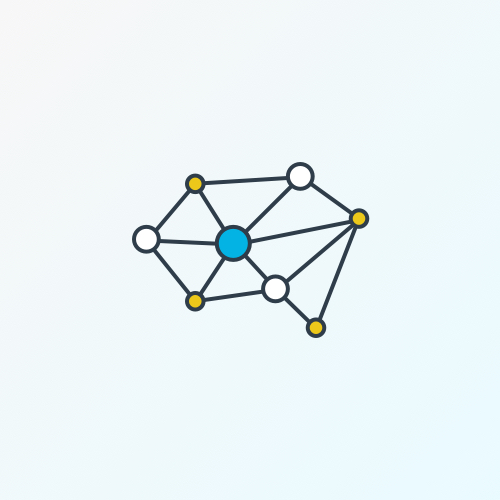
**A Note on Deep Learning**

The following lessons contain introductory and intermediate material on neural networks, building a neural network from scratch, using TensorFlow, and Convolutional Neural Networks:

* Neural Networks
* TensorFlow
* Deep Neural Networks
* Convolutional Neural Networks

While we highly suggest going through the all of the included content, if you already feel comfortable in any of these areas, feel free to skip ahead to later lessons. However, even if you have seen some of these topics before, it might be a good idea to get a refresher before you start working on the project!



Already know this content in your own neural network? Feel free to skip ahead!

A screenshot of a computer

Description automatically generated

# Linear to Logistic Regression

Linear regression helps predict values on a continuous spectrum, like predicting what the price of a house will be.

How about classifying data among discrete classes?

Here are examples of classification tasks:

* Determining whether a patient has cancer
* Identifying the species of a fish
* Figuring out who's talking on a conference call

Classification problems are important for self-driving cars. Self-driving cars might need to classify whether an object crossing the road is a car, pedestrian, and a bicycle. Or they might need to identify which type of traffic sign is coming up, or what a stop light is indicating.

In the next video, Luis will demonstrate a classification algorithm called "logistic regression". He'll use logistic regression to predict whether a student will be accepted to a university.

Linear regression leads to logistic regression and ultimately neural networks, a more advanced classification tool.

# Classification Problems

We'll start by defining what we mean by classification problems, and applying it to a simple example.

A screenshot of a cell phone

Description automatically generated

A screenshot of a cell phone

Description automatically generated

**Corrections:**

* At 3:07 in the video, the title should read "Step Function", not "Set Function".
* At 3:07 in the video, the definition of the Step function should be:

y=1 if x >= 0; y=0 if x<0

This is for 12 the video on perceptrons

A screenshot of a cell phone

Description automatically generated

# Perceptron

Now you've seen how a simple neural network makes decisions: by taking in input data, processing that information, and finally, producing an output in the form of a decision! Let's take a deeper dive into the university admission example to learn more about processing the input data.

Data, like test scores and grades, are fed into a network of interconnected nodes. These individual nodes are called [perceptrons](https://en.wikipedia.org/wiki/Perceptron), or artificial neurons, and they are the basic unit of a neural network. Each one looks at input data and decides how to categorize that data. In the example above, the input either passes a threshold for grades and test scores or doesn't, and so the two categories are: yes (passed the threshold) and no (didn't pass the threshold). These categories then combine to form a decision -- for example, if both nodes produce a "yes" output, then this student gains admission into the university.

A screenshot of a cell phone

Description automatically generated

Let's zoom in even further and look at how a single perceptron processes input data.

The perceptron above is one of the two perceptrons from the video that help determine whether or not a student is accepted to a university. It decides whether a student's grades are high enough to be accepted to the university. You might be wondering: "How does it know whether grades or test scores are more important in making this acceptance decision?" Well, when we initialize a neural network, we don't know what information will be most important in making a decision. It's up to the neural network to learn for itself which data is most important and adjust how it considers that data.

It does this with something called **weights**.

## Weights

When input comes into a perceptron, it gets multiplied by a weight value that is assigned to this particular input. For example, the perceptron above has two inputs, tests for test scores and grades, so it has two associated weights that can be adjusted individually. These weights start out as random values, and as the neural network network learns more about what kind of input data leads to a student being accepted into a university, the network adjusts the weights based on any errors in categorization that results from the previous weights. This is called **training** the neural network.

A higher weight means the neural network considers that input more important than other inputs, and lower weight means that the data is considered less important. An extreme example would be if test scores had no affect at all on university acceptance; then the weight of the test score input would be zero and it would have no affect on the output of the perceptron.

## Summing the Input Data

Each input to a perceptron has an associated weight that represents its importance. These weights are determined during the learning process of a neural network, called training. In the next step, the weighted input data are summed to produce a single value, that will help determine the final output - whether a student is accepted to a university or not. Let's see a concrete example of this.

A close up of a device

Description automatically generated

We weight x\_test by w\_test and add it to x\_grades weighted by w\_grades.

When writing equations related to neural networks, the weights will always be represented by some type of the letter **w**. It will usually look like a W*W* when it represents a **matrix** of weights or a w*w* when it represents an **individual** weight, and it may include some additional information in the form of a subscript to specify which weights (you'll see more on that next). But remember, when you see the letter **w**, think **weights**.

In this example, we'll use w\_{grades\_{}}*wgrades*​​ for the weight of grades and w\_{test}*wtest*​ for the weight of test.

For the image above, let's say that the weights are: w\_{grades} = -1, w\_{test} = -0.2*wgrades*​=−1,*wtest*​=−0.2. You don't have to be concerned with the actual values, but their relative values are important. w\_{grades\_{}}*wgrades*​​ is 5 times larger than w\_{test}*wtest*​, which means the neural network considers grades input 5 times more important than test in determining whether a student will be accepted into a university.

The perceptron applies these weights to the inputs and sums them in a process known as **linear combination**. In our case, this looks like w\_{grades} \cdot x\_{grades} + w\_{test} \cdot x\_{test} = -1 \cdot x\_{grades} - 0.2 \cdot x\_{test}*wgrades*​⋅*xgrades*​+*wtest*​⋅*xtest*​=−1⋅*xgrades*​−0.2⋅*xtest*​.

Now, to make our equation less wordy, let's replace the explicit names with numbers. Let's use 11 for grades*grades* and 22 for tests*tests*. So now our equation becomes

w\_{1} \cdot x\_{1} + w\_{2} \cdot x\_{2}*w*1​⋅*x*1​+*w*2​⋅*x*2​

In this example, we just have 2 simple inputs: grades and tests. Let's imagine we instead had m different inputs and we labeled them x\_1, x\_2, ..., x\_m*x*1​,*x*2​,...,*xm*​. Let's also say that the weight corresponding to x\_1*x*1​ is w\_1*w*1​ and so on. In that case, we would express the linear combination succintly as:

\sum\_{i=1} ^ m w\_i \cdot x\_i∑*i*=1*m*​*wi*​⋅*xi*​

Here, the Greek letter Sigma \sum∑ is used to represent **summation**. It simply means to evaluate the equation to the right multiple times and add up the results. In this case, the equation it will sum is w\_i \cdot x\_i*wi*​⋅*xi*​

But where do we get w\_i*wi*​ and x\_i*xi*​?

\sum\_{i=1} ^ m∑*i*=1*m*​ means to iterate over all i*i* values, from 11 to m*m*.

