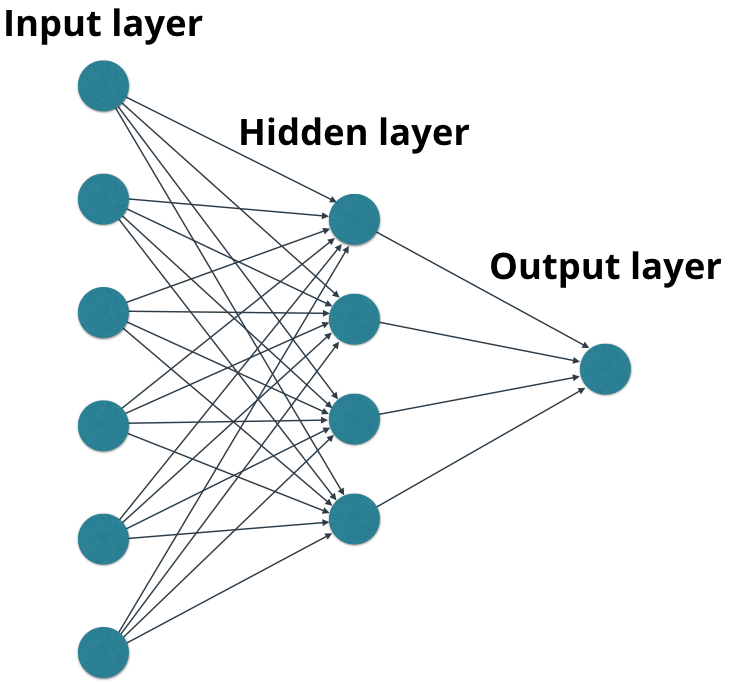
**What is a Neural Network?**



An example neural network.

A neural network is a graph of mathematical functions such as [**linear combinations**](https://en.wikipedia.org/wiki/Linear_combination) and activation functions. The graph consists of **nodes**, and **edges**.

Nodes in each layer (except for nodes in the input layer) perform mathematical functions using inputs from nodes in the previous layers. For example, a node could represent *f*(*x*,*y*)=*x*+*y*, where *x* and *y* are input values from nodes in the previous layer.

Similarly, each node creates an output value which may be passed to nodes in the next layer. The output value from the output layer does not get passed to a future layer (last layer!)

Layers between the input layer and the output layer are called **hidden layers**.

**Forward Propagation**

By propagating values from the first layer (the input layer) through all the mathematical functions represented by each node, the network outputs a value. This process is called a **forward pass**.

Here's an example of a simple forward pass.

Notice that the output layer performs a mathematical function, addition, on its inputs. There is no hidden layer.

**Graphs**

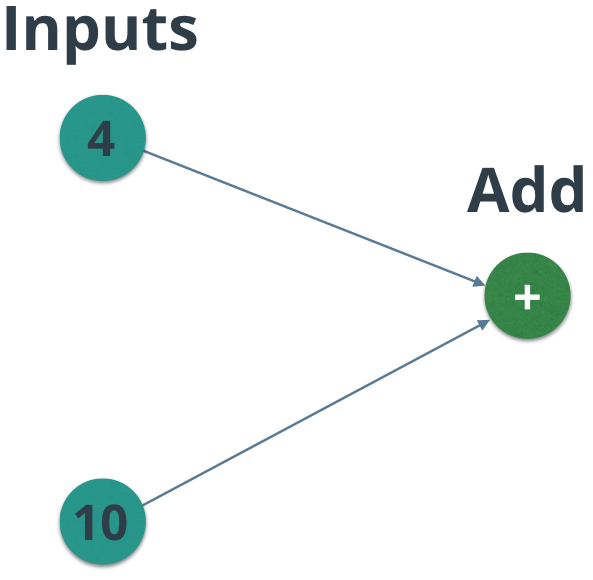
The nodes and edges create a graph structure. Though the example above is fairly simple, it isn't hard to imagine that increasingly complex graphs can calculate . . . well . . . *almost anything*.

There are generally two steps to create neural networks:

1. Define the graph of nodes and edges.
2. Propagate values through the graph.

MiniFlow works the same way. You'll define the nodes and edges of your network with one method and then propagate values through the graph with another method. MiniFlow comes with some starter code to help you out. Let's take a look.

**Graph Quiz**



QUIZ QUESTION

In the graph above, what will the value of the addition node be?

* 

4

* 

10

* 14
* 

0

# MiniFlow Architecture

Let's consider how to implement this graph structure in MiniFlow. We'll use a Python class to represent a generic node.

**class** **Node**(object):

**def** **\_\_init\_\_**(self):

*# Properties will go here!*

We know that each node might receive input from multiple other nodes. We also know that each node creates a single output, which will likely be passed to other nodes. Let's add two lists: one to store references to the inbound nodes, and the other to store references to the outbound nodes.

