TensorFlow tutorial_COMPLETED

March 5, 2021

1 Methods for Data Science

1.0.1 Deep Learning / Neural Networks and TensorFlow

1.1 Contents

Introduction

Welcome to the deep learning / neural networks section of the Methods for Data Science module!

In this section of the course, you will learn the fundamentals of deep learning models, as well as techniques for how to train, regularise and validate them.

We will cover widespread deep learning architectures such as the multilayer perceptron (MLP) and convolutional neural network (CNN), with a focus on understanding the mathematical operations and transformations included in these models. We will also look at several popular network optimisation algorithms, as well as the important error backpropagation algorithm, which is central to the training of neural networks. Regularisation techniques covered are weight regularisation, early stopping, and dropout.

The video content for this material is split into two types. There are standard 'lecture-style' videos, where the core material and theory behind deep learning models is presented, and then there are 'coding tutorial' videos, where you will learn to implement these concepts and ideas in the deep learning framework TensorFlow.

TensorFlow is an open source software library used for machine learning applications, especially deep learning. It uses symbolic mathematics (instead of purely numerical computations), which enables it to perform operations like automatic differentiation on a computational graph such as a neural network. Another major benefit is its ability to perform computations on GPU hardware, potentially leading to large speedups.

This notebook contains many blank code cells in the sections listed above. The coding tutorial videos will step through the different parts of the TensorFlow library, and show you how to fill in these code cells. The idea is that you should follow along with these videos and code in all the examples yourself. This way, you will gain familiarity in how to use TensorFlow, and you should feel free to pause the video and try things out for yourself to gain a deeper understanding.

Throughout these coding tutorials, it is a good idea to use the documentation as a regular reference for the various functions and classes that we will be looking at.

You will be able to run this notebook and follow the examples from the coding tutorial videos within the Anaconda environment you have installed for TensorFlow.

TensorFlow Tensors and Variables

In this section we will introduce some fundamental building blocks and operations in TensorFlow. Tensors and Variables are low-level objects that we will be using all the time in TensorFlow.

Tensors You can think of Tensors as being multidimensional versions of vectors and arrays. Of course, these are the objects that Tensorflow gets its name from. When we build our neural network models, what we're doing is defining a computational graph, where input data is processed through the layers of the network and sent through the graph all the way to the outputs. Tensors are the objects that get passed around within the graph, and capture those computations within the graph.

Let's take a look at some examples to get a better feel for how this works.

[1]:
```
import tensorflow as tf
```

[2]:
```
# Create a constant Tensor

a = tf.constant([1, 2, 3])
print(a)
```

```
tf.Tensor([1 2 3], shape=(3,), dtype=int32)
```

We can see that Tensors have `shape` and `dtype` properties, similar to NumPy arrays.

[3]:
```
# Examine shape property

a.shape
```

[3]:
```
TensorShape([3])
```

[4]:
```
# Examine dtype property

a.dtype
```

[4]:
```
tf.int32
```

Tensor objects can have different types, just like NumPy arrays. Take a look here for a complete list of available types.

[5]:
```
# Create Tensor objects of different type

string_tensor = tf.constant(["Hello world!"], tf.string)
float_tensor  = tf.constant([3.14159, 2.71828], tf.float32)
print(string_tensor)
print(float_tensor)
```

```
tf.Tensor([b'Hello world!'], shape=(1,), dtype=string)
tf.Tensor([3.14159 2.71828], shape=(2,), dtype=float32)
```

[6]:
```
# Create a rank-2 Tensor

b = tf.constant([[1.2, 0.4, 0.7], [-9.3, 4.5, 1.1]])
b
```

[6]:
```
<tf.Tensor: shape=(2, 3), dtype=float32, numpy=
array([[ 1.2,  0.4,  0.7],
       [-9.3,  4.5,  1.1]], dtype=float32)>
```

[7]:
```
# Get Tensor rank

tf.rank(b)
```

[7]:
```
<tf.Tensor: shape=(), dtype=int32, numpy=2>
```

[8]:
```
# Create a Tensor with tf.ones

tf.ones((2, 3))
```

[8]:
```
<tf.Tensor: shape=(2, 3), dtype=float32, numpy=
array([[1., 1., 1.],
       [1., 1., 1.]], dtype=float32)>
```

[9]:
```
# Create a Tensor with tf.zeros

tf.zeros((3,))
```

[9]:
```
<tf.Tensor: shape=(3,), dtype=float32, numpy=array([0., 0., 0.], dtype=float32)>
```

We can convert a TensorFlow Tensor into a NumPy array using the `numpy` method.

[10]:
```
# Convert Tensor to NumPy array

b_np = b.numpy()
print(type(b_np))
b_np
```

```
<class 'numpy.ndarray'>
```

[10]:
```
array([[ 1.2,  0.4,  0.7],
       [-9.3,  4.5,  1.1]], dtype=float32)
```

We can compute Tensor multiplication using `tf.tensordot` (see the docs). The `axes` argument can be an integer or list of integers. When it is a single integer `n`, the contraction is performed over the last `n` axes of the first Tensor and the first `n` axes of the second Tensor. If it is a list, then the elements of the list specify the axes to contract.

[11]:
```python
# Compute matrix-vector product

# tf.tensordot(b, a, axes=1)  # Type error

a = tf.cast(a, tf.float32)
tf.tensordot(b, a, axes=1)  # Sum over last axis of b and first axis of a
tf.tensordot(b, a, axes=[[1], [0]])  # Equivalent
```

[11]:
```
<tf.Tensor: shape=(2,), dtype=float32, numpy=array([4.1, 3. ], dtype=float32)>
```

In the case of two rank-2 Tensors, we can use the `tf.linalg.matmul` function. (In fact, we can use rank >= 2 Tensors with `tf.linalg.matmul` - see the docs.)

[12]:
```python
# Use tf.linalg.matmul to compute product

# tf.linalg.matmul(b, a)  # Shape error

print(b.shape)
print(a.shape)
```

```
(2, 3)
(3,)
```

Useful operations to manipulate Tensor shapes are `tf.expand_dims`, `tf.squeeze` and `tf.reshape`.

[13]:
```python
# Add an extra dimension to a Tensor

a = tf.expand_dims(a, 1)
print(a.shape)
```

```
(3, 1)
```

[14]:
```python
# Use tf.matmul, tf.squeeze and tf.reshape

# tf.linalg.matmul(b, a)
tf.reshape(tf.squeeze(tf.linalg.matmul(b, a)), [1, 2])
```

[14]:
```
<tf.Tensor: shape=(1, 2), dtype=float32, numpy=array([[4.1, 3. ]],
dtype=float32)>
```

It is also often useful to fill Tensors with random values.

[15]:
```python
# Create a random normal Tensor

tf.random.normal((3, 3))
```

[15]:
```
<tf.Tensor: shape=(3, 3), dtype=float32, numpy=
array([[ 0.10689038, -1.1122327 ,  0.75735706],
       [-0.8103127 , -0.21552776,  1.705053  ],
       [ 0.7772002 , -0.4163874 ,  1.2633331 ]], dtype=float32)>
```

[16]:
```python
# Create a random integer Tensor

tf.random.uniform(shape=(2, 4), minval=0, maxval=10, dtype='int32')
```

[16]:
```
<tf.Tensor: shape=(2, 4), dtype=int32, numpy=
array([[5, 6, 9, 6],
       [2, 0, 2, 4]], dtype=int32)>
```

McCulloch-Pitts neuron As an example, we will use Tensors to implement the McCulloch-Pitts neuron for a simple logical function. The McCulloch-Pitts neuron operates on boolean inputs, and uses a threshold activation to produce a boolean output. The function can be written as

$$f(\mathbf{x}) = \begin{cases} 1 & \text{if } \sum_i x_i \geq b \\ 0 & \text{if } \sum_i x_i < b \end{cases}$$

[17]:
```python
# Define the AND function

def logical_and(x):
    return tf.cast(tf.math.greater_equal(tf.reduce_sum(x), tf.reduce_sum(tf.
 ↪shape(x))), tf.int32)
```

[18]:
```python
# Test the AND function with a few examples

logical_and(tf.constant([1, 1]))
logical_and(tf.constant([1, 1, 0]))
logical_and(tf.ones((2, 3), dtype=tf.int32))
```

[18]:
```
<tf.Tensor: shape=(), dtype=int32, numpy=1>
```

[19]:
```python
# Define the OR function

def logical_or(x):
    return tf.cast(tf.math.greater_equal(tf.reduce_sum(x), 1), tf.int32)
```

[20]:
```python
# Test the OR function with a few examples

logical_or(tf.constant([1, 0]))
```

```
logical_or(tf.zeros(3,))
```

[20]: <tf.Tensor: shape=(), dtype=int32, numpy=0>

Exercise. Define the function for the NOR operation below (all inputs must be zero) for inputs `x`. *Hint: use the* `tf.math.logical_not` *function.*

[21]:
```
# Define the NOR function

def logical_nor(x):
    return tf.cast(tf.math.greater_equal(tf.reduce_sum(x), 1), dtype=tf.int32)
```

[22]:
```
# Test the NOR function with a few examples

print(logical_nor(tf.constant([1, 0])))  # False
print(logical_nor(tf.constant([0, 0])))  # True
print(logical_nor(tf.constant([0, 0, 0])))  # True
print(logical_nor(tf.constant([1, 0, 1])))  # False
```

```
tf.Tensor(1, shape=(), dtype=int32)
tf.Tensor(0, shape=(), dtype=int32)
tf.Tensor(0, shape=(), dtype=int32)
tf.Tensor(1, shape=(), dtype=int32)
```

Variables Tensors are *immutable objects*; that is, their state cannot be modified. The operations they encapsulate (or the values of a constant Tensor) are fixed. Variables are special kinds of Tensors that have *mutable state*, so their values can be updated. This is useful for parameters of a model, such as the weights and biases in a neural network.

[23]:
```
# Create a TensorFlow Variable

initial_value = tf.random.normal((2, 2))
u = tf.Variable(initial_value)
u
```

[23]:
```
<tf.Variable 'Variable:0' shape=(2, 2) dtype=float32, numpy=
array([[-0.7588623 , -0.6496014 ],
       [-1.7437831 ,  0.03645665]], dtype=float32)>
```

This looks very similar to a Tensor. However, Variables come with extra methods for updating their state, such as `assign`, `assign_add` and `assign_sub`.

[24]:
```
# Assign a new value to the Variable

new_value = 2. * tf.ones((2, 2))
u.assign(new_value)
u
```

[24]:
```
<tf.Variable 'Variable:0' shape=(2, 2) dtype=float32, numpy=
array([[2., 2.],
       [2., 2.]], dtype=float32)>
```

[25]:
```
# Add a value to the Variable

increment = tf.constant([[0., 0.], [1., 1.]])
u.assign_add(increment)
u
```

[25]:
```
<tf.Variable 'Variable:0' shape=(2, 2) dtype=float32, numpy=
array([[2., 2.],
       [3., 3.]], dtype=float32)>
```

[26]:
```
# Subtract a value from the Variable

decrement = tf.constant([[2., 0.], [2., 0.]])
u.assign_sub(decrement)
u
```

[26]:
```
<tf.Variable 'Variable:0' shape=(2, 2) dtype=float32, numpy=
array([[0., 2.],
       [1., 3.]], dtype=float32)>
```

We will often use Variables in operations within the computational graph. The result of the operation is a Tensor.

