# Quick Tutorials

<http://pandas.pydata.org/pandas-docs/stable/10min.html#min>

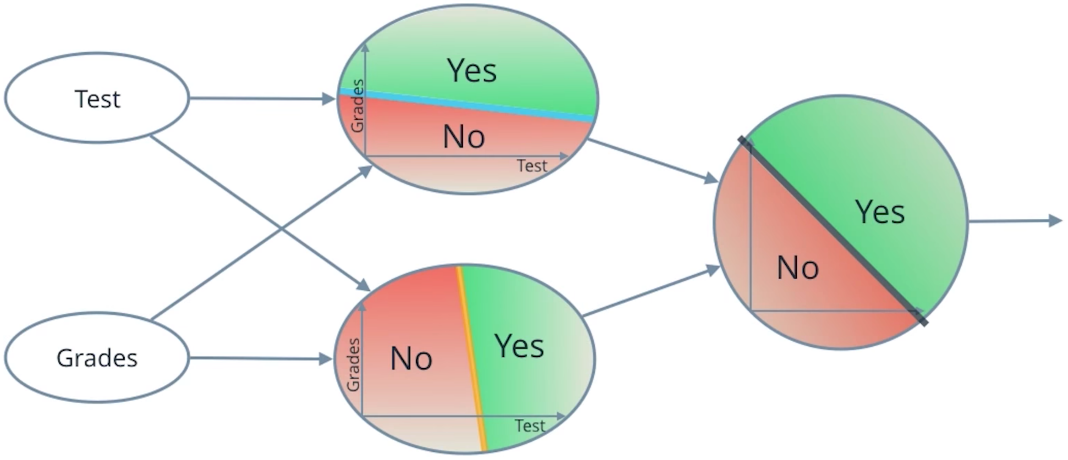
<http://scikit-learn.org/stable/tutorial/basic/tutorial.html>

<http://matplotlib.org/users/pyplot_tutorial.html>

<http://cs231n.github.io/python-numpy-tutorial/>

<https://docs.scipy.org/doc/numpy-dev/user/quickstart.html>

<https://jakevdp.github.io/blog/2016/08/25/conda-myths-and-misconceptions/>

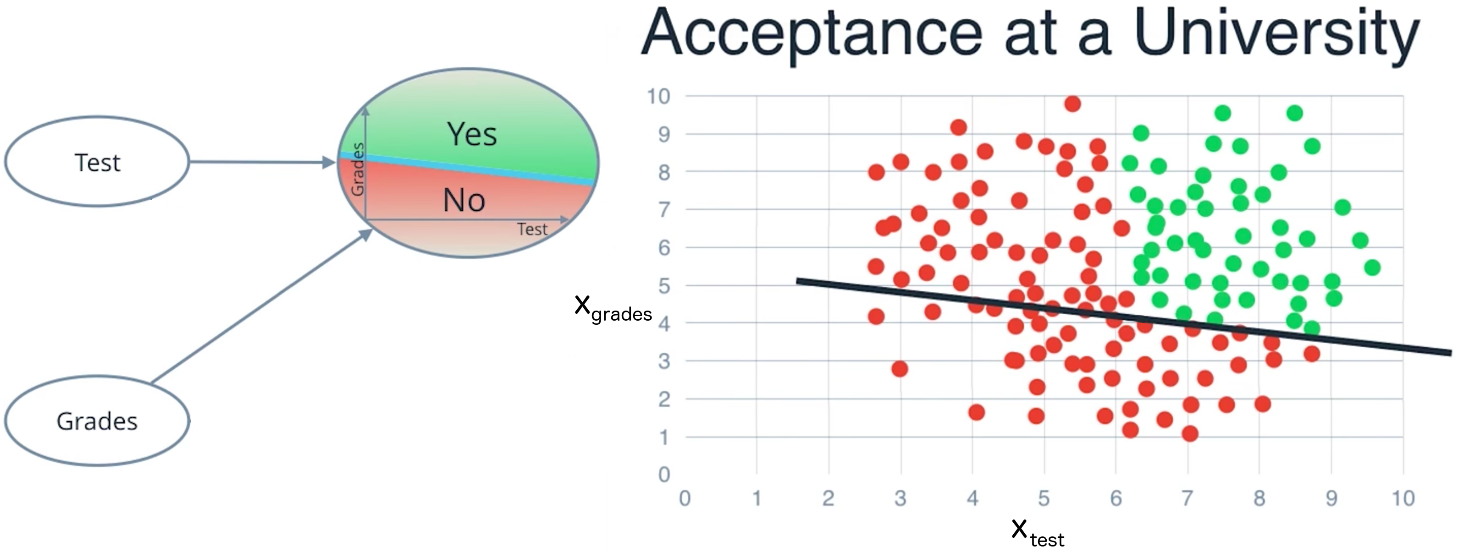


Neural Network

**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 and learn more about how this input data is processed.

Data, like test scores and grades, is fed into a network of interconnected nodes. These individual nodes are called **[perceptrons](https://en.wikipedia.org/wiki/Perceptron" \t "_blank)** or 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.



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 data comes into a perceptron, it gets multiplied by a weight value that is assigned to this particular input. For example, the perceptron above have 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 the previous weights resulted in. This is called **training** the neural network.

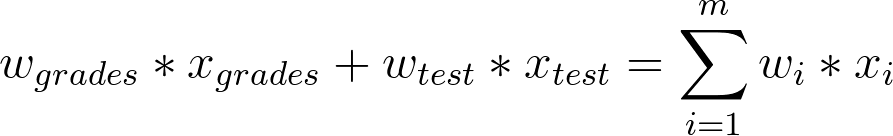
A higher weight means the neural network considers that input more important than other inputs, and lower weight means that the data is considered less important. An extreme example would be if test scores had no affect at all on university acceptance; then the weight of the test score input data would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

**Summing the Input Data**

So, each input to a perceptron has an associated weight that represents its importance and these weights are determined during the learning process of a neural network, called training. In the next step, the weighted input data is summed up 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.

