Part II

Data Processing

Chapter 9

NumPy

NumPy is a Python package that allows to create, manipulate and perform calculations with multi-dimensional arrays. As such, the basic object defined by Numpy is `numpy.ndarray`.

9.1 Create a NumPy ndarray

A common way to create a NumPy ndarray is by using the `numpy.array` helper function in a list.

```
>>> import numpy
>>> l = [1,2,3,4,5]
>>> np_l = numpy.array(l)
>>> print(np_l) # Notice the lack of commas
[1 2 3 4 5]
```

The ndarray object we just created is 1-dimensional. However, NumPy offers the possibility to have arrays with the dimensions we want. For example, if we want to create a 2-dimensional array (matrix) we can use a list of lists.

```
>>> two_dim_l = [[11,12],[21,22]]
>>> np_l = numpy.array(two_dim_l)
>>> print(np_l)
[[11 12]
 [21 22]]
```

If you wanted to create an array with 3 dimensions then you would need a nested list of lists of lists. A similar approach could be used to create an array with N dimensions from N nested lists.

In addition to the `numpy.array` function, there are other ways of creating `ndarrays`. For example, we can simply create an `ndarray` filled with zeros (or ones) by using the function `numpy.zeros` (`numpy.ones`) and specifying the pretended shape

```
>>> np_zeros = numpy.zeros((2,2))
>>> print(np_zeros)
[[ 0.  0.]
 [ 0.  0.]]
>>>
>>> np_ones = numpy.ones((2,2))
>>> print(np_ones)
[[ 1.  1.]
 [ 1.  1.]]
```

The `numpy.arange(start, stop,steps)` function is also an interesting way of creating ndarrays. Alternatively, `numpy.linspace(start, stop,num_elements)` gives an ndarray with `num_elements` elements.

```
>>> np_ar = numpy.arange(1,6,2)
>>> print(np_ar)
[1 3 5]
>>>
>>> np_ls = numpy.linspace(0,10, 5)
>>> print(np_ls)
[  0.    2.5   5.    7.5  10. ]
```

You can get the size of an ndarray from the `.shape` attribute. The number of dimensions is given by the length of the shape or the `.ndim` attribute

```
>>> np_3d = numpy.ones((2,2,2))
>>> print(np_3d.shape)
(2, 2, 2)
>>> nr_dimensions = len(np_3d.shape)
>>> print("The number of dimensions is:"); print(nr_dimensions)
The number of dimensions is:
3
>>>
>>> nr_dimensions = np_3d.ndim
>>> print("The number of dimensions is:"); print(nr_dimensions)
The number of dimensions is:
3
```

9.2 Indexing

NumPy offers an indexing mechanism that is similar to the one we saw in section 3.1 for lists but is adapted to use in higher dimensions. Let's start with some simple examples.

```
>>> np_arr = numpy.array([0,1,2,3,4,5,6,7,8,9])
>>> print(np_arr[0])
```

```
0
>>> print(np_arr[-1])
9
>>> print(np_arr[:5])
[0 1 2 3 4]
>>> print(np_arr[7:])
[7 8 9]
>>> print(np_arr[4:7])
[4 5 6]
```

Moving on to a 2-dimensional array, indexing is done as you would in a matrix. So if we wish to select the element that is in the first column of the first row we do

```
>>> twodim_np_arr = numpy.array([[0,1],[2,3]])
>>> print(twodim_np_arr)
[[0 1]
 [2 3]]
```

Indexing is done as you would in a matrix. So if we wish to select the element that is in the first column of the first row we do

```
>>> print(twodim_np_arr[0,0])
0
```

To obtain all the elements in a given dimension we use the `:` operator.

```
>>> print(twodim_np_arr[0,:])
[0 1]
```

Or to obtain the last column

```
>>> print(twodim_np_arr[:,-1])
[1 3]
```

9.3 Operations

Once we have one or more ndarrays we can perform operations with them. You can find the mean of the elements in an array with `array.mean()`

```
>>> a = numpy.array([10,20,30,40,50])
>>> print(a.mean())
30.0
```

