# DAT405 Assignment 7 – Group 67

Student Yaochen Rao - (20 hrs)
Student Wanqiu Wang - (20 hrs)

October 18, 2022

## Problem 1

```
# imports
from __future__ import print_function
import keras
from keras import utils as np_utils
import tensorflow
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten
from keras.layers import Conv2D, MaxPooling2D
from keras import backend as K
import tensorflow as tf
from matplotlib import pyplot as plt
import numpy as np
```

```
# Hyper-parameters data-loading and formatting

batch_size = 128
num_classes = 10
epochs = 10

img_rows, img_cols = 28, 28

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

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

## Preprocessing

```
# Scales the training and test data to range between 0 and 1.
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
```

```
x_train /= 255
x_test /= 255

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

**Answer 1:**

From neural_networks__intro.ipynb from module 7, we know that normalizing our data is very important, as we want the input features to be on the same order of magnitude to make our training easier. We'll use a min-max scaler from scikit-learn which scales our data to be between 0 and 1.

We convert the data set into floating point numbers, and normalize each data. And here we normalize the downloaded training and test data. We use 0-1 normalization to compress the values in the 0-1 interval to suppress the effect of outliers on the results.

$$X_{scale} = \frac{X - X_{MIN}}{X_{MAX} - X_{MIN}}$$

From problem we know that data is composed of images, so $X_{MAX}$ is 255 and $X_{MIN}$ is 0. Then we divide the data by 255 to get the data which values between 0 and 1.

And keras.utils.to_categorica() converts the category tag of an integer to the onehot encoding(binary class matrix indicating its label) which is the same means to the class label (an integer from 0 to 9).

# Problem 2

```
## Define model ##
model = Sequential()
# Configuring the network architecture
model.add(Flatten())
model.add(Dense(64, activation = 'relu'))
model.add(Dense(64, activation = 'relu'))
model.add(Dense(num_classes, activation='softmax'))

# Configuring the training process
model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=tensorflow.keras.optimizers.SGD(learning_rate = 0.1),
        metrics=['accuracy'],)
# Training the model
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 [==============================] - 5s 4ms/step - loss: 0.4695 - accuracy: 0.8651 - val_loss: 0.2515 - val_accuracy: 0.9268
Epoch 2/10
469/469 [==============================] - 2s 4ms/step - loss: 0.2281 - accuracy: 0.9337 - val_loss: 0.1797 - val_accuracy: 0.9465
Epoch 3/10
469/469 [==============================] - 2s 3ms/step - loss: 0.1744 - accuracy: 0.9489 - val_loss: 0.1535 - val_accuracy: 0.9531
Epoch 4/10
469/469 [==============================] - 2s 4ms/step - loss: 0.1428 - accuracy: 0.9582 - val_loss: 0.1297 - val_accuracy: 0.9587
Epoch 5/10
469/469 [==============================] - 2s 4ms/step - loss: 0.1210 - accuracy: 0.9641 - val_loss: 0.1169 - val_accuracy: 0.9643
Epoch 6/10
469/469 [==============================] - 2s 3ms/step - loss: 0.1047 - accuracy: 0.9687 - val_loss: 0.1149 - val_accuracy: 0.9644
Epoch 7/10
469/469 [==============================] - 2s 4ms/step - loss: 0.0919 - accuracy: 0.9730 - val_loss: 0.1089 - val_accuracy: 0.9658
Epoch 8/10
469/469 [==============================] - 2s 4ms/step - loss: 0.0813 - accuracy: 0.9753 - val_loss: 0.0998 - val_accuracy: 0.9684
Epoch 9/10
469/469 [==============================] - 2s 4ms/step - loss: 0.0742 - accuracy: 0.9774 - val_loss: 0.0915 - val_accuracy: 0.9705
Epoch 10/10
469/469 [==============================] - 2s 4ms/step - loss: 0.0674 - accuracy: 0.9799 - val_loss: 0.0961 - val_accuracy: 0.9702
Test loss: 0.0961291715502739, Test accuracy 0.9702000021934509
```

```
model.summary()
model.input_shape
```

```
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
_________________________________________________________________
(None, 28, 28, 1)
```

# 2A)

As we can see from the code and summary above:

We will be using the Sequential model, which means that we merely need to describe our layers in sequence. Our neural network has four layers:

(or we can say that has three layers if we ignore the input layer)

Input(flatten) layer: 28∗28 ->784∗1 neurons

Hidden layer 1: 64 neurons, ReLU activation

Hidden layer 2: 64 neurons, ReLU activation

Output Layer: 10 neuron, Softmax activation

Total params: 55,050

Then:

**Answer 2A)**

**1.How many layers does the network in the notebook have?:**

We have four layers(or three layers if we ignore the input layer): 1 input layer, 2 hidden layers, 1 output layers.

**2.How many neurons does each layer have?:**

Input layer has 784 neurous($28 * 28$ matrix ->784 neurous); each hidden layer has 64 neurous; output layer has 10 neurous(num_classes is 10).

**3.What activation functions and why are these appropriate for this application?:**

**ReLU**(Hidden layers): ReLU activation function is an activation function defined as the positive part of its argument: $f(x) = x^+ = max(0, x)$, so we can get better gradient propagation. When $x > 0$, the gradient is always 1, there is no gradient dissipation problem, and the convergence is fast;when $x < 0$, the output of this layer is0. This causes the sparsity of the network, and reduces the interdependence of parameters, which alleviates the overfitting problem. So by using this method, the nonlinear relationship between the layers of the neural network is increased. And because the negative neurons will not activate, the gradient problem can be better solved.

**Softmax**(Output layer): Converts a vector of K real numbers into a probability distribution of K possible outcomes. Therefore, as soon as the softmax function acts, the output of the neuron is mapped to the value of $(0, 1)$, and the cumulative sum of these values is 1 (satisfying the nature of probability), then we can understand it as probability , when the output node is finally selected, we can select the node with the highest probability (that is, the value corresponding to the largest) as our prediction target. From this, it can be seen that this method is very suitable for classification problems, and we can get the probability of each class.

**4.What is the total number of parameters for the network?:**

We can get it from the summary of model or calculate: Total number of parameters: $784 * 64 + 64 + 64 * 64 + 10 * 64 + 10 = 55050$

**5.Why does the input and output layers have the dimensions they have?:**

The input layer is a picture of pixels $28 * 28$, and an array of $(28 * 28)$ will be generated to represent it, and then the flatten layer flattens it to the one-dimensional array $(784 * 1)$. The output layer is because we There are ten categories in total. So the dimension depends on the size of the data.

# 2B)

**Answer 2B)**

**1.What loss-function is used to train the network?:**

This model use Categorical Cross-entropy loss function.

**2.What is the functional form (mathematical expression) of the loss function? and how should we interpret it?:**

$$Loss = -\sum_{i=1}^{output\ size} y_i . \log\left(\widehat{y}_i\right)$$

Where $\widehat{y}_i$ is the $i$-th scalar value in the model output, $y_i$ is the corresponding target value, and $outputsize$ is the number of scalar values in the model output.

This loss is a very good measure of how distinguishable two discrete probability distributions are from each other. In this context, $y_i$ is the probability that event $i$ occurs and the sum of all $y_i$ is 1, meaning that exactly one event may occur. The minus sign ensures that the loss gets smaller when the distributions get closer to each other.