[27]:
```
# Use a Variable in a simple operation

v = tf.Variable([2.6, -0.4])
s = v + 1
s
```

[27]:
```
<tf.Tensor: shape=(2,), dtype=float32, numpy=array([3.6, 0.6], dtype=float32)>
```

The perceptron The perceptron is also a linear binary classifier, but with more flexible weights. It can be written as the following function

$$f(\mathbf{x}) = \begin{cases} 1 & \text{if } \sum_i w_i x_i + b \geq 0 \\ 0 & \text{if } \sum_i w_i x_i + b < 0 \end{cases}$$

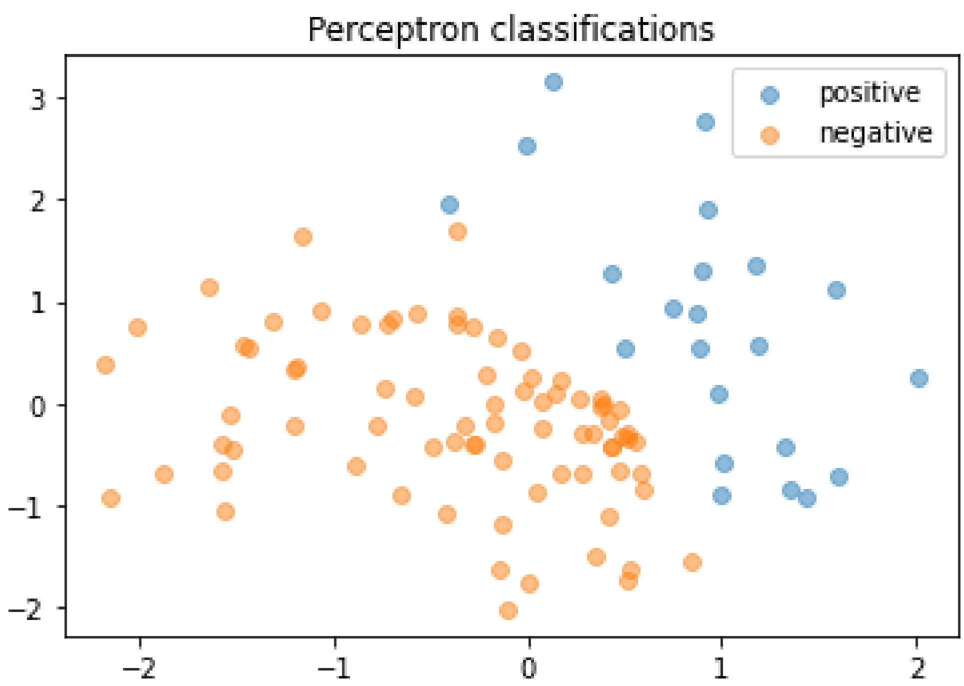
As an example, we will use Tensors and Variables to implement the perceptron classifier.

[28]:
```
# Implement the weights and bias as Variables

weights = tf.Variable(tf.constant([1., 0.5]), name='weights')
bias = tf.Variable(tf.constant(-0.5), name='bias')
```

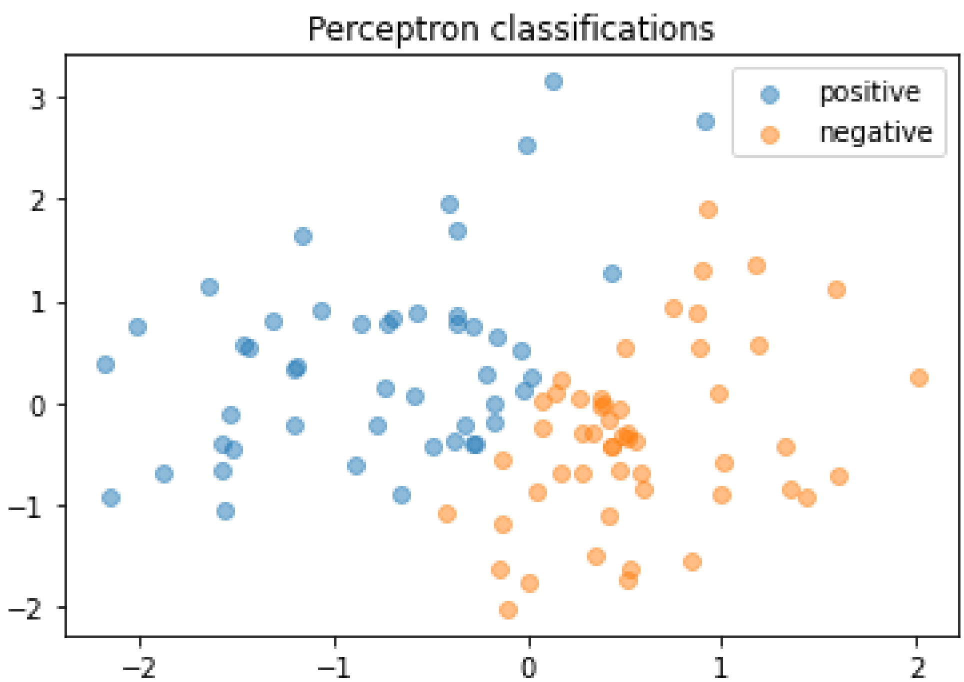
[29]:
```python
# Define the perceptron classifier

def perceptron(x):
    return tf.math.greater_equal(tf.tensordot(x, weights, axes=1) + bias, 0.)
```

[30]:
```python
# Create a random set of test points

x = tf.random.normal((100, 2))
```

[31]:
```python
# Plot the points coloured by class prediction

import matplotlib.pyplot as plt

preds = perceptron(x)
# print(preds)
positive_class = x[preds]
negative_class = x[~preds]
plt.scatter(positive_class[:, 0], positive_class[:, 1], alpha=0.5,␣
 ↪label='positive')
plt.scatter(negative_class[:, 0], negative_class[:, 1], alpha=0.5,␣
 ↪label='negative')
plt.title("Perceptron classifications")
plt.legend()
plt.show()
```



[32]:
```
# Update the weights and bias and re-plot

weights.assign_sub(tf.constant([1.5, 0.3]))
bias.assign_add(tf.constant(0.5))

preds = perceptron(x)
positive_class = x[preds]
negative_class = x[~preds]
plt.scatter(positive_class[:, 0], positive_class[:, 1], alpha=0.5,␣
 ↪label='positive')
plt.scatter(negative_class[:, 0], negative_class[:, 1], alpha=0.5,␣
 ↪label='negative')
plt.title("Perceptron classifications")
plt.legend()
plt.show()
```



Exercise. Can you find weights and bias values to implement the NOT gate for $x \in \{0, 1\}$ and the XOR gate for $x \in \{0, 1\}^2$? If yes, what are the values? If no, why not?

The Sequential class

There are multiple ways to build and apply deep learning models in Tensorflow, from high-level, quick and easy-to-use APIs, to low-level operations. In this section you will walk through the high-level Keras API for quickly building, training, evaluating and predicting from deep learning models. In particular, you will see how to use the `Sequential` class to implement MLP models.

[33]:
```
import tensorflow as tf
```

The `Dense` layer We will see how to build MLP models using the `Dense` layer class from TensorFlow.

This class implements the layer transformation $\mathbf{h}^{(k+1)} = \sigma\left(\mathbf{W}^{(k)}\mathbf{h}^{(k)} + \mathbf{b}^{(k)}\right)$.

[34]:
```
# Create a Dense layer

from tensorflow.keras.layers import Dense

dense_layer = Dense(4, activation='sigmoid')
```

[35]:
```
# Inspect the layer parameters

dense_layer.variables
```

[35]:
```
[]
```

TensorFlow models are designed to process batches of data at once, and always expect inputs to have a batch dimension in the first axis. For example, a batch of 16 inputs, each of which is a length 4 vector, should have a shape `[16, 4]`.

[36]:
```
# Call the dense layer on an input to create the weights

x = tf.ones((2, 6))
y = dense_layer(x)
y
```

[36]:
```
<tf.Tensor: shape=(2, 4), dtype=float32, numpy=
array([[0.38525614, 0.28927264, 0.3345727 , 0.2571653 ],
       [0.38525614, 0.28927264, 0.3345727 , 0.2571653 ]], dtype=float32)>
```

[37]:
```
# Inspect the layer parameters

dense_layer.weights
```

[37]:
```
[<tf.Variable 'dense/kernel:0' shape=(6, 4) dtype=float32, numpy=
 array([[-0.45606366, -0.5074954 , -0.46482536, -0.32824028],
        [ 0.38430178,  0.6727103 , -0.42831722, -0.7530881 ],
        [ 0.06628472,  0.36173153,  0.3533473 , -0.05336368],
        [-0.3686994 , -0.3892936 ,  0.60684013,  0.5333266 ],
        [-0.37225094, -0.2626431 , -0.6893156 , -0.40879175],
        [ 0.27913022, -0.77392906, -0.06530464, -0.05059743]],
       dtype=float32)>,
 <tf.Variable 'dense/bias:0' shape=(4,) dtype=float32, numpy=array([0., 0., 0.,
0.], dtype=float32)>]
```

Note that the parameters of the layer are Variable objects. This makes sense, as recall that Variables are mutable, and we will want to modify them during network training.

MLP model To construct an MLP model, we stack multiple `Dense` layers together by passing them in a list to the `Sequential` API:

[38]:
```
# Build an MLP model

from tensorflow.keras.models import Sequential

mlp = Sequential([
    Dense(4, activation='relu'), # , input_shape=(6,))
    Dense(4, activation='relu'),
    Dense(3)
])
```

The default value for the `activation` keyword argument is `None`, in which case no activation (linear activation) is applied.

[39]:
```
# Call the model on an input to create the weights

x = tf.random.normal((2, 6))
y = mlp(x)
y
```

[39]:
```
<tf.Tensor: shape=(2, 3), dtype=float32, numpy=
array([[ 0.10364395,  0.04307465,  0.15408592],
       [ 0.02053954, -0.02933363, -0.02315051]], dtype=float32)>
```

It is worth knowing that the `Sequential` class itself inherits from the `Layer` class, so all the same properties and methods are also available for `Sequential` models.