If you type instead `array.max()` (`array.min()`) you obtain the highest (lowest) element of the array. The postion of that element can be obtained with `array.argmax()` (`array.argmin()`)

```
>>> print(a.max())
50
>>> print(a.argmax())
4
```

You can add, subtract, multiply and divide ndarray's.

```
>>> a = numpy.array([1.0,2.0])
>>> b = numpy.array([3.0,4.0])
>>> print(a+b)
[4. 6.]
>>> print(a-b)
[-2. -2.]
>>> print(2 * a)
[2. 4.]
>>> print(b/2)
[1.5 2. ]
```

Be aware that if you multiply two matrices with the * operator, you will be performing a element wise multiplication. We can also use functions provided by NumPy to perform matrix operations such as the dot product (multiplication) or transposition.

```
>>> a = numpy.array([[1,2],[3,4]])
>>> b = numpy.array([[1,1],[5,2]])
>>> print(numpy.dot(a,b))
[[11  5]
 [23 11]]
>>> print(numpy.transpose(b))
[[1 5]
 [1 2]]
```

An interesting function to know is `numpy.unique` that returns the unique elements of an array with the respective counts (if `return_counts` is set to `True`)

```
>>> a = numpy.array([1,2,3,3,3,4,4,5,6,6,6,6,6])
>>> print(numpy.unique(a,return_counts = True))
(array([1, 2, 3, 4, 5, 6]), array([1, 1, 3, 2, 1, 5]))
```

These are just a few examples, you can find a comprehensive list of the methods available in NumPy's documentation page [1].

Exercises

Create Numpy code that acomplishes the following simple tasks

[1] https://docs.scipy.org/doc/numpy/reference/arrays.ndarray.html#array-methods

1. Create a Numpy array that starts in 0 and finishes at 10 and contains the even numbers between 0 and 10.
2. Create a Numpy array with the matrix $\begin{bmatrix} 2 & 1 \\ 6 & 3 \end{bmatrix}$.
3. Multiply the matrix you created with the identity matrix $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$.
4. Sum the columns of the matrix created in 2.

Chapter 10

Pandas

Pandas is a powerful Python package that can be used to create, manipulate and process datasets.

10.1 Creating a DataFrame

Pandas represents datasets as a DataFrame object. A DataFrame can be seen as a table that have rows and columns. Usually each row represents a sample and each column a feature.

Two common ways to create a DataFrame are through a dictionary of lists or with a `.csv` file.

If we have the file `cellphone.csv`:

Provider	Price	Data	Voice	Texts	Contract
A	5	0.5	150	500	1
A	7.5	2	250	Unlimited	1
B	9	0.5	200	Unlimited	1
B	14	8	Unlimited	Unlimited	1
C	13	0.5	500	Unlimited	1
C	17	3	Unlimited	Unlimited	1
B	11	4	Unlimited	Unlimited	12
C	15	5	Unlimited	Unlimited	12

We can create a DatatFrame with

```
>>> import pandas as pd
>>> df = pd.read_csv("cellphone.csv")
```

10.2 Inspecting

To check the first five rows of the dataset

```
>>> df.head()
```

```
      Provider  Price  Data      Voice      Texts  Contract
0        A    5.0   0.5        150        500         1
1        A    7.5   2.0        250  Unlimited         1
2        B    9.0   0.5        200  Unlimited         1
3        B   14.0   8.0  Unlimited  Unlimited         1
4        C   13.0   0.5        500  Unlimited         1
```

To check the columns we can type

```
>>> print(df.columns)
Index(['Provider', 'Price', 'Data', 'Voice', 'Texts', 'Contract'], dtype='object')
```

And you can have an idea of the dataset dimensions with

```
>>> print(df.shape)
(8, 6)

#This means that the dataset has 8 rows and 6 columns
```

10.3 Selecting rows and columns

We are usually interested in acessing specific rows or columns that satisify certain requirements. To index a row or a column there are two main methods: `.loc` and `.iloc`. They work in a similar way, with the difference being that `.iloc` uses integers and `.loc` labels.

