Artificial intelligence for VR/AR

Report for Lab3 Session

Xuefeng Wei
Institut Polytechnique de Paris
xuefeng.wei@ip-paris.fr

Part I.

This experiment focuses on predictive classification using CNN on the Fashion MNIST dataset, where each image has a size of 28x28x1 and has 10 classes. After loading this data, the first 9 images in the training set are displayed, as shown in Figure 1.

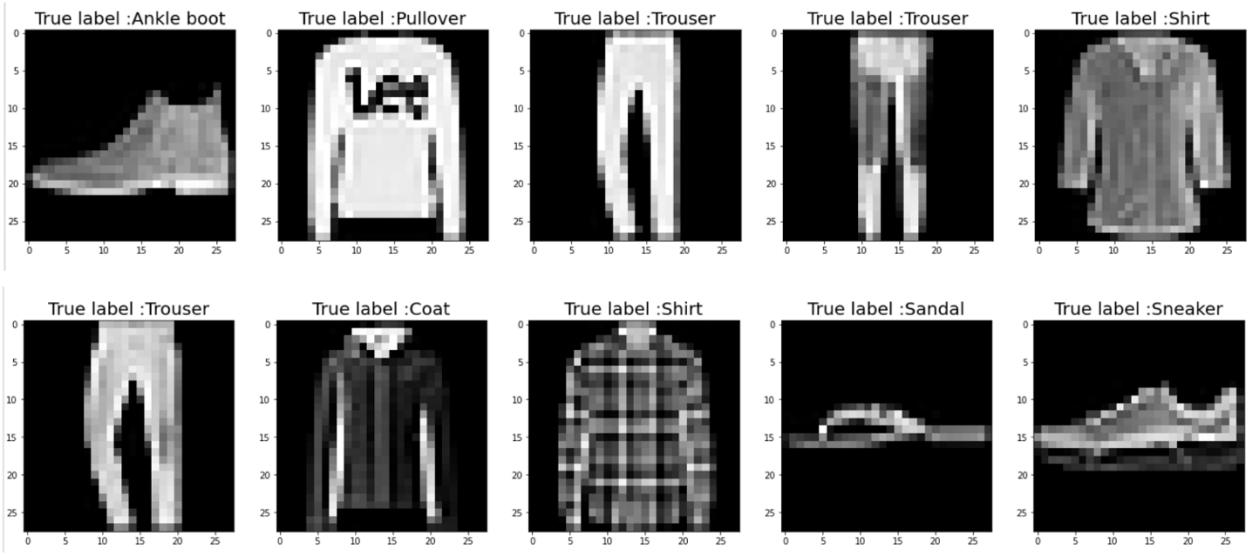

Figure1. The first 9 images existent in the training set

Then the authors classified the dataset by CNN network with the settings of epochs=5, batch size=32, optimizer=SGD, learning ratee=0.01, momentum=0.9， The structure of the network is shown in Figure 2. The results are shown in Table 1 and Figure 3.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_14 (Conv2D)           (None, 26, 26, 16)        160

max_pooling2d_14 (MaxPoolin  (None, 13, 13, 16)        0
g2D)

flatten_14 (Flatten)         (None, 2704)              0

dense_28 (Dense)             (None, 16)                43280

dense_29 (Dense)             (None, 10)                170

=================================================================
Total params: 43,610
Trainable params: 43,610
Non-trainable params: 0
```

Figure2. The first architecture of the CNN

No of filters	8	16	32	64	128

System Accuracy (%)	88.58	88.80	89.40	89.30	89.41
Time Costing (s)	28.10	28.34	30.08	31.32	33.55

Table1. System performance evaluation for various numbers of filters in the convolutional layer

```
Epoch 5/5
1875/1875 [==============================] - 5s 3ms/step - loss: 0.2800 - accuracy: 0.8992 - val_loss: 0.3189 - val_accuracy: 0.8858
When the number of filters is 8 The system accuracy is = 88.58% Time costing is 28.10308265686035s

