UNDERGRADUATE AI COURSEWORK 2

By Alba Haque Sultana

TABLE OF CONTENTS

INTRODUCTION TO COURSEWORK

I have chosen to complete lab 5 exercises as part of this coursework. The aim was to investigate a technique (convents) across similar datasets and I have conducted 12 different tests do to so. The chosen datasets for this coursework are MNIST and Fashion_MNIST.

LAB 5 PART 1 INSTRUCTIONS

1. Work through 5.1-introduction-to-convnets, running each cell and reading the linking commentary
2. Experiment with moving/adding convolutional layers, the number of filters and the size of the local receptive field.

1 MNIST

MNIST stands for (Modified National Institute of Standards and Technology database). It is a large database of handwritten digits that is commonly used for training various image processing systems. The MNIST database contains 60,000 training images and 10,000 testing images.

Image data:

Images typically have three dimensions: height, width, and color depth. Although grayscale images (like our MNIST digits) have only a single color channel and could thus be stored in 2D tensors, by convention image tensors are always 3D, with a one-dimensional color channel for grayscale images. A batch of 128 grayscale images of size 256 × 256 could thus be stored in a tensor of shape (128, 256, 256, 1), and a batch of 128 color images could be stored in a tensor of shape (128, 256, 256, 3).

1.1 Work through 5.1-introduction-to-convnets, running each cell.

I have used convnets to classify MNIST digits, a task we performed in chapter 2 using a densely connected network (our test accuracy then was 97.8%). Even though the convnet will be basic, its accuracy will increase compared to the densely connected model from chapter 2. The basic convnet contains a stack of Conv2D and MaxPooling2D layers.

A convnet takes as input tensors of shape (image_height, image_width, image_channels) excluding the batch dimension.

In this case, the convnet has been configured to process inputs of size (28, 28, 1), which is the format of MNIST images. We'll do this by passing the argument input_shape=(28, 28, 1) to the first layer.

In [9]:
```
#Listing 5.1 Instantiating a small convnet
from tensorflow.keras import models
from tensorflow.keras import layers
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1
model.add(layers.MaxPooling2D((2, 2)))
```

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

In [10]:
```
model.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_6 (Conv2D)            (None, 26, 26, 32)        320
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 13, 13, 32)        0
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 11, 11, 64)        18496
_________________________________________________________________
max_pooling2d_5 (MaxPooling2 (None, 5, 5, 64)          0
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 3, 3, 64)          36928
=================================================================
Total params: 55,744
Trainable params: 55,744
Non-trainable params: 0
_________________________________________________________________
```

Model summary with only convolutional layers

In the model's summary above, the output of every Conv2D and MaxPooling2D layer is a 3D tensor of shape (height, width, channels). The width and height dimensions tends to shrink as you go deeper in the network. The number of channels is controlled by the first argument passed to the Conv2D layers (32 or 64).

The next step is to feed the last output tensor (of shape (3, 3, 64)) into a densely connected classifier network by using a stack of Dense layers. These classifiers process vectors, which are 1D, whereas the current output is a 3D tensor. First we have to flatten the 3D outputs to 1D, and then add a few Dense layers on top.

In [11]:
```
#Listing 5.2 Adding a classifier on top of the convnet
model.add(layers.Flatten())
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))
```

In [12]:
```
model.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_6 (Conv2D)            (None, 26, 26, 32)        320
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 13, 13, 32)        0
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 11, 11, 64)        18496
_________________________________________________________________
max_pooling2d_5 (MaxPooling2 (None, 5, 5, 64)          0
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 3, 3, 64)          36928
_________________________________________________________________
flatten_2 (Flatten)          (None, 576)               0
_________________________________________________________________
dense_4 (Dense)              (None, 64)                36928
_________________________________________________________________
dense_5 (Dense)              (None, 10)                650
=================================================================
```

```
Total params: 93,322
Trainable params: 93,322
Non-trainable params: 0
_________________________________________________________________
```

Model summary with convolutional layers and Dense layers

On the model summary above the (3, 3, 64) outputs are flattened into vectors of shape (576,) before going through two Dense layers. Now, the convnet has to be trained on the MNIST digits.

In [13]:
```
# Listing 5.3 Training the convnet on MNIST images
from tensorflow.keras.datasets import mnist
from tensorflow.keras.utils import to_categorical

(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
train_images = train_images.reshape((60000, 28, 28, 1))
train_images = train_images.astype('float32') / 255
test_images = test_images.reshape((10000, 28, 28, 1))
test_images = test_images.astype('float32') / 255

train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)

model.compile(optimizer='rmsprop', loss='categorical_crossentropy',
metrics=['accuracy'])
model.fit(train_images, train_labels, epochs=5, batch_size=64)
```

```
Epoch 1/5
938/938 [==============================] - 17s 18ms/step - loss: 0.1786 - accu
racy: 0.9444
Epoch 2/5
938/938 [==============================] - 16s 17ms/step - loss: 0.0476 - accu
racy: 0.9854
Epoch 3/5
938/938 [==============================] - 16s 17ms/step - loss: 0.0337 - accu
racy: 0.9896
Epoch 4/5
938/938 [==============================] - 16s 17ms/step - loss: 0.0243 - accu
racy: 0.9931
Epoch 5/5
938/938 [==============================] - 16s 17ms/step - loss: 0.0197 - accu
racy: 0.9939
```

Out[13]:
```
<tensorflow.python.keras.callbacks.History at 0x7f8222eb0ee0>
```

MNIST compilation:

Categorical_crossentropy is the loss function that's used as a feedback signal for learning the weight tensors and which the training phase will attempt to minimize. This reduction of the loss happens via mini-batch stochastic gradient descent. The exact rules governing a specific use of gradient descent are defined by the rmsprop optimizer passed as the first argument.

MNIST training loop:

We need to call 'fit' then, the network will start to iterate on the training data in mini-batches of 64 samples, 5 times over (each iteration over all the training data is called an epoch). At each iteration, the network will compute the gradients of the weights with regard to the loss on the batch, and update the weights accordingly. After these 5 epochs, the network will

have performed 2,345 gradient updates (469 per epoch), and the loss of the network will be sufficiently low that the network will be capable of classifying handwritten digits with high accuracy.

In [14]:
```
#Let's evaluate the model on the test data
test_loss, test_acc = model.evaluate(test_images, test_labels)
test_acc
```

```
313/313 [==============================] - 1s 2ms/step - loss: 0.0295 - accura
cy: 0.9914
```

Out[14]:
```
0.9914000034332275
```

Test data evaluation

Upon the evaluation of the model on the test data, the results have improved. The densely connected network from chapter 2 had a test accuracy of 97.8%, whereas the basic convnet has a test accuracy of 99.1%.

The convolution operation

The fundamental difference between a densely connected layer and a convolution layer is: Dense layers learn global patterns in their input feature space (for example, for a MNIST digit, patterns involving all pixels), whereas convolution layers learn local patterns. In the case of images, patterns found in small 2D windows of the inputs. In the previous example, these windows were all 3 × 3.

*Images can be broken into local patterns such as edges, textures, and so on.

Key characteristic gives convnets two interesting properties:

1 - The patterns they learn are translation invariant. After learning a certain pattern in the lower-right corner of a picture, a convnet can recognize it anywhere. A densely connected network would have to learn the pattern anew if it appeared at a new location. This makes convnets data efficient when processing images (because the visual world is fundamentally translation invariant): they need fewer training samples to learn representations that have generalization power.

2 - They can learn spatial hierarchies of patterns. A first convolution layer will learn small local patterns such as edges, a second convolution layer will learn larger patterns made of the features of the first layers, and so on. This allows convnets to efficiently learn increasingly complex and abstract visual concepts.

Filters in convents

Convolutions operate over 3D tensors, called feature maps, with two spatial axes (height and width) as well as a depth axis (also called the channels axis).

For an RGB image, the dimension of the depth axis is 3, because the image has three color channels: red, green, and blue. For a black-and-white picture, like the MNIST digits, the depth is 1 (levels of gray).

The convolution operation extracts patches from its input feature map and applies the same transformation to all of these patches, producing an output feature map. This output feature map is still a 3D tensor: it has a width and a height. Its depth can be arbitrary, because the output depth is a parameter of the layer, and the different channels in that depth axis no longer stand for specific colors as in RGB input; rather, they stand for filters. Filters encode specific aspects of the input data: at a high level, a single filter could encode the concept “presence of a face in the input,” for instance.