We'll use *w*​*grades*​​ for the weight of grades and *w*​*test*​​ for the weight of test. For the image above, let's say that the weights are: *w*​*grades*​​=−1,*w*​*test*​​ =−0.2. You don't have to be concerned with the actual values, but their relative values are important. *w*​*grades*​​ is 5 times larger than *w*​*test*​​, 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. After applying these weights to the input data, the perceptron then sums these numbers to get a value we'll call the **linear combination**, which is pictured below. In the equation, *x*​*grades*​​ represents grades and *x*​*test*​​ represents test scores.

The linear combination represents how much the perceptron believes a student will be accepted into a university.

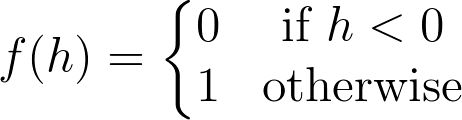


Linear Combination

**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**.

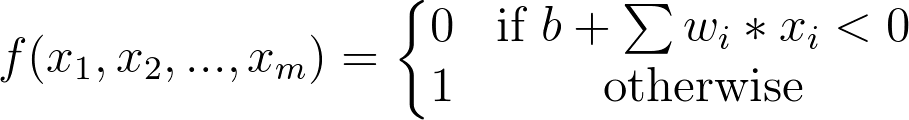
One of the simplest activation functions is the **Heaviside step function**. This function returns a 0 if the linear combination it sees is negative or equal to zero, and 1 in any other case (when the linear combination is positive). The [**Heaviside step function**](https://en.wikipedia.org/wiki/Heaviside_step_function) is shown below, where h is the calculated linear combination:



Heaviside Step Function

In the university acceptance example above, we used the weights *w*​*grades*​​=−1,*w*​*test*​​ =−0.2. Since *w*​*grades*​​and *w*​*test*​​ are negative values, the activation function will only return a 1 if grades and test are 0! This is because the range of values from the linear combination using these weights and inputs are (−∞,0].

We want more than one set of inputs to return a 1. We want a range of scores and grades that will be acceptable for the university. You can solve this problem by adding a single number to the linear combination called a **bias**, b. Just like the weights, the bias is also updated and changed by the neural network during training. Now with this value, we have a complete perceptron formula:



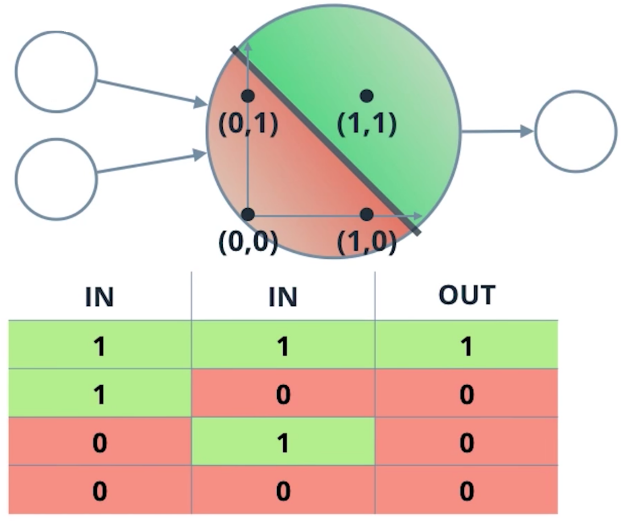
Perceptron Formula

This formula returns 1 if the input (*x*​1​​,*x*​2​​,...,*x*​*m*​​) belongs to the accepted-to-university category or returns 0if 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*​​, where *m* is the number of inputs.

Then the neural network starts to learn! Initially, the weights ( *w*​*i*​​) and bias (*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 perceptions, 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.

**AND Perceptron Quiz**



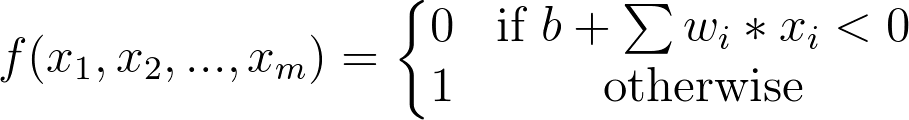
AND Perceptron

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

Set the weights (weight1, weight2) and bias bias to the correct values that calculate AND operation as shown above.

In this case, there are two inputs as seen in the table above (let's call the first column input1 and the second column input2), and based on the perceptron formula, we can calculate the output.

First, the linear combination will be the sum of the weighted inputs: linear\_combination = weight1\*input1 + weight2\*input2 then we can put this value into the *biased* Heaviside step function, which will give us our output (0 or 1):



Perceptron Formula

import pandas as pd

# TODO: Set weight1, weight2, and bias

weight1 = 0.0

weight2 = 0.0

bias = 0.0

# DON'T CHANGE ANYTHING BELOW

# Inputs and outputs

test\_inputs = [(0, 0), (0, 1), (1, 0), (1, 1)]

correct\_outputs = [False, False, False, True]

outputs = []

# Generate and check output

for test\_input, correct\_output in zip(test\_inputs, correct\_outputs):

linear\_combination = weight1 \* test\_input[0] + weight2 \* test\_input[1] + bias

output = int(linear\_combination >= 0)

is\_correct\_string = 'Yes' if output == correct\_output else 'No'

outputs.append([test\_input[0], test\_input[1], linear\_combination, output, is\_correct\_string])

# Print output

num\_wrong = len([output[4] for output in outputs if output[4] == 'No'])

output\_frame = pd.DataFrame(outputs, columns=['Input 1', ' Input 2', ' Linear Combination', ' Activation Output', ' Is Correct'])

if not num\_wrong:

print('Nice! You got it all correct.\n')

else:

print('You got {} wrong. Keep trying!\n'.format(num\_wrong))

print(output\_frame.to\_string(index=False))

If you still need a hint, think of a concrete example like so:

Consider input1 and input2 both = 1, for an AND perceptron, we want the output to also equal 1! The output is determined by the weights and Heaviside step function such that

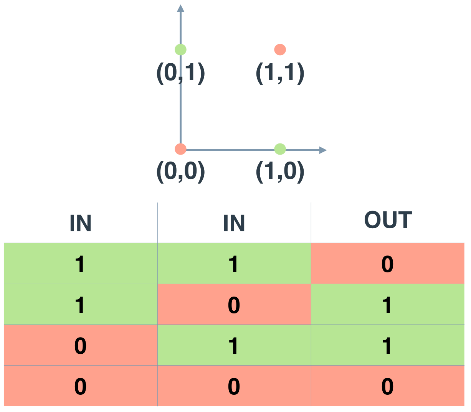
output = 1, **if** weight1\*input1 + weight2\*input2 + bias >= 0

or

output = 0, **if** weight1\*input1 + weight2\*input2 + bias < 0

So, how can you choose the values for weights and bias so that if both inputs = 1, the output = 1?