The categorical crossentropy is well suited to classification tasks, since one example can be considered to belong to a specific category with probability 1, and to other categories with probability 0.

(From https://peltarion.com/knowledge-center/modeling-view/build-an-ai-model/loss-functions/categorical-crossentropy)

**3.Why is it appropriate for the problem at hand?:**

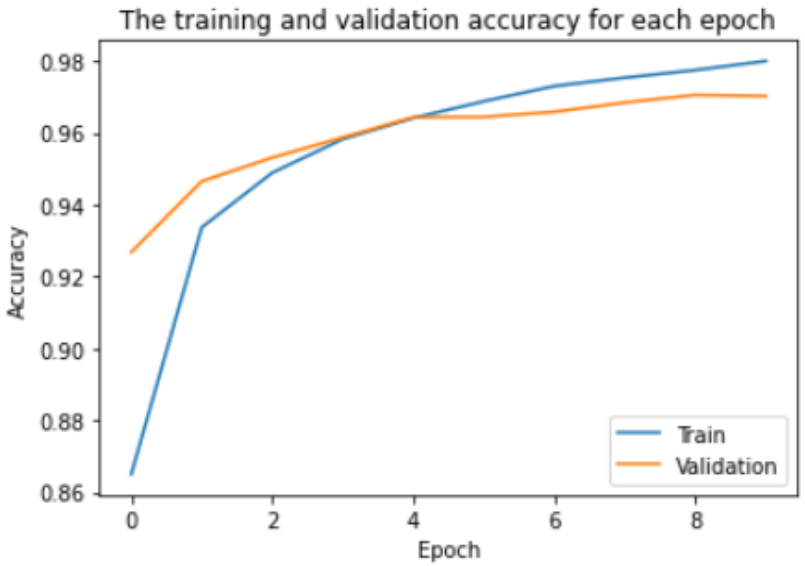
The model uses the categorical crossentropy to learn to give a high probability to the correct digit and a low probability to the other digits.

And the multi-class problem is that the softmax activation function is used to coordinate the classification cross entropy function. This is the case of multiple categories, so using this method can be very convenient to determine the high probability and low probability and turn it into the cost in the loss function.

# 2C)

The above code uses "epochs = epochs" and "epochs = 10" in the training model, so the model "fit_into" is the final model, we only need to plot the training and validation accuracy for each epoch.

```
# 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(['Train', 'Validation'], loc='lower right')
plt.show()
```




## 2D)

Update model to implement a three-layer neural network where the hidden-layers has 500 and 300 hidden units respectively. Train for 40 epochs. And using weight decay (L2 regularization of weights (kernels).

```
# Three-layer neural network where the hidden-layers has 500 and 300 hidden units
    respectively
# Train for 40 epochs

def weight_decay_model(factor):
  # Define model
  model = Sequential()
  # Configuring the network architecture
  model.add(Flatten())
  model.add(Dense(500, activation = 'relu', kernel_regularizer=keras.regularizers.l2(
    factor)))
  model.add(Dense(300, activation = 'relu', kernel_regularizer=keras.regularizers.l2(
    factor)))
  model.add(Dense(num_classes, activation='softmax'))

  # Configuring the training process
  model.compile(loss=keras.losses.categorical_crossentropy,
                optimizer=tensorflow.keras.optimizers.SGD(learning_rate = 0.1),
          metrics=['accuracy'],)
  #training the model
  fit_info = model.fit(x_train, y_train,
            batch_size=batch_size,
            epochs=40,
            verbose=2, #2 means: Output one line of records for each epoch
            validation_data=(x_test, y_test))
  score = model.evaluate(x_test, y_test, verbose=2)
  print('Test loss: {}, Test accuracy {}'.format(score[0], score[1]))
  return fit_info.history
```

Select 5 regularization factors from 0.000001 to 0.001 (we use 0.000001,0.00001,0.0001,0.001,0.000999) and train 3 replicates networks for each one.

```
# Select 5 regularization factors from 0.000001 to 0.001 (we use
    0.000001,0.00001,0.0001,0.001,0.000999)
# Train 3 replicates networks for each regularization factor.
regularization_factor = [0.000001, 0.00001, 0.0001, 0.0005, 0.001]
model_fit = []
for i in regularization_factor:
  for j in range(0,3):
    print("Model with regularization factor:{}, and it is the {} networks for {}".
    format(i,j+1,i))
    model_fit.append(weight_decay_model(i))
```

[ Because there are too many epoch processes, only the beginning and end are shown. ]

```
Model with regularization factor:1e-06, and it is the 1 networks for 1e-06
Epoch 1/40
469/469 - 2s - loss: 0.4064 - accuracy: 0.8867 - val_loss: 0.2170 - val_accuracy: 0.9358 - 2s/epoch - 4ms/step
Epoch 2/40
469/469 - 2s - loss: 0.1907 - accuracy: 0.9453 - val_loss: 0.1668 - val_accuracy: 0.9520 - 2s/epoch - 3ms/step
Epoch 3/40
469/469 - 1s - loss: 0.1390 - accuracy: 0.9604 - val_loss: 0.1328 - val_accuracy: 0.9604 - 1s/epoch - 3ms/step
Epoch 4/40
469/469 - 2s - loss: 0.1094 - accuracy: 0.9689 - val_loss: 0.1075 - val_accuracy: 0.9682 - 2s/epoch - 3ms/step
Epoch 5/40
469/469 - 2s - loss: 0.0892 - accuracy: 0.9751 - val_loss: 0.0973 - val_accuracy: 0.9708 - 2s/epoch - 3ms/step
Epoch 6/40
469/469 - 2s - loss: 0.0753 - accuracy: 0.9793 - val_loss: 0.0953 - val_accuracy: 0.9711 - 2s/epoch - 3ms/step
Epoch 7/40
469/469 - 2s - loss: 0.0637 - accuracy: 0.9822 - val_loss: 0.0807 - val_accuracy: 0.9757 - 2s/epoch - 3ms/step
Epoch 8/40
469/469 - 2s - loss: 0.0550 - accuracy: 0.9854 - val_loss: 0.0779 - val_accuracy: 0.9751 - 2s/epoch - 3ms/step
Epoch 9/40
469/469 - 1s - loss: 0.0474 - accuracy: 0.9876 - val_loss: 0.0753 - val_accuracy: 0.9776 - 1s/epoch - 3ms/step
Epoch 10/40
469/469 - 1s - loss: 0.0412 - accuracy: 0.9888 - val_loss: 0.0741 - val_accuracy: 0.9777 - 1s/epoch - 3ms/step
Epoch 11/40
469/469 - 1s - loss: 0.0364 - accuracy: 0.9905 - val_loss: 0.0714 - val_accuracy: 0.9782 - 1s/epoch - 3ms/step
Epoch 12/40
469/469 - 2s - loss: 0.0316 - accuracy: 0.9920 - val_loss: 0.0651 - val_accuracy: 0.9802 - 2s/epoch - 3ms/step
Epoch 13/40
469/469 - 1s - loss: 0.0277 - accuracy: 0.9937 - val_loss: 0.0675 - val_accuracy: 0.9792 - 1s/epoch - 3ms/step
Epoch 14/40
469/469 - 1s - loss: 0.0247 - accuracy: 0.9943 - val_loss: 0.0670 - val_accuracy: 0.9794 - 1s/epoch - 3ms/step
Epoch 15/40
469/469 - 2s - loss: 0.0215 - accuracy: 0.9955 - val_loss: 0.0652 - val_accuracy: 0.9804 - 2s/epoch - 3ms/step
Epoch 16/40
469/469 - 1s - loss: 0.0188 - accuracy: 0.9966 - val_loss: 0.0654 - val_accuracy: 0.9799 - 1s/epoch - 3ms/step
```

```
Epoch 34/40
469/469 - 1s - loss: 0.0952 - accuracy: 0.9932 - val_loss: 0.1263 - val_accuracy: 0.9808 - 1s/epoch - 3ms/step
Epoch 35/40
469/469 - 2s - loss: 0.0949 - accuracy: 0.9924 - val_loss: 0.1224 - val_accuracy: 0.9804 - 2s/epoch - 3ms/step
Epoch 36/40
469/469 - 1s - loss: 0.0930 - accuracy: 0.9934 - val_loss: 0.1247 - val_accuracy: 0.9810 - 1s/epoch - 3ms/step
Epoch 37/40
469/469 - 1s - loss: 0.0926 - accuracy: 0.9930 - val_loss: 0.1190 - val_accuracy: 0.9823 - 1s/epoch - 3ms/step
Epoch 38/40
469/469 - 1s - loss: 0.0917 - accuracy: 0.9932 - val_loss: 0.1168 - val_accuracy: 0.9818 - 1s/epoch - 3ms/step
Epoch 39/40
469/469 - 1s - loss: 0.0897 - accuracy: 0.9937 - val_loss: 0.1233 - val_accuracy: 0.9794 - 1s/epoch - 3ms/step
Epoch 40/40
469/469 - 2s - loss: 0.0900 - accuracy: 0.9933 - val_loss: 0.1180 - val_accuracy: 0.9808 - 2s/epoch - 3ms/step
313/313 - 1s - loss: 0.1180 - accuracy: 0.9808 - 651ms/epoch - 2ms/step
Test loss: 0.11801216751337051, Test accuracy 0.9807999730110168
```

