

Useful Numpy Functions

```
In [ ]: import numpy as np
result = np.dot(3, 4)
print(result)
```

12

```
In [ ]: import numpy as np
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])
result = np.dot(a, b)
print(result)
```

32

```
In [30]: import numpy as np
a = np.array([[1, 2], [3, 4]])
b = np.array([[5, 6], [7, 8]])
result = np.dot(a, b)
print(result)
```

```
[[19 22]
 [43 50]]
```

```
In [31]: print(a)
print(a.T)
```

```
[[1 2]
 [3 4]]
[[1 3]
 [2 4]]
```

```
In [32]: #np.maximum(x1, x2)
print(np.maximum(5,10))
np.maximum(a,b)
```

10

```
Out[32]: array([[5, 6],
               [7, 8]])
```

```
In [34]: #It compares element wise
a = np.array([[6, 2], [7, 8]])
b = np.array([[5, 6], [12, 4]])
np.maximum(a,b)
```

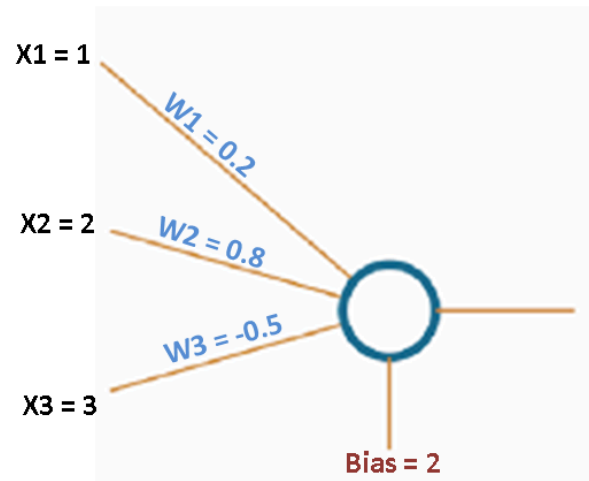
```
Out[34]: array([[ 6,  6],
               [12,  8]])
```

```
In [36]: print(np.exp(1))
print(np.exp(0))
print(np.exp(-1))
```

```
2.718281828459045
1.0
0.36787944117144233
```

A Single Neuron:

Example #1:



```
In [ ]: #The input here is a list of 3 features (1 sample).
inputs = [1, 2, 3]

#Each connection between neurons has a weight associated with it, which is a tra
weights = [0.2, 0.8, -0.5]

#The purpose of the bias is to offset the output positively or negatively
bias = 2

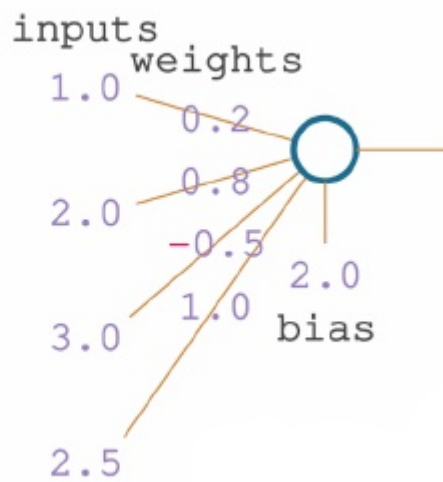
# the hypotheses  $H(x) = W \cdot X + \text{bias}$ 
outputs = inputs[0]*weights[0] + inputs[1]*weights[1] + inputs[2]*weights[2] + b
print(outputs)
```

2.3

```
In [ ]: #Solve the problem using Numpy
#       Write your code here
```

A Single Neuron (4 inputs):

Example #2:



```
In [2]: inputs = [ 1.0 , 2.0 , 3.0 , 2.5 ]
weights = [ 0.2 , 0.8 , - 0.5 , 1.0 ]

bias = 2.0

output = (inputs[0]*weights[0] +
          inputs[1]*weights[1] +
          inputs[2]*weights[2] +
          inputs[3]*weights[3] + bias)

print (output)
```

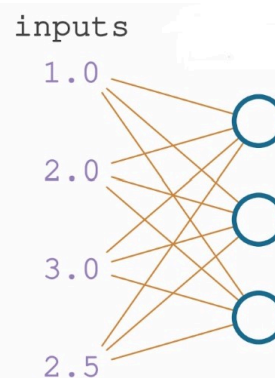
4.8

```
In [ ]: #Solve the problem using Numpy
#       Write your code here
```