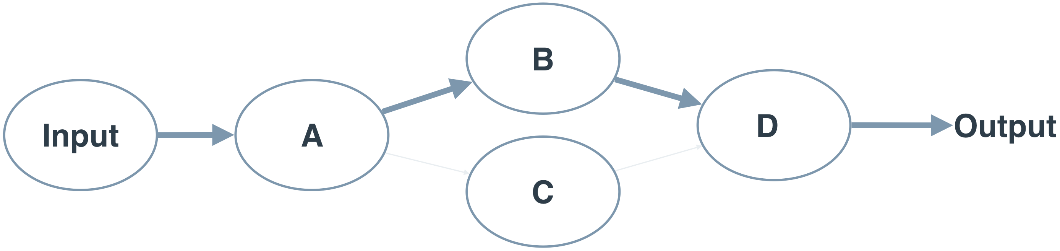
# XOR Perceptron



XOR Truth Table and Graph

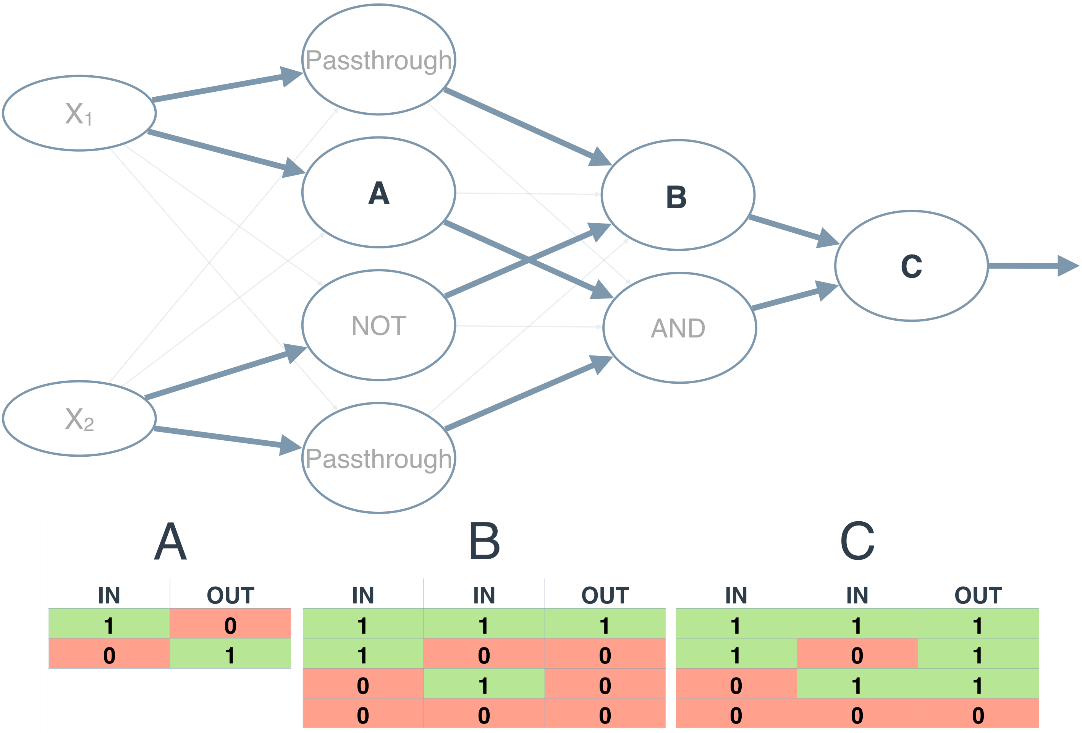
An XOR perceptron is a logic gate that outputs 0 if the inputs are the same and 1 if the inputs are different. Unlike previous perceptrons, this graph isn't linearly separable. To handle more complex problems like this, we can chain perceptrons together.

Let's build a neural network from the AND, NOT, and OR perceptrons to create XOR logic. Let's first go over what a neural network looks like.



The above neural network contains 4 perceptrons, A, B, C, and D. The input to the neural network is from the first node. The output comes out of the last node. The weights are based on the line thickness between the perceptrons. Any link between perceptrons with a low weight, like A to C, you can ignore. For perceptron C, you can ignore all input to and from it. For simplicity we wont be showing bias, but it's still in the neural network.

## Quiz



The neural network above calculates XOR. Each perceptron is a logic operation of OR, AND, Passthrough, or NOT. The **[Passthrough](https://en.wikipedia.org/wiki/Passthrough" \t "_blank)** operation just passes it's input to the output. However, the perceptrons A , B, and C don't indicate their operation. In the following quiz, set the correct operations for the three perceptrons to calculate XOR.

Note: Any line with a low weight can be ignored.

### **QUIZ QUESTION**

Set the operations for the perceptrons in the XOR neural network?

NOT

AND

OR

### PERCEPTRON

### OPERATIONS

A

B

C

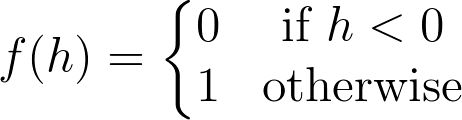
SUBMIT

You've seen that a perceptron can solve linearly separable problems. Solving more complex problems, you use more perceptrons. You saw this by calculating AND, OR, NOT, and XOR operations using perceptrons. These operations can be used to create any computer program. With enough data and time, a neural network can solve any problem that a computer can calculate. However, you don't build a Twitter using a neural network. A neural network is like any tool, you have to know when to use it.