They can be used like `.loc[rowlabels,columnlabels]` and `.iloc[rowindexes,columnindexes]`. We can also use the symbol `:` that translates to "all".

For instance, you can check what value is in the second column of the first row

```
>>> df.iloc[0,1]
5.0
```

Or the columns "Provider" and "Price"

```
>>> df.loc[:,['Provider','Price']].head()
  Provider  Price
0        A    5.0
1        A    7.5
2        B    9.0
3        B   14.0
4        C   13.0
```

The set of row labels is actually called the index of the DataFrame. In this case, the index of each row is the number you see on the left hand side and coincides with the actual position of the row in the DataFrame TODO. So, for instance, to select the first two rows we can also use the `.loc` method (with a small change).

```
>>> print(df.loc[:1,:]) #Notice that we used :1 and not :2 as with iloc.
                        #This is because how pandas is implemented.
  Provider  Price  Data Voice      Texts  Contract
0        A    5.0   0.5   150        500         1
1        A    7.5   2.0   250  Unlimited         1
```

10.4 Insert and remove

If provider C creates a new plan, you can add it to your DataFrame using the `.loc` method by setting it to an non-existent index.

```
>>> new_row = ['C', 8, 2, 100, 100, 1]
>>> df.loc[8,:] = new_row
>>> print(df)
  Provider  Price  Data      Voice      Texts  Contract
0        A    5.0   0.5        150        500       1.0
1        A    7.5   2.0        250  Unlimited       1.0
2        B    9.0   0.5        200  Unlimited       1.0
3        B   14.0   8.0  Unlimited  Unlimited       1.0
4        C   13.0   0.5        500  Unlimited       1.0
5        C   17.0   3.0  Unlimited  Unlimited       1.0
6        B   11.0   4.0  Unlimited  Unlimited      12.0
7        C   15.0   5.0  Unlimited  Unlimited      12.0
8        C    8.0   2.0        100        100       1.0
```

If it turns out that provider gives up the idea of creating a new plan we can remove with `.drop(rowlabels, axis = 0, inplace = True)`

```
>>> df.drop(8, inplace = True)
>>> print(df)
  Provider  Price  Data      Voice      Texts  Contract
0        A    5.0   0.5        150        500       1.0
1        A    7.5   2.0        250  Unlimited       1.0
2        B    9.0   0.5        200  Unlimited       1.0
3        B   14.0   8.0  Unlimited  Unlimited       1.0
4        C   13.0   0.5        500  Unlimited       1.0
5        C   17.0   3.0  Unlimited  Unlimited       1.0
6        B   11.0   4.0  Unlimited  Unlimited      12.0
7        C   15.0   5.0  Unlimited  Unlimited      12.0
```

To insert a column we can simply use the `.insert(loc, column, value)` method, specifying the position, name and the values of the column. Suppose we are interested in knowing the price per Gigabyte of data.

```
>>> new_column = df.loc[:,'Price']/df.loc[:,'Data']
>>> name_new_column =  "£/GB"
>>> position_new_column = 6
```

```
>>> df.insert(position_new_column, name_new_column, new_column)
>>> print(df)
  Provider  Price  Data      Voice      Texts  Contract       £/GB
0        A    5.0   0.5        150        500         1  10.000000
1        A    7.5   2.0        250  Unlimited         1   3.750000
2        B    9.0   0.5        200  Unlimited         1  18.000000
3        B   14.0   8.0  Unlimited  Unlimited         1   1.750000
4        C   13.0   0.5        500  Unlimited         1  26.000000
5        C   17.0   3.0  Unlimited  Unlimited         1   5.666667
6        B   11.0   4.0  Unlimited  Unlimited        12   2.750000
7        C   15.0   5.0  Unlimited  Unlimited        12   3.000000
```

