Chapter 01| Simple Artificial Neural Network Architecture

~~~~~~

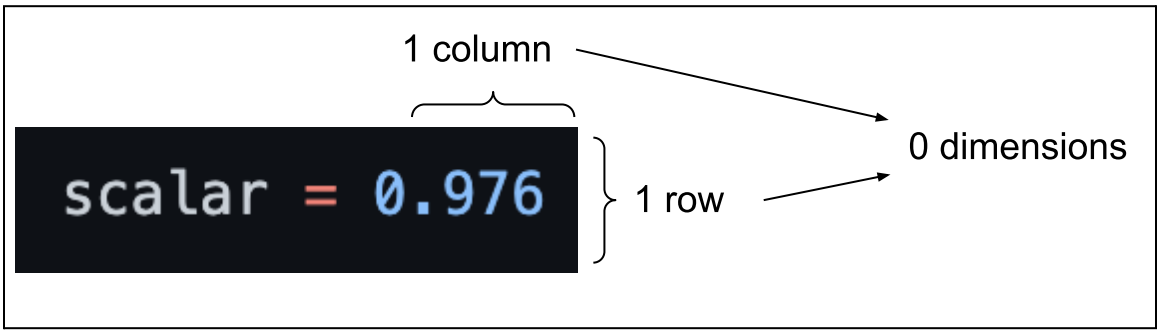
Scalars, Vectors, and Matrices

Before we learn about what happens in the cell body of an artificial neuron, we need to learn the different ways we use to store data.

Name	Python Name	Dimensions	How to Describe
Scalar	Int (Integer), Float (decimal), any other type to store 1 value	0	NA
Vector	List	1	# of elements
Matrix	List of Lists (Vector of Vectors)	2	(# of rows, # of columns)

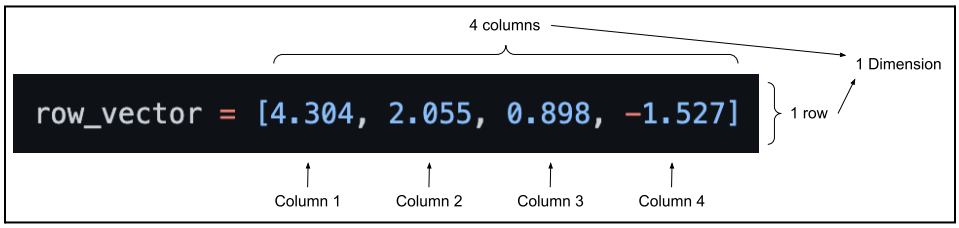
A note on storing data: when there is only 1 row or 1 column of the object, they are insignificant to the number of dimensions that object has. If this does not make sense, below are examples of a scalar, vector, and matrix to demonstrate what I mean.

Scalar in Python Example: This scalar is a float.



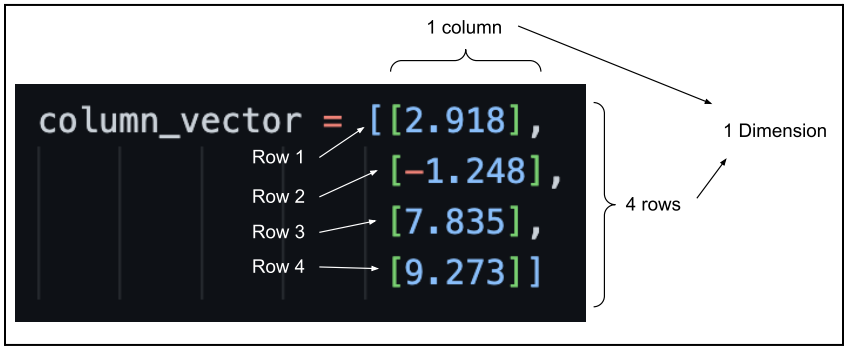
It has **zero dimensions** since there is only 1 row and 1 column so the row and column dimensions are insignificant to the number of dimensions this scalar has.
~~~~~~

Row Vector in Python Example: This is a 4-element vector (list containing 4 elements).



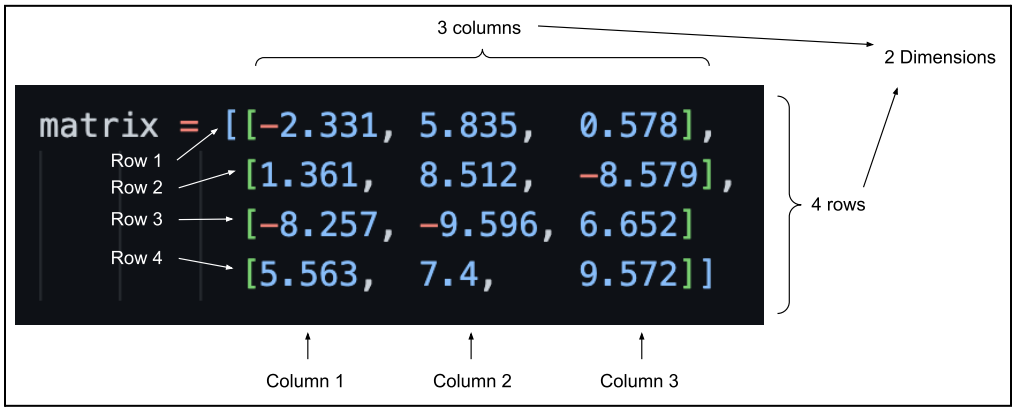
This vector has **one dimension** since there is only 1 row, which makes the row dimension insignificant. Because it has **1 row**, we call this a **4-element row vector**.

Column Vector in Python Example: This is another 4-element vector.



It also has **one dimension** since there is only 1 column, which makes the column dimension insignificant (I hope you're seeing the pattern). Since this vector has only **1 column**, we call this a **4-element column vector**. There is a bracket for each row to differentiate column vectors from a row vector (which can be manipulated to resemble a column vector).

Matrix in Python Example: This is a (4, 3) or 4 by 3 matrix.



Matrices have **two dimensions** since the number of both rows and columns is not 1. Since there are 4 rows and 3 columns in this particular example, we call this a (4, 3) or

4 by 3 matrix. The number of rows comes first, followed by the number of columns. As you can see, a matrix can be viewed as a list of lists or a vector of vectors.