A Layer of Neurons

Example #3:

here there are 4 inputs in the input layer. and 3 neurons in the output layer.



```
In [8]: inputs = [ 1, 2, 3, 2.5 ]

# each neuron has a list of weights:
weights1 = [ 0.2, 0.8, - 0.5, 1 ]
weights2 = [ 0.5, - 0.91, 0.26, - 0.5 ]
weights3 = [ - 0.26, - 0.27, 0.17, 0.87 ]

# each neuron has a bias:
bias1 = 2
bias2 = 3
bias3 = 0.5
```

```
In [9]: # here the output are 3 neurons, so it will be a list of three values.
outputs = [

    # Neuron 1:
    inputs[0] * weights1[0] + inputs[1] * weights1[1] + inputs[2] *

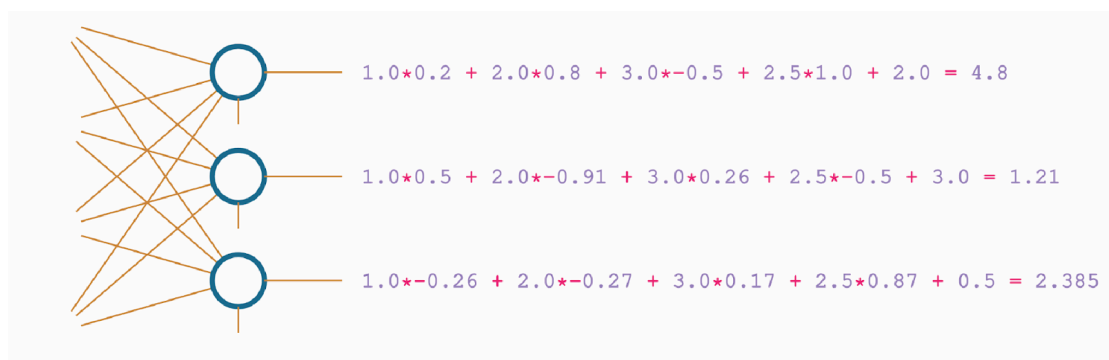
    # Neuron 2:
    inputs[0] * weights2[0] + inputs[1] * weights2[1] + inputs[2] *

    # Neuron 3:
    inputs[0] * weights3[0] + inputs[1] * weights3[1] + inputs[2] *

]
```

```
In [10]: outputs
```

```
Out[10]: [4.8, 1.21, 2.385]
```



```
In [ ]: #Solve the problem using Numpy
#       Write your code here
```

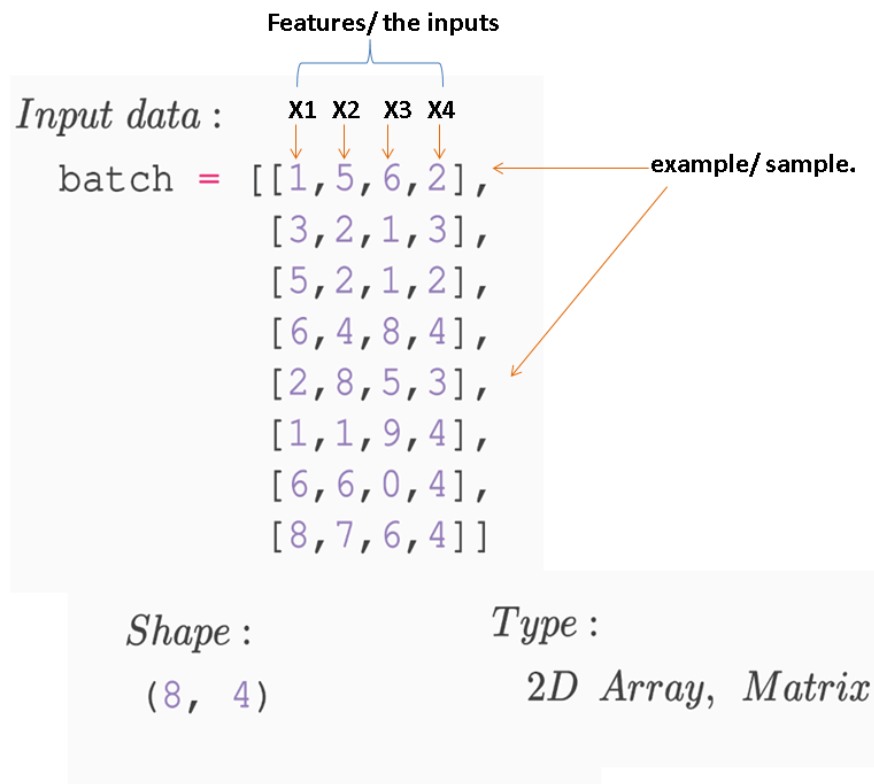
A Batch of Data

Often, neural networks expect to take in many samples at a time for two reasons. One reason is that it's faster to train in batches in parallel processing, and the other reason is that batches help with generalization during training.

If you fit (perform a step of a training process) on one sample at a time, you're highly likely to keep fitting to that individual sample, rather than slowly producing general

tweaks to weights and biases that fit the entire dataset. Fitting or training in batches gives you a higher chance of making more meaningful changes to weights and biases.

An example of a batch of data could look like:



For example, every sample is a describing for a house.