**class** **Node**(object):

**def** **\_\_init\_\_**(self, inbound\_nodes=[]):

*# Node(s) from which this Node receives values*

self.inbound\_nodes = inbound\_nodes

*# Node(s) to which this Node passes values*

self.outbound\_nodes = []

*# For each inbound Node here, add this Node as an outbound Node to \_that\_ Node.*

**for** n **in** self.inbound\_nodes:

n.outbound\_nodes.append(self)

Each node will eventually calculate a value that represents its output. Let's initialize the value to None to indicate that it exists but hasn't been set yet.

**class** **Node**(object):

**def** **\_\_init\_\_**(self, inbound\_nodes=[]):

*# Node(s) from which this Node receives values*

self.inbound\_nodes = inbound\_nodes

*# Node(s) to which this Node passes values*

self.outbound\_nodes = []

*# For each inbound Node here, add this Node as an outbound Node to \_that\_ Node.*

**for** n **in** self.inbound\_nodes:

n.outbound\_nodes.append(self)

*# A calculated value*

self.value = **None**

Each node will need to be able to pass values forward and perform backpropagation (more on that later). For now, let's add a placeholder method for forward propagation. We'll deal with backpropagation later on.

**class** **Node**(object):

**def** **\_\_init\_\_**(self, inbound\_nodes=[]):

*# Node(s) from which this Node receives values*

self.inbound\_nodes = inbound\_nodes

*# Node(s) to which this Node passes values*

self.outbound\_nodes = []

*# For each inbound Node here, add this Node as an outbound Node to \_that\_ Node.*

**for** n **in** self.inbound\_nodes:

n.outbound\_nodes.append(self)

*# A calculated value*

self.value = **None**

**def** **forward**(self):

"""

Forward propagation.

Compute the output value based on `inbound\_nodes` and

store the result in self.value.

"""

**raise** NotImplemented

### Nodes that Calculate

While Node defines the base set of properties that every node holds, only specialized [**subclasses**](https://docs.python.org/3/tutorial/classes.html#inheritance) of Node will end up in the graph. As part of this lab, you'll build the subclasses of Node that can perform calculations and hold values. For example, consider the Input subclass of Node.

**class** **Input**(Node):

**def** **\_\_init\_\_**(self):

*# An Input node has no inbound nodes,*

*# so no need to pass anything to the Node instantiator.*

Node.\_\_init\_\_(self)

*# NOTE: Input node is the only node where the value*

*# may be passed as an argument to forward().*

*#*

*# All other node implementations should get the value*

*# of the previous node from self.inbound\_nodes*

*#*

*# Example:*

*# val0 = self.inbound\_nodes[0].value*

**def** **forward**(self, value=None):

*# Overwrite the value if one is passed in.*

**if** value **is** **not** **None**:

self.value = value

Unlike the other subclasses of Node, the Input subclass does not actually calculate anything. The Inputsubclass just holds a value, such as a data feature or a model parameter (weight/bias).

You can set value either explicitly or with the forward() method. This value is then fed through the rest of the neural network.

### The Add Subclass

Add, which is another subclass of Node, actually can perform a calculation (addition).

**class** **Add**(Node):

**def** **\_\_init\_\_**(self, x, y):

Node.\_\_init\_\_(self, [x, y])

**def** **forward**(self):

"""

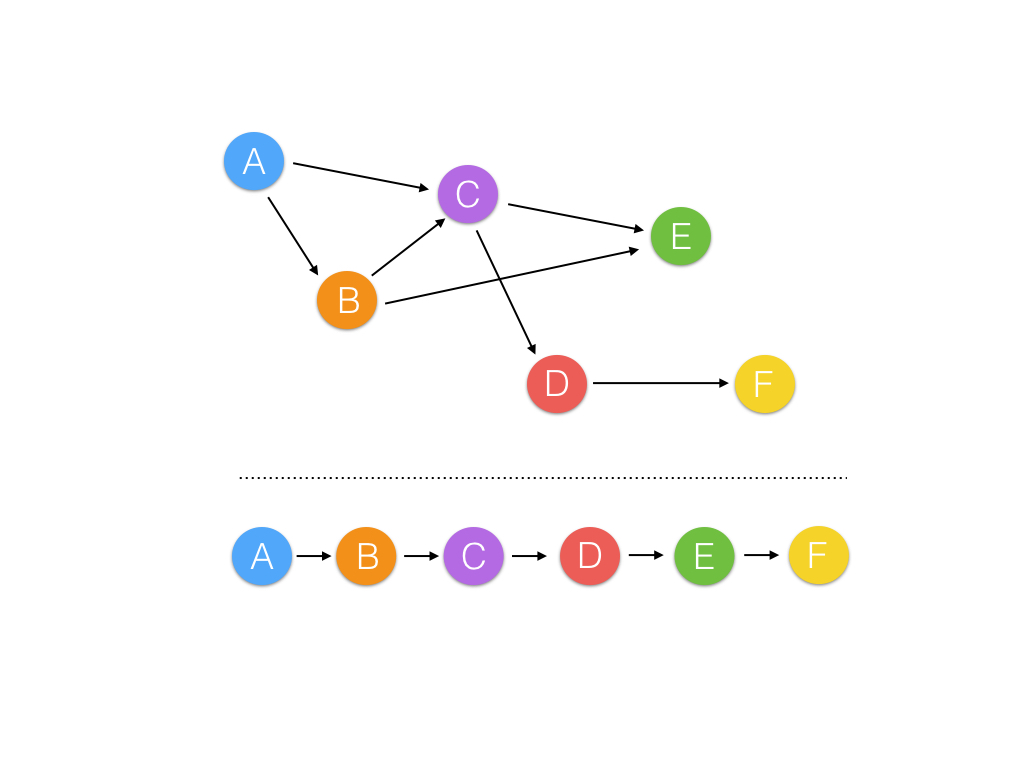
You'll be writing code here in the next quiz!