The power of a neural network isn't building it by hand, like we were doing. It's the ability to learn from examples. In the next few sections, you'll learn how a neural networks sets it's own weights and biases.

# The simplest neural network

So far you've been working with perceptrons where the output is always one or zero. The input to the output unit is passed through an activation function, *f*(*h*), in this case, the step function.



The step activation function.

Here, *h* is the input to the output unit,

*h*=∑​*i*​​*w*​*i*​​*x*​*i*​​+*b*.

You can see an example below, with the output of the perceptron labeled *a*.

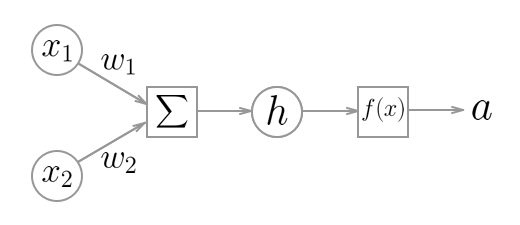


Diagram of a simple neural network. Circles are units, boxes are operations.

The cool part about this architecture, and what makes neural networks possible, is that the activation function, *f*(*h*) can be any function.

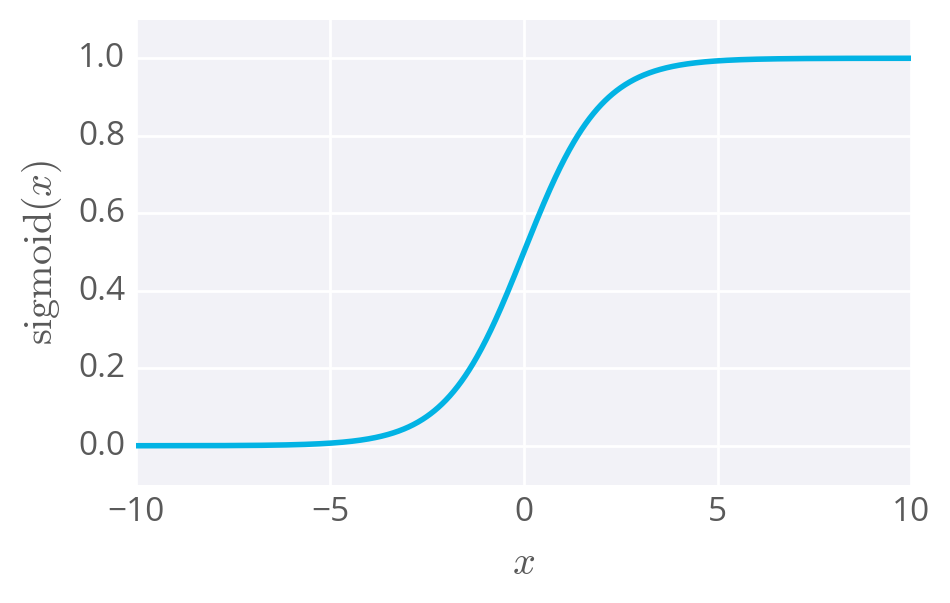
For example, if you let *f*(*h*)=*h* the output will be the same as the input. Now the output of the network is

*a*=∑​*i*​​*w*​*i*​​*x*​*i*​​+*b*.

This equation should be familiar to you, it's the same as the linear regression model!

Other activation functions you'll see are the logistic (often called the sigmoid), tanh, and softmax functions. We'll mostly be using the sigmoid function for the rest of this lesson:

sigmoid(*x*)=1/(1+*e*​−*x*​​)



The sigmoid function

The sigmoid function is bounded between 0 and 1, and as an output can be interpreted as a probability for success. It turns out, again, using a sigmoid as the activation function results in the same formulation as logistic regression.

This is where it stops being a perceptron and begins being called a neural network. In the case of simple networks like this, neural networks don't offer any advantage over general linear models such as logistic regression. But, as you saw with the XOR perceptron, stacking units will let you model linearly inseparable data, impossible to do with regression models.

Once you start using activation functions that are continuous and differentiable, it's possible to train the network using gradient descent, which you'll learn about next.

## Simple network exercise

Below, you'll use Numpy to calculate the output of a simple network with two input nodes and one output node with a sigmoid activation function. Thing's you'll need to do:

* Implement the sigmoid function.
* Calculate the output of the network.

As a reminder, the sigmoid function is

sigmoid(*x*)=1/(1+*e*​−*x*​​)

For the exponential, you can use Numpy's exponential function, np.exp.

And the output of the network is

*a*=*f*(*h*)=sigmoid(∑​*i*​​*w*​*i*​​*x*​*i*​​+*b*)

For the weights sum, you can do a simple element-wise multiplication and sum, or Numpy's [**dot product function**](https://docs.scipy.org/doc/numpy/reference/generated/numpy.dot.html).

# Learning weights

You've seen how you can use perceptrons for AND and XOR operations, but there we set the weights by hand. What if you want to perform an operation, such as predicting college admission, but don't know the correct weights? You'll need to learn the weights from example data, then use those weights to make the predictions.

To figure out how we're going to find these weights, start by thinking about the goal. We want the network to make predictions as close as possible to the real values. To measure this, we need a metric of how wrong the predictions are, the **error**. A common metric is the sum of the squared errors (SSE):

*E*=​2​​1​​∑​*μ*​​∑​*j*​​[*y*​*j*​*μ*​​−​*y*​^​​​*j*​*μ*​​]​2​​

where ​*y*​^​​ is the prediction and *y* is the true value, and you take the sum over all output units *j* and another sum over all data points *μ*. The SSE is a good choice for a few reasons. The square ensures the error is always positive and larger errors are penalized more than smaller errors. Also, it makes the math nice, always a plus.