[40]:
```
# Inspect the model parameters

mlp.weights
```

[40]:
```
[<tf.Variable 'dense_1/kernel:0' shape=(6, 4) dtype=float32, numpy=
 array([[-0.5980768 , -0.7218989 , -0.38988623, -0.54147387],
        [ 0.44846022, -0.05472422, -0.4802916 , -0.48221576],
        [ 0.4284655 , -0.173141  ,  0.08751845, -0.46963954],
        [ 0.32816756,  0.2823342 , -0.04352921, -0.19540453],
        [ 0.51533175, -0.4285431 ,  0.22099501, -0.57560444],
        [-0.5344832 , -0.688197  , -0.15963936, -0.7688281 ]],
       dtype=float32)>,
 <tf.Variable 'dense_1/bias:0' shape=(4,) dtype=float32, numpy=array([0., 0.,
0., 0.], dtype=float32)>,
 <tf.Variable 'dense_2/kernel:0' shape=(4, 4) dtype=float32, numpy=
 array([[ 0.43999845, -0.31714344,  0.3804621 ,  0.40883142],
```

```
        [-0.8632621 , -0.17998952,  0.39355093, -0.6305837 ],
        [-0.06779307,  0.09681976, -0.8602839 , -0.18632317],
        [-0.6112984 , -0.03618854,  0.7849507 , -0.04905427]],
       dtype=float32)>,
 <tf.Variable 'dense_2/bias:0' shape=(4,) dtype=float32, numpy=array([0., 0.,
0., 0.], dtype=float32)>,
 <tf.Variable 'dense_3/kernel:0' shape=(4, 3) dtype=float32, numpy=
 array([[-0.14663029, -0.885043  ,  0.24321127],
        [ 0.626385  , -0.8945743 , -0.7060105 ],
        [ 0.54812014,  0.22779989,  0.81488204],
        [ 0.8918526 ,  0.41170716,  0.33124566]], dtype=float32)>,
 <tf.Variable 'dense_3/bias:0' shape=(3,) dtype=float32, numpy=array([0., 0.,
0.], dtype=float32)>]
```

[41]:
```
# Inspect the model layers

# mlp.layers
# mlp.layers[1]
mlp.layers[1].kernel
```

[41]:
```
<tf.Variable 'dense_2/kernel:0' shape=(4, 4) dtype=float32, numpy=
array([[ 0.43999845, -0.31714344,  0.3804621 ,  0.40883142],
       [-0.8632621 , -0.17998952,  0.39355093, -0.6305837 ],
       [-0.06779307,  0.09681976, -0.8602839 , -0.18632317],
       [-0.6112984 , -0.03618854,  0.7849507 , -0.04905427]],
      dtype=float32)>
```

[42]:
```
# Print the model summary

mlp.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (2, 4)                    28
_________________________________________________________________
dense_2 (Dense)              (2, 4)                    20
_________________________________________________________________
dense_3 (Dense)              (2, 3)                    15
=================================================================
Total params: 63
Trainable params: 63
Non-trainable params: 0
_________________________________________________________________
```

`Sequential` models (and layers) also have `trainable_weights` and `non_trainable_weights` properties, as weights (Variables) that are created can be set to trainable or non-trainable.

Train an MLP model on the MNIST dataset Multidimensional inputs (i.e., with rank >= 2) can also be processed by an MLP network by simply unrolling, or flattening the dimensions. This can be done easily using the `Flatten` layer.

[43]:
```
# Load the MNIST dataset

(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-
datasets/mnist.npz
11493376/11490434 [==============================] - 3s 0us/step
```

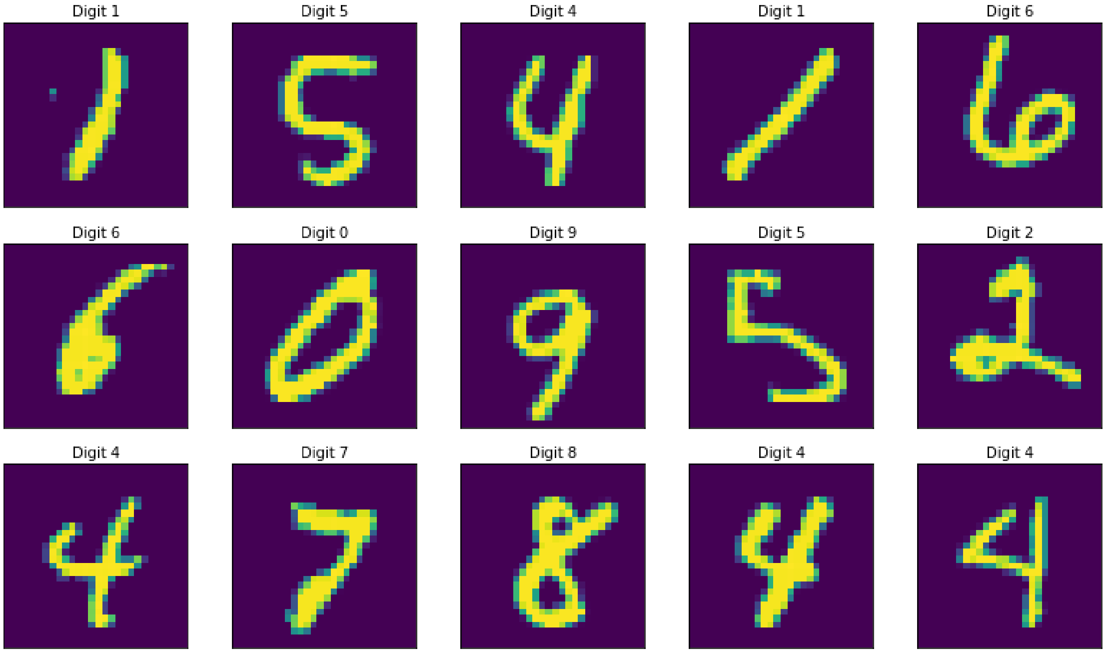
[44]:
```
! rm ~/.keras/datasets/mnist.npz
```

Several datasets are available to load using the Keras API, see the docs.

[45]:
```
# Inspect the data shapes

print(x_train.shape)
print(y_train.shape)
print(x_test.shape)
print(y_test.shape)
```

```
(60000, 28, 28)
(60000,)
(10000, 28, 28)
(10000,)
```

[46]:
```
# View a few training data examples

import numpy as np
import matplotlib.pyplot as plt

n_rows, n_cols = 3, 5
random_inx = np.random.choice(x_train.shape[0], n_rows * n_cols, replace=False)
fig, axes = plt.subplots(n_rows, n_cols, figsize=(14, 8))
fig.subplots_adjust(hspace=0.2, wspace=0.1)

for n, i in enumerate(random_inx):
    row = n // n_cols
    col = n % n_cols
    axes[row, col].imshow(x_train[i])
    axes[row, col].get_xaxis().set_visible(False)
    axes[row, col].get_yaxis().set_visible(False)
    axes[row, col].text(10., -1.5, f'Digit {y_train[i]}')
plt.show()
```



[47]:
```
# Create an MNIST classifier model

from tensorflow.keras.layers import Flatten

mnist_model = Sequential([
    Flatten(input_shape=(28, 28)),
    Dense(64, activation='tanh'),
    Dense(64, activation='tanh'),
    Dense(10, activation='softmax')
])
mnist_model.summary()
```

```
Model: "sequential_1"

_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten (Flatten)            (None, 784)               0

_________________________________________________________________
dense_4 (Dense)              (None, 64)                50240

_________________________________________________________________
dense_5 (Dense)              (None, 64)                4160

_________________________________________________________________
dense_6 (Dense)              (None, 10)                650
=================================================================
Total params: 55,050
Trainable params: 55,050
```

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

To train the model, we need to specify a loss function to minimise, and an optimisation algorithm. The average negative log-likelihood on the training set is given by the categorical cross entropy

$$L(\theta) = -\frac{1}{|\mathcal{D}_{train}|} \sum_{x_i \in \mathcal{D}_{train}} \sum_{j=1}^{10} \tilde{y}_{ij} \ln f_\theta(x_i)_j,$$

where f_θ is the neural network function (with parameters θ) that outputs a length 10 probability vector $f_\theta(x_i) \in \mathbb{R}^{10}$ for an input example image $x_i \in \mathbb{R}^{28\times 28}$, and $\tilde{y}_{ij}$ is 1 if the correct label for example i is j, and 0 otherwise.

As our labels `y_train` and `y_test` are in sparse form, we use the `sparse_categorical_crossentropy` loss function. We also will use the stochastic gradient descent (SGD) optimiser.

[48]:
```python
# Compile the model

# NOTE: enter a couple of other example loss functions, including mse
mnist_model.compile(loss='sparse_categorical_crossentropy', optimizer='sgd',␣
 ↪metrics=['accuracy'])
```

The image data is filled with integer pixel values from 0 to 255. To facilitate the training, we rescale the values to the interval $[0, 1]$.

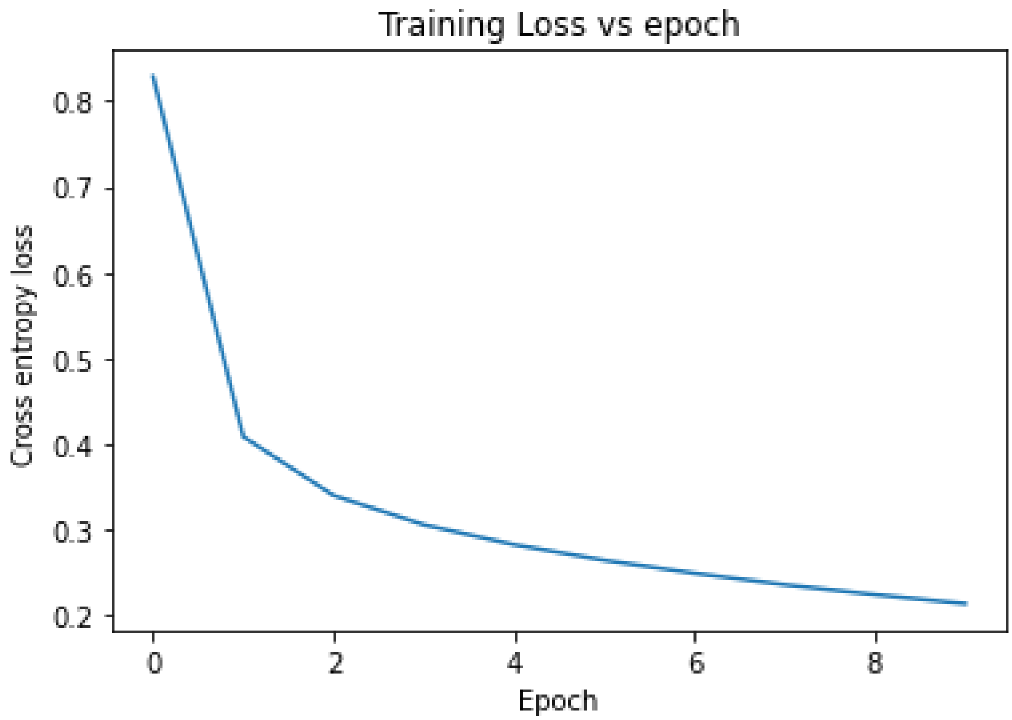
[49]:
```python
# Rescale the image data

x_train = x_train / 255.
x_test = x_test / 255.
```

[50]:
```python
# Train the model

# mnist_model.fit(x_train, y_train, epochs=10, batch_size=64)
history = mnist_model.fit(x_train, y_train, epochs=10, batch_size=64)
```

```
Epoch 1/10
938/938 [==============================] - 1s 849us/step - loss: 1.2448 -
accuracy: 0.66290s - loss: 1.3176 - accuracy:
Epoch 2/10
938/938 [==============================] - 1s 810us/step - loss: 0.4356 -
accuracy: 0.8880
Epoch 3/10
938/938 [==============================] - 1s 836us/step - loss: 0.3519 -
accuracy: 0.9014
Epoch 4/10
938/938 [==============================] - 1s 806us/step - loss: 0.3057 -
accuracy: 0.9152
```

```
Epoch 5/10
938/938 [==============================] - 1s 796us/step - loss: 0.2870 -
accuracy: 0.9198
Epoch 6/10
938/938 [==============================] - 1s 800us/step - loss: 0.2709 -
accuracy: 0.9217
Epoch 7/10
938/938 [==============================] - 1s 776us/step - loss: 0.2567 -
accuracy: 0.9264
Epoch 8/10
938/938 [==============================] - 1s 785us/step - loss: 0.2396 -
accuracy: 0.9302
Epoch 9/10
938/938 [==============================] - 1s 821us/step - loss: 0.2282 -
accuracy: 0.9346
Epoch 10/10
938/938 [==============================] - 1s 810us/step - loss: 0.2140 -
accuracy: 0.9397
```