So to put it all together, \sum\_{i=1} ^ m w\_i \cdot x\_i∑*i*=1*m*​*wi*​⋅*xi*​ means the following:

* Start at i = 1*i*=1
* Evaluate w\_1 \cdot x\_1*w*1​⋅*x*1​ and remember the results
* Move to i = 2*i*=2
* Evaluate w\_2 \cdot x\_2*w*2​⋅*x*2​ and add these results to w\_1 \cdot x\_1*w*1​⋅*x*1​
* Continue repeating that process until i = m*i*=*m*, where m*m* is the number of inputs.

One last thing: you'll see equations written many different ways, both here and when reading on your own. For example, you will often just see \sum\_{i\_{}}∑*i*​​ instead of \sum\_{i=1} ^ m∑*i*=1*m*​. The first is simply a shorter way of writing the second. That is, if you see a summation without a starting number or a defined end value, it just means perform the sum for all of the them. And sometimes, if the value to iterate over can be inferred, you'll see it as just \sum∑. Just remember they're all the same thing: \sum\_{i=1} ^ m w\_i \cdot x\_i = \sum\_{i} w\_i \cdot x\_i = \sum w\_i \cdot x\_i∑*i*=1*m*​*wi*​⋅*xi*​=∑*i*​*wi*​⋅*xi*​=∑*wi*​⋅*xi*​.

## Calculating the Output with an Activation Function

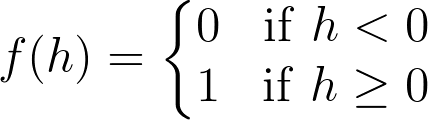
Finally, the result of the perceptron's summation is turned into an output signal! This is done by feeding the linear combination into an **activation function**.

Activation functions are functions that decide, given the inputs into the node, what should be the node's output? Because it's the activation function that decides the actual output, we often refer to the outputs of a layer as its "activations".

One of the simplest activation functions is the **Heaviside step function**. This function returns a **0** if the linear combination is less than 0. It returns a **1** if the linear combination is positive or equal to zero. The [Heaviside step function](https://en.wikipedia.org/wiki/Heaviside_step_function) is shown below, where h is the calculated linear combination:

A screenshot of a cell phone

Description automatically generated

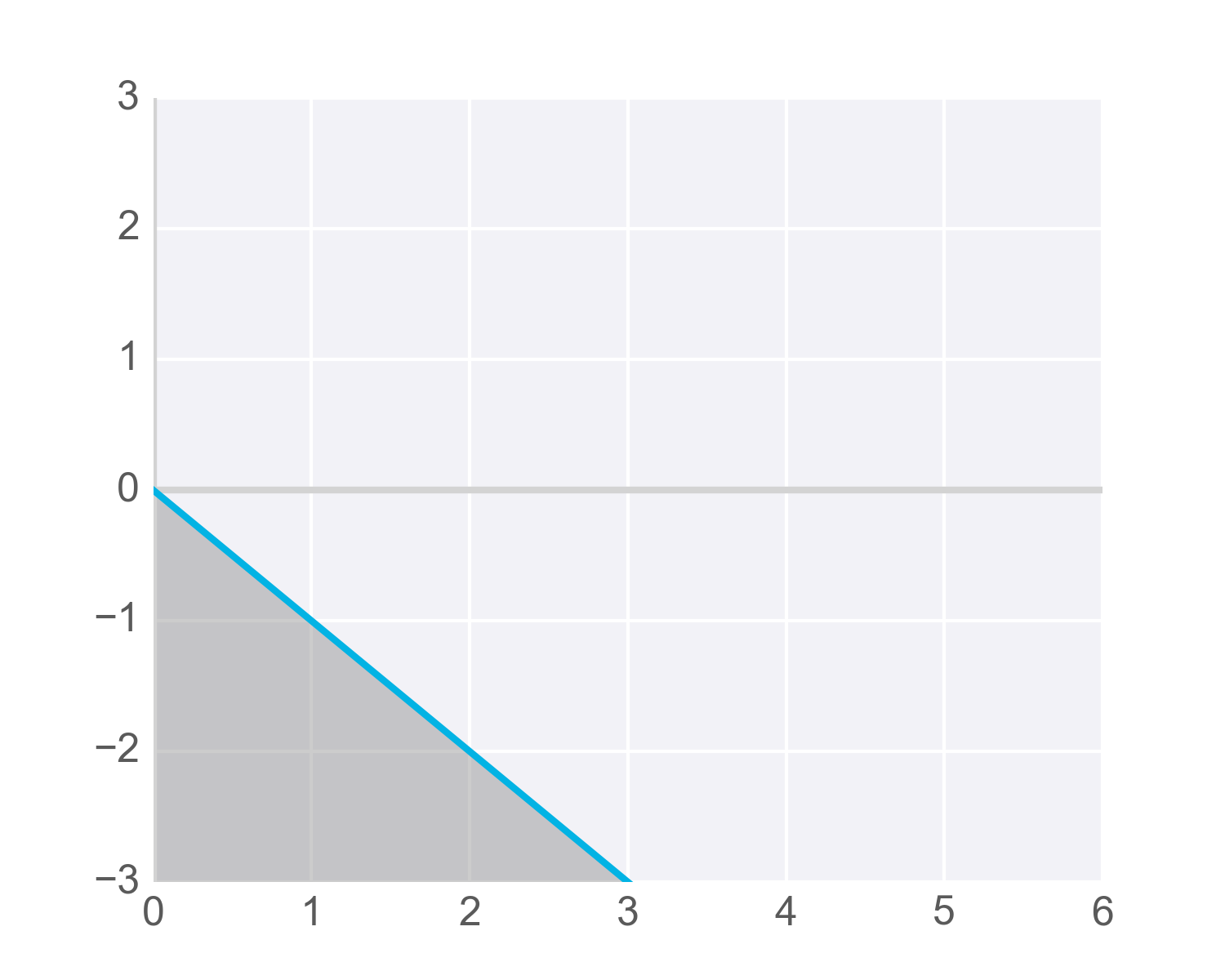


The Heaviside Step Function

In the university acceptance example above, we used the weights w\_{grades} = -1, w\_{test} = -0.2*wgrades*​=−1,*wtest*​=−0.2. Since w\_{grades\_{}}*wgrades*​​ and w\_{test}*wtest*​ are negative values, the activation function will only return a 11 if grades and test are 00! This is because the range of values from the linear combination using these weights and inputs are (-\infty, 0](−∞,0] (i.e. negative infinity to 0, including 0 itself).

It's easiest to see this with an example in two dimensions. In the following graph, imagine any points along the line or in the shaded area represent all the possible inputs to our node. Also imagine that the value along the y-axis is the result of performing the linear combination on these inputs and the appropriate weights. It's this result that gets passed to the activation function.