Filters in MNIST

The first convolution layer takes a feature map of size (28, 28, 1) and outputs a feature map of size (26, 26, 32): it computes 32 filters over its input.

Each of these 32 output channels contains a 26 × 26 grid of values, which is a response map of the filter over the input, indicating the response of that filter pattern at different locations in the input.

That is what the term feature map means: every dimension in the depth axis is a feature (or filter), and the 2D tensor output[:, :, n] is the 2D spatial map of the response of this filter over the input.

Convolutions are defined by two key parameters:

The size of the patches extracted from the inputs: These are typically 3 × 3 or 5 × 5. Below, in section 1.2 ,I have experimented with both size of patches.

And the depth of the output feature map: The number of filters computed by the convolution. The example started with a depth of 32 and ended with a depth of 64.

These parameters are the first arguments passed to the layer:

Conv2D(output_depth, (window_height, window_width))

1.2 Experiment with moving/adding convolutional layers, and the number of filters and the size of the local receptive field. -Tests

Test #	Number of Convolutional layers	Input units/filter	Size local rec.field	Number of max pooling layers	Flatten	Dense layers	Validation Loss	Validation Accuracy(%)
1	**3**	32-64-64	3 x 3	2 of (2,2)	Yes	Yes, 64-10	0.1785	99.42 at epoch 5
2	**3**	32-64-64	5 x 5	2 of (2,2)	Yes	Yes, 64-10	0.1635	99.44 at epoch 5
3	**1**	32	3 x 3	1 of (2,2)	No	No	Test Failed	Test Failed
4	**2**	32-64	3 x 3	1 of (2,2)	Yes	Yes, 64-10	0.1365	99.56 at epoch 5
5	**5**	32-32-32-64-64	3 x 3	2 of (2,2)	Yes	Yes, 64-10	0.2042	99.25 at epoch 5

Test 1 is the default experiment from the book in Chapter 5.1, it shows a test accuracy of 99.4%. In test 2, I have experimented with the size of local receptive fields, where for the first two convolutional layers are of the size 5 x 5. A difference compared to test 1, was that the number of total parameters increased to 106,122, whereas for test 1 it was 93,322. Other than that there was not any visible affects in the accuracy percentage.

In test 3, I experimented by removing some convolutional layers. This network is composed of one convolutional layer and has max pooling added too. One of the main differences compared to test 1, was the total parameters which here it is just 320. As it's a very small network. This is not ideal as the model does not have enough data to train on. For learning purposes, I experimented by not adding Flatten() and not adding and Dense layers too. And it it came to train this bottleneck model on the MNIST images, it was not possible to do so, therefore this experiment failed.

In test 4, I attempted to 'fix' test 3. I added a convolutional layer more and added used fatten and dense layers too. After which, the total number of parameters was 515,146, which in comparison to test 1's was significantly more. But there were not visible affects in the accuracy.

In test 5, I experimented by adding some convolutional layers. I had 5 convolutional layers, out of which only 2 had max pooling on them. I had also added flatten and dense layers too. Although, this convent was larger, the total parameters was smaller than in test 1s. Here, the total parameters were 79,050. Also, the loss was much greater compared to the other tests above.

How convolution works:

A convolution works by sliding these windows of size 3 × 3 or 5 × 5 over the 3D input feature map, stopping at every possible location, and extracting the 3D patch of surrounding features (shape (window_height, window_width, input_depth)).

Each such 3D patch is then transformed (via a tensor product with the same learned weight matrix, called the convolution kernel) into a 1D vector of shape (output_depth,). All of these vectors are then spatially reassembled into a 3D output map of shape (height, width, output_depth).

Every spatial location in the output feature map corresponds to the same location in the input feature map (for example, the lower-right corner of the output contains information about the lower-right corner of the input). For instance, with 3 × 3 windows, the vector output [i, j, :] comes from the 3D patch input [i-1:i+1,j-1:j+1,:].

The output width and height may differ from the input width and height. They may differ for two reasons:

•Border effects, which can be countered by padding the input feature map.

•The use of strides, which I'll define in a second

The max-pooling operation

The role of max pooling is to aggressively downsample feature maps, much like strided convolutions.

Max pooling consists of extracting windows from the input feature maps and out- putting the max value of each channel. It's conceptually similar to convolution, except that instead of transforming local patches via a learned linear transformation (the convolution kernel), they're transformed via a hardcoded max tensor operation.

A big difference from convolution is that max pooling is usually done with 2 × 2 windows and stride 2, in order to downsample the feature maps by a factor of 2. On the other hand, convolution is typically done with 3 × 3 windows and no stride.

In [21]:
```
# Build a model without the max-pooling layers.
# The convolutional base of the model would then look like this:
model_no_max_pool = models.Sequential()
model_no_max_pool.add(layers.Conv2D(32, (3, 3), activation='relu', input_shap
model_no_max_pool.add(layers.Conv2D(64, (3, 3), activation='relu'))
model_no_max_pool.add(layers.Conv2D(64, (3, 3),activation='relu'))
```

In [22]:
```
model_no_max_pool.summary()
```

```
Model: "sequential_6"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_18 (Conv2D)           (None, 26, 26, 32)        320
_________________________________________________________________
conv2d_19 (Conv2D)           (None, 24, 24, 64)        18496
_________________________________________________________________
conv2d_20 (Conv2D)           (None, 22, 22, 64)        36928
=================================================================
Total params: 55,744
Trainable params: 55,744
Non-trainable params: 0
_________________________________________________________________
```

In [23]:
```
# Me adding : Listing 5.2 Adding a classifier on top of the convnet
model_no_max_pool.add(layers.Flatten())
model_no_max_pool.add(layers.Dense(64, activation='relu'))
model_no_max_pool.add(layers.Dense(10, activation='softmax'))
```

In [24]:
```
model_no_max_pool.summary()
```

```
Model: "sequential_6"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_18 (Conv2D)           (None, 26, 26, 32)        320
_________________________________________________________________
conv2d_19 (Conv2D)           (None, 24, 24, 64)        18496
_________________________________________________________________
conv2d_20 (Conv2D)           (None, 22, 22, 64)        36928
_________________________________________________________________
flatten_5 (Flatten)          (None, 30976)             0
_________________________________________________________________
dense_10 (Dense)             (None, 64)                1982528
_________________________________________________________________
dense_11 (Dense)             (None, 10)                650
=================================================================
Total params: 2,038,922
Trainable params: 2,038,922
Non-trainable params: 0
_________________________________________________________________
```

In [ ]:
```
# Me adding: Listing 5.3 Training the convnet on MNIST images
from tensorflow.keras.datasets import mnist
from tensorflow.keras.utils import to_categorical

(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
train_images = train_images.reshape((60000, 28, 28, 1))
train_images = train_images.astype('float32') / 255
test_images = test_images.reshape((10000, 28, 28, 1))
test_images = test_images.astype('float32') / 255

train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)

model_no_max_pool.compile(optimizer='rmsprop', loss='categorical_crossentropy
metrics=['accuracy'])

model_no_max_pool.fit(train_images, train_labels, epochs=5, batch_size=64)
```

1.3 Experiment with the removal of max-pooling layers - Test

Test #	Number of Convolutional layers	Input/units	Flatten	Dense layers	Validation Loss	Validation Accuracy(%)
6	3	32-64-64	Yes	Yes, 64-10	0.1207	0.9962 at epoch 5

A very similar model in section 1.2, test 1, with max-pooling layers had a total params: 93,322.

Whereas, without it, here in test 6, it has : Total params: 2,038,922. And it is not ideal as the number has doubled and it takes a lot of time to process too.

Apart from that there are other problems with this type of setup, such as:

1)It isn't conducive to learning a spatial hierarchy of features. The 3 × 3 windows in the third layer will only contain information coming from 7 × 7 windows in the initial input. The high-level patterns learned by the convnet will still be very small with regard to the initial input, which may not be enough to learn to classify digits (try recognizing a digit by only looking at it through windows that are 7 × 7 pixels). We need the features from the last convolution layer to contain information about the totality of the input.

2)The final feature map has 22 × 22 × 64 = 30,976 total coefficients per sample. This is huge. If you were to flatten it to stick a Dense layer of size 512 on top, that layer would have 15.8 million parameters. This is far too large for such a small model and would result in intense overfitting.