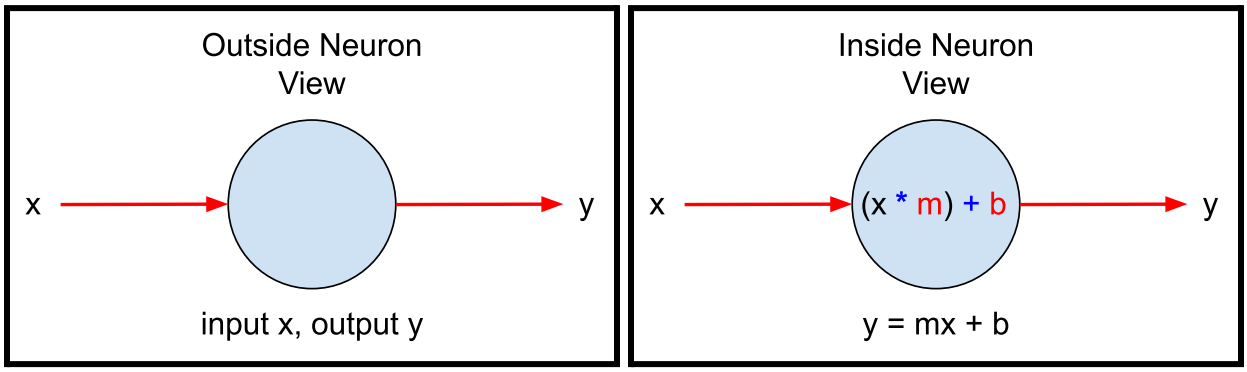
As if to better define the phrases "list of lists" and "vector of vectors", each row has an open and close bracket, signifying a list/vector. This matrix has a total of 4 vectors. When we add an open and close bracket to the outside of all 4 vectors, we get another vector. Since this large vector contains all 4 vectors inside, it is a "vector of vectors".

Keep scalers, vectors, and matrices in mind as we now go on to understand what happens inside the cell body of a neuron.

~~~~~~

What "goes on" inside one Artificial Neuron?

Let's start off by comparing the outside view of a neuron to the inside view given only 1 input feature.

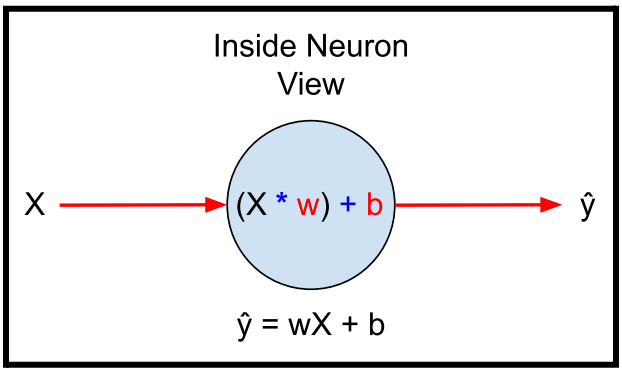


As you can see, what happens inside the cell body of an artificial neuron given one input feature is literally **y = mx + b**! When reading the equation inside the neuron, we get: "x is **multiplied by m** and that product is **increased by b**". By putting x as the first factor instead of m, we get to **imply** that the **input** is being **changed/modified** to produce the output, rather than getting random outputs. It's not a bug, it's a feature!

The problem is, we have **slightly** simplified the inside view. The only differences are the variables used (x, m, b, and y). On the next page is a table that converts each variable used in math to AI.
~~~~~~

Variable	Math Meaning	AI Conversion
x	input	Denoted with letter **X**, meaning **input to the neuron**
m	slope	Denoted with letter **w**, which is the **weight** variable
b	y-intercept	Denoted with letter **b**, which is the **bias** variable
y	output	Denoted with **ŷ (y_hat)**, meaning the **neuron's prediction**

Now here is the actual inside view given only 1 input feature:

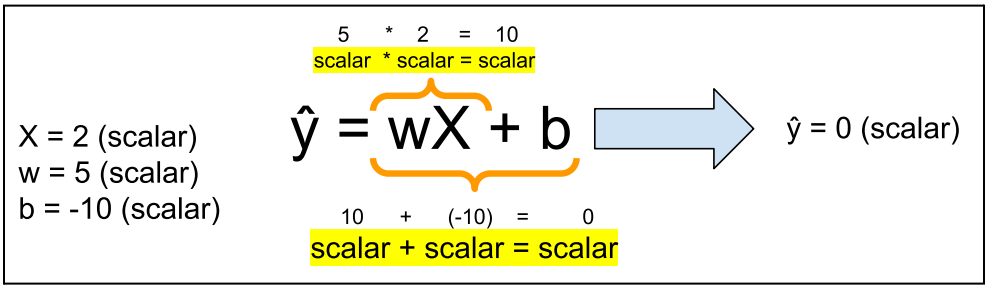


We cannot use the variable "y" for the prediction/output of the neuron because "y" is used to store the correct answer.

The prediction of the model can be either accurate or inaccurate, so to emphasize this characteristic, we give the variable "y" a hat.

Where do the variables "w" and "b" come from? We call the **weights** and **biases** of a neuron its **parameters**. At the beginning when we first define the model, we set all the parameters of every neuron as random values. Since all the parameters of the neurons are random, all the neurons are "dumb" and so is the brain. Anything the model predicts is random, so the predictions are useless. However, to make the brain **"smarter"** during training, the model will **modify** all the parameters of the model.

Now that we understand the variables of our equation for the computation of one neuron given one input feature, let's analyze our equation with random values:



Since we only have one weight value, "w" is a scalar. Similarly, we only have one bias and one input so "b" and "X" are scalars. Also, y_hat **should be** a scalar because we can only get one output value from one output neuron.

Great! Here is the code for one neuron given one input feature: (output should be 0)

```
X = 2
def one_neuron(X):    # one input feature
    w = 5
    b = -10
    return (w * X) + b

print(one_neuron(X))
```

The weight and bias of the neuron are defined inside the function because they are part of the neuron (they are the parameters of the neuron).

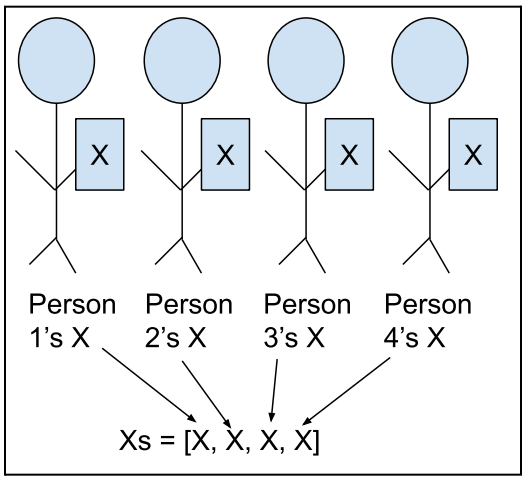
Now, what if multiple people wanted to use your one neuron that only takes in one input feature to predict something? This means that each person would have one input feature. If there were 4 people, person 1 has an X, person 2 has an X, person 3 has an X, and person 4 has an X. This also means we should get four predictions, one for each person. Here is the code:

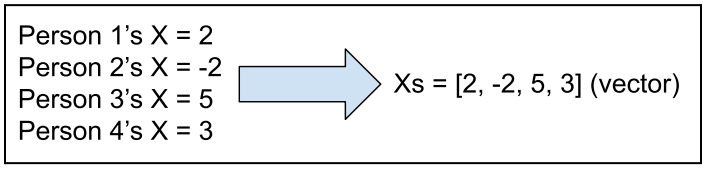
```
person1_X = 2
person2_X = -2
person3_X = 5
person4_X = 3
def one_neuron(X):    # one input feature
    w = 5
    b = -10
    return (w * X) + b

print(one_neuron(person1_X))
print(one_neuron(person2_X))
print(one_neuron(person3_X))
print(one_neuron(person4_X))
```

The output should be:
0
-20
15
5

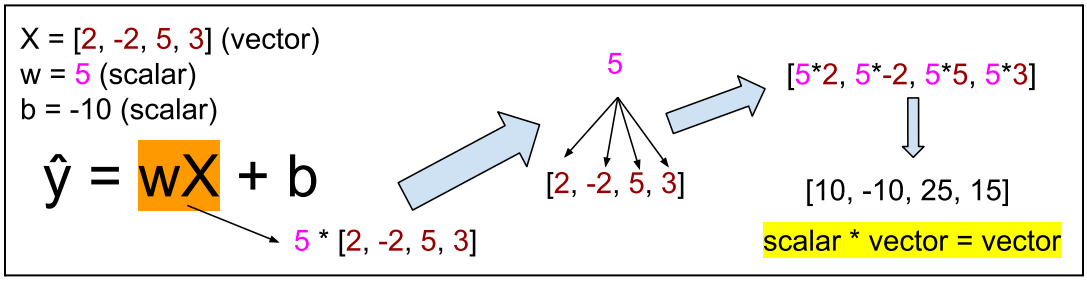
Even though the code works, it is not efficient in generating predictions for large numbers of people. Instead of calling our `one_neuron` function multiple times, we can create a **vector** called Xs which contains all the one input features of each person.



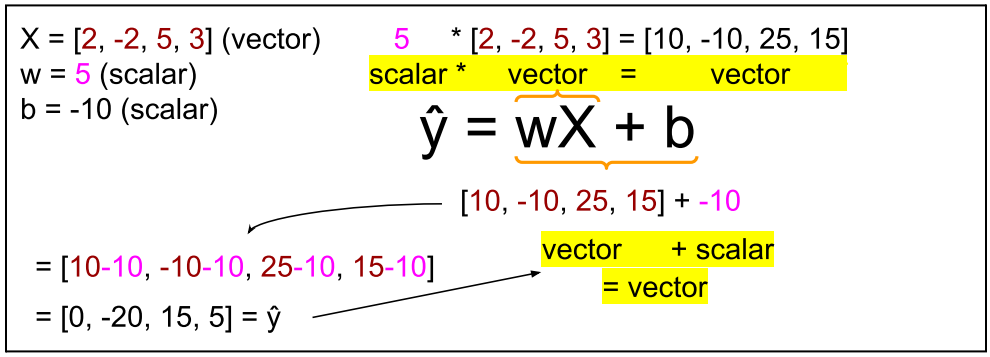


By using a vector, our code can run more efficiently for large numbers of people. When this is the case, we call our code **vectorized** and the process of using a vector to contain the inputs of multiple people is called **vectorization**.

In AI, if we have multiple scenarios (each person has a different scenario) we call them **examples**. In our situation, we have 4 examples. Let's see what happens to our equation with X in a vector instead of a scalar:



Looking at only the wX part of the equation we see that to multiply a scalar (w) by a vector (X) we need to distribute the scalar to every element of the vector, resulting in a vector. Here is the rest of the equation:



As you can see, just like when multiplying a scalar by a vector, we distribute the scalar to each of the elements of the vector when adding a vector to a scalar (with the only difference being, of course, that we add instead of multiply). In the end, the prediction of the neuron is a 4-element vector, one element for each example (one prediction for each person).

Here is the code for multiple examples (output should be [0, -20, 15, 5]):

```
import numpy as np
Xs = np.array([2, -2, 5, 3])    # 4 examples, vector
def one_neuron(Xs):    # one input feature, multiple examples
    w = 5
    b = -10
    return (w * Xs) + b

print(one_neuron(Xs))
```

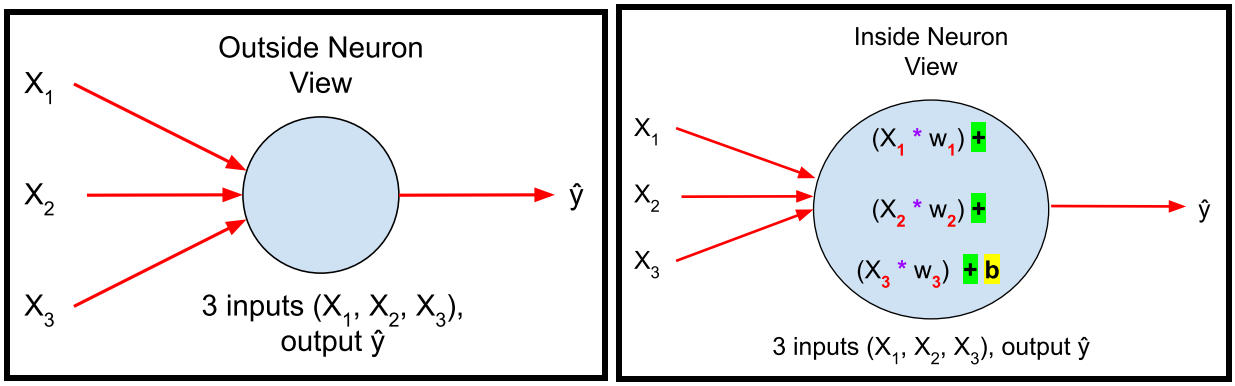
As you may already know, multiplying a list by an integer, float, or any type of scalar in Python does not give the same properties of a vector multiplied by a scalar in mathematics.

To combat this problem, we can use the 3rd party library NumPy which allows a vector or matrix to be defined as a NumPy array using `np.array`. The initialization of a NumPy array only asks for the list/list of lists to convert. If we input a list, we get a NumPy array which we can think of as a vector and if we input a list of lists, we get a NumPy array which we can think of as a matrix.

The reason NumPy is used is because when we multiply a NumPy array by a scalar or another NumPy array, the properties of the operations defined by math are applied instead of the properties of the operations defined by Python.