Now remember that the step activation function returns 11 for any inputs greater than or equal to zero. As you can see in the image, only one point has a y-value greater than or equal to zero – the point right at the origin, (0, 0)(0,0):

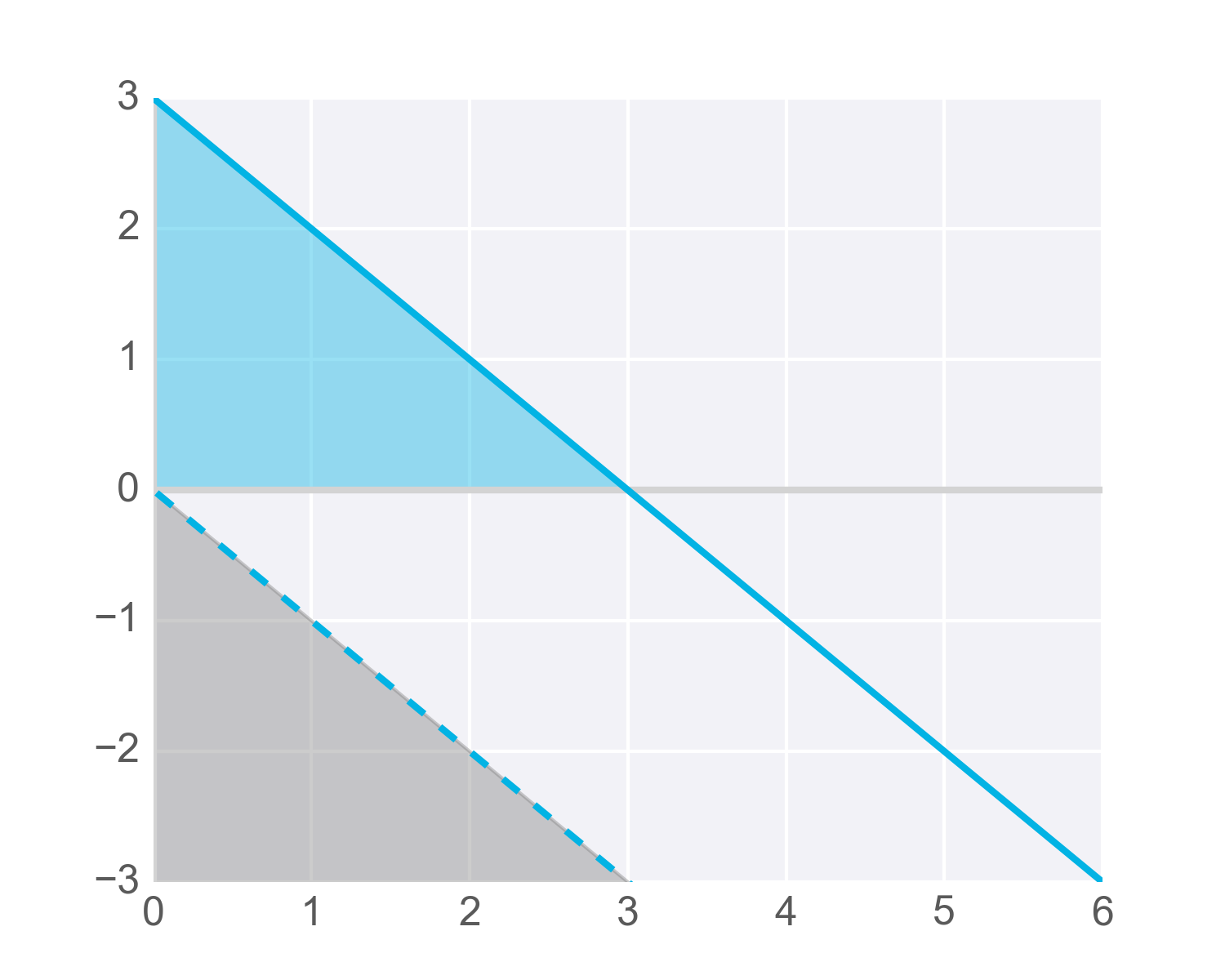


Now, we certainly want more than one possible grade/test combination to result in acceptance, so we need to adjust the results passed to our activation function so it activates – that is, returns 11 – for more inputs. Specifically, we need to find a way so all the scores we’d like to consider acceptable for admissions produce values greater than or equal to zero when linearly combined with the weights into our node.

One way to get our function to return 11 for more inputs is to add a value to the results of our linear combination, called a **bias**.

A bias, represented in equations as b*b*, lets us move values in one direction or another.

For example, the following diagram shows the previous hypothetical function with an added bias of +3+3. The blue shaded area shows all the values that now activate the function. But notice that these are produced with the same inputs as the values shown shaded in grey – just adjusted higher by adding the bias term:



Of course, with neural networks we won't know in advance what values to pick for biases. That’s ok, because just like the weights, the bias can also be updated and changed by the neural network during training. So after adding a bias, we now have a complete perceptron formula:

A picture containing clock

Description automatically generated

Perceptron Formula

This formula returns 11 if the input (x\_1, x\_2, ..., x\_m*x*1​,*x*2​,...,*xm*​) belongs to the accepted-to-university category or returns 00 if it doesn't. The input is made up of one or more [real numbers](https://en.wikipedia.org/wiki/Real_number), each one represented by x\_i*xi*​, where m*m* is the number of inputs.

Then the neural network starts to learn! Initially, the weights ( w\_i*wi*​) and bias (b*b*) are assigned a random value, and then they are updated using a learning algorithm like gradient descent. The weights and biases change so that the next training example is more accurately categorized, and patterns in data are "learned" by the neural network.

Now that you have a good understanding of perceptrons, let's put that knowledge to use. In the next section, you'll create the AND perceptron from the Neural Networks video by setting the values for weights and bias.

# Perceptrons as Logical Operators

In this lesson, we'll see one of the many great applications of perceptrons. As logical operators! You'll have the chance to create the perceptrons for the most common of these, the **AND**, **OR**, and **NOT** operators. And then, we'll see what to do about the elusive **XOR** operator. Let's dive in!

# AND Perceptron

Note: The second and third rows of the third column from 1:50-onward should be blue in color (they have the correct value of 1) for the OR perceptron.

A picture containing clock, drawing

Description automatically generated

## What are the weights and bias for the AND perceptron?

Set the weights (weight1, weight2) and bias (bias) to values that will correctly determine the AND operation as shown above.  
More than one set of values will work!

# OR Perceptron

A picture containing game

Description automatically generated

The OR perceptron is very similar to an AND perceptron. In the image below, the OR perceptron has the same line as the AND perceptron, except the line is shifted down. What can you do to the weights and/or bias to achieve this? Use the following AND perceptron to create an OR Perceptron.

A picture containing clock

Description automatically generated

### QUESTION 2 OF 4

What are two ways to go from an AND perceptron to an OR perceptron?

* Increase the weights
* Decrease the magnitude of the bias

# NOT Perceptron

Unlike the other perceptrons we looked at, the NOT operation only cares about one input. The operation returns a 0 if the input is 1 and a 1 if it's a 0. The other inputs to the perceptron are ignored.

In this quiz, you'll set the weights (weight1, weight2) and bias bias to the values that calculate the NOT operation on the second input and ignores the first input.

# XOR Perceptron

A picture containing game

Description automatically generated

# Quiz: Build an XOR Multi-Layer Perceptron

Now, let's build a multi-layer perceptron from the AND, NOT, and OR perceptrons to create XOR logic!

The neural network below contains 3 perceptrons, A, B, and C. The last one (AND) has been given for you. The input to the neural network is from the first node. The output comes out of the last node.