The reason to use downsampling is to reduce the number of feature-map coefficients to process, as well as to induce spatial-filter hierarchies by making successive convolution layers look at increasingly large windows.

*Max pooling isn't the only way you can achieve such downsampling. Strides can also be used in the prior convolution layer.And average pooling can be used instead of max pooling, where each local input patch is transformed by taking the average value of each channel over the patch, rather than the max. But max pooling tends to work better than these

alternative solutions. The reason is that features tend to encode the spatial presence of some pattern or concept over the different tiles of the feature map (hence, the term feature map), and it's more informative to look at the maximal presence of different features than at their average presence.

So the most reasonable subsampling strategy is to first produce dense maps of features (via unstrided convolutions) and then look at the maximal activation of the features over small patches, rather than looking at sparser windows of the inputs (via strided convolutions) or averaging input patches, which could cause you to miss or dilute feature-presence information.

1.4 - Evaluation of MNIST model

I have picked the book's model and evaluated it. The model has 3 convolutional layers with inputs of 32-64-64. It also has 2 max-pooling layers. Flatten has been used and two Dense layers have also been included. The validation accuracy was of 99.40%.

In [16]:
```
#----FINAL MNIST MODEL TEST-----:
from tensorflow.keras import models
from tensorflow.keras import layers

model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.Flatten())
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))

model.compile(optimizer='rmsprop', loss='categorical_crossentropy',metrics=['
model.fit(train_images, train_labels, epochs=5, batch_size=64)
results = model.evaluate(test_images, test_labels)
```

```
Epoch 1/5
938/938 [==============================] - 17s 18ms/step - loss: 0.1753 - accu
racy: 0.9452
Epoch 2/5
938/938 [==============================] - 16s 17ms/step - loss: 0.0475 - accu
racy: 0.9853
Epoch 3/5
938/938 [==============================] - 16s 18ms/step - loss: 0.0313 - accu
racy: 0.9905
Epoch 4/5
938/938 [==============================] - 16s 18ms/step - loss: 0.0245 - accu
racy: 0.9923
Epoch 5/5
938/938 [==============================] - 16s 18ms/step - loss: 0.0188 - accu
racy: 0.9941
313/313 [==============================] - 1s 2ms/step - loss: 0.0300 - accura
cy: 0.9909
```

In [17]:
```
# To check the results:
results
```

Out[17]:
```
[0.029977019876241684, 0.9908999800682068]
```

In [19]:
```
# Model's ability to recognise the digits
model.predict(test_images)
```

```
Out[19]: array([[8.52346785e-13, 3.39609718e-09, 4.94116970e-09, ...,
                 1.00000000e+00, 1.30131392e-11, 8.95525376e-10],
                [6.11505513e-09, 5.87404020e-06, 9.99994040e-01, ...,
                 1.16627645e-11, 1.20084143e-09, 7.86758083e-13],
                [1.51381269e-10, 9.99999881e-01, 1.97891019e-08, ...,
                 3.28198233e-08, 5.43064393e-09, 2.24376251e-09],
                ...,
                [1.07357978e-16, 7.16775350e-09, 9.46866023e-12, ...,
                 2.54887583e-10, 2.58044169e-10, 2.30219482e-10],
                [1.21193473e-08, 4.23171567e-11, 6.82684229e-14, ...,
                 2.09757402e-11, 7.47103695e-05, 3.46011136e-10],
                [2.23366854e-08, 2.73551515e-09, 4.11098910e-07, ...,
                 6.31227994e-14, 1.39453346e-06, 1.02359732e-10]], dtype=float32)
```

Evaluation of final model - test results and future improvements:

The validation accuracy for this model was 99.40%. The actual training accuracy is 99.04%, which is close to the validation accuracy. For this model, as per according to instructions I have followed the model of chapter 5.1 from the book 'Deep Learning with Python'. For future improvements, I would regularise the model hoping for a higher training accuracy.

LAB 5 PART 2 INSTRUCTIONS:

1. The Fashion MNIST dataset is a multiclass image classification task. Follow the MNIST pattern and download the dataset from Tensorflow.
2. Explore the data: what is the shape and size of the training and test data. Take one of the training images and display it with `matplotlib.pyplot`
3. Split the training set into a partial training set and a validation set. Reshape the partial training, validation and test sets and convert to floats in the range [0, 1].
4. Build a model similar to listing 5.1 and 5.2. Print a summary of the model.
5. Compile and run the model for 20 epochs. Plot graphs, find the optimum epochs, retrain a newly initialised model on the whole training set and then evaluate the model on the test test. Regularise your model if you see overfitting. What accuracy does your model achieve?
6. Have a look at the model's prediction on some test images by running this matplotlib code: https://colab.research.google.com/github/margaretmz/deep-learning/blob/master/fashion_mnist_keras.ipynb#scrollTo=oJv7XEk10bOv
7. Experiment with different convolutional bases. For example, remove pooling, add more convolutional layers, resize the layers etc.

2 Fashion MNIST

Fashion-MNIST is a dataset consisting of a training set of 60,000 examples and a test set of 10,000 examples. Each example is a 28×28 grayscale image, associated with a label from 10 classes. Fashion-MNIST is intended to serve as a direct drop-in replacement of the original MNIST dataset for benchmarking machine learning algorithms.

Fashion-MNIST is intended to serve as a direct drop-in replacement for the original MNIST dataset to benchmark machine learning algorithms, as it shares the same image size and the structure of training and testing splits.

The Fashion-MNIST dataset was created as a replacement for the MNIST dataset because:

1- MNIST is too easy. Convolutional nets can achieve 99.7% on MNIST. Classic machine learning algorithms can also achieve 97% easily.

2- MNIST is overused.

3- MNIST cannot represent modern computer vision tasks.

2.1 Download the fashion_mnist dataset

In [28]:
```
# TensorFlow and tf.keras
import tensorflow as tf

# Helper libraries
import numpy as np
import matplotlib.pyplot as plt

print(tf.__version__)
```

```
2.3.1
```

In [29]:
```
# Following the MNIST pattern and downloaded the dataset from Tensorflow.
fashion_mnist = tf.keras.datasets.fashion_mnist
(p_train, q_train), (p_test, q_test) = fashion_mnist.load_data()
```

2.2 Explore the dataset

In [30]:
```
#Explore the data: what is the shape and size of the training and test data.
# Name the labels
class_names = ['T-shirt/top',
               'Trouser',
               'Pullover',
               'Dress',
               'Coat',
               'Sandal',
               'Shirt',
               'Sneaker',
               'Bag',
               'Ankle boot']

# Print the  shape and size of the training and test data.
# print("Train set size:", p_train.size, "Test set size:", q_train.size)
print(p_train.shape, 'train set shape')
print(q_test.shape, 'test set shape')
```

```
(60000, 28, 28) train set shape
(10000,) test set shape
```

The following shows there are 60,000 images in the training set, with each image represented as 28 x 28 pixels. Each label is an integer between 0 and 9, from the 60,000 images in the train set. There are 10,000 images in the test set.

In [31]:
```
print("Dimension" , p_train.ndim)
print("Data type",p_train.dtype)
```

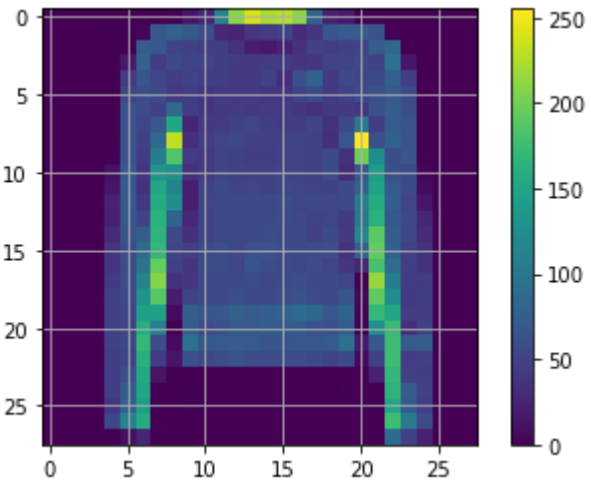
```
Dimension 3
```

```
Data type uint8
```

These images are 3D tensors of shape(samples,height,width)