[51]:
```python
# Plot the learning curve

import matplotlib.pyplot as plt

plt.plot(history.history['loss'])
plt.xlabel("Epoch")
plt.ylabel("Cross entropy loss")
plt.title("Training Loss vs epoch")
plt.show()
```



[52]:
```
# Evaluate the model on the test set

mnist_model.evaluate(x_test, y_test)
```

```
313/313 [==============================] - 0s 636us/step - loss: 0.2075 -
accuracy: 0.9404
```

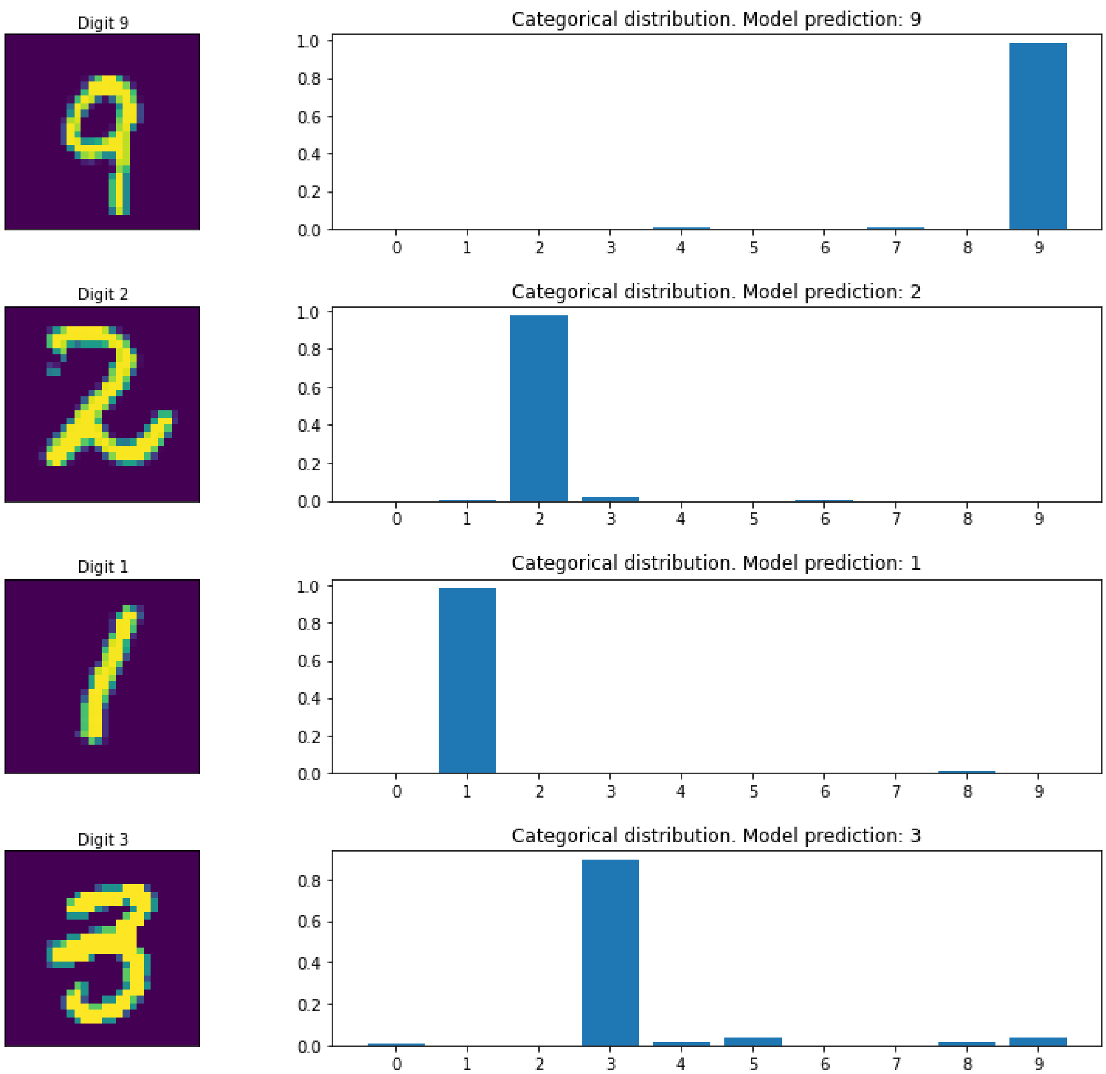
[52]: [0.20753329992294312, 0.9404000043869019]

[53]:
```
# Get predictions from model

preds = mnist_model.predict(x_test)
# preds = mnist_model(x_test)
preds.shape
```

[53]: (10000, 10)

[54]:
```
# Plot some predicted categorical distributions

num_test_images = x_test.shape[0]

random_inx = np.random.choice(num_test_images, 4)
random_preds = preds[random_inx, ...]
random_test_images = x_test[random_inx, ...]
```

```
random_test_labels = y_test[random_inx, ...]

fig, axes = plt.subplots(4, 2, figsize=(16, 12))
fig.subplots_adjust(hspace=0.4, wspace=-0.2)

for i, (prediction, image, label) in enumerate(zip(random_preds, random_test_images, random_test_labels)):
    axes[i, 0].imshow(np.squeeze(image))
    axes[i, 0].get_xaxis().set_visible(False)
    axes[i, 0].get_yaxis().set_visible(False)
    axes[i, 0].text(10., -1.5, f'Digit {label}')
    axes[i, 1].bar(np.arange(len(prediction)), prediction)
    axes[i, 1].set_xticks(np.arange(len(prediction)))
    axes[i, 1].set_title(f"Categorical distribution. Model prediction: {np.argmax(prediction)}")
plt.show()
```



Exercise. The MNIST dataset is an easy dataset, and the above model is far from optimal. Try experimenting with longer training times and/or model architecture changes to see if you can improve on the performance.

The `tf.data` module

In this section we will introduce a standard data processing pipeline in TensorFlow, using the `tf.data` module.

[55]:
```
import tensorflow as tf
```

The Fashion-MNIST dataset We will build a deep learning classifier on the Fashion-MNIST dataset to demonstrate the use of the `tf.data` module. First we load the dataset using the Keras API.

[56]:
```
# Load the Fashion-MNIST dataset

(x_train, y_train), (x_test, y_test) = tf.keras.datasets.fashion_mnist.
↪load_data()
```

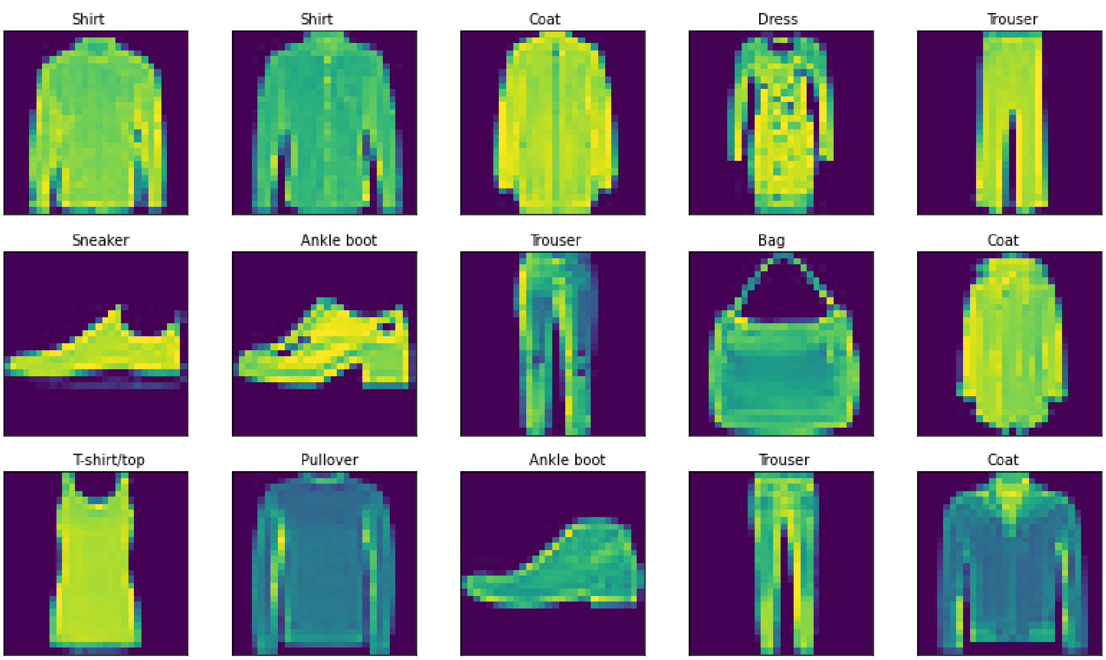
```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-
datasets/train-labels-idx1-ubyte.gz
32768/29515 [==================================] - 0s 1us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-
datasets/train-images-idx3-ubyte.gz
26427392/26421880 [==============================] - 7s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-
datasets/t10k-labels-idx1-ubyte.gz
8192/5148 [================================================] - 0s 1us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-
datasets/t10k-images-idx3-ubyte.gz
4423680/4422102 [==============================] - 1s 0us/step
```

[57]:
```
! rm -r ~/.keras/datasets/fashion-mnist
```

[58]:
```
# Get the class labels

classes = [
    "T-shirt/top",
    "Trouser",
    "Pullover",
    "Dress",
    "Coat",
    "Sandal",
    "Shirt",
    "Sneaker",
```

```
    "Bag",
    "Ankle boot"
]
```

[59]:
```
# View a few training data examples

import numpy as np
import matplotlib.pyplot as plt

n_rows, n_cols = 3, 5
random_inx = np.random.choice(x_train.shape[0], n_rows * n_cols, replace=False)
fig, axes = plt.subplots(n_rows, n_cols, figsize=(14, 8))
fig.subplots_adjust(hspace=0.2, wspace=0.1)

for n, i in enumerate(random_inx):
    row = n // n_cols
    col = n % n_cols
    axes[row, col].imshow(x_train[i])
    axes[row, col].get_xaxis().set_visible(False)
    axes[row, col].get_yaxis().set_visible(False)
    axes[row, col].text(10., -1.5, f'{classes[y_train[i]]}')
plt.show()
```



[60]:
```
# Build the model
```

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Flatten, Dense

fashion_mnist_model = Sequential([
    Flatten(input_shape=(28, 28)),
    Dense(64, activation='relu'),
    Dense(64, activation='relu'),
    Dense(10)
], name='fashion_mnist_classifier')
```

[61]:
```
# Print the model summary

fashion_mnist_model.summary()
```

```
Model: "fashion_mnist_classifier"

_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_1 (Flatten)          (None, 784)               0

_________________________________________________________________
dense_7 (Dense)              (None, 64)                50240

_________________________________________________________________
dense_8 (Dense)              (None, 64)                4160

_________________________________________________________________
dense_9 (Dense)              (None, 10)                650
=================================================================
Total params: 55,050
Trainable params: 55,050
Non-trainable params: 0

_________________________________________________________________
```

The main class that we will be working with is the `Dataset` class from the `tf.data` module.

[62]:
```
# Load the data into tf.data.Dataset objects

train_dataset = tf.data.Dataset.from_tensor_slices((x_train, y_train)) # pause␣
↪after train_dataset
test_dataset = tf.data.Dataset.from_tensor_slices((x_test, y_test))
train_dataset.element_spec
```

[62]:
```
(TensorSpec(shape=(28, 28), dtype=tf.uint8, name=None),
 TensorSpec(shape=(), dtype=tf.uint8, name=None))
```

[63]:
```
# Iterate over the Dataset object

for inputs, labels in train_dataset.take(2):
    print(type(inputs))
    print(type(labels))
```