Then calculate the final validation accuracy and the mean/standard/max of accuracy from three replicates for each regularization factor.

```
accuracy = []
final_accuracy = []
mean_accuracy = []
max_accuracy = []
std_accuracy = []
#acc_factor = []
tmp=[]
for i in range(0,5):
  for j in range(0,3):
    a = model_fit[3*i+j]['val_accuracy'][-1]
    # All accuracy (5*3=15)
    accuracy.append(a)
    tmp.append(a)
  # Final validation accuracy from 3 replicates networks for each regularization
    factor
  b = model_fit[3*i+2]['val_accuracy'][-1]
  final_accuracy.append(accuracy[3*i+2])
  # The mean of accuracy from 3 replicates networks for each regularization factor
  mean_accuracy.append(np.mean(tmp))
  # The standard deviation of accuracy from 3 replicates networks for each
    regularization factor
  std_accuracy.append(np.std(tmp))
  # The max of accuracy from 3 replicates networks for each regularization factor
  max_accuracy.append(np.max(tmp))
  #acc_factor.append([accuracy[3*i+2],regularization_factor[i]])
```

We can display the max accuracy for each regularization factor.

```
max_accuracy
```

```
[0.982200026512146,
 0.9822999835014343,
 0.983299970626831,
 0.983299970626831,
 0.983299970626831]
```

And We can display the max accuracy for all regularization factors.

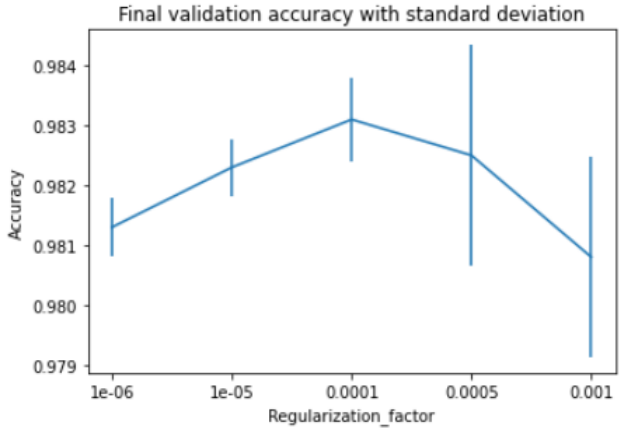
```
max_totalaccuracy = max(accuracy)
max_totalaccuracy
```

```
0.983299970626831
```

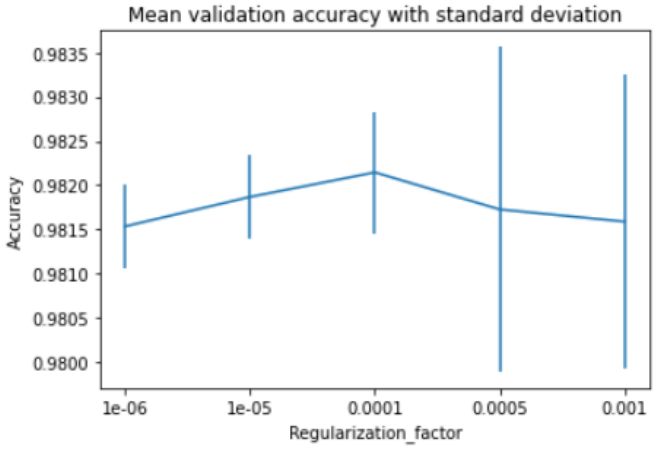
Plot the final validation accuracy with standard deviation(which the problem2D) require); and plot the mean/max validation accuracy with standard deviation.

```
# Plot the final validation accuracy with standard deviation
x = list(map(str, [0.000001, 0.00001, 0.0001, 0.0005, 0.001]))
#y = list(mean_acc)
```

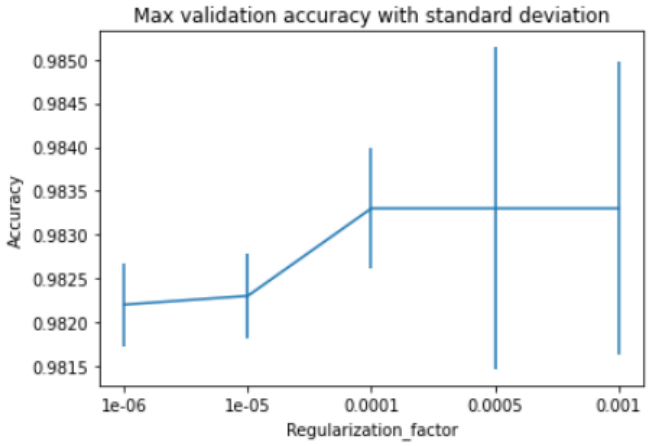
```
y = list(final_accuracy)
e = list(std_accuracy)
plt.errorbar(x, y, yerr=e)
plt.title("Final validation accuracy with standard deviation")
plt.xlabel('Regularization_factor')
plt.ylabel('Accuracy')
#plt.ylim(0.9800,0.9847)
plt.show()
```