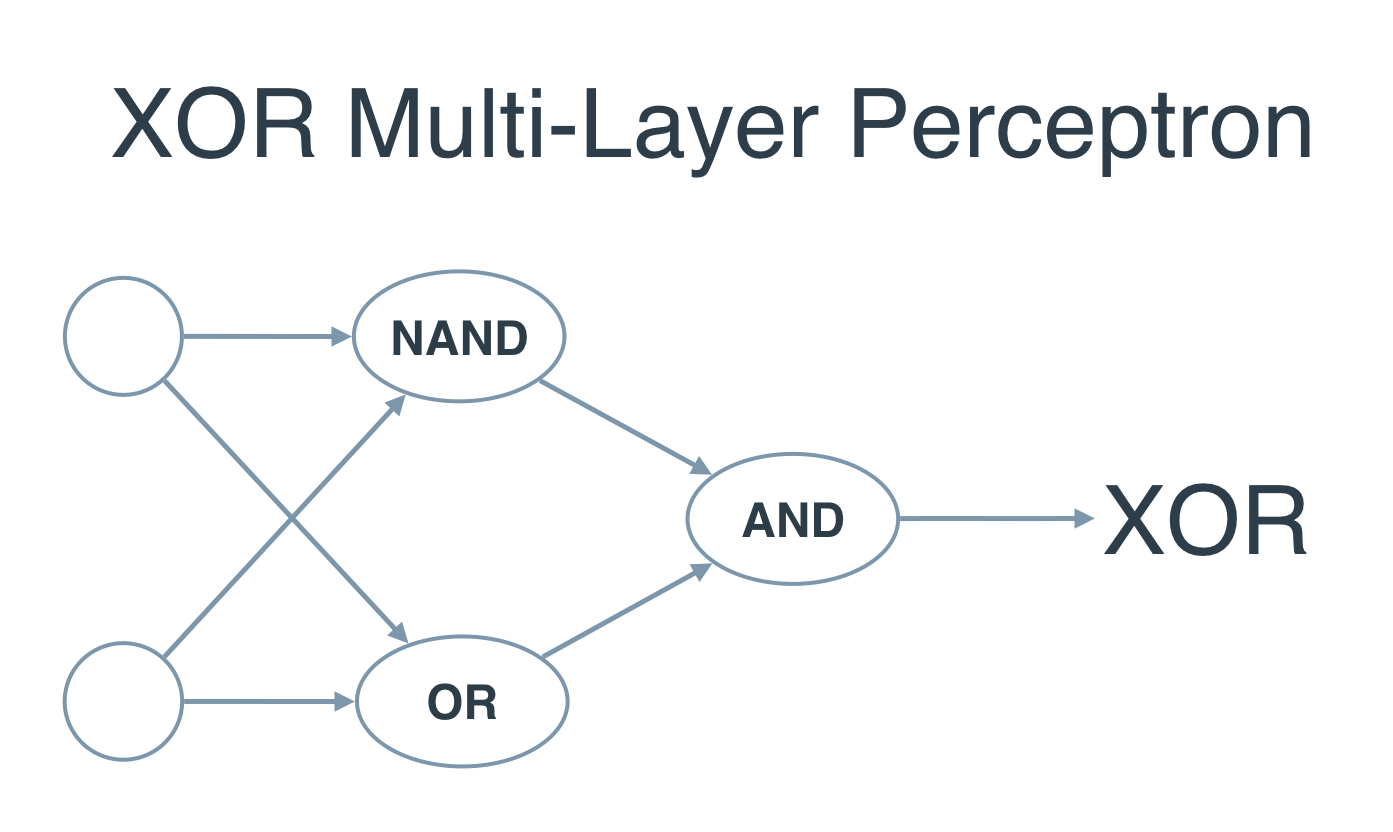
The multi-layer perceptron below calculates XOR. Each perceptron is a logic operation of AND, OR, and NOT. However, the perceptrons A, B, and C don't indicate their operation. In the following quiz, set the correct operations for the perceptrons to calculate XOR.

A close up of a map

Description automatically generated

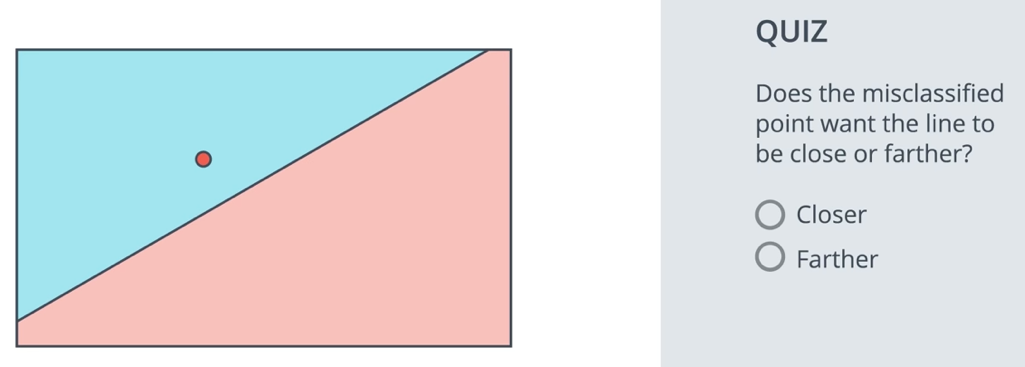
That's Correct!

And if we introduce the **NAND** operator as the combination of **AND** and **NOT**, then we get the following two-layer perceptron that will model **XOR**. That's our first neural network!



# Perceptron Trick

In the last section you used your logic and your mathematical knowledge to create perceptrons for some of the most common logical operators. In real life, though, we can't be building these perceptrons ourselves. The idea is that we give them the result, and they build themselves. For this, here's a pretty neat trick that will help us.



### Time for some math!

Now that we've learned that the points that are misclassified, want the line to move closer to them, let's do some math. The following video shows a mathematical trick that modifies the equation of the line, so that it comes closer to a particular point.

For the second example, where the line is described by 3x1+ 4x2 - 10 = 0, if the learning rate was set to 0.1, how many times would you have to apply the perceptron trick to move the line to a position where the blue point, at (1, 1), is correctly classified?

Answer: 10

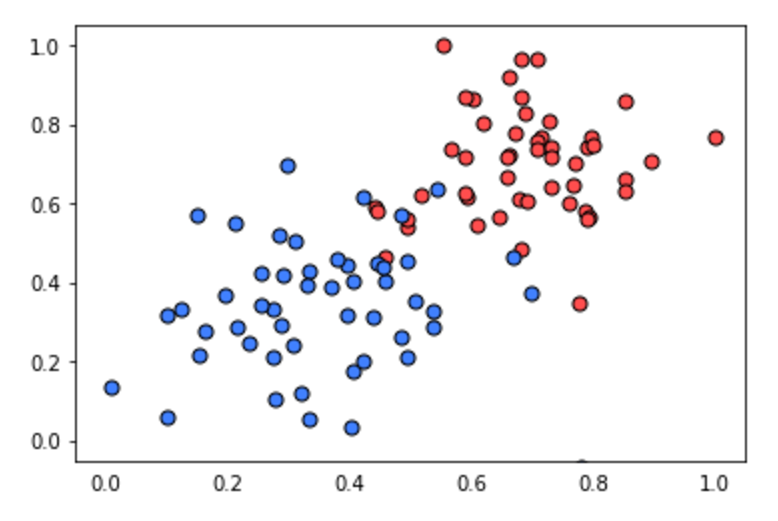
# Perceptron Algorithm

And now, with the perceptron trick in our hands, we can fully develop the perceptron algorithm! The following video will show you the pseudocode, and in the quiz below, you'll have the chance to code it in Python.