2.3 Visualize an image from the dataset

In [32]:
```
# Took one of the trainig images and display it with matplotlib.pyplot
plt.figure()
#Below I can choose to display any picture from the set from [0-59999]
plt.imshow(p_train[5670])
plt.colorbar()
plt.grid(True)
plt.show()
```



2.4 Data normalization

In [33]:
```
#Split the training set into a partial training set and a validation set.
p_train = p_train.astype('float32') / 255
p_test = p_test.astype('float32') / 255
print("Normalize the data dimensions so that they are of approximately the sa
```

```
Normalize the data dimensions so that they are of approximately the same scal
e.
```

In [34]:
```
print("Number of train data: " + str(len(p_train)))
print("Number of test data: " + str(len(p_test)))
```

```
Number of train data: 60000
Number of test data: 10000
```

2.5 Split the data into training, validation and testing data sets

In [35]:
```
#Reshape the partial training, validation and test sets and convert to floats

(p_train, p_valid) = p_train[5000:], p_train[:5000]
(q_train, q_valid) = q_train[5000:], q_train[:5000]

# Reshape input data from (28, 28) to (28, 28, 1)
w, h = 28, 28
p_train = p_train.reshape(p_train.shape[0], w, h, 1)
p_valid = p_valid.reshape(p_valid.shape[0], w, h, 1)
p_test = p_test.reshape(p_test.shape[0], w, h, 1)

# One-hot encode the labels
q_train = tf.keras.utils.to_categorical(q_train, 10)
```

```
q_valid = tf.keras.utils.to_categorical(q_valid, 10)
q_test = tf.keras.utils.to_categorical(q_test, 10)

# Print training set shape
print("p_train shape:", p_train.shape, ", q_train shape:", q_train.shape)

# Print the number of training, validation, and test datasets
print(p_train.shape, 'train set')
print(p_valid.shape, 'validation set')
print(p_test.shape, 'test set')
```

```
p_train shape: (55000, 28, 28, 1) , q_train shape: (55000, 10)
(55000, 28, 28, 1) train set
(5000, 28, 28, 1) validation set
(10000, 28, 28, 1) test set
```

2.6 Build a model similar to listing 5.1 and 5.2. Print a summary of the model.

In [36]:
```
# Build a model similar to listing book sections 5.1 and 5.2. Print a summary
from tensorflow.keras import layers
from tensorflow.keras import models

model1 = models.Sequential()
model1.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28,
model1.add(layers.MaxPooling2D((2, 2)))
model1.add(tf.keras.layers.Dropout(0.3))
model1.add(layers.Conv2D(64, (3, 3), activation='relu'))
model1.add(layers.MaxPooling2D((2, 2)))
model1.add(tf.keras.layers.Dropout(0.3))
model1.add(layers.Conv2D(64, (3, 3), activation='relu'))

model1.add(layers.Flatten())
# model1.add(tf.keras.layers.Dropout(0.3))
model1.add(layers.Dense(128, activation='relu'))
model1.add(layers.Dense(10, activation='softmax'))
```

In [37]:
```
model1.summary()
```

```
Model: "sequential_8"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_24 (Conv2D)           (None, 26, 26, 32)        320
_________________________________________________________________
max_pooling2d_12 (MaxPooling (None, 13, 13, 32)        0
_________________________________________________________________
dropout_1 (Dropout)          (None, 13, 13, 32)        0
_________________________________________________________________
conv2d_25 (Conv2D)           (None, 11, 11, 64)        18496
_________________________________________________________________
max_pooling2d_13 (MaxPooling (None, 5, 5, 64)          0
_________________________________________________________________
dropout_2 (Dropout)          (None, 5, 5, 64)          0
_________________________________________________________________
conv2d_26 (Conv2D)           (None, 3, 3, 64)          36928
_________________________________________________________________
flatten_7 (Flatten)          (None, 576)               0
_________________________________________________________________
dense_14 (Dense)             (None, 128)               73856
_________________________________________________________________
dense_15 (Dense)             (None, 10)                1290
=================================================================
Total params: 130,890
Trainable params: 130,890
```

```
Non-trainable params: 0
_________________________________________________________________
```

Model summary with convolutional and Dense layers

In the model's summary above, the output of every Conv2D and MaxPooling2D layer is a 3D tensor of shape (height, width, channels). The width and height dimensions tends to shrink as you go deeper in the network. The number of channels is controlled by the first argument passed to the Conv2D layers (32 or 64).

The next step is to feed the last output tensor (of shape (3, 3, 64)) into a densely connected classifier network by using a stack of Dense layers. These classifiers process vectors, which are 1D, whereas the current output is a 3D tensor. First we have to flatten the 3D outputs to 1D, and then add a few Dense layers on top.

On the model summary above the (3, 3, 64) outputs are flattened into vectors of shape (576,) before going through two Dense layers. Now, the convnet has to be trained on the Fashion_MNIST data.

In the section below 2.11, different variations of this model have been implemented and regularised where necessary for experimental and learning purposes.

2.7 Compile and run the model

To prepare the model for training, three main components have to be set. These are added during the model's compile step:

Loss function: This measures the accuracy of the model is during the training stage. The aim is to minimize this function to "steer" the model in the right direction. The model is a multiclass, single-label classification. The loss function therefore is going to be categorical_crossentropy.

Optimizer: It implements what's called the Backpropagation algorithm. The mechanism through which the network will update itself based on the data it sees and its loss function. For this model, I will be using RMSProp optimizer.

Metrics: Used to monitor the training and testing steps. This model will use accuracy, which will show the fraction of the images that are correctly classified. The hypothesis for model is that the outputs can be predicted given the inputs. And it is because the available data is sufficiently informative to learn the relationship between inputs and outputs. The validation accuracy of the model will be used to measure the success rate.

The evaluation protocol for this model will be the hold-out validation. In hold-out validation, a fraction of the data is set aside as the test set. The remaining data set is trained and then evaluated on the test set.This type of evaluation protocol validation is not suitable with small datasets. But for the Fashion_MNIST it's the appropriate validation type to use, as this dataset is quite large.

In [38]:
```
model1.compile(optimizer='rmsprop',
               loss='categorical_crossentropy',
               metrics=['accuracy'])
```

In [40]:
```
# Compile and run the model for 20 epochs.
history = model1.fit(p_train, q_train, batch_size=64, epochs=20, validation_d
```

```
Epoch 1/20
860/860 [==============================] - 19s 22ms/step - loss: 0.6127 - accu
racy: 0.7689 - val_loss: 0.4240 - val_accuracy: 0.8420
Epoch 2/20
860/860 [==============================] - 19s 22ms/step - loss: 0.3940 - accu
racy: 0.8548 - val_loss: 0.3347 - val_accuracy: 0.8736
Epoch 3/20
860/860 [==============================] - 18s 21ms/step - loss: 0.3363 - accu
racy: 0.8775 - val_loss: 0.3536 - val_accuracy: 0.8598
Epoch 4/20
860/860 [==============================] - 18s 21ms/step - loss: 0.3079 - accu
racy: 0.8863 - val_loss: 0.2734 - val_accuracy: 0.8968
Epoch 5/20
860/860 [==============================] - 19s 22ms/step - loss: 0.2912 - accu
racy: 0.8913 - val_loss: 0.2582 - val_accuracy: 0.9032
Epoch 6/20
860/860 [==============================] - 19s 22ms/step - loss: 0.2782 - accu
racy: 0.8974 - val_loss: 0.2810 - val_accuracy: 0.8938
Epoch 7/20
860/860 [==============================] - 19s 22ms/step - loss: 0.2688 - accu
racy: 0.9005 - val_loss: 0.2395 - val_accuracy: 0.9108
Epoch 8/20
860/860 [==============================] - 19s 22ms/step - loss: 0.2600 - accu
racy: 0.9036 - val_loss: 0.2560 - val_accuracy: 0.9044
Epoch 9/20
860/860 [==============================] - 18s 21ms/step - loss: 0.2575 - accu
racy: 0.9063 - val_loss: 0.2529 - val_accuracy: 0.9110
Epoch 10/20
860/860 [==============================] - 17s 20ms/step - loss: 0.2528 - accu
racy: 0.9079 - val_loss: 0.2406 - val_accuracy: 0.9126
Epoch 11/20
860/860 [==============================] - 17s 20ms/step - loss: 0.2503 - accu
racy: 0.9070 - val_loss: 0.2431 - val_accuracy: 0.9120
Epoch 12/20
860/860 [==============================] - 17s 20ms/step - loss: 0.2480 - accu
racy: 0.9101 - val_loss: 0.2523 - val_accuracy: 0.9088
Epoch 13/20
860/860 [==============================] - 17s 20ms/step - loss: 0.2502 - accu
racy: 0.9090 - val_loss: 0.2394 - val_accuracy: 0.9146
Epoch 14/20
860/860 [==============================] - 17s 20ms/step - loss: 0.2511 - accu
racy: 0.9106 - val_loss: 0.2340 - val_accuracy: 0.9176
Epoch 15/20
860/860 [==============================] - 17s 20ms/step - loss: 0.2495 - accu
racy: 0.9096 - val_loss: 0.2432 - val_accuracy: 0.9144
Epoch 16/20
860/860 [==============================] - 17s 20ms/step - loss: 0.2530 - accu
racy: 0.9097 - val_loss: 0.2622 - val_accuracy: 0.9046
Epoch 17/20
860/860 [==============================] - 17s 20ms/step - loss: 0.2497 - accu
racy: 0.9103 - val_loss: 0.2467 - val_accuracy: 0.9088
Epoch 18/20
860/860 [==============================] - 17s 20ms/step - loss: 0.2502 - accu
racy: 0.9098 - val_loss: 0.2493 - val_accuracy: 0.9130
Epoch 19/20
860/860 [==============================] - 17s 20ms/step - loss: 0.2491 - accu
racy: 0.9101 - val_loss: 0.3413 - val_accuracy: 0.8812
Epoch 20/20
860/860 [==============================] - 17s 20ms/step - loss: 0.2506 - accu
racy: 0.9097 - val_loss: 0.2761 - val_accuracy: 0.9030
```