Remember that the output of a neural network, the prediction, depends on the weights

​*y*​^​​​*j*​*μ*​​=*f*(∑​*i*​​*w*​*ij*​​*x*​*i*​*μ*​​)

and accordingly the error depends on the weights

*E*=​2​​1​​∑​*μ*​​∑​*j*​​[*y*​*j*​*μ*​​−*f*(∑​*i*​​*w*​*ij*​​*x*​*i*​*μ*​​)]​2​​

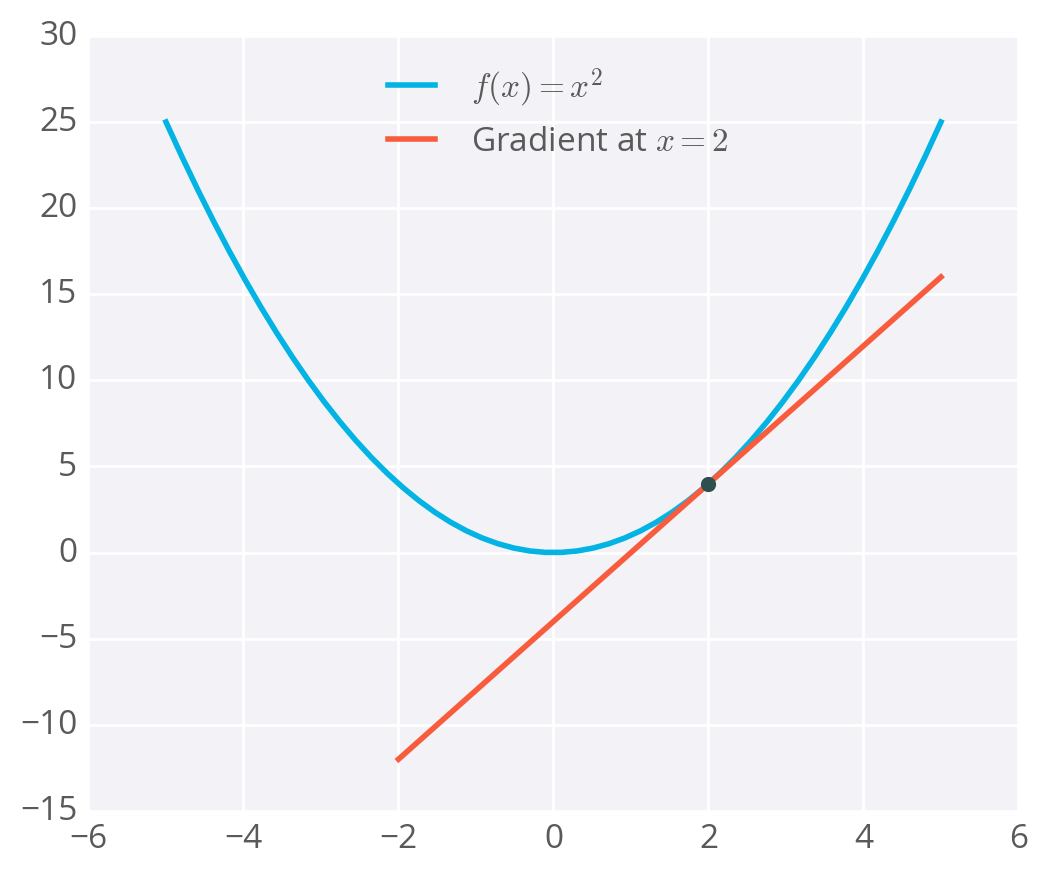
We want the network's prediction error to be as small as possible and the weights are the knobs we can use to make that happen. Our goal is to find weights *w*​*ij*​​ that minimize the squared error *E*. To do this with a neural network, typically you'd use **gradient descent**.

## Enter Gradient Descent

As Luis said, with gradient descent, we take multiple small steps towards our goal. In this case, we want to change the weights in steps that reduce the error. Continuing the analogy, the error is our mountain and we want to get to the bottom. Since the fastest way down a mountain is in the steepest direction, the steps taken should be in the direction that minimizes the error the most. We can find this direction by calculating the gradient of the squared error.

Gradient is another term for rate of change or slope. If you need to brush up on this concept, check out Khan Academy's [**great lectures**](https://www.khanacademy.org/math/multivariable-calculus/multivariable-derivatives/gradient-and-directional-derivatives/v/gradient) on the topic.

To calculate a rate of change, we turn to calculus, specifically derivatives. A derivative of a function *f*(*x*) gives you another function *f*​′​​(*x*) that returns the slope of *f*(*x*) at point *x*. For example, consider *f*(*x*)=*x*​2​​. The derivative of *x*​2​​ is *f*​′​​(*x*)=2*x*. So, at *x*=2, the slope is *f*​′​​(2)=4. Plotting this out, it looks like:

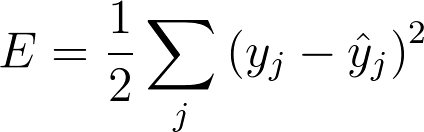


Example of a gradient

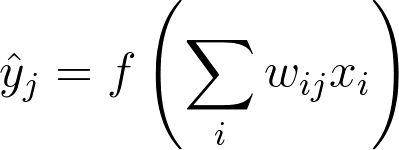
The gradient is just a derivative generalized to functions with more than one variable. We can use calculus to find the gradient at any point in our error function, which depends on the input weights.

### Math time

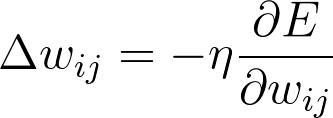
Our goal here is to calculate the gradient of the error which means we need the partial derivatives of the error with respect to each of the weights. To keep things simple, I'm only going to consider one update step from one data point. From before, the error is:



and the prediction ​*y*​^​​​*j*​​ is



We want to update our weights by an amount Δ*w*​*ij*​​ proportional to the gradient of the error, but in the opposite direction (hence the negative sign below). This gives us the following equation for Δ*w*​*ij*​​:

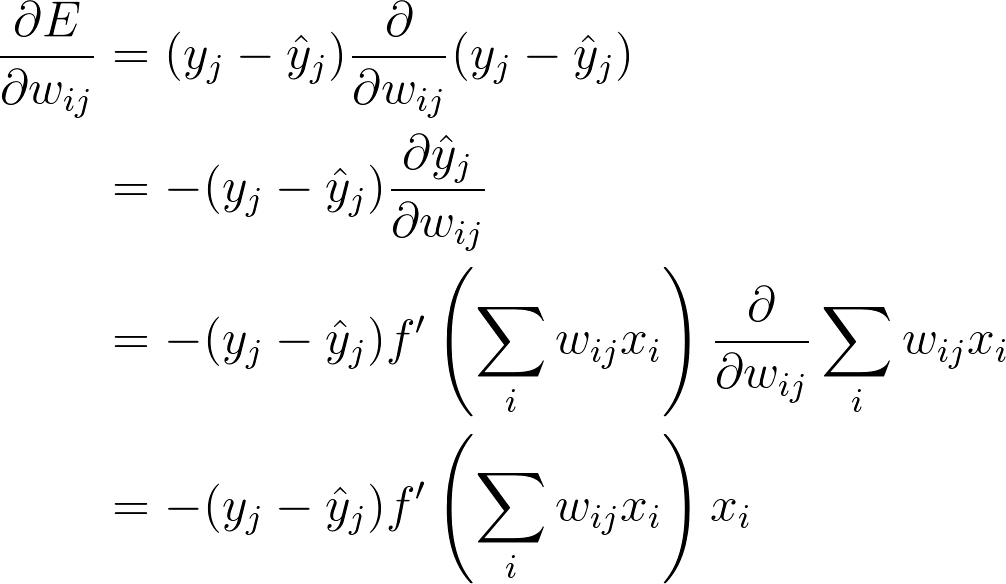


Here, *η* is the learning rate which allows us to scale the step size in the gradient descent updates.

Let's now understand what ​∂*w*​*ij*​​​​∂*E*​​ will be. We're going to be using the chain rule here so be sure to brush up on it [**here**](https://www.khanacademy.org/math/ap-calculus-ab/product-quotient-chain-rules-ab/chain-rule-ab/v/chain-rule-introduction).

Remember, *E*=​2​​1​​∑​*j*​​[(*y*​*j*​​−​*y*​*j*​​​^​​)​2​​].

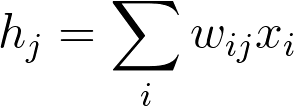
Using the chain rule,



And finally, putting everything together,

https://d17h27t6h515a5.cloudfront.net/topher/2017/January/588ac7c8_grad-weight-step/grad-weight-step.png

where *h*​*j*​​ is the input to the output unit *j*



Let's go over the intuition for this result. (*y*​*j*​​−​*y*​^​​​*j*​​) is the prediction error. The larger this error is, the larger the step should be. When the error is small, our steps can be smaller since the weights are near the minimum. Similarly, there's the input term, *x*​*i*​​. Since larger inputs drive more error, we scale the weight proportional to this term.

The next part is the gradient, *f*​′​​(*h*​*j*​​). This is proportional to the effect unit *j* has on the output. Remember that the gradient is a rate of change. If the gradient is small, then a change in the unit input *h*​*j*​​ will have a small effect on the error. And conversely, if the gradient is large, a change in the unit input will have a large effect. This term produces larger gradient descent steps for units that have larger gradients and therefore effect the change in error more.

If we define the errors, *δ*​*j*​​, as

https://d17h27t6h515a5.cloudfront.net/topher/2017/January/588ac949_delta/delta.png

then we can write the gradient descent step as

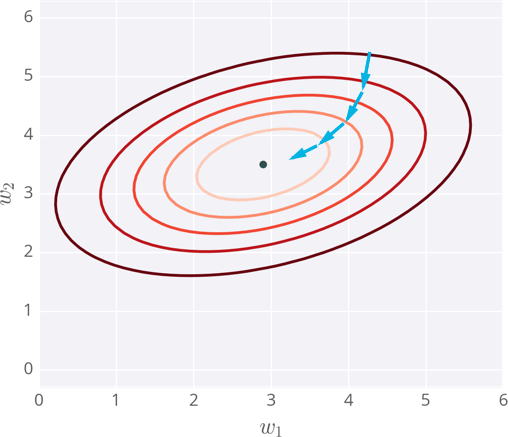
https://d17h27t6h515a5.cloudfront.net/topher/2017/January/588ac981_weight-step-delta/weight-step-delta.png

This notation will come in handy later when we're dealing with backpropagation.

Note: You might be wondering how bias terms work with this. For most of this lesson, the bias term will be implicit in derivations and code examples. It's really easy to deal with biases though, you use the normal weight results, but set *x*​*i*​​=1. So, Δ*b*​*j*​​=*ηδ*​*j*​​.

Below I've plotted an example of the error of a neural network with two inputs, and accordingly, two weights. You can read this like a topographical map where points on a contour line have the same error and darker contour lines correspond to larger errors.

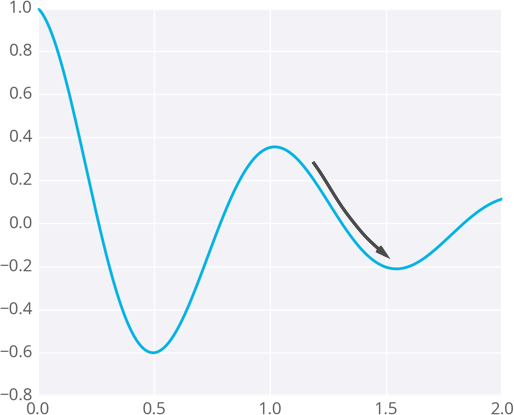
At each step, you calculate the error and the gradient, then use those to determine how much to change each weight. Repeating this process will eventually find weights that are close to the minimum of the error function, the block dot in the middle.



Gradient descent steps to the lowest error

# Caveats

Since the weights will just go where ever the gradient takes them, they can end up where the error is low, but not the lowest. These spots are called local minima. If the weights are initialized with the wrong values, gradient descent could lead the weights into a local minimum, illustrated below.



Gradient descent leading into a local minimum

There are methods to avoid this, such as using [**momentum**](http://sebastianruder.com/optimizing-gradient-descent/index.html#momentum).

## Gradient descent exercise

Below, you'll calculate one gradient descent step for the weights of a simple network with two inputs and one output unit. Your goal here is to calculate the correct weight step using gradient descent.