Here is the pip install link for NumPy: https://pypi.org/project/numpy/
For reference, this book uses **NumPy 1.21.2.**

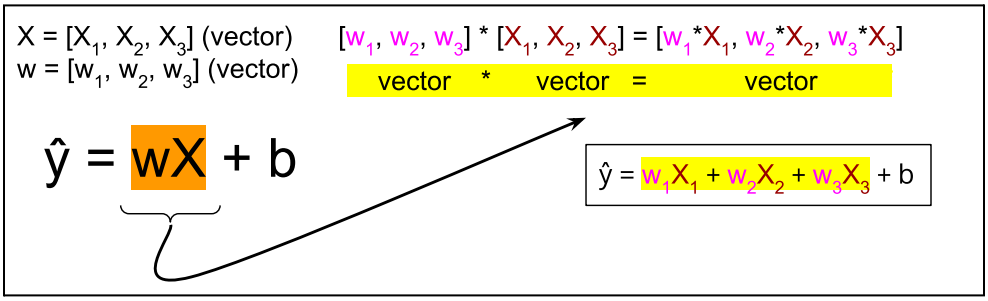
Now that we understand how an artificial neuron works given one input feature, let's take a look at an artificial neuron given multiple input features:



This neuron receives 3 input features and this is its equation: $\hat{y} = w_1X_1 + w_2X_2 + w_3X_3 + b$. As you can see 3 input features result in 3 weights but one bias. It might seem strange that there are more weights than biases, but in reality, there is no point to have more than one bias in a neuron: $\hat{y} = w_1X_1 + w_2X_2 + w_3X_3 + b_1 + b_2 + b_3$; if $b = b_1 + b_2 + b_3$, then: $\hat{y} = w_1X_1 + w_2X_2 + w_3X_3 + b$.

As you can see, having multiple biases does nothing but add more parameters to the model. Setting one bias to the sum of all the biases of the neuron achieves the same result. The key takeaway is that an artificial neuron only has 1 bias and the number of weights is equal to the number of input features. So if there are *n* input features to the artificial neuron, the artificial neuron's computation is: $\hat{y} = w_1X_1 + w_2X_2 + \ldots + w_nX_n + b$.

If we put each weight into a vector and each input feature into a vector, the equation for an artificial neuron is actually ŷ = sum(wX) + b. Below is a diagram that visualizes how this is the case.



When we multiply a vector of weights by a vector of input features, we get another vector that multiplies each weight value with its corresponding input feature. Then, we sum up all the elements of that resulting vector which will result in a scalar. That scalar added to the bias value (a scalar) results in another scalar which is the neuron's prediction ŷ.

Instead of that messy equation, we now have ŷ = sum(wX) + b. Thus for an artificial neuron (no matter how many input features that neuron receives), it will always output a scalar for each example.

Here is the code for one neuron with multiple input features but only one example:

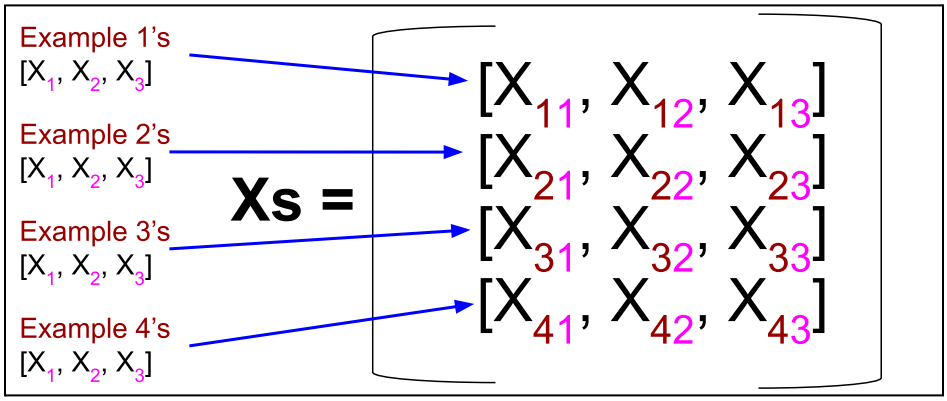
```
import numpy as np
X = np.array([2, 0, -7])    # 3 input features, vector
def one_neuron(X):    # multiple input features
    w = np.array([5, -2, -10])
    b = -10
    return np.sum(w * X) + b

print(one_neuron(X))
```

Since X and w are now vectors, we need to initialize them as NumPy arrays so that vector multiplication will work exactly as defined by mathematics. To sum up all the elements of a vector, we can use `np.sum`. The output is 70 and to confirm that this is the correct answer, you can always hand-calculate the math:
ŷ = sum(wX) + b = sum([5, -2, -10] * [2, 0, -7]) + (-10) = sum([10, 0, 70]) - 10 = 80 - 10 = 70

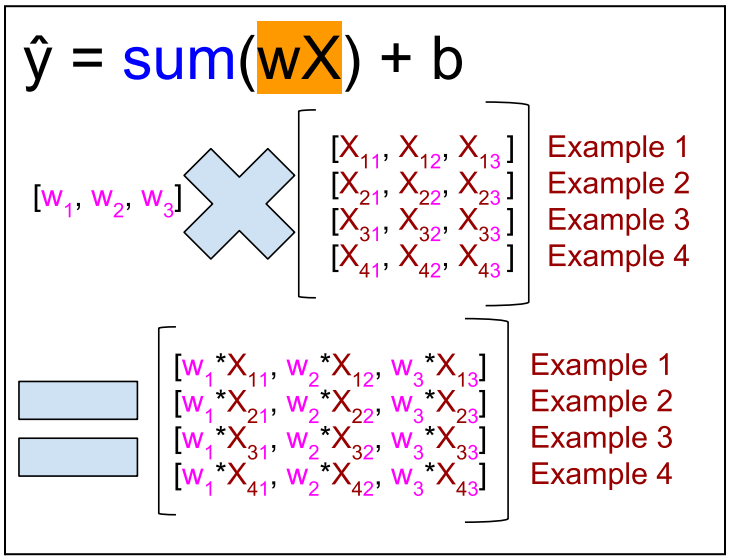
So that was the calculation for one artificial neuron with multiple input features but only one example. Let's scale up to calculate a neuron's prediction of multiple examples.

Each example is a vector containing its input features. To combine each example together into one variable, we need to use a matrix:



Each example takes up a row in the matrix and each input feature of each example takes up a column in the matrix. Since there are 4 examples and each example has 3 input features we have 4 rows and 3 columns, making this a (4, 3) or 4 by 3 matrix. Each element in this matrix is differentiated by its example/row number followed by its input feature/column number. For instance, example 1's 2nd input feature is X_{12}.