Epoch 5/5
1875/1875 [==============================] - 5s 3ms/step - loss: 0.2581 - accuracy: 0.9044 - val_loss: 0.3131 - val_accuracy: 0.8880
When the number of filters is 16 The system accuracy is = 88.80% Time costing is 28.34582757949829s

Epoch 5/5
1875/1875 [==============================] - 6s 3ms/step - loss: 0.2533 - accuracy: 0.9083 - val_loss: 0.2980 - val_accuracy: 0.8940
When the number of filters is 32 The system accuracy is = 89.40% Time costing is 30.08874273300171s

Epoch 5/5
1875/1875 [==============================] - 6s 3ms/step - loss: 0.2435 - accuracy: 0.9114 - val_loss: 0.2917 - val_accuracy: 0.8930
When the number of filters is 64 The system accuracy is = 89.30% Time costing is 31.321433544158936s

Epoch 5/5
1875/1875 [==============================] - 7s 4ms/step - loss: 0.2517 - accuracy: 0.9086 - val_loss: 0.2961 - val_accuracy: 0.8941
When the number of filters is 128 The system accuracy is = 89.41% Time costing is 33.55412936210632s
```

Figure3. Results shown in Google Colab console for exercise 2

The results in Table 1 show that increasing the number of filters increases the test accuracy slightly, but not significantly, while the runtime increases. Next, the number of neurons in the dense hidden layer is discussed to explore its effect on the model, and the number of filters in the convolutional layer is fixed to 32. The results are shown in Table 2 and Figure 4.

No of neurons	16	64	128	256	512
System Accuracy (%)	89.48	89.81	89.12	90.02	90.71
Time Costing (s)	30.57	31.05	41.42	31.03	41.44

Table2. System performance evaluation for various numbers of neurons in the dense hidden layer

```
Epoch 5/5
1875/1875 [==============================] - 6s 3ms/step - loss: 0.2508 - accuracy: 0.9090 - val_loss: 0.2948 - val_accuracy: 0.8948
Then number of neurons in the dense hidden layer is 16 When the number of filters is 32
 The system accuracy is = 89.48% Time costing is 30.57835102081299s

Epoch 5/5
1875/1875 [==============================] - 6s 3ms/step - loss: 0.2201 - accuracy: 0.9189 - val_loss: 0.2783 - val_accuracy: 0.8981
Then number of neurons in the dense hidden layer is 64 When the number of filters is 32
 The system accuracy is = 89.81% Time costing is 31.059712648391724s

Epoch 5/5
1875/1875 [==============================] - 6s 3ms/step - loss: 0.2107 - accuracy: 0.9216 - val_loss: 0.3050 - val_accuracy: 0.8912
Then number of neurons in the dense hidden layer is 128 When the number of filters is 32
 The system accuracy is = 89.12% Time costing is 41.42394208908081s

Epoch 5/5
1875/1875 [==============================] - 7s 4ms/step - loss: 0.2061 - accuracy: 0.9240 - val_loss: 0.2694 - val_accuracy: 0.9002
Then number of neurons in the dense hidden layer is 256 When the number of filters is 32
 The system accuracy is = 90.02% Time costing is 31.03911304473877s

Epoch 5/5
1875/1875 [==============================] - 7s 4ms/step - loss: 0.2016 - accuracy: 0.9257 - val_loss: 0.2569 - val_accuracy: 0.9071
Then number of neurons in the dense hidden layer is 512 When the number of filters is 32
 The system accuracy is = 90.71% Time costing is 41.44890069961548s
```

Figure4. Results shown in Google Colab console for exercise 3

From the results in Table 2, it can be seen that as the number of neurons in the hidden layer increases, the accuracy of the model on the test set also improves, while the training time also increases.

The effect of the size of the epochs on the model is also explored, The number of neurons in the hidden layer is set to 16, and the number of filters in the convolution layer is fixed to 32. The results can be seen in Table

3 and Figure 5.

No of epochs	1	2	5	10	20
System Accuracy (%)	86.04	88.08	89.41	90.07	90.15
Time Costing (s)	10.72	21.05	29.39	82.40	142.42

Table3. System performance evaluation for different values of the number of epochs