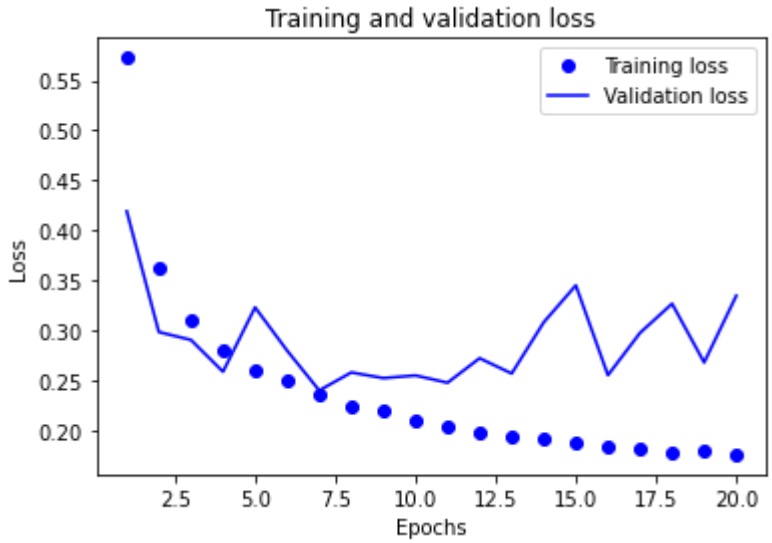
Save the model

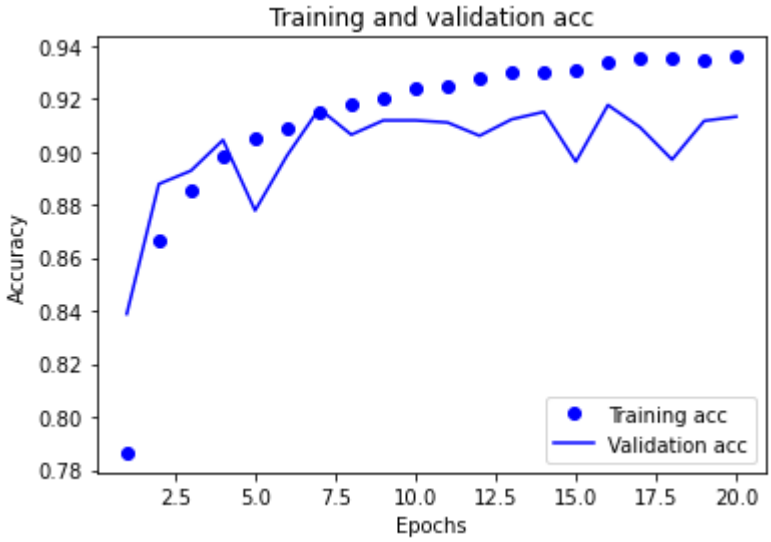
In [49]:
```
#I will not be submitting any saved models, here I just tested it for learnin
# model.save('fash_mnist7.h5')
```

2.8 Plot the graphs

In [129…

```
import matplotlib.pyplot as plt
def plot_loss():

    history_dict = history.history
    loss = history_dict['loss']
    val_loss = history_dict['val_loss']

    epochs = range(1, len(loss) + 1)

    blue_dots = 'bo'
    solid_blue_line = 'b'

    plt.plot(epochs, loss, blue_dots, label = 'Training loss')
    plt.plot(epochs, val_loss, solid_blue_line, label = 'Validation loss')
    plt.title('Training and validation loss')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.legend()
#     plt.savefig('7c.png', format = 'png')
    plt.show()
def plot_acc():

    plt.clf()
    history_dict = history.history
    acc = history_dict['accuracy']
    val_acc = history_dict['val_accuracy']

    epochs = range(1, len(acc) + 1)

    plt.plot(epochs, acc, 'bo', label = 'Training acc')
    plt.plot(epochs, val_acc, 'b', label = 'Validation acc')
    plt.title('Training and validation acc')
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy')
    plt.legend()
#     plt.savefig('7d.png', format = 'png')
    plt.show()

plot_loss()
plot_acc()
```





Load Model with the best validation accuracy

In [51]:
```
# Load the weights with the best validation accuracy
# model.load_weights('fash_mnist7.h5')
```

2.9 Test Accuracy

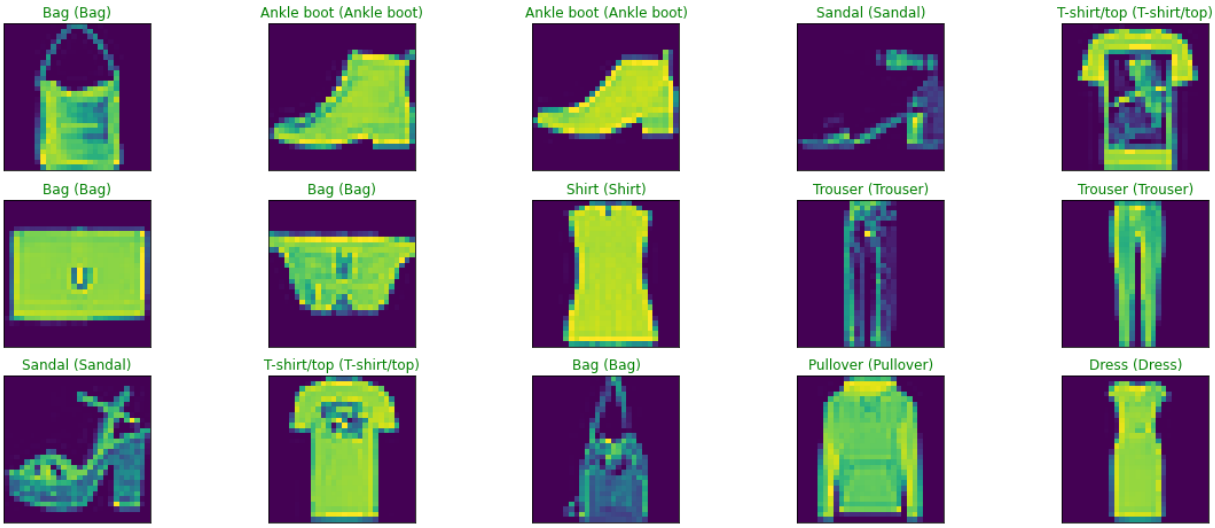
In [130…
```
# Evaluate the model on test set
score = model.evaluate(p_test, q_test, verbose=0)

# Print test accuracy
print('\n', 'Test accuracy:', score[1])
```

```
Test accuracy: 0.9035000205039978
```

The test accuracy for every model has been 90% and higher which is good compared to other model's accuracy which does not used convents like the IMDB model in chapter 3 of the 'Deep Learning with Python' book which had an accuracy of 88%.