So: X1: the area of this house. X2: number of rooms. ... And so on.

```
In [3]: import numpy as np

# Batch of 3 examples:
inputs = [
    [ 1.0 , 2.0 , 3.0 , 2.5 ],
    [ 2.0 , 5.0 , - 1.0 , 2.0 ],
    [ - 1.5 , 2.7 , 3.3 , - 0.8 ]
]
weights = [
    [ 0.2 , 0.8 , - 0.5 , 1.0 ],
    [ 0.5 , - 0.91 , 0.26 , - 0.5 ],
    [ - 0.26 , - 0.27 , 0.17 , 0.87 ]
]
biases = [ 2.0 , 3.0 , 0.5 ]

outputs = np.dot(inputs, np.array(weights).T) + biases
print(outputs)

[[ 4.8    1.21  2.385]
 [ 8.9   -1.81  0.2 ]
 [ 1.41  1.051  0.026]]
```

As you can see, our neural network takes in a group of samples (inputs) and outputs a group of predictions.

Why we use W.T (Transported of weights array)?

```
inputs = [[1.0, 2.0, 3.0, 2.5],  
          [2.0, 5.0, -1.0, 2.0],  
          [-1.5, 2.7, 3.3, -0.8]]  
weights = [[0.2, 0.8, -0.5, 1.0],  
            [0.5, -0.91, 0.26, -0.5],  
            [-0.26, -0.27, 0.17, 0.87]]
```

Inputs – Batch


$$\begin{bmatrix} 1.0 & 2.0 & 3.0 & 2.5 \\ 2.0 & 5.0 & -1.0 & 2.0 \\ -1.5 & 2.7 & 3.3 & -0.8 \end{bmatrix}$$

(3, 4)
Matrix

Weights

$$\begin{bmatrix} 0.2 & 0.8 & -0.5 & 1.0 \\ 0.5 & -0.91 & 0.26 & -0.5 \\ -0.26 & -0.27 & 0.17 & 0.87 \end{bmatrix}$$

(3, 4)
Matrix

Inputs – Batch


$$\begin{bmatrix} 1.0 & 2.0 & 3.0 & 2.5 \\ 2.0 & 5.0 & -1.0 & 2.0 \\ -1.5 & 2.7 & 3.3 & -0.8 \end{bmatrix}$$

(3, 4)
Matrix

Weights

$$\begin{bmatrix} 0.2 & 0.5 & -0.26 \\ 0.8 & -0.91 & -0.27 \\ -0.5 & 0.26 & 0.17 \\ 1.0 & -0.5 & 0.87 \end{bmatrix}$$

(4, 3)
Matrix

we need to perform the dot product of each input and each weight set in all of their combinations. The dot product takes the row from the first array and the column from the second one, but currently the data in both arrays are row-aligned. Transposing the second array shapes the data to be column-aligned. The matrix product of inputs and transposed weights will result in a matrix containing all atomic dot products that we need to calculate. The resulting matrix consists of outputs of all neurons after operations performed on each input sample:

$$\begin{bmatrix} 1.0 & 2.0 & 3.0 & 2.5 \\ 2.0 & 5.0 & -1.0 & 2.0 \\ -1.5 & 2.7 & 3.3 & -0.8 \end{bmatrix} \begin{bmatrix} 0.2 & 0.5 & -0.26 \\ 0.8 & -0.91 & -0.27 \\ -0.5 & 0.26 & 0.17 \\ 1.0 & -0.5 & 0.87 \end{bmatrix} = \begin{bmatrix} 2.8 & -1.79 & 1.885 \\ 6.9 & -4.81 & -0.3 \\ -0.59 & -1.949 & -0.474 \end{bmatrix}$$

$$\begin{bmatrix} 2.8 & -1.79 & 1.885 \\ 6.9 & -4.81 & -0.3 \\ -0.59 & -1.949 & -0.474 \end{bmatrix} + \begin{bmatrix} 2.0 & 3.0 & 0.5 \end{bmatrix} = \begin{bmatrix} 4.8 & 1.21 & 2.385 \\ 8.9 & -1.81 & 0.2 \\ 1.41 & 1.051 & 0.026 \end{bmatrix}$$

```
In [ ]: def relu(x):
        """
        ReLU (Rectified Linear Unit) activation function

        Args:
            x: Input array (numpy array or scalar)

        Returns:
            Element-wise ReLU activation: max(0, x)
        """
        # Write your code here
        return
```

```
In [ ]: def sigmoid(x):
        """
        Sigmoid activation function

        Args:
            x: Input array (numpy array or scalar)

        Returns:
            Element-wise sigmoid activation: 1 / (1 + exp(-x))
        """
        # Write your code here
        return
```

```
In [ ]: test_inputs = np.array([-2, -1, 0, 1, 2])
print("Input:", test_inputs)
print("ReLU:", relu(test_inputs))
print("Sigmoid:", sigmoid(test_inputs))
```