Remember that the weight step is the learning rate times the error times the input values:

Δ*w*​*ij*​​=*ηδ*​*j*​​*x*​*i*​​.

You'll need to calculate the error gradient, *δ*​*j*​​=(*y*−​*y*​^​​)*f*​′​​(*h*), which consists of the output error (the target *y*minus the output ​*y*​^​​) and the gradient of the activation function. A nice thing about using the sigmoid function for the activations is that it's derivative can be written in terms of the sigmoid function itself:

*f*​′​​(*h*)=*f*(*h*)(1−*f*(*h*)).

You calculate *f*(*h*) to get the output, it's the activation of the output unit. So you can just use that for the derivative in the error.

import numpy as np

def sigmoid(x):

"""

Calculate sigmoid

"""

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

learnrate = 0.5

x = np.array([1, 2])

y = np.array(0.5)

# Initial weights

w = np.array([0.5, -0.5])

# Calculate one gradient descent step for each weight

# TODO: Calculate output of neural network

nn\_output = None

# TODO: Calculate error of neural network

error = None

# TODO: Calculate change in weights

del\_w = None

print('Neural Network output:')

print(nn\_output)

print('Amount of Error:')

print(error)

print('Change in Weights:')

print(del\_w)

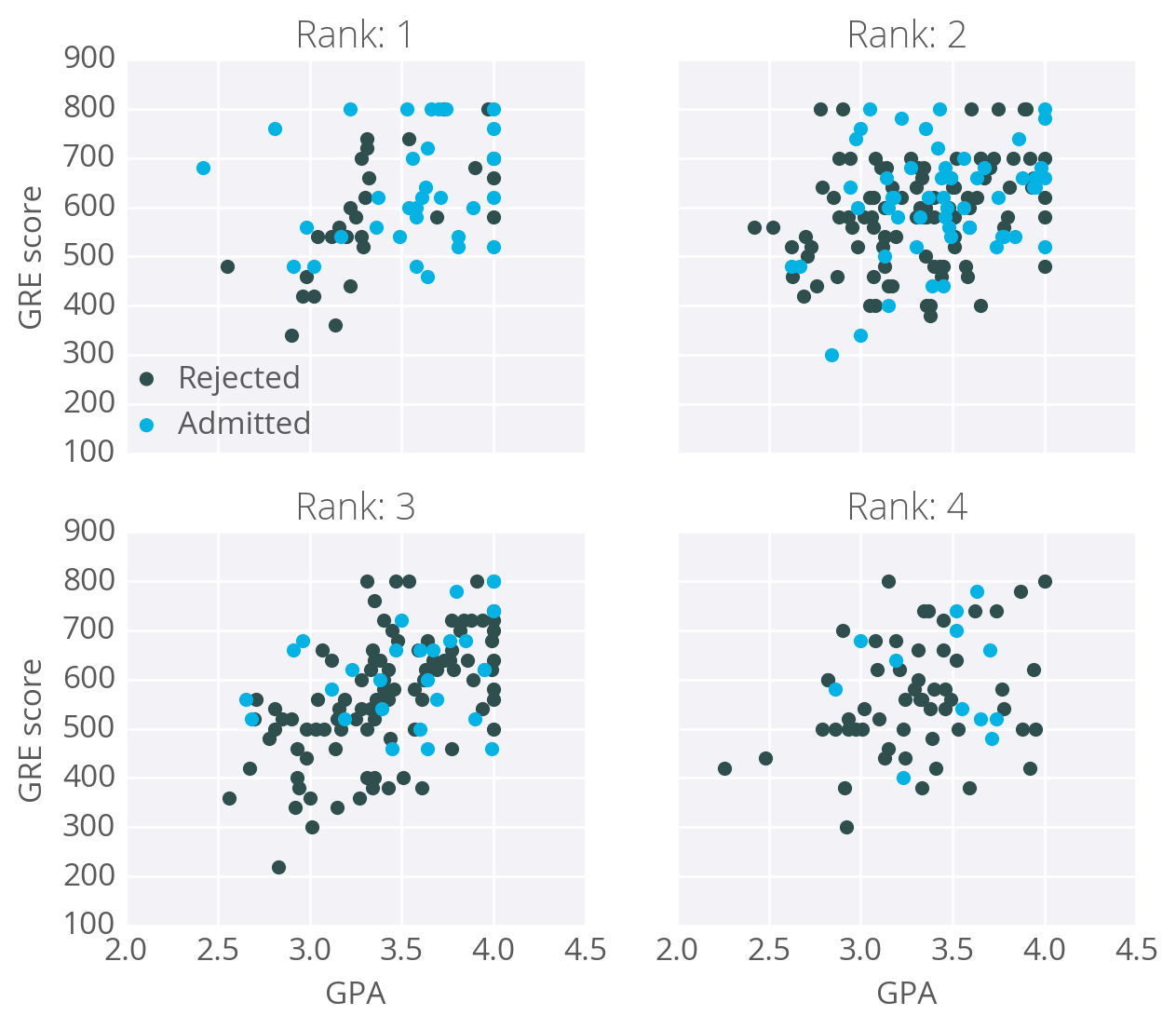
# Implementing gradient descent

Okay, now we know how to update our weights

Δ*w*​*ij*​​=*ηδ*​*j*​​*x*​*i*​​,

how do we translate this into code?

As an example, I'm going to have you use gradient descent to train a network on graduate school admissions data (found at [**http://www.ats.ucla.edu/stat/data/binary.csv**](http://www.ats.ucla.edu/stat/data/binary.csv). This dataset has three input features: GRE score, GPA, and the rank of the undergraduate school (numbered 1 through 4). Institutions with rank 1 have the highest prestige, those with rank 4 have the lowest.