2.10 Model's prediction on some test images

In [131…
```
q_hat = model.predict(p_test)

# Plot a random sample of 10 test images, their predicted labels and ground t
figure = plt.figure(figsize=(20, 8))
for i, index in enumerate(np.random.choice(p_test.shape[0], size=15, replace=
    ax = figure.add_subplot(3, 5, i + 1, xticks=[], yticks=[])
    # Display each image
    ax.imshow(np.squeeze(p_test[index]))
    predict_index = np.argmax(q_hat[index])
    true_index = np.argmax(q_test[index])
    # Set the title for each image
    ax.set_title("{} ({})".format(class_names[predict_index],
                                  class_names[true_index]),
                                  color=("green" if predict_index == true_ind
```

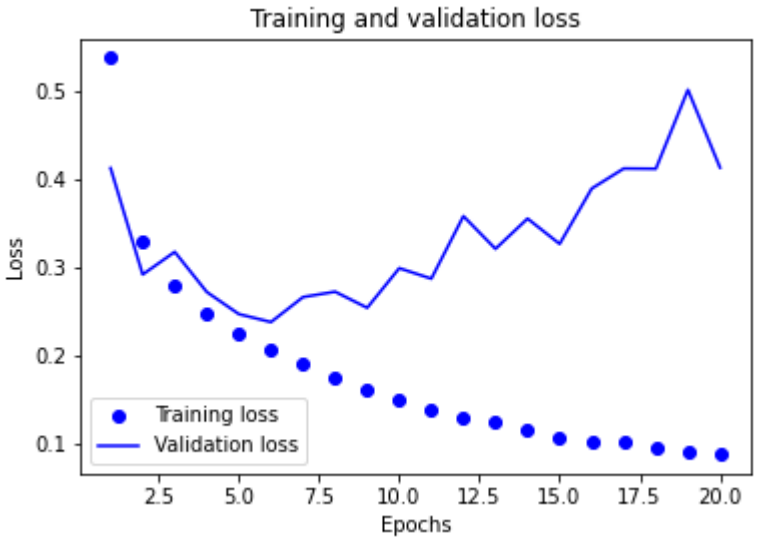


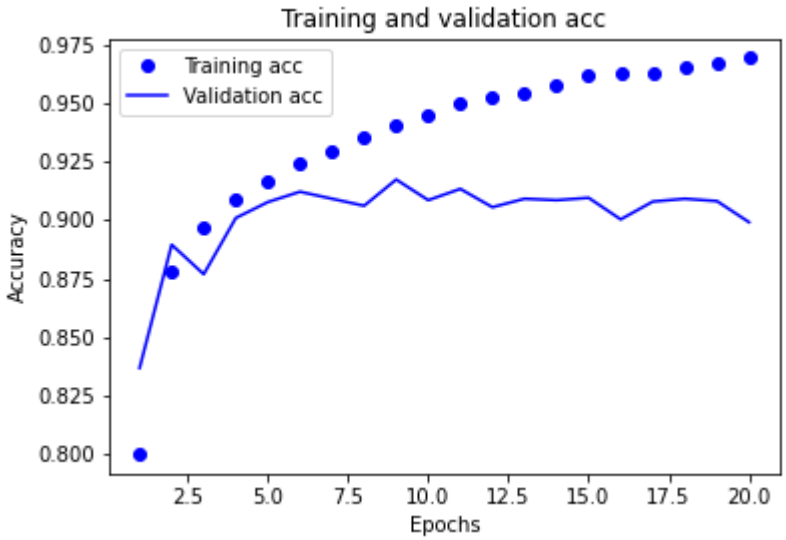
The model is successfully able to predict on some test images.

2.11 Experiments and results table - Tests

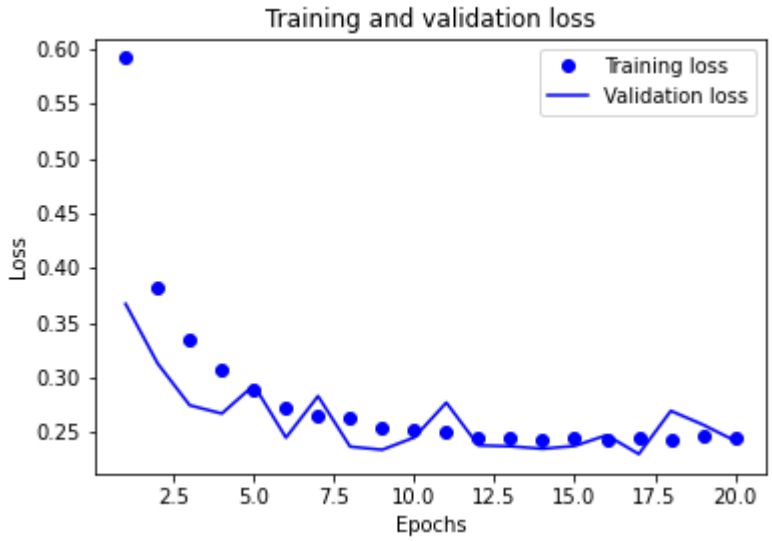
Test #	Number of Convolutional layers	Input units/filter	Size local rec.field	Regularised	Number of max pooling layers	Flatten	Dense layers	Validation Loss	Va Acc
7	**3**	32-64-64	3 x 3	No	2 of (2,2)	Yes	Yes, 128-10	0.5026	91.7 epc
7.1	**3**	32-64-64	3 x 3	Yes, 2 droput 0.3	2 of (2,2)	Yes	Yes, 128-10	0.3673	91.4 epc
7.2	**3**	32-64-64	3 x 3	Yes, 1 droput 0.3 after flatten	2 of (2,2)	Yes	Yes, 128-10	0.4194	91.5 epc
8	**3**	32-64-64	3 x 3	No	No	Yes	Yes, 128-10	1.1800	92.1 epc
9	**1**	32	5 x 5	No	1 of (2,2)	Yes	Yes, 64-10	0.3267	92.1 epc
9.1	**1**	32	5 x 5	Yes, droput 0.3	1 of (2,2)	Yes	Yes, 64-10	0.3436	92.1 epc

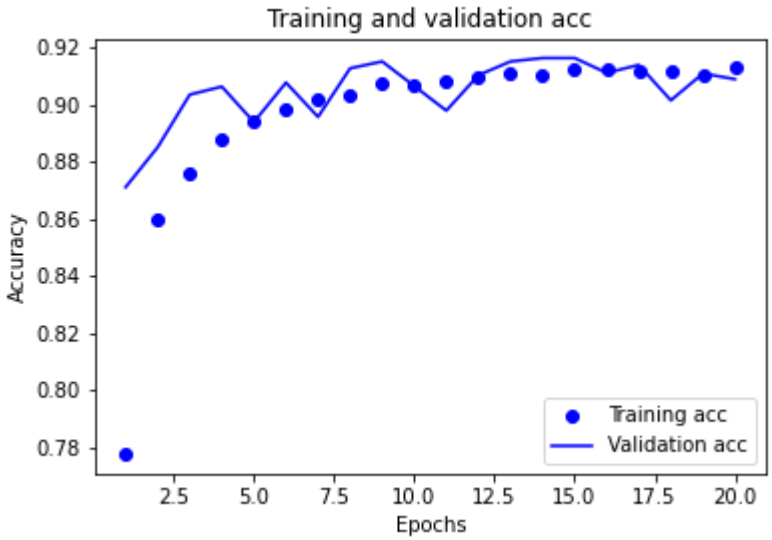
In test 7, the model has 130,890 trainable parameters. I did not apply any regularisation, and the model was overfitted. This model had a test accuracy of 89.25%.