```
    print(inputs.shape)
    print(labels.shape)
```

```
<class 'tensorflow.python.framework.ops.EagerTensor'>
<class 'tensorflow.python.framework.ops.EagerTensor'>
(28, 28)
()
<class 'tensorflow.python.framework.ops.EagerTensor'>
<class 'tensorflow.python.framework.ops.EagerTensor'>
(28, 28)
()
```

`Dataset` objects come with `map` and `filter` methods for data preprocessing on the fly. For example, we can normalise the pixel values to the range $[0, 1]$ with the `map` method:

[64]:
```
# Normalise the pixel values

def normalise_pixels(image, label):
    return (tf.cast(image, tf.float32) / 255., label)  # Maybe add the tf.cast␣
↪after the error

train_dataset = train_dataset.map(normalise_pixels)
test_dataset = test_dataset.map(normalise_pixels)
train_dataset.element_spec
```

[64]:
```
(TensorSpec(shape=(28, 28), dtype=tf.float32, name=None),
 TensorSpec(shape=(), dtype=tf.uint8, name=None))
```

We could also filter out data examples according to some criterion with the `filter` method. For example, if we wanted to exclude all data examples with label 9 from the training:

[65]:
```
# Filter out all examples with label 9 (ankle boot)

train_dataset = train_dataset.filter(lambda x, y: tf.math.logical_not(tf.
↪equal(y, 9)))
test_dataset = test_dataset.filter(lambda x, y: tf.math.logical_not(tf.equal(y,␣
↪9)))
```

[66]:
```
# Shuffle the training dataset

train_dataset = train_dataset.shuffle(buffer_size=1024)
```

[67]:
```
# Batch the datasets

batch_size = 64
train_dataset = train_dataset.batch(batch_size)  # drop_remainder=True
test_dataset = test_dataset.batch(batch_size)
```

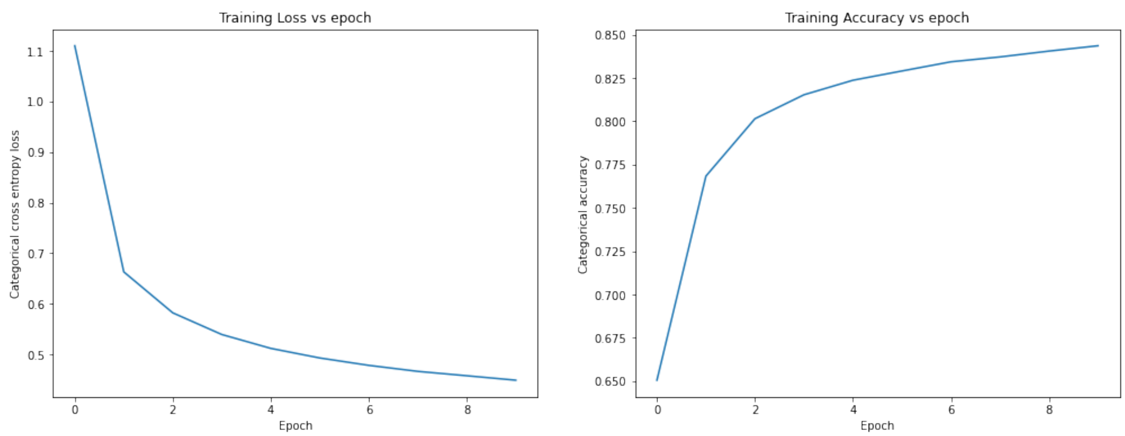
[68]:
```python
# Print the element_spec

train_dataset.element_spec
```

[68]:
```
(TensorSpec(shape=(None, 28, 28), dtype=tf.float32, name=None),
 TensorSpec(shape=(None,), dtype=tf.uint8, name=None))
```

[69]:
```python
# Compile and fit the model

sgd = tf.keras.optimizers.SGD(learning_rate=0.005)
loss_fn = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)

fashion_mnist_model.compile(optimizer=sgd, loss=loss_fn, metrics=['accuracy'])
history = fashion_mnist_model.fit(train_dataset, epochs=10)
```

```
Epoch 1/10
844/844 [==============================] - 4s 5ms/step - loss: 1.5057 -
accuracy: 0.5200
Epoch 2/10
844/844 [==============================] - 3s 4ms/step - loss: 0.6950 -
accuracy: 0.7551
Epoch 3/10
844/844 [==============================] - 3s 4ms/step - loss: 0.5962 -
accuracy: 0.7965
Epoch 4/10
844/844 [==============================] - 3s 4ms/step - loss: 0.5471 -
accuracy: 0.8124
Epoch 5/10
844/844 [==============================] - 3s 3ms/step - loss: 0.5158 -
accuracy: 0.8215
Epoch 6/10
844/844 [==============================] - 3s 3ms/step - loss: 0.4937 -
accuracy: 0.8285
Epoch 7/10
844/844 [==============================] - 3s 3ms/step - loss: 0.4791 -
accuracy: 0.8339
Epoch 8/10
844/844 [==============================] - 3s 3ms/step - loss: 0.4663 -
accuracy: 0.8371
Epoch 9/10
844/844 [==============================] - 3s 3ms/step - loss: 0.4564 -
accuracy: 0.8411
Epoch 10/10
844/844 [==============================] - 3s 3ms/step - loss: 0.4488 -
accuracy: 0.8430
```

[70]:
```python
# Plot the learning curves

import matplotlib.pyplot as plt

fig = plt.figure(figsize=(17, 6))
fig.add_subplot(121)
plt.plot(history.history['loss'])
plt.xlabel("Epoch")
plt.ylabel("Categorical cross entropy loss")
plt.title("Training Loss vs epoch")
fig.add_subplot(122)
plt.plot(history.history['accuracy'])
plt.xlabel("Epoch")
plt.ylabel("Categorical accuracy")
plt.title("Training Accuracy vs epoch")
plt.show()
```



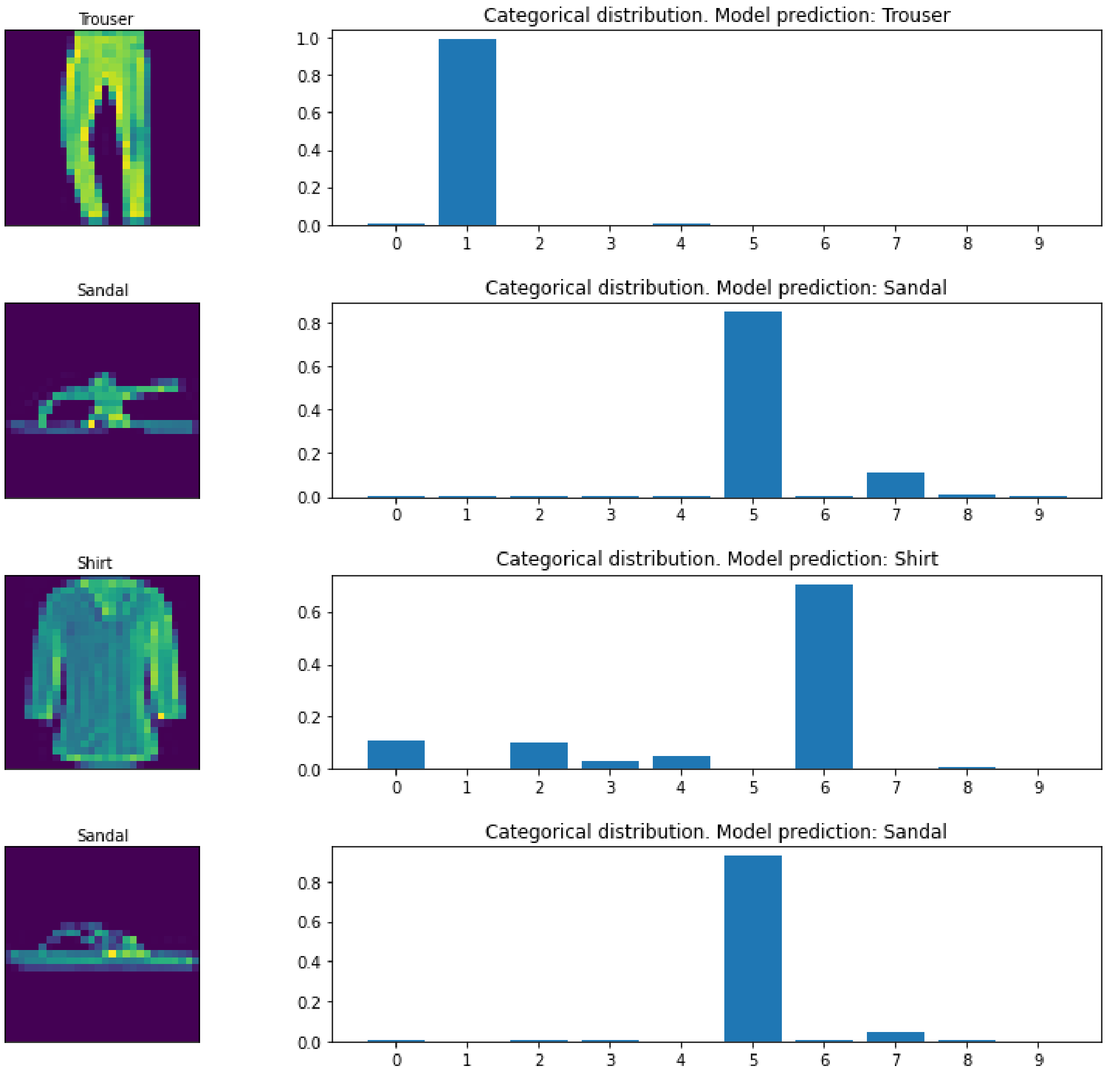
[71]:
```python
# Evaluate the model on the test set

fashion_mnist_model.evaluate(test_dataset)
```

```
141/141 [==============================] - 1s 4ms/step - loss: 0.4801 -
accuracy: 0.8304
```

[71]: [0.48009902238845825, 0.8304444551467896]

[72]:
```python
# Get predictions from model

for images, labels in test_dataset.take(1):
    preds = fashion_mnist_model.predict(images)
    preds = tf.nn.softmax(preds, axis=-1).numpy()
```

[73]:
```
# Plot some predicted categorical distributions

num_test_images = preds.shape[0]

random_inx = np.random.choice(num_test_images, 4)
random_preds = preds[random_inx, ...]
random_test_images = images.numpy()[random_inx, ...]
random_test_labels = labels.numpy()[random_inx, ...]

fig, axes = plt.subplots(4, 2, figsize=(16, 12))
fig.subplots_adjust(hspace=0.4, wspace=-0.2)

for i, (prediction, image, label) in enumerate(zip(random_preds,␣
↪random_test_images, random_test_labels)):
    axes[i, 0].imshow(np.squeeze(image))
    axes[i, 0].get_xaxis().set_visible(False)
    axes[i, 0].get_yaxis().set_visible(False)
    axes[i, 0].text(10., -1.5, f'{classes[label]}')
    axes[i, 1].bar(np.arange(len(prediction)), prediction)
    axes[i, 1].set_xticks(np.arange(len(prediction)))
    axes[i, 1].set_title(f"Categorical distribution. Model prediction:␣
↪{classes[np.argmax(prediction)]}")
plt.show()
```



Exercise. Rewrite the model to make it a binary classifier, and change the dataset processing steps above, to map 'Sandal', 'Sneaker' and 'Ankle boot' to a single label 0, and all other categories to label 1.

TensorFlow regularisers, Dropout layers and callbacks

In this section we will build on what we have covered already with the `Sequential` API, and include weight regularisers, `Dropout` layers, and introduce callback objects - these are very useful objects for dynamically performing operations during the training run. An example is the `EarlyStopping` callback.

[74]:
```
import tensorflow as tf
```

For this tutorial we will use the diabetes dataset from `sklearn`.

[75]:

```python
# Load the diabetes dataset

from sklearn.datasets import load_diabetes
diabetes_dataset = load_diabetes()
```

[76]:

```python
# Print dataset description

print(diabetes_dataset["DESCR"])
```

```
.. _diabetes_dataset:

Diabetes dataset
----------------

Ten baseline variables, age, sex, body mass index, average blood
pressure, and six blood serum measurements were obtained for each of n =
442 diabetes patients, as well as the response of interest, a
quantitative measure of disease progression one year after baseline.