There's a small error in the above video in that W\_i*Wi*​ should be updated to W\_i = W\_i + \alpha x\_i*Wi*​=*Wi*​+*αxi*​ (plus or minus depending on the situation).

**Coding the Perceptron Algorithm**

Time to code! In this quiz, you'll have the chance to implement the perceptron algorithm to separate the following data (given in the file data.csv).



Recall that the perceptron step works as follows. For a point with coordinates (p,q)(*p*,*q*), label y*y*, and prediction given by the equation \hat{y} = step(w\_1x\_1 + w\_2x\_2 + b)*y*^​=*step*(*w*1​*x*1​+*w*2​*x*2​+*b*):

* If the point is correctly classified, do nothing.
* If the point is classified positive, but it has a negative label, subtract \alpha p, \alpha q,*αp*,*αq*, and \alpha*α* from w\_1, w\_2,*w*1​,*w*2​, and b*b* respectively.
* If the point is classified negative, but it has a positive label, add \alpha p, \alpha q,*αp*,*αq*, and \alpha*α* to w\_1, w\_2,*w*1​,*w*2​, and b*b* respectively.

Then click on test run to graph the solution that the perceptron algorithm gives you. It'll actually draw a set of dotted lines, that show how the algorithm approaches to the best solution, given by the black solid line.

Feel free to play with the parameters of the algorithm (number of epochs, learning rate, and even the randomizing of the initial parameters) to see how your initial conditions can affect the solution!

import numpy as np  
import pandas as pd  
# Setting the random seed, feel free to change it and see different solutions.  
np.random.seed(42)  
  
def stepFunction(t):  
 if t >= 0:  
 return 1  
 return 0  
  
def prediction(X, W, b):  
 return stepFunction((np.matmul(X,W)+b)[0])  
  
# TODO: Fill in the code below to implement the perceptron trick.  
# The function should receive as inputs the data X, the labels y,  
# the weights W (as an array), and the bias b,  
# update the weights and bias W, b, according to the perceptron algorithm,  
# and return W and b.  
def perceptronStep(X, y, W, b, lr):  
 for i in range(len((X))):  
 pred = prediction(X[i], W, b)  
 if pred == 0 and y[i] == 1:  
 W[0] = W[0] + lr \* X[i][0]  
 W[1] = W[0] + lr \* X[i][1]  
 b += lr  
 elif pred == 1 and y[i] == 0:  
 W[0] = W[0] - lr \* X[i][0]  
 W[1] = W[1] - lr \* X[i][1]  
 b = b - lr  
 return W, b  
  
# This function runs the perceptron algorithm repeatedly on the dataset,  
# and returns a few of the boundary lines obtained in the iterations,  
# for plotting purposes.  
# Feel free to play with the learning rate and the num\_epochs,  
# and see your results plotted below.  
def trainPerceptronAlgorithm(X, y, lr = 0.01, num\_epochs = 25):  
 x\_min, x\_max = min(X.T[0]), max(X.T[0])  
 y\_min, y\_max = min(X.T[1]), max(X.T[1])  
 W = np.array(np.random.rand(2,1))  
 b = np.random.rand(1)[0] + x\_max  
 # These are the solution lines that get plotted below.  
 boundary\_lines = []  
 for i in range(num\_epochs):  
 # In each epoch, we apply the perceptron step.  
 W, b = perceptronStep(X, y, W, b, lr)  
 boundary\_lines.append((-W[0]/W[1], -b/W[1]))  
 return boundary\_lines  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 data = pd.read\_csv('data.csv', sep = ',', header = None)  
 X = np.array(data[[0, 1]])  
 y = np.array(data[2])  
  
 boundary\_lines = trainPerceptronAlgorithm(X, y)

### QUIZ QUESTION

Which of the following conditions should be met in order to apply gradient descent? (Check all that apply.)

* The error function should be discrete
* The error function should contain only positive values
* The error function should be differentiable
* The error function should be normalized
* The error function should be continuous

### QUIZ QUESTION

The sigmoid function is defined as sigmoid(x) = 1/(1+e-x). If the score is defined by 4x1 + 5x2 - 9 = score, then which of the following points has exactly a 50% probability of being blue or red? (Choose all that are correct.)

* (1, 1)
* (2, 4)
* (5, -5)
* (-4, 5)

# Multi-Class Classification and Softmax

# The Softmax Function

In the next video, we'll learn about the softmax function, which is the equivalent of the sigmoid activation function, but when the problem has 3 or more classes.

### QUESTION 1 OF 2

What function turns every number into a positive number?

* sin
* cos
* log
* exp

# Maximum Likelihood

Probability will be one of our best friends as we go through Deep Learning. In this lesson, we'll see how we can use probability to evaluate (and improve!) our models.

### QUIZ QUESTION

Based on the above video, which of the following is true for a very high value for P(all)?

* The model classifies most blue points correctly.
* The model classifies most red points correctly.
* The model classifies most points correctly with P(all) indicating how accurate the model is.
* The model classifies all points correctly.

SUBMIT

# Maximizing Probabilities

In this lesson and quiz, we will learn how to maximize a probability,

### QUIZ QUESTION

What function turns products into sums?

* sin
* cos
* log
* exp

Cross entropy 1 video notes:

*Correction:* At 2:18, the top right point should be labelled -log(0.7) instead of -log(0.2).

NEXT

# Cross-Entropy

So we're getting somewhere, there's definitely a connection between probabilities and error functions, and it's called **Cross-Entropy**. This concept is tremendously popular in many fields, including Machine Learning. Let's dive more into the formula, and actually code it!

A screenshot of a cell phone

Description automatically generated

Nice work! Yes, cross-entropy is inversely proportional to the total probability of an outcome.

# Logistic Regression

Now, we're finally ready for one of the most popular and useful algorithms in Machine Learning, and the building block of all that constitutes Deep Learning. The **Logistic Regression** Algorithm. And it basically goes like this:

* Take your data
* Pick a random model
* Calculate the error
* Minimize the error, and obtain a better model
* Enjoy!

### Calculating the Error Function

Let's dive into the details. The next video will show you how to calculate an error function.

# Gradient Descent

In this lesson, we'll learn the principles and the math behind the gradient descent algorithm.

# Gradient Calculation

In the last few videos, we learned that in order to minimize the error function, we need to take some derivatives. So let's get our hands dirty and actually compute the derivative of the error function. The first thing to notice is that the sigmoid function has a really nice derivative. Namely,

\sigma'(x) = \sigma(x) (1-\sigma(x))*σ*′(*x*)=*σ*(*x*)(1−*σ*(*x*))

The reason for this is the following, we can calculate it using the quotient formula:

A picture containing clock

Description automatically generated

And now, let's recall that if we have m*m* points labelled x^{(1)}, x^{(2)}, \ldots, x^{(m)},*x*(1),*x*(2),…,*x*(*m*), the error formula is:

E = -\frac{1}{m} \sum\_{i=1}^m \left( y\_i \ln(\hat{y\_i}) + (1-y\_i) \ln (1-\hat{y\_i}) \right)*E*=−*m*1​∑*i*=1*m*​(*yi*​ln(*yi*​^​)+(1−*yi*​)ln(1−*yi*​^​))

where the prediction is given by \hat{y\_i} = \sigma(Wx^{(i)} + b).*yi*​^​=*σ*(*Wx*(*i*)+*b*).