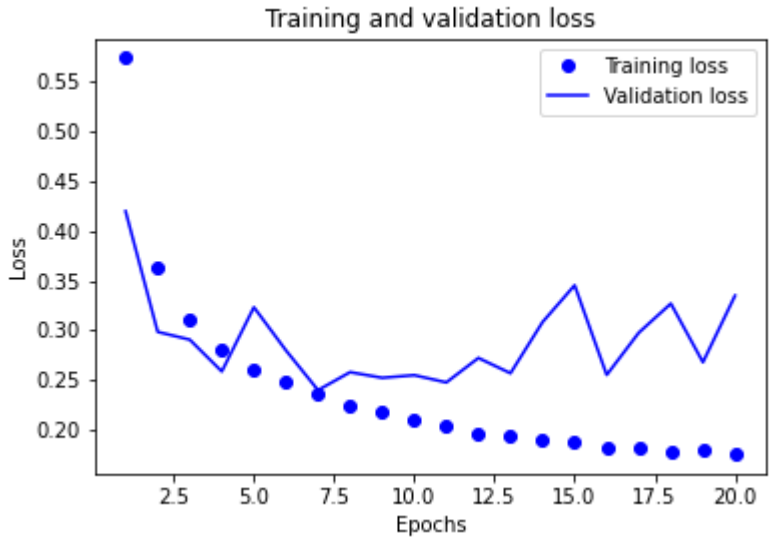


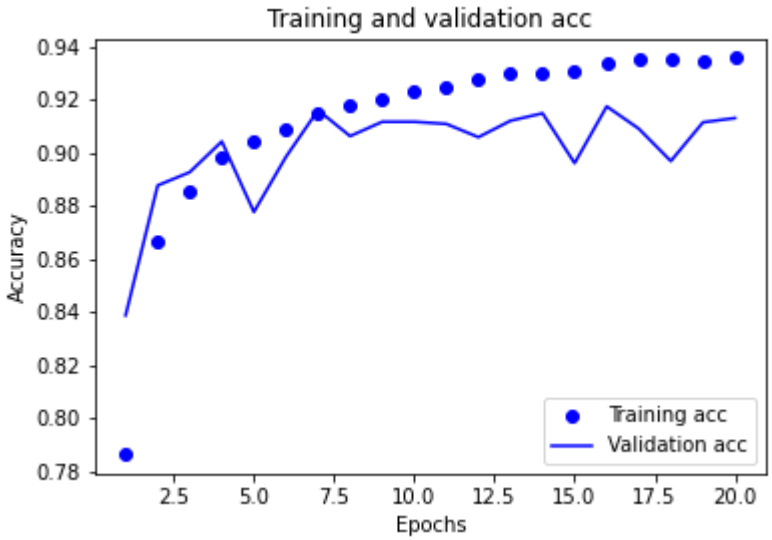
In test 7.1, the aim was to regularise test 7, therefore 2 dropouts was added after the convolutional layers. The model did become a little underfitted. The model has 130,890 trainable parameters.This model's test accuracy improved to 90.26%.



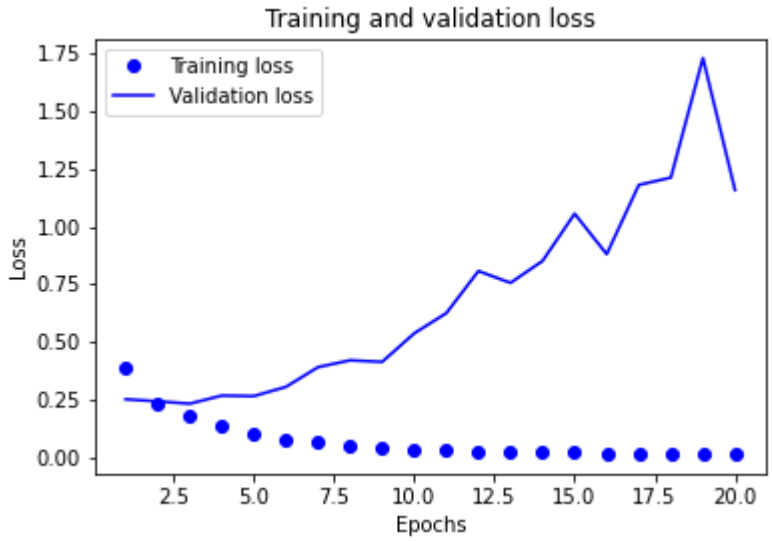


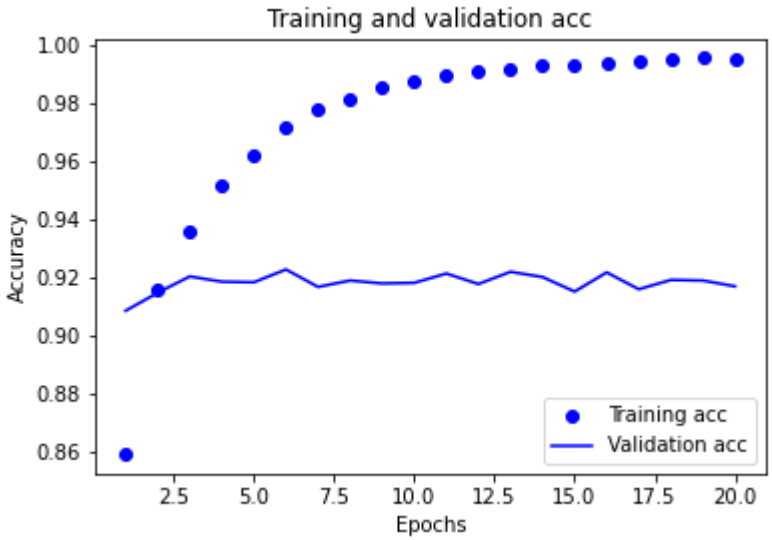
In test 7.2, the aim was to regularise test 7.1, therefore a dropout was added after flatten. The model is regularised .The model has 130,890 trainable parameters. This model's test accuracy improved to 90.35%.



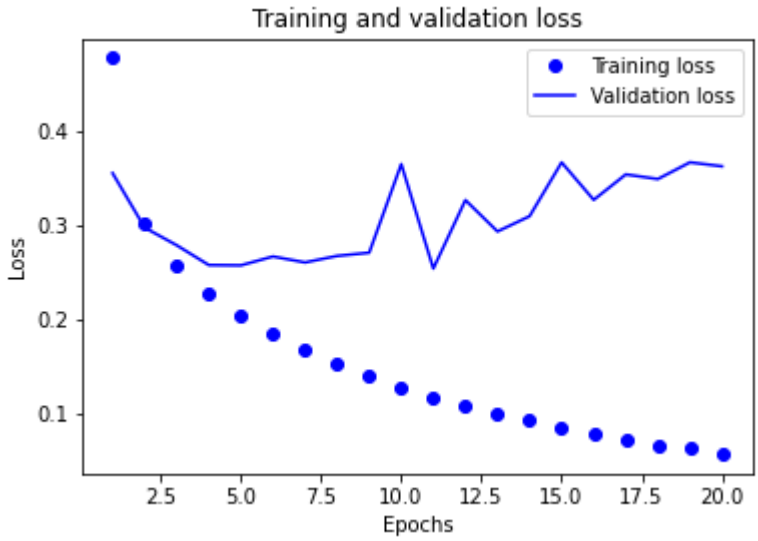


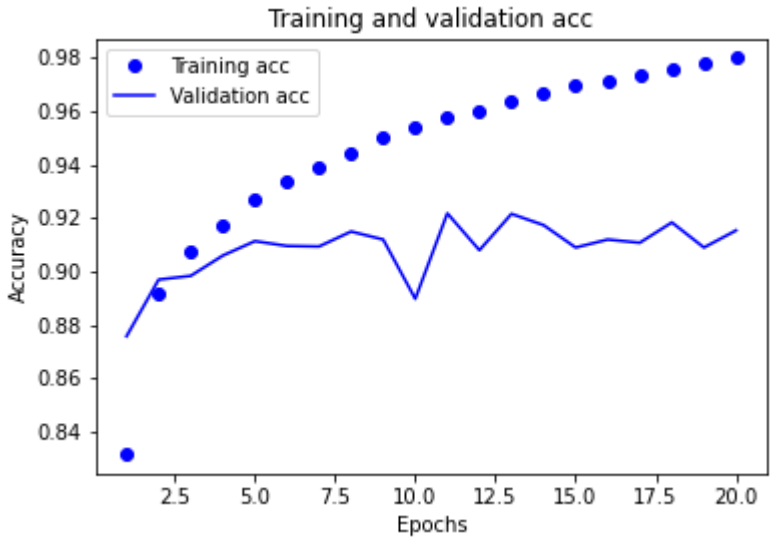
In test 8, I have experimented with by removing max-pooling. The model had 4,022,090 parameters, which is considerably much more than when I added max-pooling. Apart from the long time it to run, the results weren't good either. The accuracy level has higher than other tests, but it was a bit overfitted. And the loss has very high too.