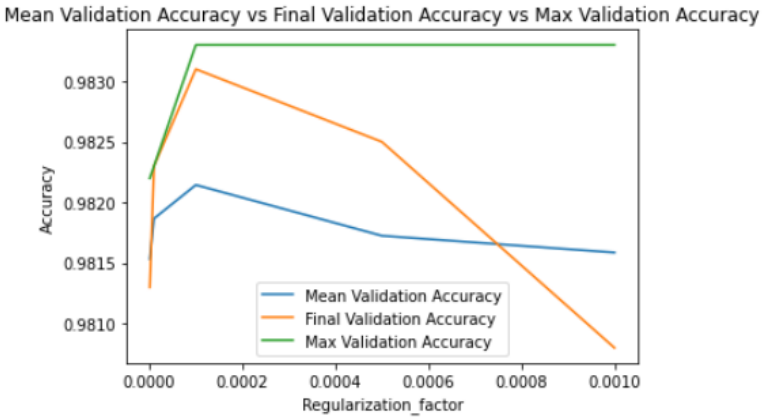

```
# Plot the mean validation accuracy with standard deviation
x = list(map(str, [0.000001, 0.00001, 0.0001, 0.0005, 0.001]))
y = list(mean_accuracy)
#y = list(final_accuracy)
e = list(std_accuracy)
plt.errorbar(x, y, yerr=e)
plt.title("Mean validation accuracy with standard deviation")
plt.xlabel('Regularization_factor')
plt.ylabel('Accuracy')
#plt.ylim(0.9800,0.9847)
plt.show()
```



```
# Plot the max validation accuracy with standard deviation
x = list(map(str, [0.000001, 0.00001, 0.0001, 0.0005, 0.001]))
y = list(max_accuracy)
#y = list(final_accuracy)
e = list(std_accuracy)
plt.errorbar(x, y, yerr=e)
plt.title("Max validation accuracy with standard deviation")
plt.xlabel('Regularization_factor')
plt.ylabel('Accuracy')
#plt.ylim(0.9800,0.9847)
plt.show()
```




```
# Compare 'Mean Validation Accuracy', 'Final Validation Accuracy' and 'Max Validation
    Accuracy'
plt.plot(regularization_factor, mean_accuracy)
plt.plot(regularization_factor, final_accuracy)
plt.plot(regularization_factor, max_accuracy)
plt.legend(['Mean Validation Accuracy', 'Final Validation Accuracy', 'Max Validation
    Accuracy'])
plt.title("Mean Validation Accuracy vs Final Validation Accuracy vs Max Validation
    Accuracy")
plt.xlabel("Regularization_factor")
plt.ylabel("Accuracy")
plt.show()
```




**Answer 2D)**

**How close do you get to Hintons result?**

The max accuracy we get is 0.983299970626831, it is close to 0.9847, but not the same results.

**If you do not get the same results, what factors may influence this?**

Because there is no way of knowing exactly what factors caused the difference between our accuracy results and Hintons' results, we can only guess based on the test results. According to the above image, it will be found that the selection of different regularization factors will lead to different accuracy, and according to the image, the highest accuracy will be found around 0.0001. So we thought that if we modified the regularization factor to have a value around 0.0001, we might get a closer answer.

In addition to this, other reason would be that our optimizer uses the SGD method (Stochastic Gradient Descent) here. As shown in the picture of the plot above, the accuracy of the model obtained by each epoch is not the same. Because it is random, there may be errors that cannot be avoided. Because of the error, the gradient of each iteration is greatly affected by sampling, that is to say, the gradient contains relatively large noise and cannot reflect the real gradient well. Therefore, the learning rate needs to be gradually reduced (such as 0.01) to converge and reduce the error.