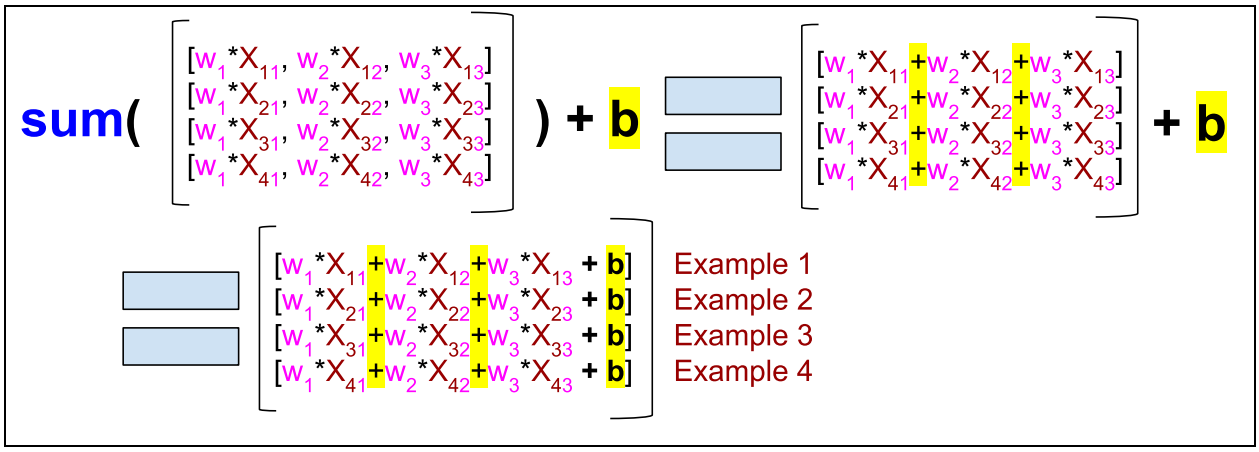
We have 4 examples so $\hat{y}$ should be a 4-element vector (one prediction for each example). Below is a diagram that shows the new interactions from using a matrix of inputs and a vector of weights:



As you can see, multiplying a vector by a matrix results in a matrix. For each example, each weight value is still multiplied by its corresponding input feature.

Continuing on, do we add each element of the matrix? No, because doing so would result in a scalar even though we have 4 examples. Instead, for each example's row vector, we have to add/sum up each element.

By doing so, we would get a column vector of 4 examples:



The last step is to add the bias (a scalar). We know that when adding a vector to a scalar, we distribute the scalar to each element of the vector, resulting in a vector. As you can see, with multiple examples, the calculation for each example is the same but by using **vectorization**, we do not have to change the equation.

Here is the code for a neuron receiving multiple input features with multiple examples:

```
Xs = np.array([[2, 0, -7],
               [-2, 9, 0],
               [5, 1, -1],
               [3, 0, -4]])
def one_neuron(Xs):     # multiple input features, multiple examples
    w = np.array([5, -2, -10])
    b = -10
    return np.dot(Xs, w.T) + b

print(one_neuron(Xs))
```

The output should be [70, -38, 23, 45]. For multiple examples, the sum(wX) part of the equation is called a **dot product**. In addition to multiplying the two arrays together like normal, the dot product will add/sum up each element of each row's vector. This is a plus since we have to write less code and just use NumPy's `np.dot` function.

However, we cannot use the dot product **blindly**. Unlike multiplication which is commutative (a * b = b * a ; order does not matter), a dot product is **not commutative** (order matters). Here is the rule for a dot product: The 1st array must have a **shape** (# of rows, # of columns) of (a, b) and the 2nd array must have a shape of (b, c). The resulting shape of the dot product will be (a, c).

The shape of "Xs" is (4, 3) – (4 rows, 3 columns). The shape of "w" can be thought of as (1, 3) – (1 row, 3 columns). For a dot product to work the **second value** of the 1st array's (Xs) shape must equal the **first value** of the 2nd array's (w) shape. We see that the **common value** in both shapes of the array is 3, but the problem is that the **first value** of the 2nd array's shape is 1 while the **second value** of the 2nd array's shape is 3.

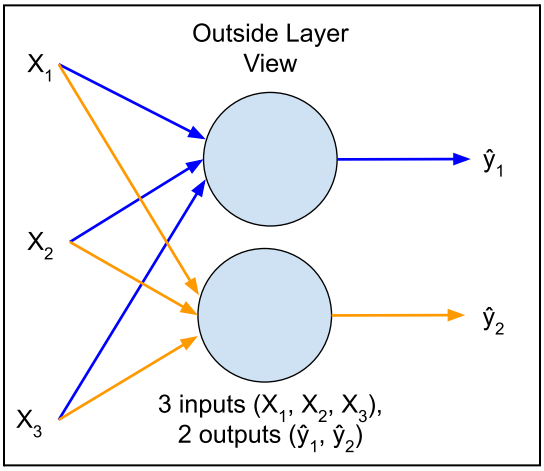
To fix this, we can **transpose** the 2nd (w) array. Transposing means that we swap the **first value** of the array's shape with the **second value** of the array. To get the transposed array, we can use the `.T` attribute on the "w" array. Now the second array to the `np.dot` function has shape (3, 1) which means the dot product can now work correctly. The result of the dot product Xs (4, 3) * w (3, 1) will be (4, 1). The (4, 1) shape makes sense because we have 4 examples and each example requires one prediction.

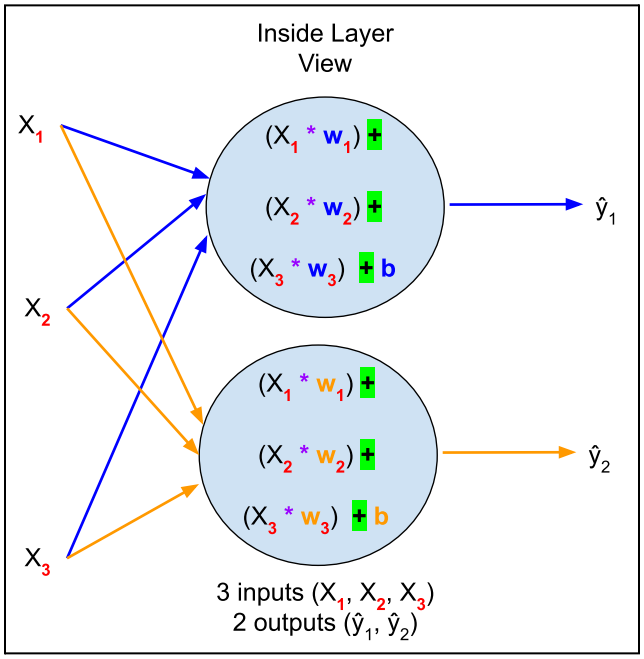
That is it! That is what "goes on" inside one artificial neuron given multiple input features and examples. In the next section, we will simulate a **layer of neurons**.

~~~~~~

What "goes on" inside one layer of Artificial Neurons?

Let's take a look at the outside and inside view of a layer of artificial neurons:

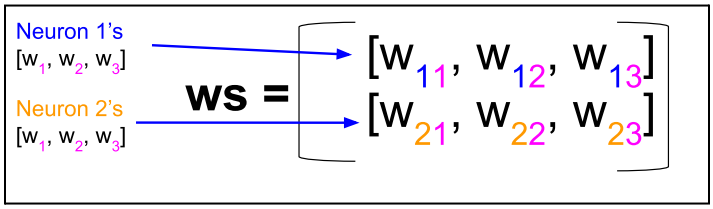


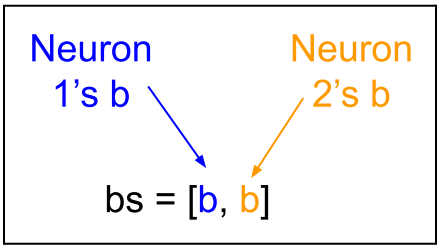


Each neuron in a layer receives all input features. They receive the same 3 input features and output one value for each example.
~~~~~~

As you can see by the color-coding of the arrows connecting each input feature to the neuron, each neuron has their **own set of weights** (one weight for each input feature) and each neuron has their own bias value. Since each neuron's parameters hold different values, each neuron's corresponding output (prediction) will also hold different values. This allows each neuron to have a unique purpose to the prediction of the model. Next, let's combine all the weights of this layer and all the biases of this layer into a variable (ws, for the weights) and into another variable (b, for the biases).