To remove a column , you can perform a similar operation as removing a row but have to change the `axis` parameter to 1

```
>>> df.drop('£/GB', axis = 1, inplace = True)
>>> print(df)
  Provider  Price  Data      Voice      Texts  Contract
0        A    5.0   0.5        150        500       1.0
1        A    7.5   2.0        250  Unlimited       1.0
2        B    9.0   0.5        200  Unlimited       1.0
3        B   14.0   8.0  Unlimited  Unlimited       1.0
4        C   13.0   0.5        500  Unlimited       1.0
5        C   17.0   3.0  Unlimited  Unlimited       1.0
6        B   11.0   4.0  Unlimited  Unlimited      12.0
7        C   15.0   5.0  Unlimited  Unlimited      12.0
```

10.5 Filtering

Filtering is usually done in two steps. If you want the plans that cost less than £10 you must first obtain a boolean array

```
>>> filtered_rows = df.loc[:,'Price'] < 10
>>> print(filtered_rows)
0     True
1     True
2     True
3    False
4    False
5    False
6    False
7    False
Name: Price, dtype: bool
```

And then we use this array with the .loc method.

```
>>> print(df[filtered_rows])
  Provider  Price  Data Voice      Texts  Contract
0        A    5.0   0.5   150        500         1
1        A    7.5   2.0   250  Unlimited         1
2        B    9.0   0.5   200  Unlimited         1
```

10.6 Analysing

After select a row or a column we can process the information in it. For example, if we want to know the highest amount or the mean of data in the plans in this dataset we can

```
>>> max_data = df.loc[:,'Data'].max()
>>> mean_data = df.loc[:,'Data'].mean()
>>> print(max_data)
8.0
>>> print(mean_data)
2.9375
```

10.7 Group

Sometimes we are interested grouping information to have a more aggregated view of the dataset. For instance, if want to know how many plans there are for each provider we group the dstaset by provider

```
grouped = df.groupby(['Provider'])
```

And then find the size

```
>>> grouped.size()
Provider
A    2
B    3
C    3
dtype: int64
```

It is also possible to group by more than one column

```
>>> grouped = df.groupby(['Provider','Voice'])
>>> grouped.size()
Provider  Voice
A         150          1
          250          1
B         200          1
          Unlimited    2
C         500          1
          Unlimited    2
dtype: int64
```

The `.size()` method of the `grouped` object will give the number of rows that belong to each of the possible combinations of the group. There are also other possibilities, for example the following example shows the average prices of each provider

```
>>> print(df.groupby(['Provider']).mean()["Price"])
Provider
A     6.250000
B    11.333333
C    15.000000
Name: Price, dtype: float64
```

Exercises

Use Pandas to process and query the wireshark file you used in Chapter 6. Namely:

1. Load the Wireshark file that you used in chapter 6 into a pandas DataFrame.
2. How many packets (rows) are there in the DataFrame?
3. How many fields (columns) are there in the DataFrame?
4. Remove the last column.
5. insert a row/column
6. What is the maximun length of a packet?
7. What is the mean length of TCP packets?
8. Repeat the exercise 6.2 with pandas.

Part III

Machine Learning

Chapter 11

Machine Learning

The last years have seen an incredible growth in interest in machine learning. The field is revolutionizing all aspects of our lives: health care, transport, security, entertainment. Obviously, it is also changing dramatically the fields of network and telecommunications.

This chapter provides a very small overview of one of the most important machine learning tools: neural networks and deep learning. You should not expect to have a comprehensive understanding of the field but we hope that it gives you an insight of the main ideas and sparks your interest in studying it a bit more.

11.1 Machine Learning techniques

Machine learning follows a completely different approach than traditional programming. The driver for this is that for many problems (character recognition, autonomic driving, etc) it has proven impossible to come up with algorithms that solve the problem. Humans are just not that smart.