# Problem 3

## 3A)

```
epochs = 40
#Defining CNN Model
cnn_model = Sequential()

#Add convenlutional layer & polling layer
cnn_model.add(Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape = (28, 28,
     1)))
cnn_model.add(MaxPooling2D(pool_size=(2, 2)))
cnn_model.add(Conv2D(64, (3, 3), activation='relu'))
cnn_model.add(MaxPooling2D(pool_size=(2, 2)))
#Add flatten layer to convert data to one-dimension array
cnn_model.add(Flatten())
#Add fully connected layer to deeply connected
cnn_model.add(Dense(200, activation='relu'))
cnn_model.add(Dense(128, activation='relu'))
#Add dropout layer to reduce overfitting
cnn_model.add(Dropout(0.5))
cnn_model.add(Dense(num_classes, activation='softmax'))

cnn_model.compile(loss=keras.losses.categorical_crossentropy, optimizer=keras.
    optimizers.SGD(lr = 0.1), metrics=['accuracy'])
cnn_model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs, verbose=1,
    validation_data=(x_test,y_test))
cnn_score = cnn_model.evaluate(x_test, y_test, verbose=0)

print('Test loss of CNN is: {}, Test accuracy of CNN is {}'.format(cnn_score[0],
    cnn_score[1]))
```

```
Epoch 1/40
469/469 [==============================] - 28s 60ms/step - loss: 0.4533 - accuracy: 0.8551 - val_loss: 0.0916 - val_accuracy: 0.9716
Epoch 2/40
469/469 [==============================] - 28s 60ms/step - loss: 0.1237 - accuracy: 0.9633 - val_loss: 0.0673 - val_accuracy: 0.9795
Epoch 3/40
469/469 [==============================] - 28s 59ms/step - loss: 0.0889 - accuracy: 0.9745 - val_loss: 0.0558 - val_accuracy: 0.9815
Epoch 4/40
469/469 [==============================] - 28s 59ms/step - loss: 0.0693 - accuracy: 0.9798 - val_loss: 0.0401 - val_accuracy: 0.9868
Epoch 5/40
469/469 [==============================] - 28s 59ms/step - loss: 0.0596 - accuracy: 0.9830 - val_loss: 0.0377 - val_accuracy: 0.9877
Epoch 6/40
469/469 [==============================] - 28s 59ms/step - loss: 0.0504 - accuracy: 0.9854 - val_loss: 0.0378 - val_accuracy: 0.9881
Epoch 7/40
469/469 [==============================] - 28s 59ms/step - loss: 0.0437 - accuracy: 0.9873 - val_loss: 0.0345 - val_accuracy: 0.9880
Epoch 8/40
469/469 [==============================] - 28s 59ms/step - loss: 0.0399 - accuracy: 0.9888 - val_loss: 0.0324 - val_accuracy: 0.9897
Epoch 9/40
469/469 [==============================] - 28s 60ms/step - loss: 0.0351 - accuracy: 0.9895 - val_loss: 0.0289 - val_accuracy: 0.9902
Epoch 10/40
469/469 [==============================] - 28s 60ms/step - loss: 0.0318 - accuracy: 0.9909 - val_loss: 0.0299 - val_accuracy: 0.9903
Epoch 11/40
469/469 [==============================] - 28s 60ms/step - loss: 0.0289 - accuracy: 0.9916 - val_loss: 0.0310 - val_accuracy: 0.9906
Epoch 12/40
469/469 [==============================] - 28s 59ms/step - loss: 0.0258 - accuracy: 0.9919 - val_loss: 0.0316 - val_accuracy: 0.9897
Epoch 13/40
469/469 [==============================] - 28s 59ms/step - loss: 0.0230 - accuracy: 0.9928 - val_loss: 0.0311 - val_accuracy: 0.9898
Epoch 14/40
469/469 [==============================] - 28s 59ms/step - loss: 0.0212 - accuracy: 0.9937 - val_loss: 0.0312 - val_accuracy: 0.9908
Epoch 15/40
469/469 [==============================] - 28s 59ms/step - loss: 0.0197 - accuracy: 0.9941 - val_loss: 0.0284 - val_accuracy: 0.9916
```

```
Epoch 16/40
469/469 [==============================] - 28s 60ms/step - loss: 0.0193 - accuracy: 0.9940 - val_loss: 0.0282 - val_accuracy: 0.9912
Epoch 17/40
469/469 [==============================] - 28s 59ms/step - loss: 0.0177 - accuracy: 0.9946 - val_loss: 0.0265 - val_accuracy: 0.9915
Epoch 18/40
469/469 [==============================] - 28s 60ms/step - loss: 0.0151 - accuracy: 0.9955 - val_loss: 0.0276 - val_accuracy: 0.9913
Epoch 19/40
469/469 [==============================] - 28s 59ms/step - loss: 0.0137 - accuracy: 0.9959 - val_loss: 0.0258 - val_accuracy: 0.9916
Epoch 20/40
469/469 [==============================] - 28s 60ms/step - loss: 0.0123 - accuracy: 0.9961 - val_loss: 0.0262 - val_accuracy: 0.9920
Epoch 21/40
469/469 [==============================] - 28s 60ms/step - loss: 0.0118 - accuracy: 0.9962 - val_loss: 0.0290 - val_accuracy: 0.9915
Epoch 22/40
469/469 [==============================] - 28s 60ms/step - loss: 0.0112 - accuracy: 0.9965 - val_loss: 0.0347 - val_accuracy: 0.9905
Epoch 23/40
469/469 [==============================] - 28s 59ms/step - loss: 0.0098 - accuracy: 0.9972 - val_loss: 0.0323 - val_accuracy: 0.9911
Epoch 24/40
469/469 [==============================] - 28s 59ms/step - loss: 0.0102 - accuracy: 0.9966 - val_loss: 0.0314 - val_accuracy: 0.9899
Epoch 25/40
469/469 [==============================] - 28s 59ms/step - loss: 0.0087 - accuracy: 0.9973 - val_loss: 0.0649 - val_accuracy: 0.9838
Epoch 26/40
469/469 [==============================] - 28s 59ms/step - loss: 0.0076 - accuracy: 0.9976 - val_loss: 0.0303 - val_accuracy: 0.9919
Epoch 27/40
469/469 [==============================] - 28s 59ms/step - loss: 0.0084 - accuracy: 0.9971 - val_loss: 0.0310 - val_accuracy: 0.9907
Epoch 28/40
469/469 [==============================] - 28s 59ms/step - loss: 0.0072 - accuracy: 0.9979 - val_loss: 0.0298 - val_accuracy: 0.9916
Epoch 29/40
469/469 [==============================] - 28s 59ms/step - loss: 0.0067 - accuracy: 0.9979 - val_loss: 0.0299 - val_accuracy: 0.9916
Epoch 30/40
469/469 [==============================] - 28s 59ms/step - loss: 0.0066 - accuracy: 0.9978 - val_loss: 0.0297 - val_accuracy: 0.9920
```


```
469/469 [==============================] - 28s 59ms/step - loss: 0.0076 - accuracy: 0.9976 - val_loss: 0.0303 - val_accuracy: 0.9919
Epoch 27/40
469/469 [==============================] - 28s 59ms/step - loss: 0.0084 - accuracy: 0.9971 - val_loss: 0.0310 - val_accuracy: 0.9907
Epoch 28/40
469/469 [==============================] - 28s 59ms/step - loss: 0.0072 - accuracy: 0.9979 - val_loss: 0.0298 - val_accuracy: 0.9916
Epoch 29/40
469/469 [==============================] - 28s 59ms/step - loss: 0.0067 - accuracy: 0.9979 - val_loss: 0.0299 - val_accuracy: 0.9916
Epoch 30/40
469/469 [==============================] - 28s 59ms/step - loss: 0.0066 - accuracy: 0.9978 - val_loss: 0.0297 - val_accuracy: 0.9920
```


```
Epoch 31/40
469/469 [==============================] - 28s 59ms/step - loss: 0.0066 - accuracy: 0.9978 - val_loss: 0.0318 - val_accuracy: 0.9914
Epoch 32/40
469/469 [==============================] - 28s 59ms/step - loss: 0.0060 - accuracy: 0.9980 - val_loss: 0.0306 - val_accuracy: 0.9919
Epoch 33/40
469/469 [==============================] - 28s 59ms/step - loss: 0.0058 - accuracy: 0.9982 - val_loss: 0.0327 - val_accuracy: 0.9913
Epoch 34/40
469/469 [==============================] - 28s 59ms/step - loss: 0.0044 - accuracy: 0.9986 - val_loss: 0.0382 - val_accuracy: 0.9904
Epoch 35/40
469/469 [==============================] - 28s 59ms/step - loss: 0.0044 - accuracy: 0.9986 - val_loss: 0.0320 - val_accuracy: 0.9914
Epoch 36/40
469/469 [==============================] - 28s 59ms/step - loss: 0.0043 - accuracy: 0.9988 - val_loss: 0.0339 - val_accuracy: 0.9914
Epoch 37/40
469/469 [==============================] - 28s 59ms/step - loss: 0.0051 - accuracy: 0.9984 - val_loss: 0.0328 - val_accuracy: 0.9908
Epoch 38/40
469/469 [==============================] - 28s 59ms/step - loss: 0.0033 - accuracy: 0.9991 - val_loss: 0.0313 - val_accuracy: 0.9922
Epoch 39/40
469/469 [==============================] - 28s 59ms/step - loss: 0.0037 - accuracy: 0.9989 - val_loss: 0.0308 - val_accuracy: 0.9920
Epoch 40/40
469/469 [==============================] - 28s 59ms/step - loss: 0.0040 - accuracy: 0.9986 - val_loss: 0.0302 - val_accuracy: 0.9912
Test loss of CNN is: 0.030204854905605316, Test accuracy of CNN is 0.9911999702453613
```

In this part, we designed a model with multiple layers. The first layer is the convolutional layer, which is used to extract features, where we set the dimensionality of the output space to 32 and 64 for the two convolutional layers respectively.

The next layer is the pooling layer, which is used for compression and dimensionality reduction. Since the convolved image will be large and may have too many weak features extracted, thus the pooling operation is performed using the Maxpolling2d method. In this layer we choose the size of the input window to be (2*2), i.e. the layer is to take the maximum value in the (2*2) window and move and

aggregate it. This model has 2 convolutional layers and 2 pooling layers arranged in a convolution-pooling-convolution-pooling manner.

Then we add the flatten layer, which is used to convert the pooled data into one-dimensional arrays that are ready to be passed into the next layer. The fully connected layer is used to deeply connect the neurons between layers, and the dropout layer is added to the fully connected layer to prevent overfitting of the data by reducing the number of intermediate features, thus reducing redundancy. After above layers, our model has accuracy of 99.1%, which has reached our goal.