**Data Set Characteristics:**

  :Number of Instances: 442

  :Number of Attributes: First 10 columns are numeric predictive values

  :Target: Column 11 is a quantitative measure of disease progression one year
after baseline

  :Attribute Information:
      - Age
      - Sex
      - Body mass index
      - Average blood pressure
      - S1
      - S2
      - S3
      - S4
      - S5
      - S6

Note: Each of these 10 feature variables have been mean centered and scaled by
the standard deviation times `n_samples` (i.e. the sum of squares of each column
totals 1).

Source URL:
https://www4.stat.ncsu.edu/~boos/var.select/diabetes.html
```

```
For more information see:
Bradley Efron, Trevor Hastie, Iain Johnstone and Robert Tibshirani (2004) "Least
Angle Regression," Annals of Statistics (with discussion), 407-499.
(https://web.stanford.edu/~hastie/Papers/LARS/LeastAngle_2002.pdf)
```

[77]:
```python
# Get the input and target data

print(diabetes_dataset.keys())  # Run this first
data = diabetes_dataset["data"]
targets = diabetes_dataset["target"]
```

```
dict_keys(['data', 'target', 'DESCR', 'feature_names', 'data_filename',
'target_filename'])
```

[78]:
```python
# Normalise the target data (this will make clearer training curves)

targets = (targets - targets.mean()) / targets.std()
```

[79]:
```python
# Partition the data into training and validation sets

from sklearn.model_selection import train_test_split

train_data, val_data, train_targets, val_targets = train_test_split(data,␣
 ↪targets, test_size=0.2)  # Type train_test_split first

print(train_data.shape)
print(val_data.shape)
print(train_targets.shape)
print(val_targets.shape)
```

```
(353, 10)
(89, 10)
(353,)
(89,)
```

[80]:
```python
# Load the data into training, validation and test Dataset objects

train_dataset = tf.data.Dataset.from_tensor_slices((train_data, train_targets))
val_dataset = tf.data.Dataset.from_tensor_slices((val_data, val_targets))

train_dataset = train_dataset.shuffle(353)

train_dataset = train_dataset.batch(128)
val_dataset = val_dataset.batch(89)

train_dataset = train_dataset.prefetch(tf.data.experimental.AUTOTUNE)
```

[81]:
```
# Build the MLP model

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

model = Sequential([
    Dense(256, activation="relu", input_shape=(train_data.shape[-1],)),
    Dense(256, activation="relu"),
    Dense(256, activation="relu"),
    Dense(1)
])
```

[82]:
```
# Print the model summary

model.summary()
```

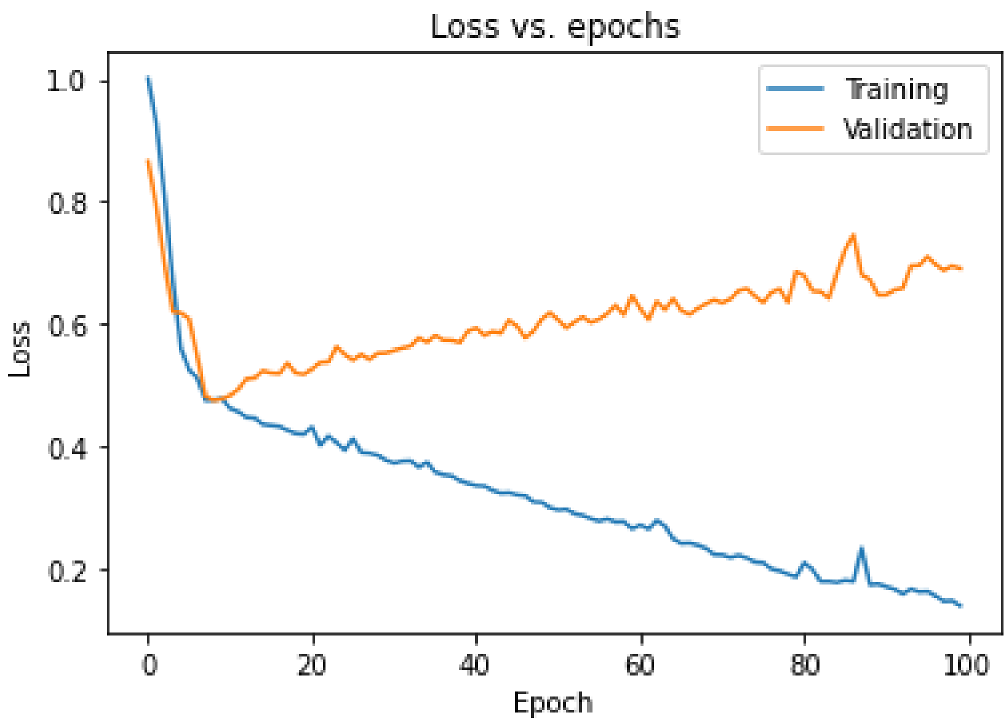
```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_10 (Dense)             (None, 256)               2816
_________________________________________________________________
dense_11 (Dense)             (None, 256)               65792
_________________________________________________________________
dense_12 (Dense)             (None, 256)               65792
_________________________________________________________________
dense_13 (Dense)             (None, 1)                 257
=================================================================
Total params: 134,657
Trainable params: 134,657
Non-trainable params: 0
_________________________________________________________________
```

[83]:
```
# Compile the model

model.compile(optimizer='adam', loss="mse")
```

[84]:
```
# Train the model, including validation

history = model.fit(train_dataset, epochs=100, validation_data=val_dataset,␣
 ↪verbose=False)
```

[85]:
```
# Plot the training and validation loss

import matplotlib.pyplot as plt

plt.plot(history.history['loss'])
```

```
plt.plot(history.history['val_loss'])
plt.title('Loss vs. epochs')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Training', 'Validation'], loc='upper right')
plt.show()
```



Regularise the model Both ℓ^2 and ℓ^1 regularisation can easily be included using the `kernel_regularizer` and `bias_regularizer` keyword arguments in the `Dense` layer.

Dropout can also be easily included as an additional layer of our model.

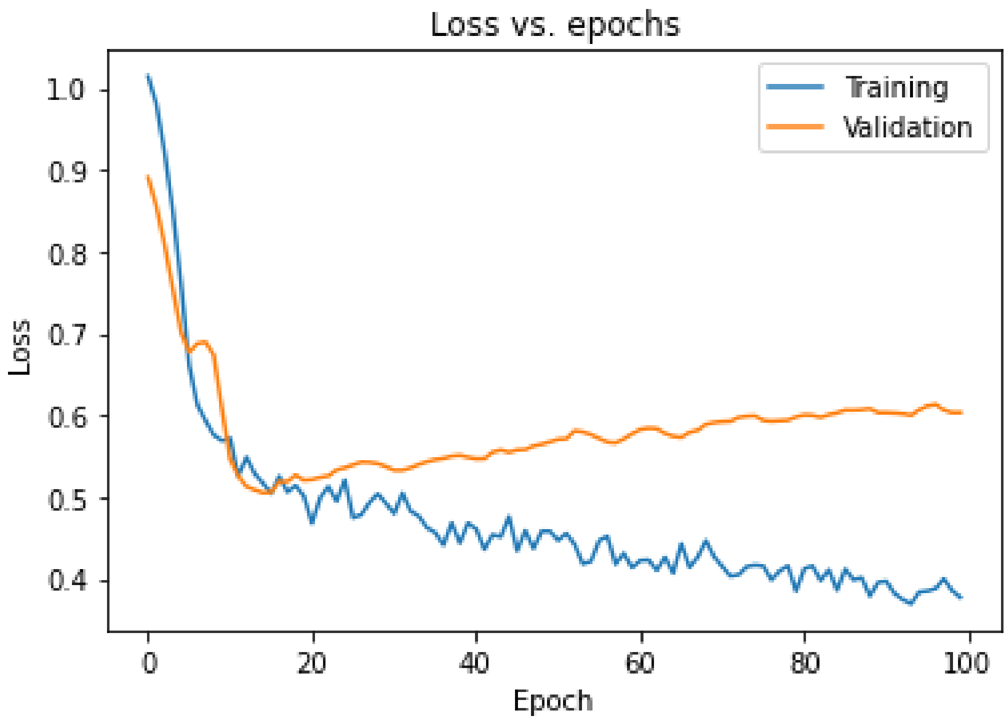
[86]:
```
# Redefine the model using l2 regularisation and dropout

from tensorflow.keras.layers import Dropout
from tensorflow.keras import regularizers

l2_coeff = 1e-5 # 1e-5
rate = 0.5 # 0.3

def get_regularised_model():
    model = Sequential([
        Dense(256, kernel_regularizer=regularizers.l2(l2_coeff), activation="relu", input_shape=(train_data.shape[-1],)),
```

```
        Dropout(rate),
        Dense(256, kernel_regularizer=regularizers.l2(l2_coeff),␣
↪activation="relu"),
        Dropout(rate),
        Dense(256, kernel_regularizer=regularizers.l2(l2_coeff),␣
↪activation="relu"),
        Dropout(rate),
        Dense(1)
    ])
    return model
model = get_regularised_model()
```

[87]:
```
# Compile the model

model.compile(optimizer='adam', loss="mse")
```

[88]:
```
# Train the model, including validation

history = model.fit(train_dataset, epochs=100, validation_data=val_dataset,␣
↪verbose=False)
```

[89]:
```
# Plot the training and validation loss

plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Loss vs. epochs')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Training', 'Validation'], loc='upper right')
plt.show()
```



The ℓ^2 regularisation and dropout have helped to reduce the overfitting of the model.

Callbacks We can go one step further and introduce early stopping as well, and save the model weights at the best validation score. We can do this with callbacks.

[90]:
```
# Create a new model