In ML we use **data** to **train** a system to **learn**. We then than test the system with new/other data and hopefully the system knows how to handle the data. This is illustrated in Figure 11.1

Machine learning is seen as a field of Artificial Intelligence which is a wider field. However, in today's world when people mention A.I. they generally mean Machine Learning (ML). See ML as the AI that really works.

Machine learning can be divided in the following sub-fields

- Supervised Learning: In a supervised learning model, the algorithm learns on a labeled dataset, providing an answer key that the algorithm can use to evaluate its accuracy on training data. The most used example of SL is Neural Networks (Deep Learning is just neural networks with lots of layers)

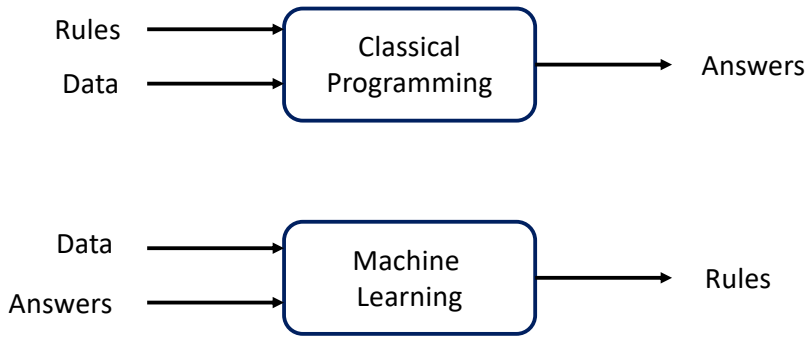

Figure 11.1: Machine Learning vs Traditional Programming

- Unsupervised Learning: in contrast, provides unlabeled data that the algorithm tries to make sense of by extracting features and patterns on its own.
- Reinforcement Learning: is an area of machine learning concerned with how software agents ought to take actions in an environment so as to maximize some notion of cumulative reward.

11.2 Neural Networks

Artificial Neural Networks (or Neural Networks (NN) for short) is a field of machine learning with already an "old history". The intellectual roots are in the 1940s but the first breakthroughs were done in the 1970s. The field stayed relatively under the radar until the deep learning breakthroughs of the beginning of the 21st century.

11.2.1 The MNIST data set

Let us illustrate the use of NNs with a famous example. The MNIST dataset. One problem that is incredibly difficult (if not impossible) to solve with traditional programming is hand-written recognition. This is a clear example where Machine learning can be very advantageous. The MNIST dataset http://yann.lecun.com/exdb/mnist/ is illustrated in Figure 11.2.

A black box illustration of a neural network is illustrated in Figure 11.3. The data including a character (in this case, an array of 28x28 pixels with a grey value in each element) is inputted in the system which then identifies it as an 8.

In slightly more detail the system will output an array of values, ideally with a 1 for the digit 8 and 0 for the others. This can be seen in Figure 11.4.