Our goal is to calculate the gradient of E,*E*, at a point x = (x\_1, \ldots, x\_n),*x*=(*x*1​,…,*xn*​), given by the partial derivatives

\nabla E =\left(\frac{\partial}{\partial w\_1}E, \cdots, \frac{\partial}{\partial w\_n}E, \frac{\partial}{\partial b}E \right)∇*E*=(∂*w*1​∂​*E*,⋯,∂*wn*​∂​*E*,∂*b*∂​*E*)

To simplify our calculations, we'll actually think of the error that each point produces, and calculate the derivative of this error. The total error, then, is the average of the errors at all the points. The error produced by each point is, simply,

E = - y \ln(\hat{y}) - (1-y) \ln (1-\hat{y})*E*=−*y*ln(*y*^​)−(1−*y*)ln(1−*y*^​)

In order to calculate the derivative of this error with respect to the weights, we'll first calculate \frac{\partial}{\partial w\_j} \hat{y}.∂*wj*​∂​*y*^​. Recall that \hat{y} = \sigma(Wx+b),*y*^​=*σ*(*Wx*+*b*), so:

A close up of a logo

Description automatically generated

The last equality is because the only term in the sum which is not a constant with respect to w\_j*wj*​ is precisely w\_j x\_j,*wj*​*xj*​, which clearly has derivative x\_j.*xj*​.

Now, we can go ahead and calculate the derivative of the error E*E* at a point x,*x*, with respect to the weight w\_j.*wj*​.

A close up of text on a white background

Description automatically generated

A similar calculation will show us that

A picture containing object, clock

Description automatically generated

This actually tells us something very important. For a point with coordinates (x\_1, \ldots, x\_n),(*x*1​,…,*xn*​), label y,*y*, and prediction \hat{y},*y*^​, the gradient of the error function at that point is \left(-(y - \hat{y})x\_1, \cdots, -(y - \hat{y})x\_n, -(y - \hat{y}) \right).(−(*y*−*y*^​)*x*1​,⋯,−(*y*−*y*^​)*xn*​,−(*y*−*y*^​)). In summary, the gradient is

\nabla E = -(y - \hat{y}) (x\_1, \ldots, x\_n, 1).∇*E*=−(*y*−*y*^​)(*x*1​,…,*xn*​,1).

If you think about it, this is fascinating. The gradient is actually a scalar times the coordinates of the point! And what is the scalar? Nothing less than a multiple of the difference between the label and the prediction. What significance does this have?

### QUIZ QUESTION

What does the scalar we obtained above signify? (Check all that are true.)

* Closer the label to the prediction, larger the gradient.
* Closer the label to the prediction, smaller the gradient.
* Farther the label from the prediction, larger the gradient.
* Farther the label to the prediction, smaller the gradient.

So, a small gradient means we'll change our coordinates by a little bit, and a large gradient means we'll change our coordinates by a lot.

If this sounds anything like the perceptron algorithm, this is no coincidence! We'll see it in a bit.

# Gradient Descent Step

Therefore, since the gradient descent step simply consists in subtracting a multiple of the gradient of the error function at every point, then this updates the weights in the following way:

w\_i' \leftarrow w\_i -\alpha [-(y - \hat{y}) x\_i],*wi*′​←*wi*​−*α*[−(*y*−*y*^​)*xi*​],

which is equivalent to

w\_i' \leftarrow w\_i + \alpha (y - \hat{y}) x\_i.*wi*′​←*wi*​+*α*(*y*−*y*^​)*xi*​.

Similarly, it updates the bias in the following way:

b' \leftarrow b + \alpha (y - \hat{y}),*b*′←*b*+*α*(*y*−*y*^​),

Note: Since we've taken the average of the errors, the term we are adding should be \frac{1}{m} \cdot \alpha*m*1​⋅*α* instead of \alpha,*α*, but as \alpha*α* is a constant, then in order to simplify calculations, we'll just take \frac{1}{m} \cdot \alpha*m*1​⋅*α* to be our learning rate, and abuse the notation by just calling it \alpha.*α*.

# Gradient Descent: The Code

From before we saw that one weight update can be calculated as:

\Delta w\_i = \alpha \* \delta \* x\_iΔ*wi*​=*α*∗*δ*∗*xi*​

where \alpha*α* is the learning rate and \delta*δ* is the error term.

Previously, we utilized the loss (error) function for logistic regression, which was because we were performing a binary classification task. This time we'll try to get the function to learn a value instead of a class. Therefore, we'll use a simpler loss function, as defined below in the error term \delta*δ*.

\delta = (y - \hat y) f'(h) = (y - \hat y) f'(\sum w\_i x\_i)*δ*=(*y*−*y*^​)*f*′(*h*)=(*y*−*y*^​)*f*′(∑*wi*​*xi*​)

Note that f'(h)*f*′(*h*) is the derivative of the activation function f(h)*f*(*h*), and h*h* is defined as the output, which in the case of a neural network is a sum of the weights times the inputs.

Now I'll write this out in code for the case of only one output unit. We'll also be using the sigmoid as the activation function f(h)*f*(*h*).

*# Defining the sigmoid function for activations*

**def** **sigmoid**(x):

**return** 1/(1+np.exp(-x))

*# Derivative of the sigmoid function*

**def** **sigmoid\_prime**(x):

**return** sigmoid(x) \* (1 - sigmoid(x))

*# Input data*

x = np.array([0.1, 0.3])

*# Target*

y = 0.2

*# Input to output weights*

weights = np.array([-0.8, 0.5])

*# The learning rate, eta in the weight step equation*

learnrate = 0.5

*# The neural network output (y-hat)*

nn\_output = sigmoid(x[0]\*weights[0] + x[1]\*weights[1])

*# or nn\_output = sigmoid(np.dot(x, weights))*

*# output error (y - y-hat)*

error = y - nn\_output

*# error term (lowercase delta)*

error\_term = error \* sigmoid\_prime(np.dot(x,weights))

*# Gradient descent step*

del\_w = [ learnrate \* error\_term \* x[0],

learnrate \* error\_term \* x[1]]

*# or del\_w = learnrate \* error\_term \* x*

**Perceptron vs gradient descent video**

In the video at 0:12 mark, the instructor said y hat minus y. It should be y minus y hat instead as stated on the slide.

# Neural Network Architecture

Ok, so we're ready to put these building blocks together, and build great Neural Networks! (Or Multi-Layer Perceptrons, however you prefer to call them.)

This first two videos will show us how to combine two perceptrons into a third, more complicated one.

### **Multiple layers**

Now, not all neural networks look like the one above. They can be way more complicated! In particular, we can do the following things:

* Add more nodes to the input, hidden, and output layers.
* Add more layers.

We'll see the effects of these changes in the next video.

### **Multi-Class Classification**

And here we elaborate a bit more into what can be done if our neural network needs to model data with more than one output.

# Feedforward

Feedforward is the process neural networks use to turn the input into an output. Let's study it more carefully, before we dive into how to train the networks

# Error Function

Just as before, neural networks will produce an error function, which at the end, is what we'll be minimizing. The following video shows the error function for a neural network.

# Implementing the hidden layer

##### Prerequisites

Below, we are going to walk through the math of neural networks in a multilayer perceptron. With multiple perceptrons, we are going to move to using vectors and matrices. To brush up, be sure to view the following:

1. Khan Academy's [introduction to vectors](https://www.khanacademy.org/math/linear-algebra/vectors-and-spaces/vectors/v/vector-introduction-linear-algebra).
2. Khan Academy's [introduction to matrices](https://www.khanacademy.org/math/precalculus/precalc-matrices).