```
1875/1875 [==============================] - 6s 3ms/step - loss: 0.5285 - accuracy: 0.8141 - val_loss: 0.3861 - val_accuracy: 0.8604
The epochs is 1 The system accuracy is = 86.04% Time costing is 10.724628686904907s

Epoch 1/2
1875/1875 [==============================] - 6s 3ms/step - loss: 0.5410 - accuracy: 0.8039 - val_loss: 0.4202 - val_accuracy: 0.8462
Epoch 2/2
1875/1875 [==============================] - 6s 3ms/step - loss: 0.3466 - accuracy: 0.8775 - val_loss: 0.3313 - val_accuracy: 0.8808
The epochs is 2 The system accuracy is = 88.08% Time costing is 21.057918787002563s

Epoch 5/5
1875/1875 [==============================] - 6s 3ms/step - loss: 0.2510 - accuracy: 0.9088 - val_loss: 0.2983 - val_accuracy: 0.8941
The epochs is 5 The system accuracy is = 89.41% Time costing is 29.3931667804718s

Epoch 10/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.1971 - accuracy: 0.9262 - val_loss: 0.2848 - val_accuracy: 0.9007
The epochs is 10 The system accuracy is = 90.07% Time costing is 82.40949249267578s

Epoch 20/20
1875/1875 [==============================] - 6s 3ms/step - loss: 0.1314 - accuracy: 0.9510 - val_loss: 0.3396 - val_accuracy: 0.9015
The epochs is 20 The system accuracy is = 90.15% Time costing is 142.42768883705l4s
```

Figure5. Results shown in Google Colab console for exercise 4

The results in Table 3 show that more epochs allow the model to converge more fully, i.e., the model performs better on the test set. The effect of learning rate on the model is also discussed by the authors, set the epochs to 5, and then change the learning rate of the optimizer, the results can be seen in Table 4 and Figure 6.

Learning rate	0.1	0.01	0.001	0.0001	0.00001
System Accuracy (%)	82.87	89.05	86.11	79.41	47.45
Time Costing (s)	41.43	29.00	41.42	41.43	41.43

Table4. System performance evaluation for various learning rates of the SGD

```
Epoch 5/5
1875/1875 [==============================] - 6s 3ms/step - loss: 0.4676 - accuracy: 0.8414 - val_loss: 0.5492 - val_accuracy: 0.8287
The learnng rate s 0.1 The system accuracy is = 82.87% Time costing is 41.43734383583069s

Epoch 5/5
1875/1875 [==============================] - 6s 3ms/step - loss: 0.2569 - accuracy: 0.9063 - val_loss: 0.3009 - val_accuracy: 0.8905
The learnng rate s 0.01 The system accuracy is = 89.05% Time costing is 29.00029706954956s

Epoch 5/5
1875/1875 [==============================] - 6s 3ms/step - loss: 0.3852 - accuracy: 0.8640 - val_loss: 0.3982 - val_accuracy: 0.8611
The learnng rate s 0.001 The system accuracy is = 86.11% Time costing is 41.42604422569275s

Epoch 5/5
1875/1875 [==============================] - 6s 3ms/step - loss: 0.5809 - accuracy: 0.8011 - val_loss: 0.5890 - val_accuracy: 0.7941
The learnng rate s 0.0001 The system accuracy is = 79.41% Time costing is 41.43814516067505s