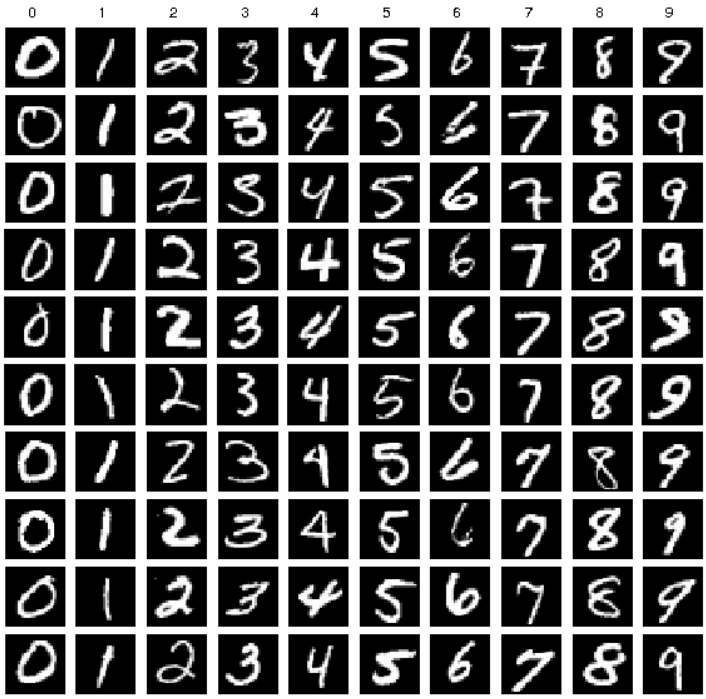

Figure 11.2: The MNIST dataset

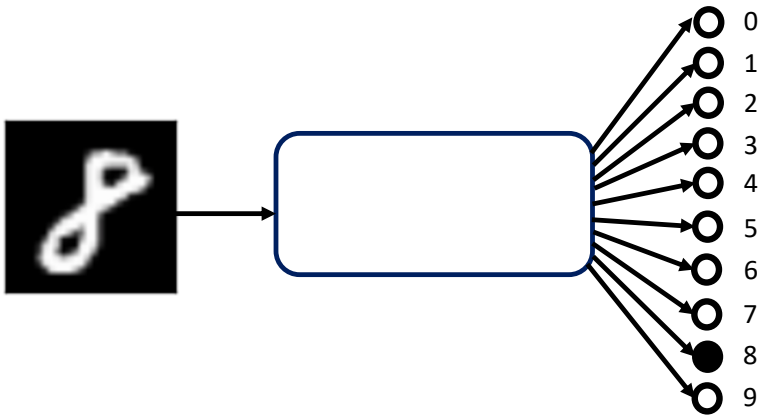

Figure 11.3: Neural Network black box

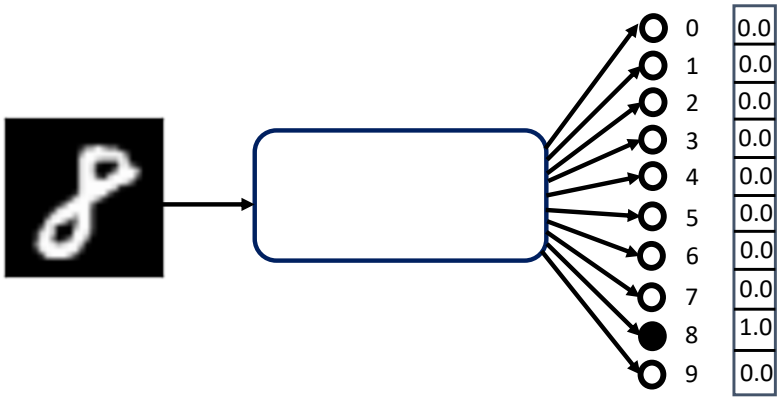

Figure 11.4: Neural Network structure

Let us see now, how a neural network is organized. One can see this in Figure 11.5. A Neural network is made up of **neurons** represented by the circles interconnected. They are organized in several layers. The first layer of neurons will have the inputs. The last layer will have the outputs. All joint layers are fully connected (this is not illustrated for clarity).

Let us now see what happens for each neuron. This is illustrated in Figure **??**.

- Each link will have an associated weight with it (w1, w2, etc)
- The value of the neuron will consist of the sum of products of the input times the weight as illustrated in the formula in the image
- The total sum is then passed through a function that transforms the value between 0 and 1. There are several alternatives for this.
- This applies to every subsequent layer until we have the final values for the output.

11.2.2 Backpropagation

Now the big question is: how do we calculate the weights. This is what the **backpropagation** algorithm does. In a nutshell, the algorithm takes labeled data which in this case is a picture of a 8 with the label 8. It then inputs the data and adjusts the weights in order to make the output as close as possible to 8 (this is called minimizing the loss function as illustrated in Figure **??**). This is repeated many many times (this is the training or learning). When it is finished we should have weights that identify pictures you haven't used in the training part. The system has learned !