model = get_regularised_model()
```

[91]:
```
# Compile the model

# model.compile(optimizer='adam', loss="mse")
model.compile(optimizer='adam', loss="mse", metrics=['mae'])
```

The `EarlyStopping` callback is a built-in callback in the `tf.keras.callbacks` module. You can see a complete list of built-in callbacks here.

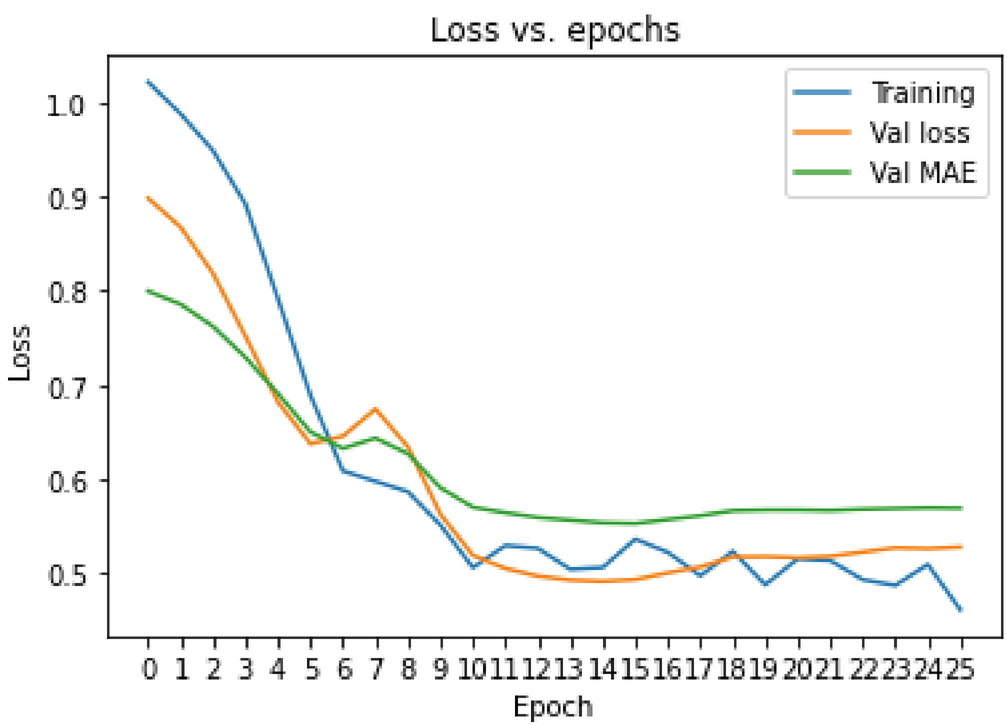
[92]:
```
# Create an EarlyStopping callback

# earlystopping = tf.keras.callbacks.EarlyStopping(monitor='val_loss',␣
 ↪patience=10)
earlystopping = tf.keras.callbacks.EarlyStopping(monitor='val_mae', patience=10)
```

[93]:

```
# Train the model, including validation

history = model.fit(train_dataset, epochs=100, validation_data=val_dataset,␣
↪verbose=False,
                    callbacks=[earlystopping])
```

[94]:

```
# Plot the training and validation metrics

import numpy as np

plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.plot(history.history['val_mae']) # Added in a second pass
plt.title('Loss vs. epochs')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.xticks(np.arange(len(history.history['loss'])))
# plt.legend(['Training', 'Validation'], loc='upper right')
plt.legend(['Training', 'Val loss', 'Val MAE'], loc='upper right')
plt.show()
```



Exercise. Take a look at some more of the callbacks available in the callbacks module in TensorFlow, and have a go at implemented some of them in your model here.

CNNs and feature maps

In this section we will use the `Conv2D` and `MaxPool2D` layer to implement the convolution and pooling operations described above, and see how these easily fits into our existing model-building workflow.

We will also see the effect of different kernel tensor choices on the output feature maps, and look at more complex feature maps from a pre-trained model.

[95]:
```
import tensorflow as tf
```

The `Conv2D` and `MaxPool2D` classes are imported from the `tf.keras.layers` module just as the `Flatten` and `Dense` layers we have already worked with. Note that there are also 1-D and 3-D variants of these layers available, which both work in a similar way.

[96]:
```
# Define a dummy model with Conv2D and MaxPool2D layers

from tensorflow.keras import Sequential # pause
from tensorflow.keras.layers import Conv2D, MaxPool2D # pause before import
# pause

model = Sequential([
    Conv2D(8, (3, 5), activation='relu', input_shape=(32, 32, 3)),  # pause␣
 ↪before Conv2D, and after each arg
    MaxPool2D((2, 2)), # pause before arg
    Conv2D(16, 3, activation='relu'),  # pause after each arg
    MaxPool2D(2)
])
```

[97]:
```
# Print the model summary

model.summary()
```

```
Model: "sequential_5"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 30, 28, 8)         368
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 15, 14, 8)         0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 13, 12, 16)        1168
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 6, 6, 16)          0
=================================================================
Total params: 1,536
Trainable params: 1,536
Non-trainable params: 0
_________________________________________________________________
```

[98]:
```
# Inspect the layer variables' shapes

print(model.layers[0].kernel.shape)
print(model.layers[0].bias.shape)

# Add afterwards
print(model.layers[2].kernel.shape)
print(model.layers[2].bias.shape)
```

```
(3, 5, 3, 8)
(8,)
(3, 3, 8, 16)
(16,)
```

Edge detection filters The kernels (or filters) in CNNs are typically learned with backpropagation. However, simple low-level features such as edge detection kernels can also be designed by hand. In this section we will see the output of such low-level kernels.

[99]:
```
# Define a simple model with a Conv2D layer

model = Sequential([
    Conv2D(1, (3, 3), activation=None, use_bias=False, input_shape=(None, None,␣
 ↪1))  # slight pauses in between args
])
```

A shape dimension of `None` indicates that the model can take flexible input sizes in this dimension.

[100]:
```
# Inspect the model's weights

model.weights
```

[100]:
```
[<tf.Variable 'conv2d_2/kernel:0' shape=(3, 3, 1, 1) dtype=float32, numpy=
 array([[[[ 0.03013647]],

          [[ 0.3100946 ]],

          [[-0.39117697]]],


         [[[ 0.32159257]],

          [[-0.49464744]],

          [[ 0.34914166]]],


         [[[ 0.32474768]],
```

```
[[-0.3876173 ]],

[[ 0.5624256 ]]]], dtype=float32)>]
```

[101]:
```python
# Load an image as grayscale

import matplotlib.pyplot as plt

image = tf.io.read_file("./figures/oscar.png")
image = tf.io.decode_png(image, channels=1)
plt.figure(figsize=(8, 6))
plt.imshow(image, cmap='gray')
plt.axis('off')
plt.show()
```

A simple and intuitive edge detection kernel is the Sobel operator:

[102]:
```python
# Define simple edge detection filters

sobel_x = tf.constant([[1, 0, -1], [2, 0, -2], [1, 0, -1]], dtype=tf.float32)
sobel_y = tf.constant([[1, 2, 1], [0, 0, 0], [-1, -2, -1]], dtype=tf.float32)

print(sobel_x)
```

```
# print(sobel_y)
```

```
tf.Tensor(
[[ 1.  0. -1.]
 [ 2.  0. -2.]
 [ 1.  0. -1.]], shape=(3, 3), dtype=float32)
```

[103]:
```
# Set the model kernel

def assign_filter(arr):
    model.weights[0].assign(arr[:, :, tf.newaxis, tf.newaxis])  # first just␣
 ↪write arr
```

[104]:
```
# Compute the feature maps

assign_filter(sobel_x)
gx = model(image[None, ...])[0]  # Maybe run without the None (error), and then␣
 ↪without the [0] first

# Add this after running the above
assign_filter(sobel_y)
gy = model(image[None, ...])[0]

g = tf.sqrt(tf.square(gx) + tf.square(gy))
```

[105]:
```
# View the image and feature map

fig = plt.figure(figsize=(17, 6))
fig.add_subplot(121)
plt.imshow(image, cmap='gray')
plt.axis('off')
fig.add_subplot(122)
plt.imshow(g, cmap='gray')  # First gx, then gy, then g
plt.axis('off')
plt.show()  # After executing, show the forehead markings with the cursor␣
 ↪(after both gx and gy)
```

Extract learned features from a pre-trained model In this section we will load a CNN model that has been pre-trained on the ImageNet dataset, which is a large scale image classification dataset which to date has over 20,000 categories and over 14 million images. Large deep learning models trained on this dataset tend to learn general, useful representations of image features that can be used for a range of image processing tasks.

Below we will load the VGG-19 model (Simonyan and Zisserman 2015), which is available to load as a pre-trained model in the `tf.keras.applications` module. This might take a minute or two to download the first time you run the cell.

[106]:
```
# Load the VGG-19 model

# vgg = tf.keras.applications.VGG19(weights='imagenet')  # pause after vgg,␣
↪applications
vgg = tf.keras.applications.VGG19(weights='imagenet', include_top=False)
```

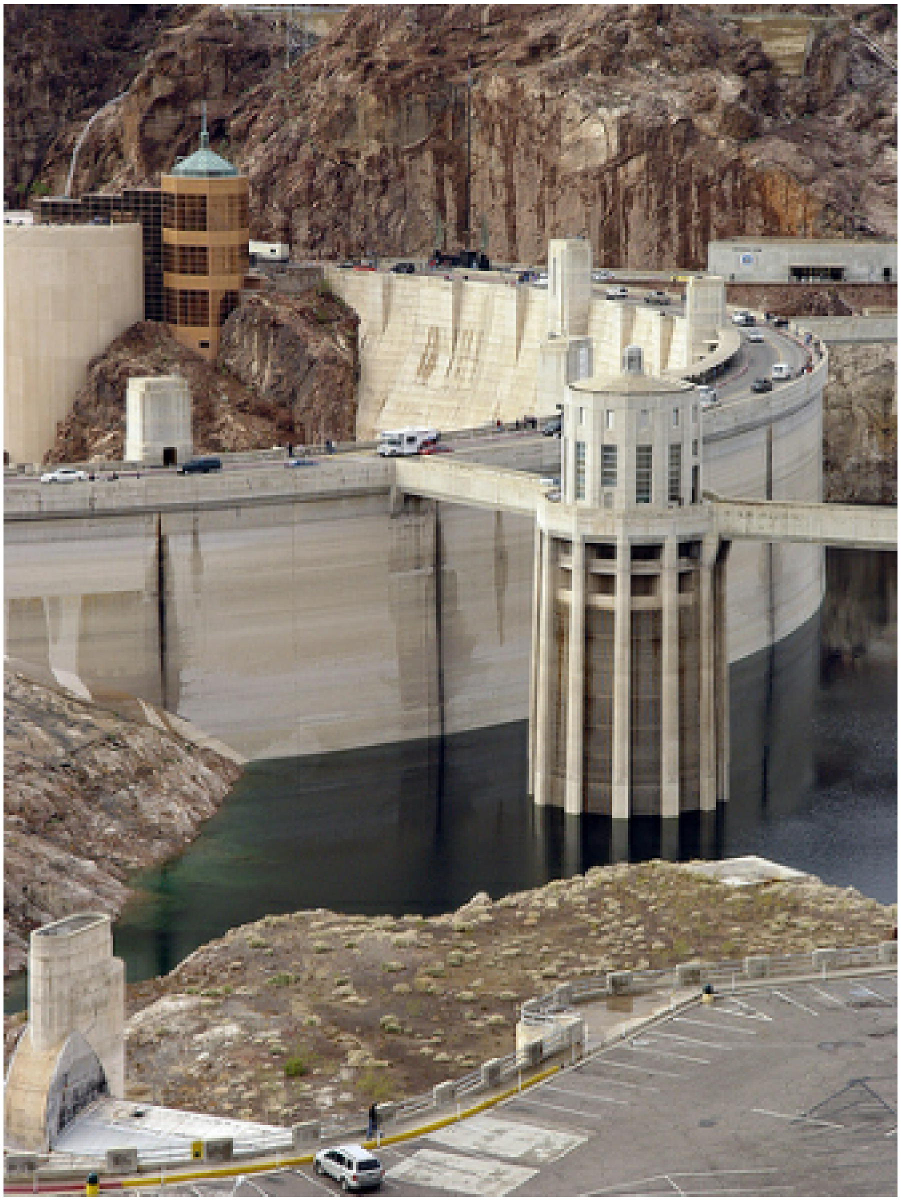
[107]:
```
# Print the model summary

vgg.summary()  # pause, slowly scroll to the bottom, then later to the top
```

```
Model: "vgg19"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, None, None, 3)]   0
_________________________________________________________________
block1_conv1 (Conv2D)        (None, None, None, 64)    1792
_________________________________________________________________
block1_conv2 (Conv2D)        (None, None, None, 64)    36928
_________________________________________________________________
block1_pool (MaxPooling2D)   (None, None, None, 64)    0
_________________________________________________________________
block2_conv1 (Conv2D)        (None, None, None, 128)   73856
_________________________________________________________________
block2_conv2 (Conv2D)        (None, None, None, 128)   147584
_________________________________________________________________
block2_pool (MaxPooling2D)   (None, None, None, 128)   0
_________________________________________________________________
block3_conv1 (Conv2D)        (None, None, None, 256)   295168
_________________________________________________________________
block3_conv2 (Conv2D)        (None, None, None, 256)   590080
_________________________________________________________________
block3_conv3 (Conv2D)        (None, None, None, 256)   590080
_________________________________________________________________
block3_conv4 (Conv2D)        (None, None, None, 256)   590080
```