# 3B)

**Differences:**

- Fully-connected layers are fully-connected, but convolutional layers are locally-connected.
- A fully-connected layer, because it is fully-connected, has every input affecting its output, however this does not happen with a convolutional layer because it is locally-connected.
- The weights of the fully-connected layer are applied to all inputs, whereas the weights of the convolutional layer are only applied to part of the inputs, which means that the fully-connected layer accepts any form of input, whereas the convolutional layer only accepts input in the form of images.

**Potential Benefits:**

- The convolutional layer saves computation time. Since the neurons in the convolutional column share parameters, they are sharing computational resources during computation, which saves computation time. The fully-connected layer, on the other hand, takes multiple repetitions of the pooling operation for the same regions sampled, meaning that the fully-connected layer's computation will be more time-consuming.
- Convolutional layers can have multi-scale input images, whereas fully-connected layers must limit the image input size. Since the fully-connected layer uses global information about the image and has a requirement on the size of the input feature map, the network starts with a fixed size of the input image. The convolutional layer, on the other hand, takes a window sliding operation on local regions, so the convolutional layer does not need to fix the size of the input.
- Convolutional layers are more efficient for image recognition. Since convolutional layers can extract image features, they can learn the shape and texture of an image more efficiently, and are definitely more efficient for image recognition problems.

# Problem 4

## 4A)

```
import numpy as np
def salt_and_pepper(input, noise_level=0.5):
    """
    This applies salt and pepper noise to the input tensor - randomly setting bits to
    1 or 0.
    Parameters
    ----------
    input : tensor
        The tensor to apply salt and pepper noise to.
    noise_level : float
        The amount of salt and pepper noise to add.
    Returns
    -------
    tensor
        Tensor with salt and pepper noise applied.
    """
    # salt and pepper noise
    a = np.random.binomial(size=input.shape, n=1, p=(1 - noise_level))
    b = np.random.binomial(size=input.shape, n=1, p=0.5)
    c = (a==0) * b
    return input * a + c


#data preparation
flattened_x_train = x_train.reshape(-1,784)
flattened_x_train_seasoned = salt_and_pepper(flattened_x_train, noise_level=0.4)

flattened_x_test = x_test.reshape(-1,784)
flattened_x_test_seasoneed = salt_and_pepper(flattened_x_test, noise_level=0.4)
```

```
latent_dim = 96

input_image = keras.Input(shape=(784,))
encoded = Dense(128, activation='relu')(input_image)
encoded = Dense(latent_dim, activation='relu')(encoded)
decoded = Dense(128, activation='relu')(encoded)
decoded = Dense(784, activation='sigmoid')(decoded)

autoencoder = keras.Model(input_image, decoded)
encoder_only = keras.Model(input_image, encoded)

encoded_input = keras.Input(shape=(latent_dim,))
decoder_layer = Sequential(autoencoder.layers[-2:])
decoder = keras.Model(encoded_input, decoder_layer(encoded_input))

autoencoder.compile(optimizer='adam', loss='binary_crossentropy')
```

```
fit_info_AE = autoencoder.fit(flattened_x_train_seasoned, flattened_x_train,
                epochs=32,
                batch_size=64,
                shuffle=True,
                validation_data=(flattened_x_test_seasoneed, flattened_x_test))
```

**Explain what the model does: use the data-preparation and model definition code to explain how the goal of the model is achieved.**

Overview: This part of the code adds noise to the dataset and then removes it by encoding and decoding.

The code first defines the salt_and_pepper function, which is used to add noise randomly.

Next, the image is turned into a 1×784 array by a reshape operation and noise is added to the training and test sets, which are flattened_x_train_seasoned and flattened_x_test_seasoned respectively.

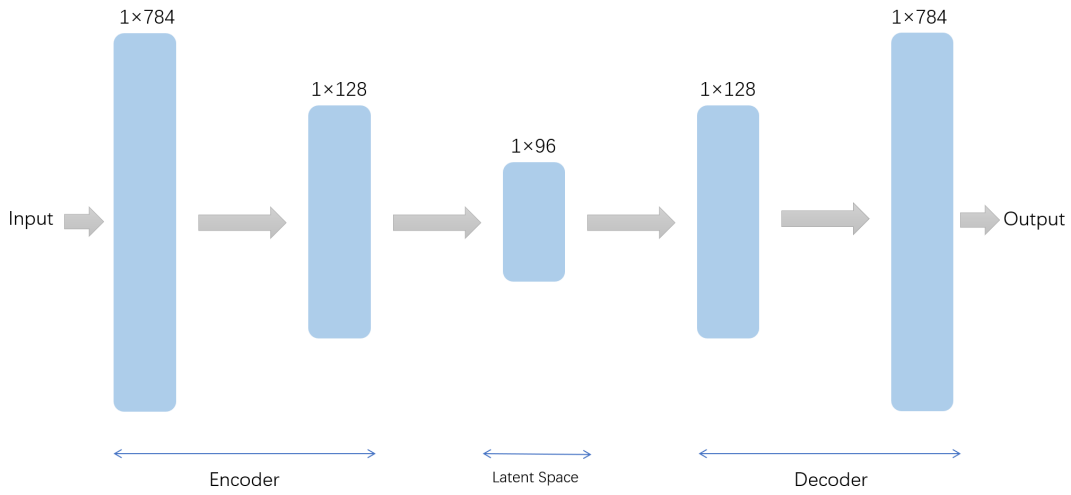
The compression of the image is then started and the image is encoded for representation and then passed to the next layer for further compression, which is the maximum of the compression and consists of 96 neurons.

Decoding and reconstruction of the image is performed. The encoded compressed image is passed to the decoding layer, which decodes the image into a fully connected layer of 128 neurons and finally reconstructs it as a 784 (28*28) image. The sigmoid function is used in the last layer because we want the output to be either 0 or 1.

The autoencoder is created to preserve the most characteristic and useful information in the image by compressing it and then reconstructing it by decoder, which removes the noise in the process. Finally we write the noise added data and the original data to the autoencoder and compare the results to see how well the autoencoder removes the noise.

**Explain the role of the loss function?** The loss function in this part of the code is used to compare the original image with the noise added image denoised by the autoencoder, it is used to see how correct the autoencoder is, and this is why it is binary.