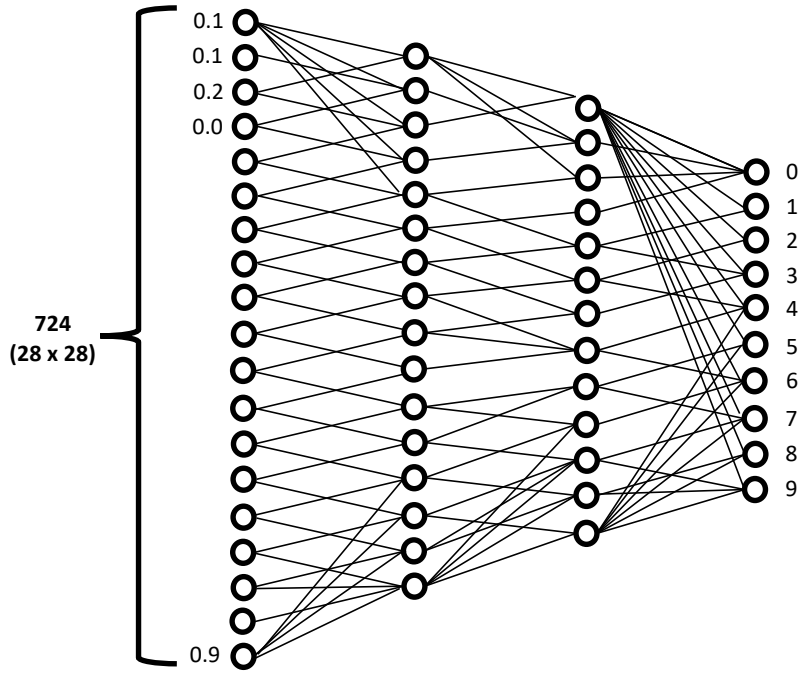

Figure 11.5: Neural Network

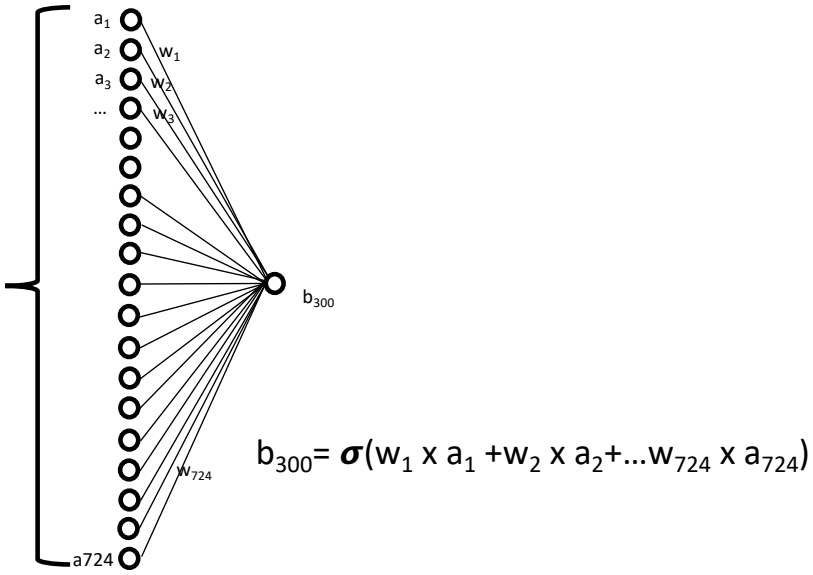

Figure 11.6: fig:NN4

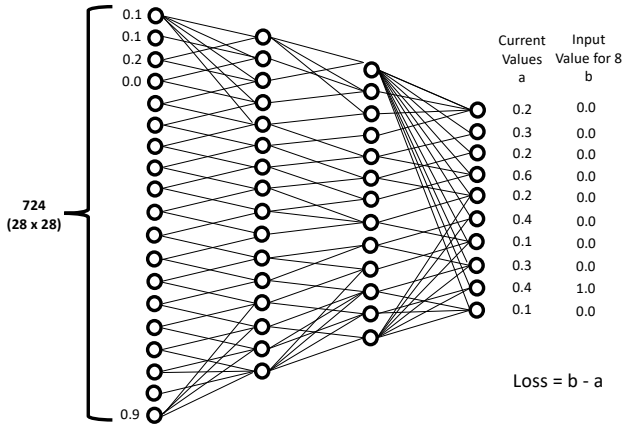

Figure 11.7: fig:NN5

11.3 Using Neural Networks in Keras

Until recently, to implement a neural networks was very cumbersome. Fortunately we have the Keras library. Figure **??** shows the code that does this.

Exercises

1. Run the above code. Change some of the parameters and observe the changes
2. Use the UK home broadband data to predict what the speed of a house will be. https://data.gov.uk/dataset/dfe843da-06ca-4680-9ba0-fbb27319e402/uk-home-broadband-performance

```
1  import keras
2  from keras.datasets import mnist
3  from keras.models import Sequential
4  from keras.layers import Dense, Dropout
5  from keras.optimizers import RMSprop
6
7  # The batch size is the number of samples
8  batch_size = 128
9  # There are 10 possible digits
10 num_classes = 10
11 # Epochs is the number of times the training set is used
12 epochs = 20
13
14 # the data, split between train and test sets
15 (x_train, y_train), (x_test, y_test) = mnist.load_data()
16
17
18 # Each image in the MNIST dataset has 28*28 = 74 pixels.
19 # We reshape this 28x28 matrix into a 784 array
20 x_train = x_train.reshape(60000, 784)
21 x_test = x_test.reshape(10000, 784)
22 x_train = x_train.astype('float32')
23 x_test = x_test.astype('float32')
24
25 # The RGB values are between 0 and 255,
26 # and we want to input values between 0 and 1.
27 x_train /= 255
28 x_test /= 255
29 print(x_train.shape[0], 'train samples')
30 print(x_test.shape[0], 'test samples')
31
32 # convert class vectors to binary class matrices