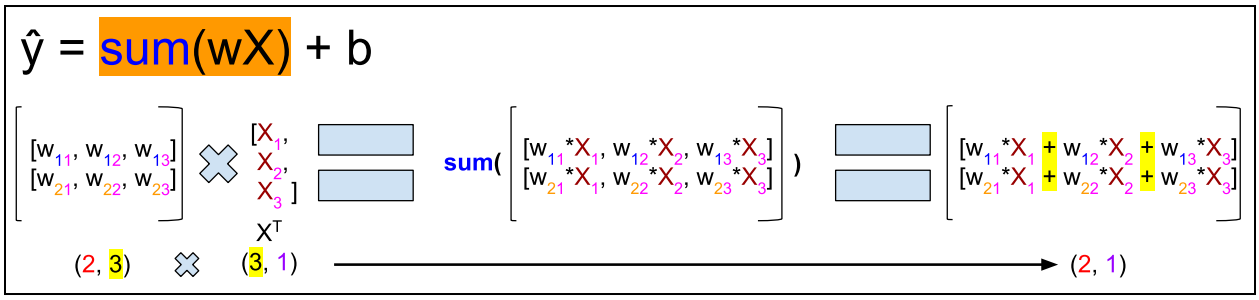
Similar to having multiple examples (**each example's vector** of **input features** is **contained in a vector** to get a matrix), by having multiple neurons, **each neuron's vector** of **weights** will be **contained in a vector** to get a matrix. Furthermore, similar to having multiple input features (**each input feature** will be **contained in a vector**), by having multiple neurons, **each neuron's bias** will be **contained in a vector**. Here are visuals to understand what we are doing:



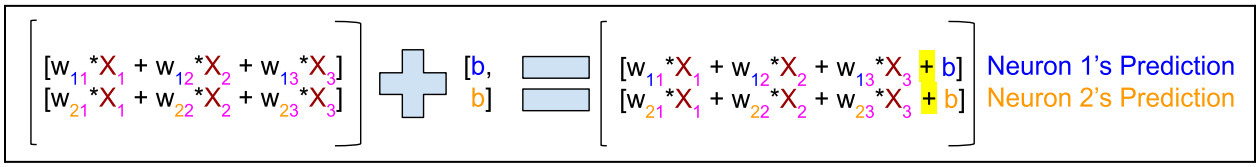


We get a matrix of weights with the **number of rows equal to the number of neurons** and the **number of columns equal to the number of input features**. In this case, since we have 2 neurons, each receiving 3 input features, we have a (2, 3) matrix.
We also get a vector of biases with the **number of elements equal to the number of neurons**. In this case since we have 2 neurons, we have a 2-element vector.

Now let us see the new interactions between an input vector (one example, multiple input features), a matrix of weight values and a vector of bias values. Since we have two neurons, the output (ŷ) should be a 2-element vector (one output for each neuron).



For the dot product to work, we have to transpose the inputs vector (X). The superscript "T" stands for "transposed". Each neuron's weight values are multiplied by their corresponding input values and the dot product always sums the elements of each row vector. Lastly, we can add the bias vector to the result of the dot product:



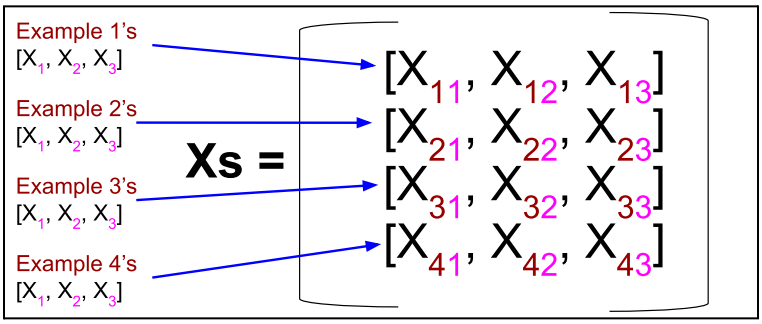
The first neuron's bias is added to the first row and the second neuron's bias is added to the second row. By putting each neuron's weight vector into a matrix and their bias into a vector we have **vectorized** the calculation. The calculation to get each neuron's prediction is still the same. The result is a 2-element vector, so we have successfully completed the calculation.

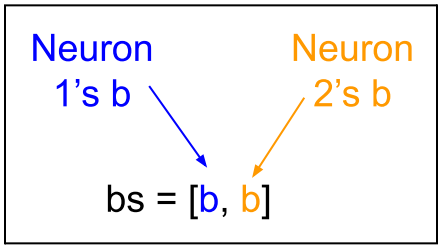
Here is the code (output should be [70, 42]):

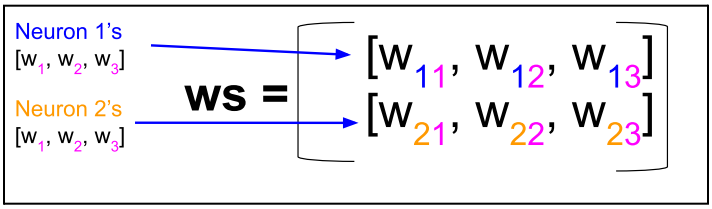
```
import numpy as np
X = np.array([2, 0, -7])    # 3 input features, vector
def multiple_neurons(X):    # multiple input features
    w = np.array([[5, -2, -10],    # neuron 1's weights
                  [0, 9, -5]])    # neuron 2's weights
    b = np.array([-10, 7])    # 2 biases, one for each neuron, vector
    return np.dot(w, X.T) + b

print(multiple_neurons(X))
```

Now that we understand how to simulate the calculation of a layer of neurons given one example, let's scale up to simulate the calculation of a layer given multiple examples! Just to review, X and w are matrices while b is a vector:







X has shape (4, 3) – 4 examples/rows, 3 input features/columns

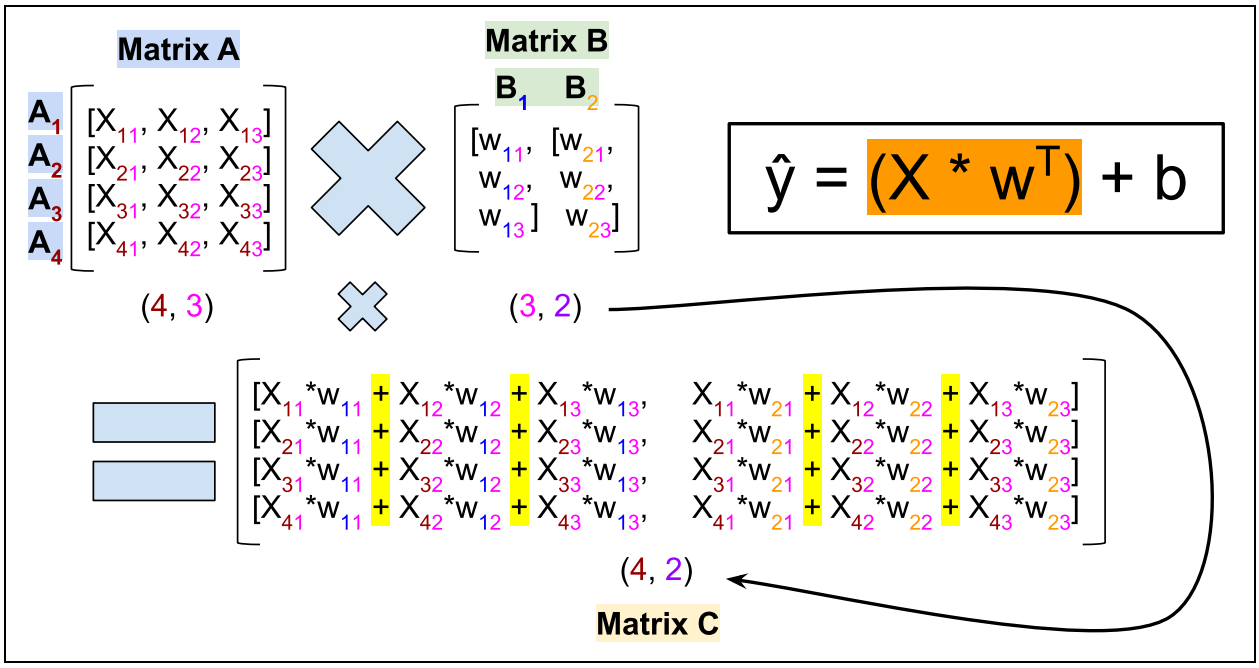
w has shape (2, 3) – 2 neurons/rows, 3 input features/columns

For multiple examples, we have 2 options to do the dot product:
Option 1 – $X * w^T$: (4, 3) * (3, 2) → (4, 2) ; Option 2 – $w * X^T$: (2, 3) * (3, 4) → (2, 4)

So which option do we choose? The first option results in a matrix with a shape that is more intuitive to understand since the number of rows equals the number of examples and the number of columns equals the number of neurons. Assume that go with option 1 where the output is a (4, 2) matrix, we are saying that each example (4 examples total) has a prediction from each neuron (2 neurons total).

With option 2, however, this would be reversed: each neuron has 4 predictions, which sounds unnatural. This is mainly due to the fact that it is by convention to start with the meaning of each row followed by the meaning of each column. To keep everything intuitive, we are going to use option 1 for the dot product.

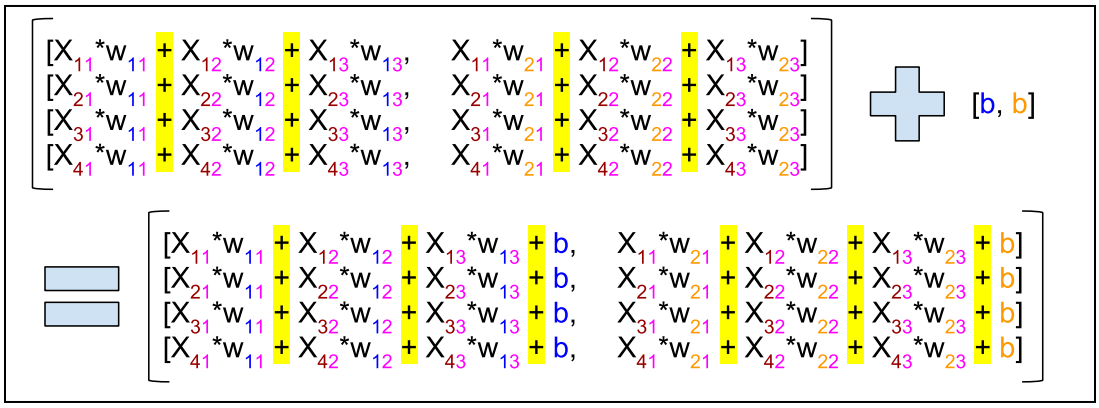
Now let's understand how a dot product between two matrices works:



To understand what is happening, let's say the 1st matrix (X) is matrix A, the 2nd matrix (w^T) is matrix B and the result of the dot product is matrix C. Given any row, i, and any column, j, $C_{i,j}$ would be equal to the **dot product** between matrix A's row i and matrix B's column j (explanation based on Khan Academy).

For example, $C_{1,1}$ is the dot product between matrix A's row 1, $[X_{11}, X_{12}, X_{13}]$ and matrix B's column 1, $[w_{11}, w_{12}, w_{13}]$. Simply multiplying these vectors yields: $[X_{11}*w_{11}, X_{12}*w_{12}, X_{13}*w_{13}]$, but since this is a dot product, we sum up each element of this resulting vector to get: $[X_{11}*w_{11} + X_{12}*w_{12} + X_{13}*w_{13}]$ which is $C_{1,1}$.

Lastly, we have to add in the bias vector:



Since there are 2 elements in each row vector and there are only 2 biases, in each row vector, we add the 1st bias to the 1st element and add the 2nd bias to the 2nd element. Now the 1st column of the resulting matrix contains all the predictions of each example from the 1st neuron. Similarly, the 2nd column contains all the predictions of each example from the 2nd neuron.

As you can see, the calculation for the predictions of each example for each neuron is still the same. By **vectorizing** the inputs, weights, and biases, we can easily use NumPy to calculate all these values. Here is the code to calculate the prediction of multiple neurons receiving multiple examples:

```
Xs = np.array([[2, 0, -7],
               [-2, 9, 0],
               [5, 1, -1],
               [3, 0, -4]])

def multiple_neurons(Xs):    # multiple input features, multiple examples
    w = np.array([[5, -2, -10],    # neuron 1's weights
                  [0, 9, -5]])    # neuron 2's weights
    b = np.array([-10, 7])    # 2 biases, one for each neuron, vector
    return np.dot(Xs, w.T) + b

print(multiple_neurons(Xs))
```

Like before, to get the transposed version of an array, use the `.T` attribute.