Epoch 5/5
1875/1875 [==============================] - 6s 3ms/step - loss: 1.7468 - accuracy: 0.4664 - val_loss: 1.6827 - val_accuracy: 0.4745
The learnng rate s 0.00001 The system accuracy is = 47.45% Time costing is 41.43887662887573s
```

Figure6. Results shown in Google Colab console for exercise 5

From Table 4, we can see that after the epochs are fixed at 5, the smaller learning rate does not allow the

model to converge, and more epochs are needed to allow the model to reach convergence. Little impact on training time. In the dropout layer, the number of neurons per dropout will also have an impact on the model, so the size of the dropout percentage on the model is also discussed, the results of the convergence speed with and without dropout layer (0.2) can be seen in Figure6. The results of the effect of the different dropout percentage can be seen in Table 5 and Figure 7.

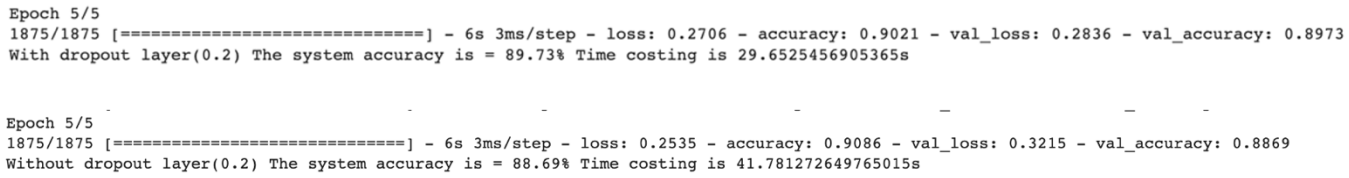

Figure6. Results shown in Google Colab console for exercise 6-1

From the results in Figure 6, the speed of training increases when the dropout layer is added, which is since there are only fewer neurons during training.

Dropout percentage	0.1	0.2	0.3	0.4	0.5
System Accuracy (%)	89.11	89.31	89.09	89.23	88.61
Time Costing (s)	32.06	31.28	30.05	29.88	29.85

Table5. System performance evaluation for number of neurons dropped in the dropout layer

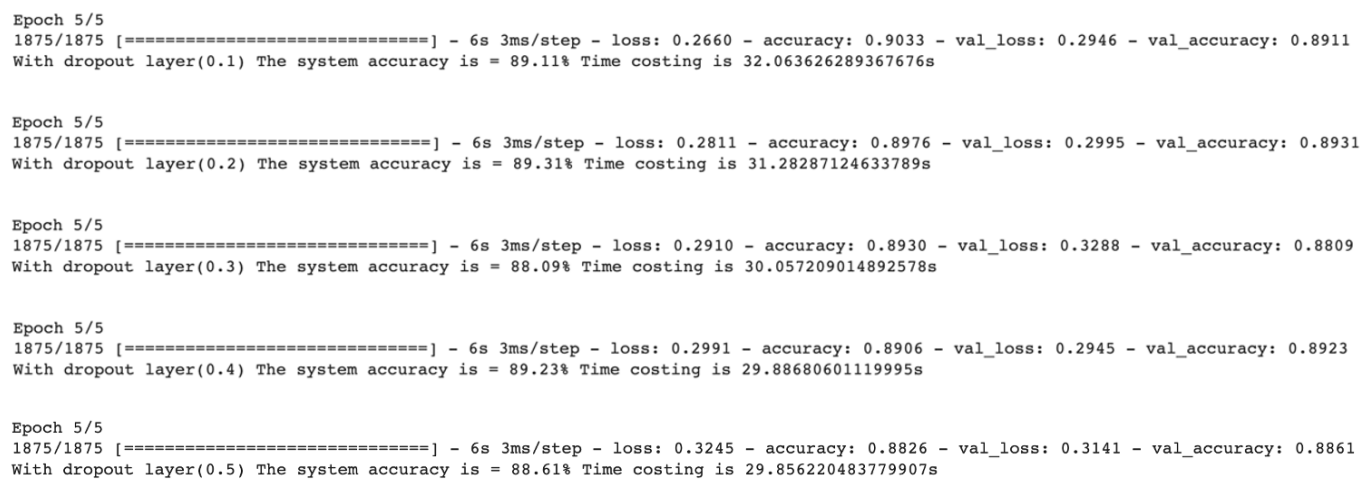

Figure7. Results shown in Google Colab console for exercise 6-2

From the results in Table 7, we can observe that as the dropout ratio of the dropout layer increases, the time needed for training decreases, but the performance of the model also decreases, and at a ratio of 20%, the model performs the best and the time is moderate, so the appropriate ratio should be chosen. Then the authors choose the optimal hyperparameters from the previous experiments which are dropout percentage = 0.2, learning rate = 0.01, number of epochs = 20, number of neurons = 128, number of filters = 32 or 128. because when number of filters=32 and 128, so the authors did a control experiment to select the optimal parameters, and the results are shown in Figure 8.