##### Derivation

Before, we were dealing with only one output node which made the code straightforward. However now that we have multiple input units and multiple hidden units, the weights between them will require two indices: w\_{ij}*wij*​ where i*i* denotes input units and j*j* are the hidden units.

For example, the following image shows our network, with its input units labeled x\_1, x\_2,*x*1​,*x*2​, and x\_3*x*3​, and its hidden nodes labeled h\_1*h*1​ and h\_2*h*2​:

A picture containing sitting, holding, small, clock

Description automatically generated

The lines indicating the weights leading to h\_1*h*1​ have been colored differently from those leading to h\_2*h*2​ just to make it easier to read.

Now to index the weights, we take the input unit number for the \_i*i*​ and the hidden unit number for the \_j*j*​. That gives us

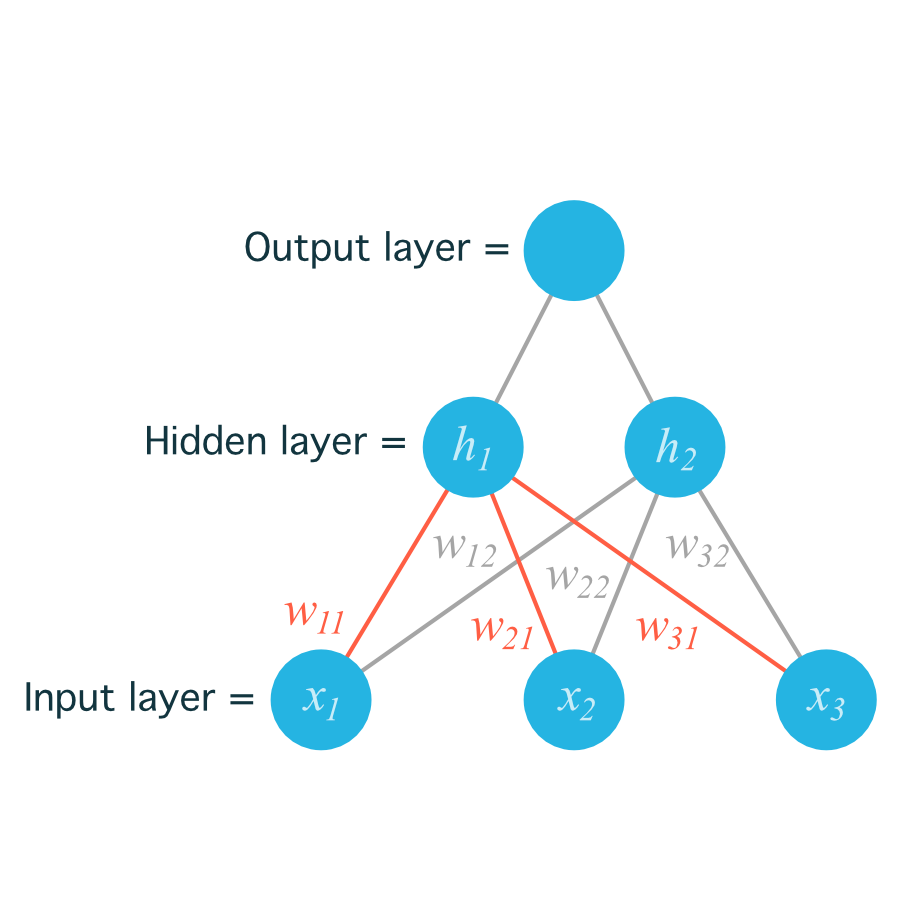
w\_{11}*w*11​

for the weight leading from x\_1*x*1​ to h\_1*h*1​, and

w\_{12}*w*12​

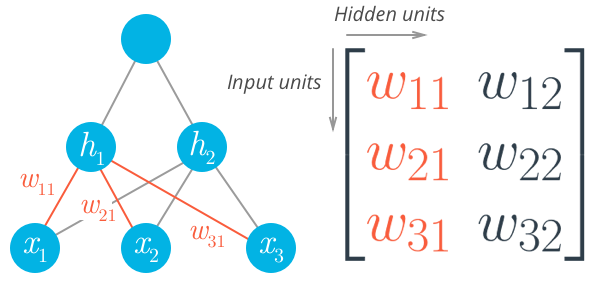
for the weight leading from x\_1*x*1​ to h\_2*h*2​.

The following image includes all of the weights between the input layer and the hidden layer, labeled with their appropriate w\_{ij}*wij*​ indices:



Before, we were able to write the weights as an array, indexed as w\_i*wi*​.

But now, the weights need to be stored in a **matrix**, indexed as w\_{ij}*wij*​. Each **row** in the matrix will correspond to the weights **leading out** of a **single input unit**, and each **column** will correspond to the weights **leading in** to a **single hidden unit**. For our three input units and two hidden units, the weights matrix looks like this:



Be sure to compare the matrix above with the diagram shown before it so you can see where the different weights in the network end up in the matrix.

To initialize these weights in Numpy, we have to provide the shape of the matrix. If features is a 2D array containing the input data:

*# Number of records and input units*

n\_records, n\_inputs = features.shape

*# Number of hidden units*

n\_hidden = 2

weights\_input\_to\_hidden = np.random.normal(0, n\_inputs\*\*-0.5, size=(n\_inputs, n\_hidden))

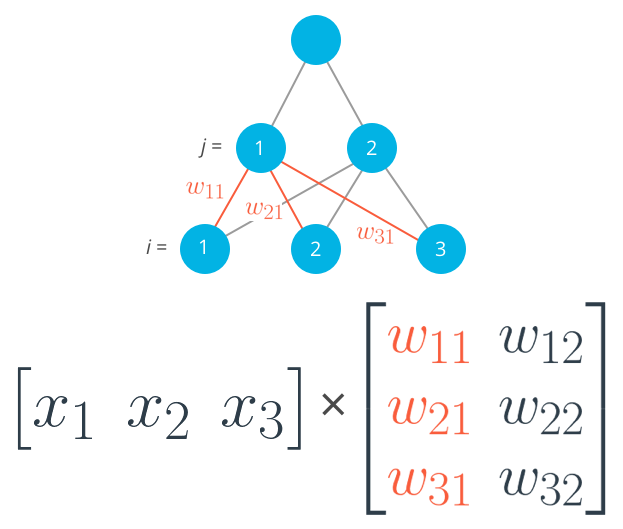
This creates a 2D array (i.e. a matrix) named weights\_input\_to\_hidden with dimensions n\_inputs by n\_hidden. Remember how the input to a hidden unit is the sum of all the inputs multiplied by the hidden unit's weights. So for each hidden layer unit, h\_j*hj*​, we need to calculate the following:

A close up of a logo

Description automatically generated

To do that, we now need to use [matrix multiplication](https://en.wikipedia.org/wiki/Matrix_multiplication).

In this case, we're multiplying the inputs (a row vector here) by the weights. To do this, you take the dot (inner) product of the inputs with each column in the weights matrix. For example, to calculate the input to the first hidden unit, j = 1*j*=1, you'd take the dot product of the inputs with the first column of the weights matrix, like so:



Calculating the input to the first hidden unit with the first column of the weights matrix.



And for the second hidden layer input, you calculate the dot product of the inputs with the second column. And so on and so forth.