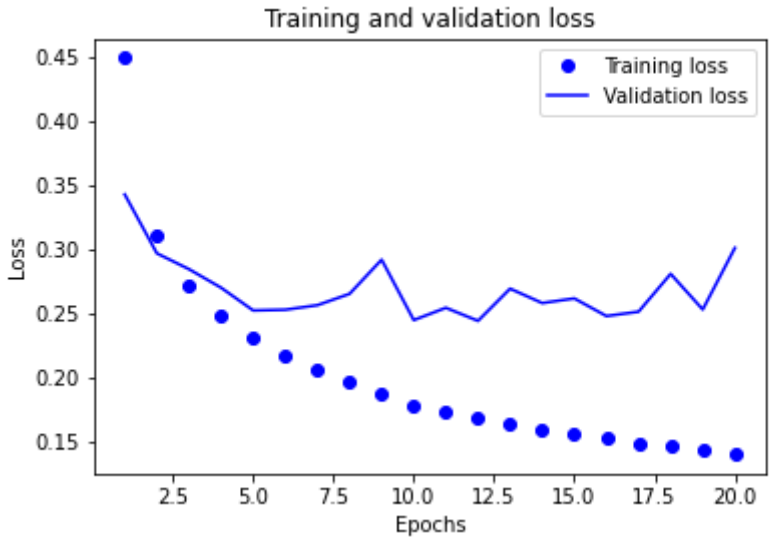


In test 9 I have experimented with removing convolutional layers and resized the dense layers too. This model has 347,146 parameters, which is much more than in test 7.1. The model is slightly overfitted, I want to try regularising it on the next test 9.1.

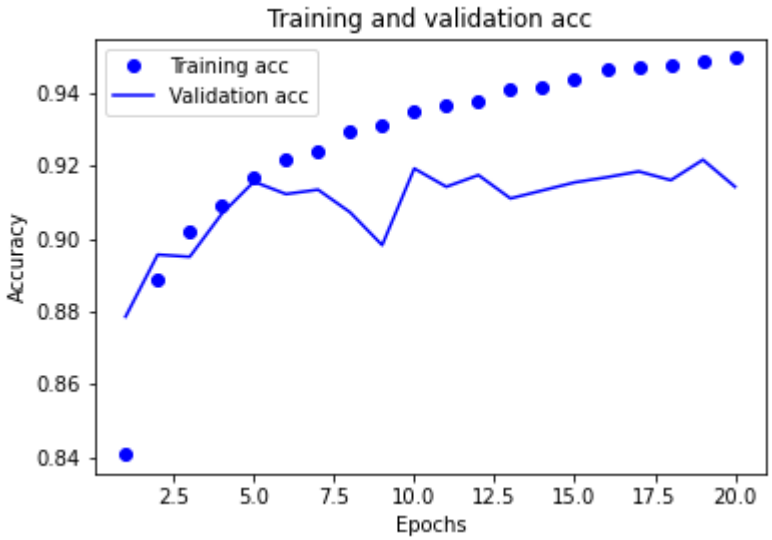




Below are the results after regularising test 9. It has been slightly regularised. This model has a test accuracy of 91.47%. This model has 347,146 parameters, which is much more than in



test 7.1.



2.12 - Evaluation of FASHION MNIST model

In [42]:
```
# ----FINAL FASHION MNIST MODEL EVALUATION-----:
from tensorflow.keras import models
from tensorflow.keras import layers
from tensorflow.keras import regularizers

model1 = models.Sequential()
model1.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28,
model1.add(layers.MaxPooling2D((2, 2)))
model.add(tf.keras.layers.Dropout(0.3))
model1.add(layers.Conv2D(64, (3, 3), activation='relu'))
model1.add(layers.MaxPooling2D((2, 2)))
model.add(tf.keras.layers.Dropout(0.3))
model1.add(layers.Conv2D(64, (3, 3), activation='relu'))
model1.add(layers.Flatten())
model1.add(layers.Dense(128, activation='relu'))
model1.add(layers.Dense(10, activation='softmax'))
model1.compile(optimizer='rmsprop',
               loss='categorical_crossentropy',
               metrics=['accuracy'])
model1.fit(p_train, q_train, batch_size=64, epochs=15)
results = model1.evaluate(p_test, q_test)
```

```
Epoch 1/15
860/860 [==============================] - 16s 18ms/step - loss: 0.5428 - accu
racy: 0.7985
```

```
Epoch 2/15
860/860 [==============================] - 16s 18ms/step - loss: 0.3324 - accu
racy: 0.8769
Epoch 3/15
860/860 [==============================] - 16s 18ms/step - loss: 0.2804 - accu
racy: 0.8973
Epoch 4/15
860/860 [==============================] - 15s 18ms/step - loss: 0.2495 - accu
racy: 0.9075
Epoch 5/15
860/860 [==============================] - 16s 18ms/step - loss: 0.2254 - accu
racy: 0.9187
Epoch 6/15
860/860 [==============================] - 16s 18ms/step - loss: 0.2048 - accu
racy: 0.9245
Epoch 7/15
860/860 [==============================] - 16s 18ms/step - loss: 0.1874 - accu
racy: 0.9302
Epoch 8/15
860/860 [==============================] - 16s 18ms/step - loss: 0.1728 - accu
racy: 0.9359
Epoch 9/15
860/860 [==============================] - 16s 18ms/step - loss: 0.1617 - accu
racy: 0.9409
Epoch 10/15
860/860 [==============================] - 16s 18ms/step - loss: 0.1488 - accu
racy: 0.9444
Epoch 11/15
860/860 [==============================] - 16s 18ms/step - loss: 0.1379 - accu
racy: 0.9486
Epoch 12/15
860/860 [==============================] - 16s 18ms/step - loss: 0.1313 - accu
racy: 0.9511
Epoch 13/15
860/860 [==============================] - 15s 18ms/step - loss: 0.1218 - accu
racy: 0.9547
Epoch 14/15
860/860 [==============================] - 16s 18ms/step - loss: 0.1157 - accu
racy: 0.9567
Epoch 15/15
860/860 [==============================] - 16s 18ms/step - loss: 0.1099 - accu
racy: 0.9590
313/313 [==============================] - 1s 2ms/step - loss: 0.3573 - accura
cy: 0.9070
```

In [43]:
```
# To check the results:
results
```

Out[43]:
```
[0.35726428031921387, 0.9070000052452087]
```

In [44]:
```
# Model ability to recognise the items
model1.predict(p_test)
```

Out[44]:
```
array([[9.9549490e-21, 4.9383859e-22, 1.0299471e-21, ..., 4.6117524e-12,
        1.1991853e-19, 1.0000000e+00],
       [4.3576094e-07, 1.1669768e-19, 9.9999952e-01, ..., 3.6316651e-23,
        2.1162966e-15, 1.0239427e-18],
       [1.9626175e-21, 1.0000000e+00, 2.1208481e-22, ..., 8.6734849e-30,
        4.9246693e-23, 8.1707878e-33],
       ...,
       [2.2500833e-21, 0.0000000e+00, 2.3110305e-20, ..., 6.5679511e-23,
        1.0000000e+00, 2.5542740e-28],
       [8.0414074e-19, 1.0000000e+00, 6.7912977e-22, ..., 1.3984447e-24,
        2.6431889e-20, 5.5031012e-28],
       [3.6655931e-07, 3.1536936e-19, 3.3183552e-13, ..., 1.8835227e-06,
        4.5369113e-08, 1.0761685e-09]], dtype=float32)
```

The validation accuracy for this model was 91%. The actual training accuracy is 90.70 which

isn't too far from the validation accuracy. As a conclusion on experimenting with convents on different datasets some of the key observations have been that:

Convnets are the best tool for attacking visual-classification problems compared to just using Dense layers alone.

Convnets work by learning a hierarchy of modular patterns and concepts to represent the visual world.

The representations they learn are easy to inspect—convnets are the opposite of black boxes, which I experienced whilst training the MNIST dataset.

Also that training my own convnet from scratch to solve an image-classification problem is not too complicated, as I experienced it first-hand whilst training the Fashion MNIST dataset.

For future improvements to increase the accuracy, I'd like to use other regularisation methods apart from dropout, in particular data augmentation.