```
Epoch 20/20
1875/1875 [==============================] - 6s 3ms/step - loss: 0.0875 - accuracy: 0.9668 - val_loss: 0.2969 - val_accuracy: 0.9136
With dropout layer(0.2), epochs =20, learning ratee = 0.01,
 number of neurons = 128, number of filters=32,
 The system accuracy is = 91.36% Time costing is 122.67604064941406s


Epoch 20/20
1875/1875 [==============================] - 8s 4ms/step - loss: 0.0663 - accuracy: 0.9748 - val_loss: 0.3206 - val_accuracy: 0.9148
With dropout layer(0.2), epochs =20, learning ratee = 0.01,
 number of neurons = 128, number of filters=128,
 The system accuracy is = 91.48% Time costing is 148.4873342514038s
```

Figure8. Results shown in Google Colab console for exercise 7

From the results, when dropout percentage = 0.2, learning rate = 0.01, number of epochs = 20, number of neurons = 128, number of filters = 128, the model can reach the highest performance. Next, the trained optimal model is saved for testing at the time, Then the author created a new CNN network in another python script, and then used the saved weight file to predict the new image, the image is shown in Figure 9, and the prediction result is shown in Figure 10.

Figure9. Tested sample in exercise 8

```
1/1 [==============================] - 0s 101ms/step
The predicted label is Pullover