In NumPy, you can do this for all the inputs and all the outputs at once using np.dot

hidden\_inputs = np.dot(inputs, weights\_input\_to\_hidden)

You could also define your weights matrix such that it has dimensions n\_hidden by n\_inputs then multiply like so where the inputs form a column vector:

A close up of a logo

Description automatically generated

**Note:** The weight indices have changed in the above image and no longer match up with the labels used in the earlier diagrams. That's because, in matrix notation, the row index always precedes the column index, so it would be misleading to label them the way we did in the neural net diagram. Just keep in mind that this is the same weight matrix as before, but rotated so the first column is now the first row, and the second column is now the second row. If we were to use the labels from the earlier diagram, the weights would fit into the matrix in the following locations:

A close up of a blackboard

Description automatically generated

Weight matrix shown with labels matching earlier diagrams.

Remember, the above is **not** a correct view of the **indices**, but it uses the labels from the earlier neural net diagrams to show you where each weight ends up in the matrix.

The important thing with matrix multiplication is that the dimensions match. For matrix multiplication to work, there has to be the same number of elements in the dot products. In the first example, there are three columns in the input vector, and three rows in the weights matrix. In the second example, there are three columns in the weights matrix and three rows in the input vector. If the dimensions don't match, you'll get this:

# Same weights and features as above, but swapped the order

hidden\_inputs = np.dot(weights\_input\_to\_hidden, features)

---------------------------------------------------------------------------

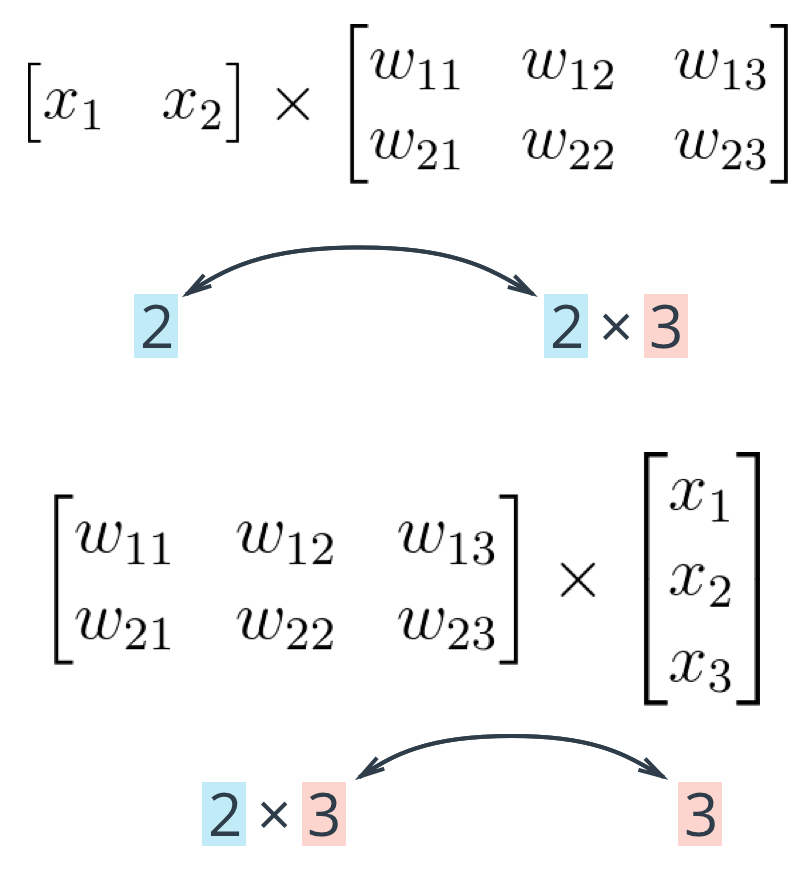
ValueError Traceback (most recent call last)

<ipython-input-11-1bfa0f615c45> in <module>()

----> 1 hidden\_in = np.dot(weights\_input\_to\_hidden, X)

ValueError: shapes (3,2) and (3,) not aligned: 2 (dim 1) != 3 (dim 0)

The dot product can't be computed for a 3x2 matrix and 3-element array. That's because the 2 columns in the matrix don't match the number of elements in the array. Some of the dimensions that could work would be the following:



The rule is that if you're multiplying an array from the left, the array must have the same number of elements as there are rows in the matrix. And if you're multiplying the matrix from the left, the number of columns in the matrix must equal the number of elements in the array on the right.

### Making a column vector

You see above that sometimes you'll want a column vector, even though by default Numpy arrays work like row vectors. It's possible to get the transpose of an array like so arr.T, but for a 1D array, the transpose will return a row vector. Instead, use arr[:,None] to create a column vector:

print(features)

> array([ 0.49671415, -0.1382643 , 0.64768854])

print(features.T)

> array([ 0.49671415, -0.1382643 , 0.64768854])

print(features[:, **None**])

> array([[ 0.49671415],

[-0.1382643 ],

[ 0.64768854]])

Alternatively, you can create arrays with two dimensions. Then, you can use arr.T to get the column vector.

np.array(features, ndmin=2)

> array([[ 0.49671415, -0.1382643 , 0.64768854]])

np.array(features, ndmin=2).T

> array([[ 0.49671415],

[-0.1382643 ],

[ 0.64768854]])

I personally prefer keeping all vectors as 1D arrays, it just works better in my head.

## Programming quiz

Below, you'll implement a forward pass through a 4x3x2 network, with sigmoid activation functions for both layers.

Things to do:

* Calculate the input to the hidden layer.
* Calculate the hidden layer output.
* Calculate the input to the output layer.
* Calculate the output of the network.

**Backpropagation**

Now, we're ready to get our hands into training a neural network. For this, we'll use the method known as **backpropagation**. In a nutshell, backpropagation will consist of:

* Doing a feedforward operation.
* Comparing the output of the model with the desired output.
* Calculating the error.
* Running the feedforward operation backwards (backpropagation) to spread the error to each of the weights.
* Use this to update the weights, and get a better model.
* Continue this until we have a model that is good.

Sounds more complicated than what it actually is. Let's take a look in the next few videos. The first video will show us a conceptual interpretation of what backpropagation is.

### **Backpropagation Math**

And the next few videos will go deeper into the math. Feel free to tune out, since this part gets handled by Keras pretty well. If you'd like to go start training networks right away, go to the next section. But if you enjoy calculating lots of derivatives, let's dive in!

In the video below at 1:24, the edges should be directed to the sigmoid function and not the bias at that last layer; the edges of the last layer point to the bias currently which is incorrect.

### **Chain Rule**

We'll need to recall the chain rule to help us calculate derivatives.

### **Calculation of the derivative of the sigmoid function**

Recall that the sigmoid function has a beautiful derivative, which we can see in the following calculation. This will make our backpropagation step much cleaner.

A picture containing clock

Description automatically generated

## **Further reading**

Backpropagation is fundamental to deep learning. TensorFlow and other libraries will perform the backprop for you, but you should really really understand the algorithm. We'll be going over backprop again, but here are some extra resources for you:

* From Andrej Karpathy: [Yes, you should understand backprop](https://medium.com/@karpathy/yes-you-should-understand-backprop-e2f06eab496b#.vt3ax2kg9)
* Also from Andrej Karpathy, [a lecture from Stanford's CS231n course](https://www.youtube.com/watch?v=59Hbtz7XgjM)