33 # e.g instead of "8" we want [0,0,0,0,0,0,0,0,1,0]
34 y_train = keras.utils.to_categorical(y_train, num_classes)
35 y_test = keras.utils.to_categorical(y_test, num_classes)
36
37 # We initialize an empty Sequential model
38 model = Sequential()
39
40 # And then sequentially add new layers.
41 # A Dense layer is the one we covered this chapter,
42 # where a neuron connects to all the neurons in the,
43 # following layer.
44 # For each layer, we have to specify the activation,
45 # function and the output size. In the first layer,
46 # we also have to specify the input shape.
47 model.add(Dense(512, activation='relu', input_shape=(784,)))
48
49 # Dropout is a regularization technique (to prevent) overfitting
50 model.add(Dropout(0.2))
51 model.add(Dense(512, activation='relu'))
52 model.add(Dropout(0.2))
```

```
50 model.add(Dense(num_classes, activation='softmax'))
51
52 model.summary()
53 # Once the neural network structure is set we compile it.
54 # That means associate a loss function and an optimzer with it.
55 model.compile(loss='categorical_crossentropy',
56              optimizer=RMSprop(),
57              metrics=['accuracy'])
58
59 # After the network is compiled we can train it, using our
60 # training set.
61 history = model.fit(x_train, y_train,
62                     batch_size=batch_size,
63                     epochs=epochs,
64                     verbose=1,
65                     validation_data=(x_test, y_test))
66
67 #Finally, we check the performance of the model
68 # in the test set
69 score = model.evaluate(x_test, y_test, verbose=0)
70 print('Test loss:', score[0])
71 print('Test accuracy:', score[1])
72
```

Figure 11.8: MNIST in Keras