Process finished with exit code 0
```

Figure10. Predicted result in exercise 8

From the results in Figure 10, the neural network loaded with the weight file can work successfully and its prediction is consistent with our expectation, that is, the network's prediction is Pullover.

Part II.

Unlike Part I where the entire test set data is used for testing, in Part II, the authors split the train set into 2 parts, i.e., the test on and the validation set, with this method, the model's performance on the test set will be more objective because we will select the best performing model to test by selecting it on the validation set, rather than directly by selecting the best model on the test set.

The 5-fold cross-validation allows us to evaluate the system more objectively. The learning curve and loss curve of the 5-fold cross-validation are drawn as shown in Figure 10, and the accuracy values on the 5-fold and the final average are shown in Table 6.

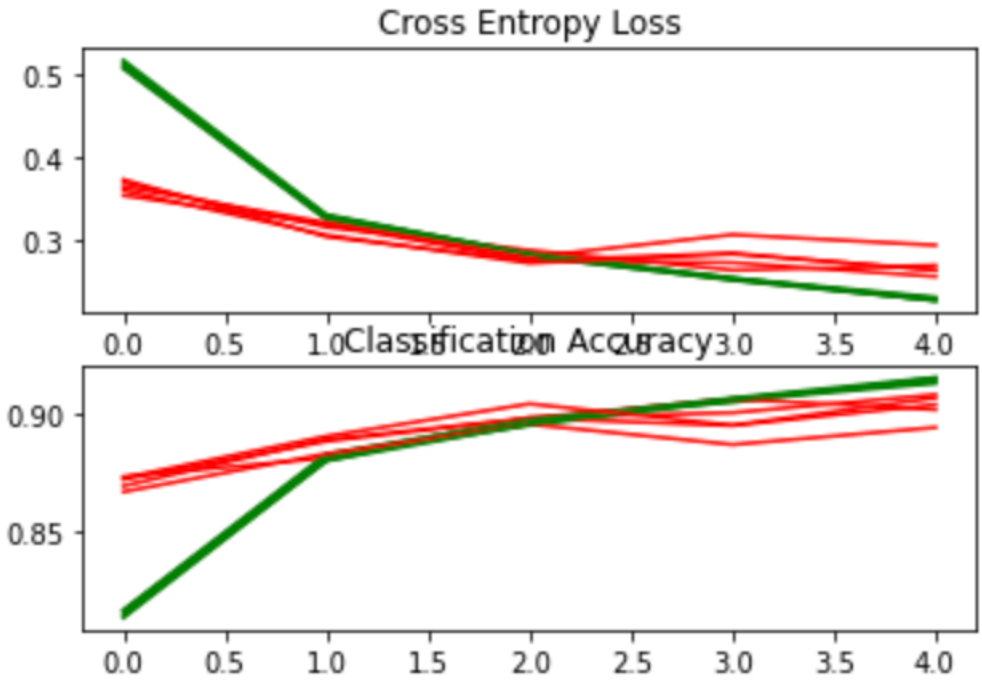

Figure10. Accuracy curves and loss curves for Application 2

Set	0	1	2	3	4
Accuracy (%)	90.09%	89.82%	89.12%	89.16%	89.73%
Mean Acc (%)	89.58				

Table5. Testing accuracy in 5 sets and the average testing accuracy

The effect of padding operation on the model is discussed when the number of filters is small. In this problem, the model of Application1 will be used, with hyperparameters set to filtrres=64, number of neurons in the hidden layer=128, batchsize=32, epoch=20. The accuracy and loss curves of the model with and without the paddng operation are shown in Figure 11

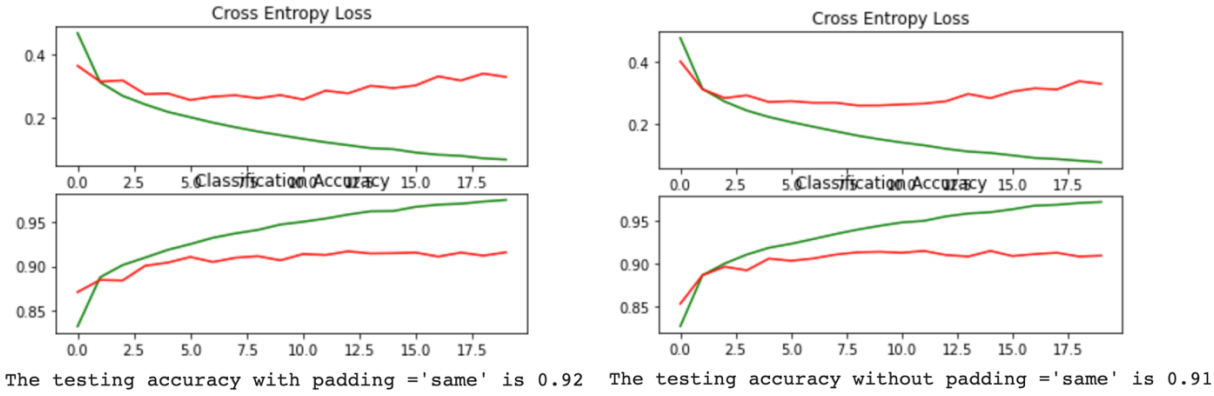

Figure11. Accuracy and loss curve for CNN with and without padding operation

From the results in Figure 11, the CNN is able to achieve higher test accuracy after adding the padding operation. This is because the padding operation allows the CNN to contain more data information without discarding the original graph information and allows the deeper layers and inputs to remain large enough, so the model performs better. From the curve, we can see that after adding the padding operation, there are more inflection points in the learning process, and the learning curve is not as smooth as the general case.

Appendix

Code for Exercise 1

```
def main():
    (trainX,    trainY), (testX, testY) = fashion_mnist.load_data()
    fig = plt.figure(figsize=(8, 8))
    rows, cols = 2, 5
    for j in range(0, 9):
        fig.add_subplot(rows, cols, j + 1)
        plt.imshow(trainX[j],'gray')
    plt.show()
    return

if __name__ == '__main__':
    main()
```

Comment: The for loop controls the number of output images to 9, and then the imshow function can display the images

Code for Exercise 2

```
def prepareData(trainX, trainY, testX, testY):

    trainX = trainX.reshape(60000,28,28,1)
    testX = testX.reshape(10000,28,28,1)
    trainX = trainX / 255
    testX = testX / 255
    trainY = np_utils.to_categorical(trainY)
    testY = np_utils.to_categorical(testY)

    return trainX, trainY, testX, testY

def defineModel(input_shape, num_classes):
    model = Sequential()
    model.add(Conv2D(16, (3, 3), input_shape=(28, 28, 1),activation='relu')) # Change the number of filters here
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Flatten())
    model.add(Dense(16,activation='relu',kernel_initializer='he_uniform'))
    model.add(Dense(10, activation='softmax'))
    model.compile(loss='categorical_crossentropy'                                                                    ,
    optimizer=gradient_descent_v2.SGD(learning_rate=0.01,momentum = 0.9),
```