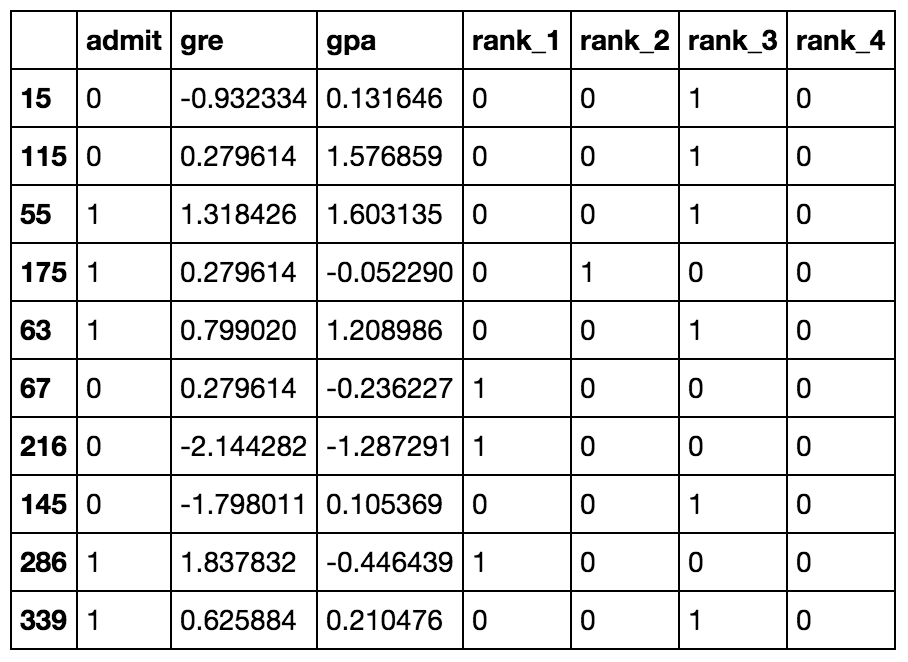
The goal here is to predict if a student will be admitted to a graduate program based on these features. For this, we'll use a network with one output layer with one unit. We'll use a sigmoid function for the output unit activation.

## Data cleanup

You might think there will be three input units, but we actually need to transform the data first. The rankfeature is categorical, the numbers don't encode any sort of relative values. Rank 2 is not twice as much as rank 1, rank 3 is not 1.5 more than rank 2. Instead, we need to use [**dummy variables**](https://en.wikipedia.org/wiki/Dummy_variable_(statistics)) to encode rank, splitting the data into four new columns encoded with ones or zeros. Rows with rank 1 have one in the rank 1 dummy column, and zeros in all other columns. Rows with rank 2 have one in the rank 2 dummy column, and zeros in all other columns. And so on.

We'll also need to standardize the GRE and GPA data, which means to scale the values such they have zero mean and a standard deviation of 1. This is necessary because the sigmoid function squashes really small and really large inputs. The gradient of really small and large inputs is zero, which means that the gradient descent step will go to zero too. Since the GRE and GPA values are fairly large, we have to be really careful about how we initialize the weights or the gradient descent steps will die off and the network won't train. Instead, if we standardize the data, we can initialize the weights easily and everyone is happy.

This is just a brief run-through, you'll learn more about preparing data later. If you're interested in how I did this, check out the data\_prep.py file in the programming exercise below.



Ten rows of the data after transformations.

Now that the data is ready, we see that there are six input features: gre, gpa, and the four rank dummy variables.

Here's the general algorithm for updating the weights with gradient descent:

* Set the weight step to zero: Δ*w*​*i*​​=0
* For each record in the training data:
  + Make a forward pass through the network, calculating the output ​*y*​^​​=*f*(∑​*i*​​*w*​*i*​​*x*​*i*​​)
  + Calculate the error gradient in the output unit, *δ*=(*y*−​*y*​^​​)∗*f*​′​​(∑​*i*​​*w*​*i*​​*x*​*i*​​)
  + Update the weight step Δ*w*​*i*​​=Δ*w*​*i*​​+*δx*​*i*​​
* Update the weights *w*​*i*​​=*w*​*i*​​+*η*Δ*w*​*i*​​/*m* where *η* is the learning rate and *m* is the number of records. Here we're averaging the weight steps to help reduce any large variations in the training data.
* Repeat for *e* epochs.

You can also update the weights on each record instead of averaging the weight steps after going through all the records.

Remember that we're using the sigmoid for the activation function, *f*(*h*)=1/(1+*e*​−*x*​​)

And the gradient of the sigmoid is *f*​′​​(*h*)=*f*(*h*)(1−*f*(*h*))

where *h* is the input to the output unit,

*h*=∑​*i*​​*w*​*i*​​*x*​*i*​​

## Implementing with Numpy

For the most part, this is pretty straightforward with Numpy.

First, you'll need to initialize the weights. We want these to be small such that the input to the sigmoid is in the linear region near 0 and not squashed at the high and low ends. It's also important to initialize them randomly so that they all have different starting values and diverge, breaking symmetry. So, we'll initialize the weights from a normal distribution centered at 0. A good value for the scale is 1/√​*n*​​​ where *n* is the number of input units. This keeps the input to the sigmoid low for increasing numbers of input units.

weights = np.random.normal(scale=1/n\_features\*\*-.5, size=n\_features)

Numpy provides a function that calculates the dot product of two arrays, which conveniently calculates *h* for us. The dot product multiplies two arrays element-wise, the first element in array 1 is multiplied by the first element in array 2, and so on. Then, each product is summed.

*# input to the output layer*

output\_in = np.dot(weights, inputs)

And finally, we can update Δ*w*​*i*​​ and *w*​*i*​​ by incrementing them with weights += ... which is shorthand for weights = weights + ....

### Efficiency tip!

You can save some calculations since we're using a sigmoid here. For the sigmoid function, *f*​′​​(*h*)=*f*(*h*)(1−*f*(*h*)). That means that once you calculate *f*(*h*), the activation of the output unit, you can use it to calculate the gradient for the error gradient.

## Programming exercise

Below, you'll implement gradient descent and train the network on the admissions data. Your goal here is to train the network until you reach a minimum in the mean square error (MSE) on the training set. You need to implement:

* The network output: output.
* The error gradient: error.
* Update the weight step: del\_w +=.
* Update the weights: weights +=.

After you've written these parts, run the training by pressing "Test Run". The MSE will print out, as well as the accuracy on a test set, the fraction of correctly predicted admissions.

Feel free to play with the hyperparameters and see how it changes the MSE.