Output

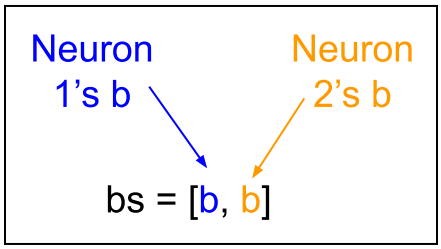
[[70, 42],
[-38, 88],
[23, 21],
[45, 27]

Currently, the weights and biases (parameters) of our layers are randomly hand-generated. Continuing the hand-generation of random parameters for more layers that may require more input features or more neurons, will tire us out. In the next section, we will use NumPy to randomly initialize parameters and be able to calculate the output of the neurons given any number of examples, input features, and neurons.

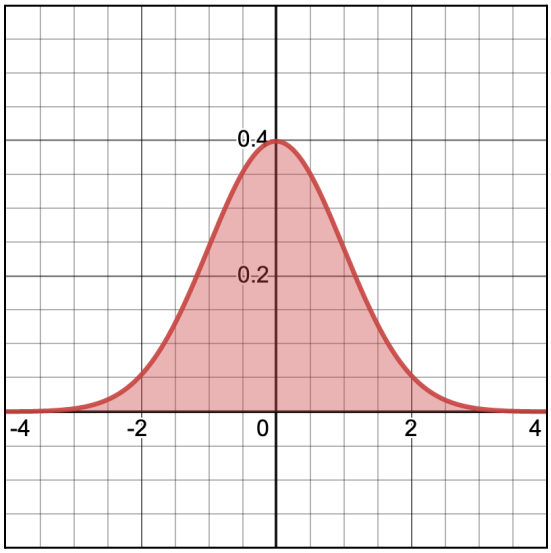
~~~~~~

Virtual Layer (object) of Artificial Neurons

In practice, we only randomly intialize the weights for each neuron. To understand why we do not randomly initialize the bias value for each neuron, think about a newly formed brain. A newly formed brain has no biases to anything in the world so when we initialize the values for the biases we do not want to add any "bias" to the brain. To achieve this, we can initialize each bias value to zero (neural) which is not considered a positive or a negative value.



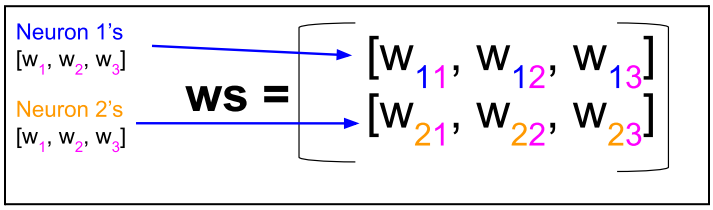
To create an array of zeros, for the bias vector, we can use Numpy's `np.zeros` function which requires the **shape** of the array to be filled with zeros. What is the **shape** of a bias vector? Well, take the example of having 2 neurons. When there are 2 neurons, we get a 2-element **row vector** (only one row). Also since each neuron has one bias value, the shape of a bias vector will always be: **(1, # of neurons)**.



For the weight matrix, we can use Numpy's `np.random.randn` function to generate random values given the **shape** of the array. The random values are generated based on the **"standard normal" distribution** which is a **normal distribution with mean 0 and standard deviation of 1**. This may seem familiar to those who have taken statistics, but for those who haven't, the random values are generated based on the **"bell-shaped" curve** (graph on the right).
~~~~~~

This curve resesmbles the shape of a bell, hence its name: "bell-shaped" curve. The y-values are the probability the corresponding x-value is picked. This means that the probability that a randomly generated weight value will be 40% (0.4 * 100). In addition, about 68% of the randomly generated weight values will be between [-1, 1], about 95% of them will be between [-2, 2] and about 99.7% of them will be between [-3, 3].

The question we still haven't asked is: what is the shape of the weight matrix?
Take the example of having 2 neurons, each receiving 3 input features. When this is the case, we have a (2, 3) matrix:



The number of rows is equal to the number of neurons and the number of columns is equal to the number of input features.
Meaning that, the shape of a weight matrix will always be:
(# of neurons, # of input features).

The only problem is that, when we do the dot product between the inputs matrix with shape (# of examples, # of input features) and the weights matrix, we put the inputs matrix first and then use the **transposed** weight matrix with shape (# of input features, # of neurons) as the second matrix. Instead of always using the `.T` attribute of the weights matrix in the dot product, we can randomly initialize the weights matrix to shape (# of input features, # of neurons).

Here is the code for randomly initializing the weights matrix and setting the bias values to an array of zeros:

```
import numpy as np
class Dense_Layer:
    def __init__(self, n_inputs, n_neurons):
        self.weights = 0.01 * np.random.randn(n_inputs, n_neurons)
        self.biases = np.zeros([1, n_neurons])
```

A dense layer is an artificial layer of neurons where all the input features are received by all the neurons of the layer. Sometimes people also refer to this layer as a fully-connected layer since all the input features are connected to all the neurons.

We multiply the randomly generated matrix of weights by 0.01 is because we want the weights closer to zero and it is not good for a model with huge weight values (in AI

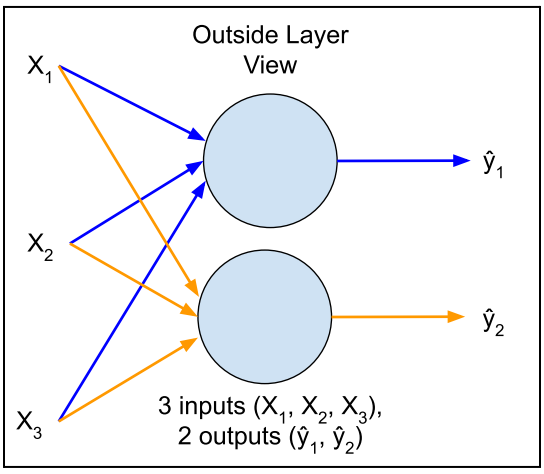
"huge weight values" means weight values greater than 1 or less than -1). Starting the model with huge weight values may cause problems in training or slow down training.

Here is the code of the dense layer with a method to calculate the outputs of the neurons:

```python
class Dense_Layer:
    def __init__(self, n_inputs, n_neurons):
        self.weights = 0.01 * np.random.randn(n_inputs, n_neurons)
        self.biases = np.zeros([1, n_neurons])

    def forward(self, inputs):    # inputs is X
        self.outputs = np.dot(inputs, self.weights) + self.biases
```

We have replaced inputs with X because if we had multiple layers, the inputs to the next layer is the output (prediction) of the previous layer instead of the original X (original inputs). The reason why we named `forward` method, "forward" is because the input features are moving forward to the neurons and the outputs of neurons are moving forward. There is a diagram on the right to help you remember how it looks like to get the prediction of a layer of neurons.



Technically we have completed Step 1 of creating an Artifical Neural Network: starting with a "dumb" brain. In the next chapters we will be focusing on the Training Phase, making this "dumb" brain "smart"!