Epoch 9/50
25/25 [==============================] - 1s 22ms/step - loss: 0.1007 - val_loss: 0.1951
Epoch 10/50
25/25 [==============================] - 1s 22ms/step - loss: 0.0977 - val_loss: 0.2702
Epoch 11/50
25/25 [==============================] - 1s 23ms/step - loss: 0.0732 - val_loss: 0.2175
Epoch 12/50
25/25 [==============================] - 1s 27ms/step - loss: 0.1772 - val_loss: 0.2255
Epoch 13/50
25/25 [==============================] - 1s 24ms/step - loss: 0.0669 - val_loss: 0.3063
Epoch 14/50
25/25 [==============================] - 1s 22ms/step - loss: 0.1402 - val_loss: 0.2856
Epoch 15/50
25/25 [==============================] - 1s 22ms/step - loss: 0.0691 - val_loss: 0.2973
Epoch 16/50
25/25 [==============================] - 1s 23ms/step - loss: 0.0817 - val_loss: 0.6316
Epoch 17/50
25/25 [==============================] - 1s 22ms/step - loss: 0.1267 - val_loss: 0.3448
Epoch 18/50
25/25 [==============================] - 1s 22ms/step - loss: 0.1155 - val_loss: 0.2470
Epoch 19/50
25/25 [==============================] - 1s 23ms/step - loss: 0.0553 - val_loss: 0.2071
Epoch 20/50
25/25 [==============================] - 1s 22ms/step - loss: 0.0882 - val_loss: 0.2395
Epoch 21/50
25/25 [==============================] - 1s 22ms/step - loss: 0.0464 - val_loss: 0.3634
Epoch 22/50
25/25 [==============================] - 1s 22ms/step - loss: 0.1257 - val_loss: 0.4139
Epoch 23/50
25/25 [==============================] - 1s 22ms/step - loss: 0.1080 - val_loss: 0.3234
Epoch 24/50
25/25 [==============================] - 1s 23ms/step - loss: 0.0469 - val_loss: 0.2971
Epoch 25/50
25/25 [==============================] - 1s 23ms/step - loss: 0.0690 - val_loss: 0.2414
Epoch 26/50
25/25 [==============================] - 1s 23ms/step - loss: 0.0502 - val_loss: 0.3879
Epoch 27/50
25/25 [==============================] - 1s 22ms/step - loss: 0.0574 - val_loss: 0.3431
Epoch 28/50
25/25 [==============================] - 1s 23ms/step - loss: 0.0813 - val_loss: 0.4010
Epoch 29/50
25/25 [==============================] - 1s 23ms/step - loss: 0.0944 - val_loss: 0.4872
Epoch 30/50
25/25 [==============================] - 1s 22ms/step - loss: 0.0926 - val_loss: 0.2677
Epoch 31/50
25/25 [==============================] - 1s 24ms/step - loss: 0.0911 - val_loss: 0.4110
Epoch 32/50
25/25 [==============================] - 1s 26ms/step - loss: 0.0460 - val_loss: 0.2774
Epoch 33/50
25/25 [==============================] - 1s 24ms/step - loss: 0.0560 - val_loss: 0.5786
Epoch 34/50
25/25 [==============================] - 1s 22ms/step - loss: 0.1071 - val_loss: 0.4297
Epoch 35/50
25/25 [==============================] - 1s 23ms/step - loss: 0.0367 - val_loss: 0.4476
Epoch 36/50
25/25 [==============================] - 1s 23ms/step - loss: 0.0661 - val_loss: 0.3022
```

```
Epoch 37/50
25/25 [==============================] - 1s 22ms/step - loss: 0.0448 - val_loss: 0.4308
Epoch 38/50
25/25 [==============================] - 1s 23ms/step - loss: 0.0769 - val_loss: 0.3597
Epoch 39/50
25/25 [==============================] - 1s 22ms/step - loss: 0.0513 - val_loss: 0.5548
Epoch 40/50
25/25 [==============================] - 1s 22ms/step - loss: 0.0663 - val_loss: 0.3917
Epoch 41/50
25/25 [==============================] - 1s 28ms/step - loss: 0.0877 - val_loss: 0.4091
Epoch 42/50
25/25 [==============================] - 1s 23ms/step - loss: 0.0233 - val_loss: 0.5054
Epoch 43/50
25/25 [==============================] - 1s 22ms/step - loss: 0.0830 - val_loss: 0.4251
Epoch 44/50
25/25 [==============================] - 1s 26ms/step - loss: 0.0699 - val_loss: 0.3184
Epoch 45/50
25/25 [==============================] - 1s 25ms/step - loss: 0.1073 - val_loss: 0.4423
Epoch 46/50
25/25 [==============================] - 1s 23ms/step - loss: 0.0784 - val_loss: 0.5067
Epoch 47/50
25/25 [==============================] - 1s 23ms/step - loss: 0.1368 - val_loss: 0.4503
```

Further information

For ideas about hyper-parameter tuning, take a look at the strategies described in the sklearn documentation under model selection, or in this blog post from TensorFlow. For a more thorough discussion about optimizers see this video discussing the article Descending through a Crowded Valley -- Benchmarking Deep Learning Optimizers.

Good luck!