**Draw a diagram of the model and include it in your report.**




**Train the model with the settings given.**

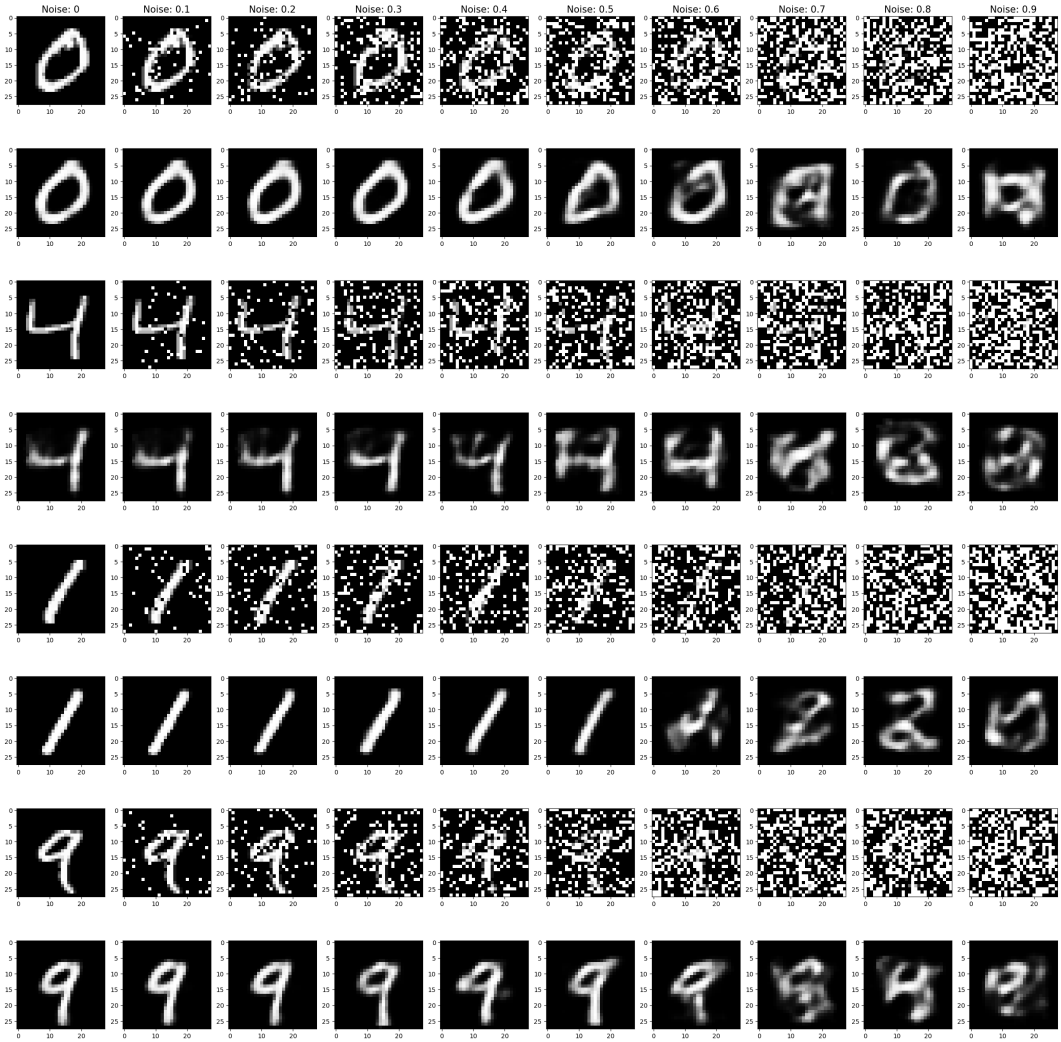
The result is as follows:

```
Epoch 1/32
938/938 [==============================] - 7s 7ms/step - loss: 0.1904 - val_loss: 0.1517
Epoch 2/32
938/938 [==============================] - 6s 7ms/step - loss: 0.1458 - val_loss: 0.1387
Epoch 3/32
938/938 [==============================] - 6s 6ms/step - loss: 0.1364 - val_loss: 0.1330
Epoch 4/32
938/938 [==============================] - 6s 7ms/step - loss: 0.1317 - val_loss: 0.1298
Epoch 5/32
938/938 [==============================] - 6s 7ms/step - loss: 0.1286 - val_loss: 0.1277
Epoch 6/32
938/938 [==============================] - 6s 7ms/step - loss: 0.1266 - val_loss: 0.1260
Epoch 7/32
938/938 [==============================] - 6s 7ms/step - loss: 0.1249 - val_loss: 0.1253
Epoch 8/32
938/938 [==============================] - 6s 6ms/step - loss: 0.1236 - val_loss: 0.1246
Epoch 9/32
938/938 [==============================] - 6s 7ms/step - loss: 0.1226 - val_loss: 0.1244
Epoch 10/32
938/938 [==============================] - 6s 7ms/step - loss: 0.1217 - val_loss: 0.1229
Epoch 11/32
938/938 [==============================] - 6s 7ms/step - loss: 0.1209 - val_loss: 0.1227
Epoch 12/32
938/938 [==============================] - 6s 7ms/step - loss: 0.1203 - val_loss: 0.1237
Epoch 13/32
938/938 [==============================] - 6s 7ms/step - loss: 0.1197 - val_loss: 0.1217
Epoch 14/32
938/938 [==============================] - 6s 6ms/step - loss: 0.1193 - val_loss: 0.1221
Epoch 15/32
938/938 [==============================] - 6s 7ms/step - loss: 0.1187 - val_loss: 0.1214
```

```
Epoch 18/32
938/938 [==============================] - 6s 7ms/step - loss: 0.1177 - val_loss: 0.1214
Epoch 19/32
938/938 [==============================] - 6s 7ms/step - loss: 0.1174 - val_loss: 0.1206
Epoch 20/32
938/938 [==============================] - 6s 7ms/step - loss: 0.1171 - val_loss: 0.1207
Epoch 21/32
938/938 [==============================] - 6s 6ms/step - loss: 0.1168 - val_loss: 0.1204
Epoch 22/32
938/938 [==============================] - 6s 7ms/step - loss: 0.1166 - val_loss: 0.1204
Epoch 23/32
938/938 [==============================] - 6s 7ms/step - loss: 0.1164 - val_loss: 0.1205
Epoch 24/32
938/938 [==============================] - 6s 7ms/step - loss: 0.1162 - val_loss: 0.1204
Epoch 25/32
938/938 [==============================] - 6s 6ms/step - loss: 0.1160 - val_loss: 0.1199
Epoch 26/32
938/938 [==============================] - 6s 6ms/step - loss: 0.1158 - val_loss: 0.1205
Epoch 27/32
938/938 [==============================] - 6s 7ms/step - loss: 0.1156 - val_loss: 0.1205
Epoch 28/32
938/938 [==============================] - 6s 7ms/step - loss: 0.1154 - val_loss: 0.1201
Epoch 29/32
938/938 [==============================] - 6s 7ms/step - loss: 0.1154 - val_loss: 0.1197
Epoch 30/32
938/938 [==============================] - 6s 7ms/step - loss: 0.1152 - val_loss: 0.1198
Epoch 31/32
938/938 [==============================] - 6s 7ms/step - loss: 0.1150 - val_loss: 0.1200
Epoch 32/32
938/938 [==============================] - 6s 7ms/step - loss: 0.1149 - val_loss: 0.1199
```