```
                    metrics = ['accuracy'])

    return model

def defineTrainAndEvaluateClassic(trainX, trainY, testX, testY):

    model=defineModel(trainX, 10)
    time_start= time.time()
    model.fit(trainX,trainY,
                batch_size=32,
                epochs=5,
                validation_data=(testX, testY))
    time_end = time.time()
    scores = model.evaluate(testX, testY, verbose=0)
    print("When the number of filters is 16","The system accuracy is = {:.2f}%".format(scores[1]*100),"Time costing is {}s".format(time_end-time_start))


    return

def main():

    (trainX,  trainY), (testX, testY) = fashion_mnist.load_data()
    trainX, trainY, testX, testY = prepareData(trainX, trainY, testX, testY)
    defineTrainAndEvaluateClassic(trainX, trainY, testX, testY)

    return
```

Comment: The number of filters can be controlled by changing the Con2D function in the line of code marked in red.

Code for Exercise 3

```
def defineModel(input_shape, num_classes):
    model = Sequential()
    model.add(Conv2D(32, (3, 3), input_shape=(28, 28, 1),activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Flatten())
    model.add(Dense(512,activation='relu',kernel_initializer='he_uniform')) //Change the number of neurons in the hidden layer here
    model.add(Dense(10, activation='softmax'))
    #TODO - Application 1 - Step 6g - Compile the model
    model.compile(loss='categorical_crossentropy' , optimizer=gradient_descent_v2.SGD(learning_rate=0.01,momentum = 0.9),
                    metrics = ['accuracy'])
    return model
```

Comment: The number of neurons in the hidden layer can be changed by changing the first argument of the dense function in the blue code line

Code for Exercise 4

```
def defineTrainAndEvaluateClassic(trainX, trainY, testX, testY):

    #TODO - Application 1 - Step 6 - Call the defineModel function
    model=defineModel(trainX, 10)
    model.summary()
    #TODO - Application 1 - Step 7 - Train the model
    time_start= time.time()
    model.fit(trainX,trainY,
                batch_size=32,
                epochs=20, #change the number here we can change the epochs for the trainig
                validation_data=(testX, testY))
    time_end = time.time()
    #TODO - Application 1 - Step 8 - Evaluate the model
    scores = model.evaluate(testX, testY, verbose=0)
    print("The epochs is 20","The system accuracy is = {:.2f}%".format(scores[1]*100),"Time costing is {}s".format(time_end-time_start))


    return
```

Code for Exercise 5

```
def defineModel(input_shape, num_classes):
    model = Sequential()
    model.add(Conv2D(32, (3, 3), input_shape=(28, 28, 1),activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Flatten())
    model.add(Dense(16,activation='relu',kernel_initializer='he_uniform'))
    model.add(Dense(10, activation='softmax'))
    #TODO - Application 1 - Step 6g - Compile the model

    model.compile(loss='categorical_crossentropy',optimizer=gradient_descent_v2.SGD(learning_rate=0.0
    0001,momentum = 0.9), #change the learning rate here
                  metrics = ['accuracy'])

    return model
```

Code for Exercise 6

```
def defineModel(input_shape, num_classes):
	model = Sequential()
	model.add(Conv2D(32, (3, 3), input_shape=(28, 28, 1),activation='relu'))
	model.add(MaxPooling2D(pool_size=(2, 2)))
	model.add(Dropout(0.5)) # change the number here to change the percentage of dropout
	model.add(Flatten())
	model.add(Dense(16,activation='relu',kernel_initializer='he_uniform'))
	model.add(Dense(10, activation='softmax'))
	#TODO - Application 1 - Step 6g - Compile the model
	model.compile(loss='categorical_crossentropy' ,
optimizer=gradient_descent_v2.SGD(learning_rate=0.01,momentum = 0.9),
			metrics = ['accuracy'])