```
_________________________________________________________________
block3_pool (MaxPooling2D)   (None, None, None, 256)   0
_________________________________________________________________
block4_conv1 (Conv2D)        (None, None, None, 512)   1180160
_________________________________________________________________
block4_conv2 (Conv2D)        (None, None, None, 512)   2359808
_________________________________________________________________
block4_conv3 (Conv2D)        (None, None, None, 512)   2359808
_________________________________________________________________
block4_conv4 (Conv2D)        (None, None, None, 512)   2359808
_________________________________________________________________
block4_pool (MaxPooling2D)   (None, None, None, 512)   0
_________________________________________________________________
block5_conv1 (Conv2D)        (None, None, None, 512)   2359808
_________________________________________________________________
block5_conv2 (Conv2D)        (None, None, None, 512)   2359808
_________________________________________________________________
block5_conv3 (Conv2D)        (None, None, None, 512)   2359808
_________________________________________________________________
block5_conv4 (Conv2D)        (None, None, None, 512)   2359808
_________________________________________________________________
block5_pool (MaxPooling2D)   (None, None, None, 512)   0
=================================================================
Total params: 20,024,384
Trainable params: 20,024,384
Non-trainable params: 0
_________________________________________________________________
```

We will visualise the features extracted by this model at different levels of hierarchy for the following image:

[108]:
```python
# Load a colour image

image = tf.io.read_file("./figures/hoover_dam.JPEG")
image = tf.io.decode_jpeg(image, channels=3)
plt.figure(figsize=(6, 10))
plt.imshow(image)
plt.axis('off')
plt.show()
```



We will use the functional API to create a multi-output model that outputs different hidden layer outputs within the model.

[109]:
```
# Define the multi-output model

from tensorflow.keras.models import Model

inputs = vgg.inputs
layer_names = ['block1_conv2', 'block2_conv2', 'block3_conv4', 'block4_conv4',␣
↪'block5_conv4']
```

```
outputs = [vgg.get_layer(layer_name).output for layer_name in layer_names] #␣
↪add .output at the end
vgg_features = Model(inputs=inputs, outputs=outputs)
```

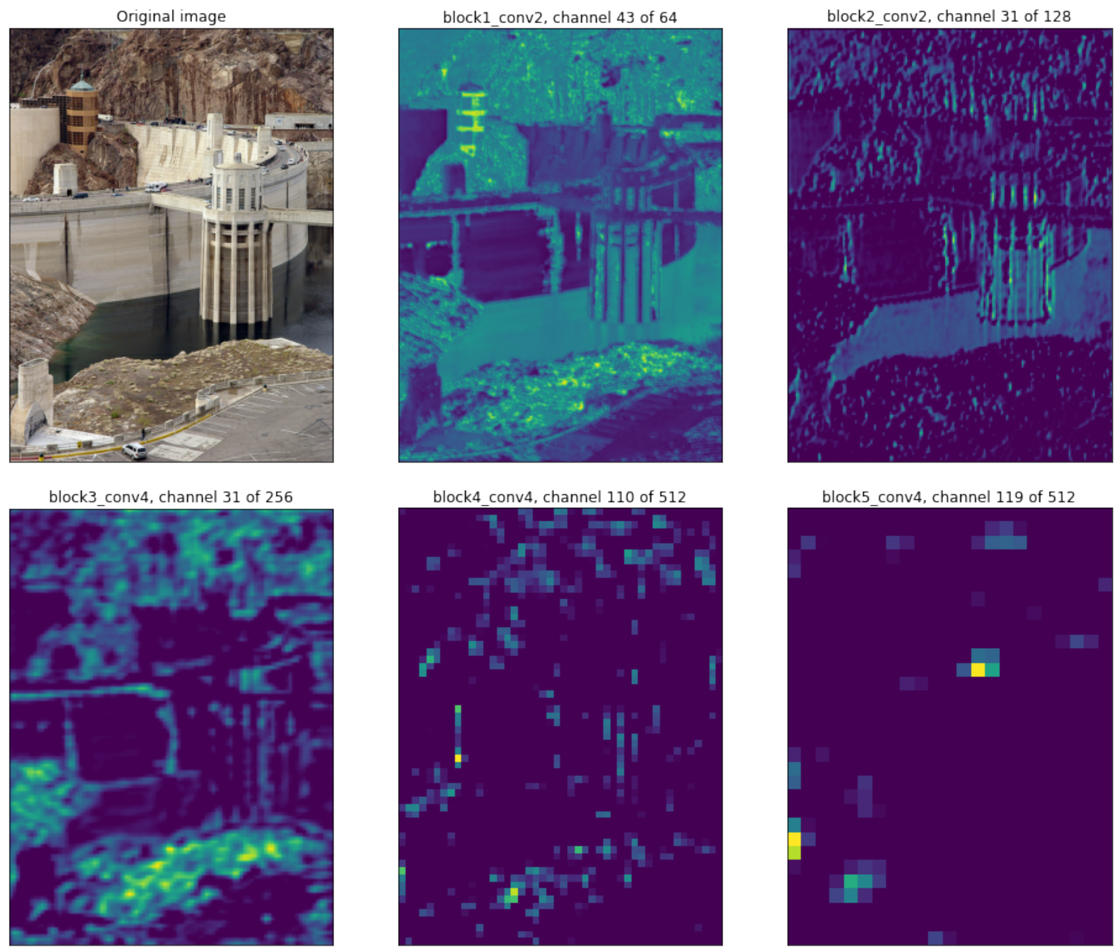
[110]:
```
# View the model inputs and outputs Tensors

vgg_features.inputs # inputs, then inputs, then vgg.inputs (then delete this)
vgg_features.outputs
```

[110]:
```
[<KerasTensor: shape=(None, None, None, 64) dtype=float32 (created by layer
'block1_conv2')>,
 <KerasTensor: shape=(None, None, None, 128) dtype=float32 (created by layer
'block2_conv2')>,
 <KerasTensor: shape=(None, None, None, 256) dtype=float32 (created by layer
'block3_conv4')>,
 <KerasTensor: shape=(None, None, None, 512) dtype=float32 (created by layer
'block4_conv4')>,
 <KerasTensor: shape=(None, None, None, 512) dtype=float32 (created by layer
'block5_conv4')>]
```

[111]:
```
# Extract the hierarchical features for this image

image_processed = tf.keras.applications.vgg19.preprocess_input(image)
features = vgg_features(image_processed[tf.newaxis, ...]) # pause after␣
↪vgg_features and image_processed
features = [image] + features
```

[112]:
```
# Visualise the features

import numpy as np

n_rows, n_cols = 2, 3
fig, axes = plt.subplots(n_rows, n_cols, figsize=(16, 14))
fig.subplots_adjust(hspace=0.05, wspace=0.2)

for i in range(len(features)):
    feature_map = features[i]
    num_channels = feature_map.shape[-1]
    row = i // n_cols
    col = i % n_cols
    if i == 0:
        axes[row, col].imshow(image)
        axes[row, col].set_title('Original image')
    else:
        random_feature = np.random.choice(num_channels)
        axes[row, col].imshow(feature_map[0, ..., random_feature])
```

```
        axes[row, col].set_title('{}, channel {} of {}'.
↪format(layer_names[i-1], random_feature + 1, num_channels))

    axes[row, col].get_xaxis().set_visible(False)
    axes[row, col].get_yaxis().set_visible(False)
plt.show()
```



Exercise: load one of your own images to view the features extracted by the VGG-19 network.

References

- Chen, J. & Kyrillidis, A., (2019), "Decaying Momentum Helps Neural Network Training", arXiv preprint arXiv:1910.04952.
- Duchi, J., Hazan, E., & Singer, Y. (2011), "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization", *Journal of Machine Learning Research*, **12**, 2121–2159.
- Dumoulin, V. & Visin, F. (2016), "A guide to convolution arithmetic for deep learning", arXiv preprint, abs/1603.07285.
- Hochreiter, S. (1991), "Untersuchungen zu dynamischen neuronalen Netzen", Diploma thesis, Institut für Informatik, Lehrstuhl Prof. Brauer, Technische Universität München.

- Kingma, D. P. & Ba, J. L. (2015), "Adam: a Method for Stochastic Optimization", International Conference on Learning Representations, 1–13.
- McCulloch, W. & Pitts, W. (1943), "A Logical Calculus of Ideas Immanent in Nervous Activity", Bulletin of Mathematical Biophysics, **5**, 127-147.
- LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., & Jackel, L. D. (1989) "Backpropagation Applied to Handwritten Zip Code Recognition", AT&T Bell Laboratories.
- Mitchell, T. (1997), "Machine Learning", McGraw-Hill, New York.
- Nesterov, Y. (1983), "A method for unconstrained convex minimization problem with the rate of convergence o(1/k2)", Doklady ANSSSR (translated as Soviet. Math. Docl.), **269**, 543–547.
- Qian, N. (1999), "On the momentum term in gradient descent learning algorithms", Neural Networks: The Official Journal of the International Neural Network Society, **12** (1), 145–151.
- Robbins, H. and Monro, S. (1951), "A stochastic approximation method", *The annals of mathematical statistics*, 400–407.
- Rosenblatt, F. (1958), "The Perceptron: A Probabilistic Model for Information Storage and Organization in The Brain", Psychological Review, 65-386.
- Rosenblatt, F. (1961), "Principles of Neurodynamics. Perceptrons and the Theory of Brain Mechanisms", Defense Technical Information Center.
- Rumelhart, D. E., McClelland, J. L. and the PDP Research Group (1986a), "Parallel Distributed Processing: Explorations in the Microstructure of Cognition", MIT Press, Cambridge.
- Rumelhart, D. E., Hinton, G., & Williams, R. (1986b), "Learning representations by back-propagating errors", Nature, **323**, 533-536.
- Simonyan, K. and Zisserman, A. (2015), "Very Deep Convolutional Networks for Large-Scale Image Recognition", in *3rd International Conference on Learning Representations, (ICLR) 2015*, San Diego, CA, USA.
- Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014), "Dropout: A Simple Way to Prevent Neural Networks from Overfitting", Journal of Machine Learning Research, **15**, 1929-1958.