## 4B)

```
#Define noise level with 0.1 interval
noise = [0,0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1]

f, ax = plt.subplots(nrows = 8, ncols = 10, figsize = (30,30))

for i in range (8):
    for j in range (10):
        #Display seasoned image
        ax[0,j].imshow(tf.reshape(salt_and_pepper(flattened_x_train[1].reshape(1,-1),
    noise_level = noise[j]),(28,28)), cmap='gray')
        #Display denoised image by using autoencoder
        ax[1,j].imshow(tf.reshape(autoencoder(salt_and_pepper(flattened_x_train[1].
    reshape(1,-1),noise_level = noise[j])), shape = (28,28)), cmap='gray')

        #Display seasoned image
        ax[2,j].imshow(tf.reshape(salt_and_pepper(flattened_x_train[2].reshape(1,-1),
    noise_level = noise[j]),(28,28)), cmap='gray')
        #Display denoised image by using autoencoder
        ax[3,j].imshow(tf.reshape(autoencoder(salt_and_pepper(flattened_x_train[2].
    reshape(1,-1),noise_level = noise[j])), shape = (28,28)), cmap='gray')

        #Display seasoned image
        ax[4,j].imshow(tf.reshape(salt_and_pepper(flattened_x_train[3].reshape(1,-1),
    noise_level = noise[j]),(28,28)), cmap='gray')
        #Display denoised image by using autoencoder
        ax[5,j].imshow(tf.reshape(autoencoder(salt_and_pepper(flattened_x_train[3].
    reshape(1,-1),noise_level = noise[j])), shape = (28,28)), cmap='gray')

        #Display seasoned image
        ax[6,j].imshow(tf.reshape(salt_and_pepper(flattened_x_train[4].reshape(1,-1),
    noise_level = noise[j]),(28,28)), cmap='gray')
        #Display denoised image by using autoencoder
        ax[7,j].imshow(tf.reshape(autoencoder(salt_and_pepper(flattened_x_train[4].
    reshape(1,-1),noise_level = noise[j])), shape = (28,28)), cmap='gray')

#Make label of noise level for each columns
label = list(map(lambda n: f"Noise: {n}", noise))
for ax, j in zip(ax[0], label):
    ax.set_title(j, size=15)

plt.show()
```




**At what noise level does it become difficult to identify the digits for you?**

As we can see from above, at 0.4-0.5 noise level it became difficult to identify.

**At what noise level does the denoising stop working?**

At about 0.7 noise level the denoising stop working.

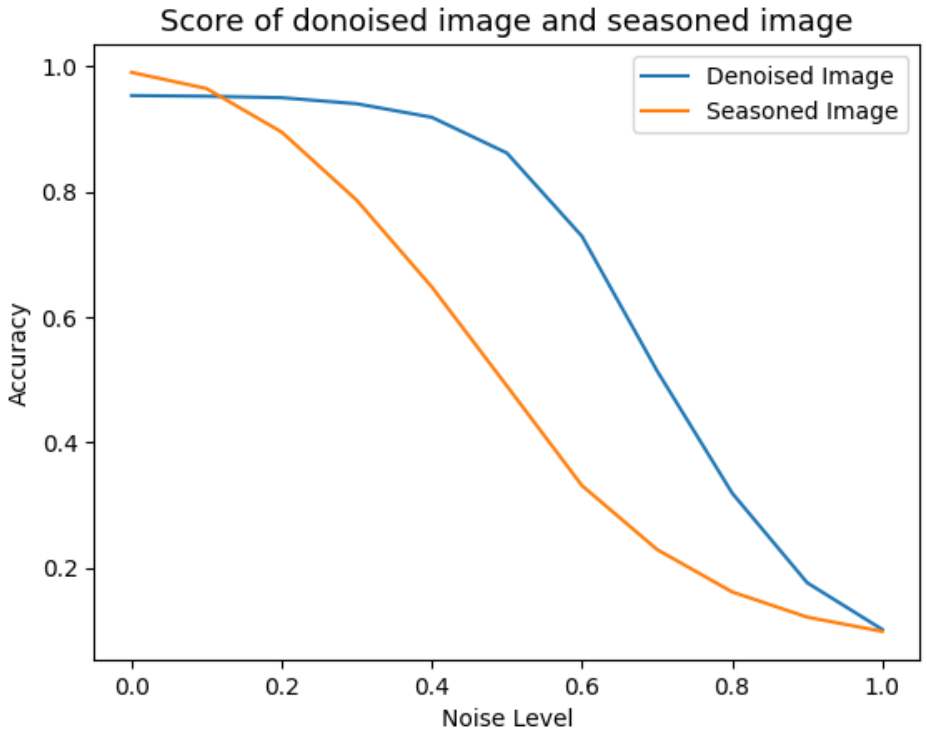
# 4C)

```
denoised_score = []
seasoned_score = []

for i,j in enumerate(noise):
    #Add noise to the image
    seasoned_x_test = tf.reshape(salt_and_pepper(flattened_x_test, noise_level=j),
    (10000,28,28,1))
    #Add noise to image and then use autoencoder to denoised
```

```
    denoised_x_test = tf.reshape(autoencoder(salt_and_pepper(flattened_x_test,
    noise_level=j)), (10000,28,28,1))
    #Use model from Q3 to compute score
    score_denoised = cnn_model.evaluate(denoised_x_test, y_test, verbose=0)[1]
    score_seasoned = cnn_model.evaluate(seasoned_x_test, y_test, verbose=0)[1]

    #Append denoised score and seasoned score
    denoised_score.append(score_denoised)
    seasoned_score.append(score_seasoned)

print('The denoised score are:', denoised_score)
print('The seasoned score are:', seasoned_score)

plt.plot(noise, denoised_score, label="Denoised Image")
plt.plot(noise, seasoned_score, label="Seasoned Image")
plt.ylabel("Accuracy")
plt.xlabel("Noise Level")
plt.title("Score of donoised image and seasoned image", fontsize=13)
plt.legend()
plt.show()
```

The denoised score are: [0.9538999795913696, 0.9527999758720398, 0.9505000114440918, 0.9408000111579895, 0.9192000031471252, 0.8621000051498413, 0.729200005531311, 0.5142999887466431, 0.3190999925136566, 0.17579999566078186, 0.10119999945163727]

The seasoned score are: [0.9908999800682068, 0.9650999903678894, 0.8956000208854675, 0.7867000102996826, 0.6481999754905701, 0.4900999963283539, 0.3310000002384186, 0.2289000004529953, 0.16120000183582306, 0.120899997651577, 0.09809999912977219]



From the above graph we can see that when the noise level is between (0, 0.15), better results are obtained without using the autoencoder on the image, probably because a lot of useful feature data is lost during the encoding process (especially during pooling stage), leading to the final elimination of not only noise but also useful data even when reconstructing, which makes the model perform vary badly. When the noise level is between (0.15, 1), better results are achieved when using the autoencoder on the image, and the slope of the denoised image is flatter when the noise level is 0.15-0.4, indicating that the autoencoder can achieve better performance when the noise level is between 0.15 and 0.4. However, when the noise level is (0.5, 1), the slope of the denoised image becomes larger and the denoised image becomes less accurate, indicating that the autoencoder cannot well eliminate noise after the noise level of 0.5. This is consistent with what we observed intuitively at 3c. For the seasoned image, its slope is relatively smooth between (0.15, 0.6). When the noise level is in this part, its accuracy decreases gently, while when the noise level is between (0.6, 1), its slope becomes larger and its accuracy drops sharply. At a noise level of 1, they are equally accurate (both perform badly) and the autoencoder fails completely.

# 4D)

In the above, we used the decoder in the autoencoder to take the data in the latent space and reconstruct the compressed image, so in theory it can extract the feature values from any data in the latent spcae and reconstruct the image, even if we add noise directly to the latent space to make it a hand-written digit, i.e. an irregular number, the decoder can reconstruct the image and determine which digit it is, even if there are some digits that we cannot recognise with the human eye.