	return model
```

Code for Exercise 7

```
def defineModel(input_shape, num_classes):
	model = Sequential()
	model.add(Conv2D(128, (3, 3), input_shape=(28, 28, 1),activation='relu'))
	model.add(MaxPooling2D(pool_size=(2, 2)))
	model.add(Dropout(0.2))
	model.add(Flatten())
	model.add(Dense(128,activation='relu',kernel_initializer='he_uniform'))
	model.add(Dense(10, activation='softmax'))
	#TODO - Application 1 - Step 6g - Compile the model
	model.compile(loss='categorical_crossentropy' ,
optimizer=gradient_descent_v2.SGD(learning_rate=0.01,momentum = 0.9),
			metrics = ['accuracy'])

	return model

def defineTrainAndEvaluateClassic(trainX, trainY, testX, testY):

	#TODO - Application 1 - Step 6 - Call the defineModel function
	model=defineModel(trainX, 10)
	model.summary()
	#TODO - Application 1 - Step 7 - Train the model
```

```
    time_start= time.time()
    model.fit(trainX,trainY,
                batch_size=32,
                epochs=20,
                validation_data=(testX, testY))
    time_end = time.time()
    #TODO - Application 1 - Step 8 - Evaluate the model
    scores = model.evaluate(testX, testY, verbose=0)
    print("With dropout layer(0.2), epochs =20, learning ratee = 0.01,\n number of neurons = 128, number of filters=128,\n","The system accuracy is = {:.2f}%".format(scores[1]*100),"Time costing is {}s".format(time_end-time_start))


    return
```

Code for Exercise 8

```
def defineTrainAndEvaluateClassic(trainX, trainY, testX, testY):

    classes = ['T-shirt/top', 'Trouser' , 'Pullover','Dress','Coat','Sandal','Shirt','Sneaker','Bag','Ankle boot']
    model = defineModel()
    image_path = 'sample_image.png'
    image = tf.keras.preprocessing.image.load_img(image_path, color_mode='grayscale', target_size=(28,28))
    tf_image = np.array(image)
    image_tensor = tf.convert_to_tensor(tf_image)
    image_tensor = tf.expand_dims(image_tensor, axis=0)
    model = load_model('./Fashion_MNIST_model.h5')
    predictedLabel = np.argmax(model.predict(image_tensor))
    print("The predicted label is {} ".format(classes[predictedLabel]))

    return
```

Comment: The tf.convert_to_tensor function converts the array into a tensor, and then converts the 2D tensor into a 3D tensor to fit the CNN model, and then the predict function can be used to make predictions.

Code for Exercise 9&10

```
def plotaccandloss(history, acc):
  pyplot.subplot(211)
```

```
    pyplot.title('Cross Entropy Loss')
    pyplot.plot(history.history['loss'], color='green', label='train')
    pyplot.plot(history.history['val_loss'], color='red', label='test')

    pyplot.subplot(212)
    pyplot.title('Classification Accuracy')
    pyplot.plot(history.history['accuracy'], color='green', label='train')
    pyplot.plot(history.history['val_accuracy'], color='red', label='test')

    pyplot.show()
    print("The testing accuracy without padding ='same' is {:.2f}".format(acc))

def defineModel(input_shape, num_classes):
        model = Sequential()
        model.add(Conv2D(64, (3, 3), input_shape=(28, 28, 1),activation='relu', padding=’same)) //add padding
here
        model.add(MaxPooling2D(pool_size=(2, 2)))
        model.add(Dropout(0.2))
        model.add(Flatten())
        model.add(Dense(128,activation='relu',kernel_initializer='he_uniform'))
        model.add(Dense(10, activation='softmax'))
        #TODO - Application 1 - Step 6g - Compile the model
        model.compile(loss='categorical_crossentropy'                                                    ,
optimizer=gradient_descent_v2.SGD(learning_rate=0.01,momentum = 0.9),
                              metrics = ['accuracy'])

        return model
```

Comment: Adding padding='same' to the Conv2D function adds a padding operation to the CNN.