"""

Notice the difference in the \_\_init\_\_ method, Add.\_\_init\_\_(self, [x, y]). Unlike the Input class, which has no inbound nodes, the Add class takes 2 inbound nodes, x and y, and adds the values of those nodes.

# Forward propagation

MiniFlow has two methods to help you define and then run values through your graphs: topological\_sort() and forward\_pass().



An example of topological sorting

In order to define your network, you'll need to define the order of operations for your nodes. Given that the input to some node depends on the outputs of others, you need to flatten the graph in such a way where all the input dependencies for each node are resolved before trying to run its calculation. This is a technique called a [**topological sort**](https://en.wikipedia.org/wiki/Topological_sorting).

The topological\_sort() function implements topological sorting using [**Kahn's Algorithm**](https://en.wikipedia.org/wiki/Topological_sorting#Kahn.27s_algorithm). The details of this method are not important, the result is; topological\_sort() returns a sorted list of nodes in which all of the calculations can run in series. topological\_sort() takes in a feed\_dict, which is how we initially set a value for an Input node. The feed\_dict is represented by the Python dictionary data structure. Here's an example use case:

*# Define 2 `Input` nodes.*

x, y = Input(), Input()

*# Define an `Add` node, the two above`Input` nodes being the input.*

add = Add(x, y)

*# The value of `x` and `y` will be set to 10 and 20 respectively.*

feed\_dict = {x: 10, y: 20}

*# Sort the nodes with topological sort.*

sorted\_nodes = topological\_sort(feed\_dict=feed\_dict)

(You can find the source code for topological\_sort() in miniflow.py in the programming quiz below.)

The other method at your disposal is forward\_pass(), which actually runs the network and outputs a value.

**def** **forward\_pass**(output\_node, sorted\_nodes):

"""

Performs a forward pass through a list of sorted nodes.

Arguments:

`output\_node`: The output node of the graph (no outgoing edges).

`sorted\_nodes`: a topologically sorted list of nodes.

Returns the output node's value

"""

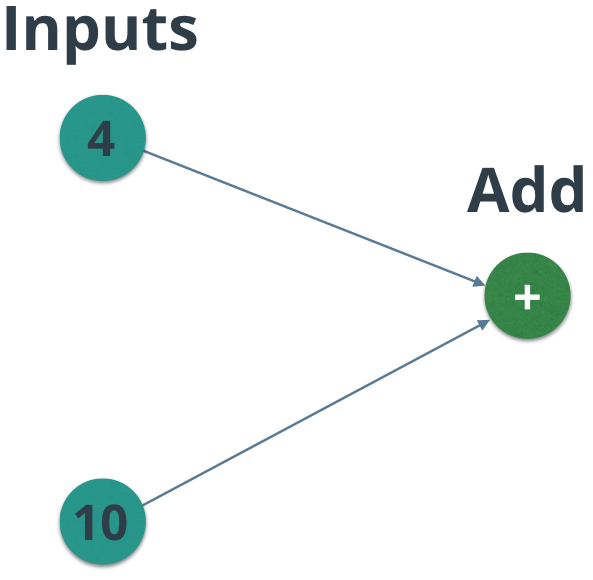
**for** n **in** sorted\_nodes:

n.forward()

**return** output\_node.value

### Quiz 1 - Passing Values Forward

Create and run this graph!



The graph you'll run in this quiz. The node values may change, though!

### Setup

Review nn.py and miniflow.py.

The neural network architecture is already there for you in nn.py. It's your job to finish MiniFlow to make it work.

For this quiz, I want you to:

1. Open nn.py below. **You don't need to change anything.** I just want you to see how MiniFlow works.
2. Open miniflow.py. **Finish the forward method on the Add class. All that's required to pass this quiz is a correct implementation of forward.**
3. Test your network by hitting "Test Run!" When the output looks right, hit "Submit!"

(You'll find the solution on the next page.)

* [**nn.py**](https://classroom.udacity.com/nanodegrees/nd101/parts/2a9dba0b-28eb-4b0e-acfa-bdcf35680d90/modules/269a4aad-9025-4354-a0ea-2623e889540a/lessons/b6deebe4-7f78-4947-b2c6-fc660ca942fb/concepts/48bad937-5bfd-4e38-9e61-337ff69f9fb7)
* [**miniflow.py**](https://classroom.udacity.com/nanodegrees/nd101/parts/2a9dba0b-28eb-4b0e-acfa-bdcf35680d90/modules/269a4aad-9025-4354-a0ea-2623e889540a/lessons/b6deebe4-7f78-4947-b2c6-fc660ca942fb/concepts/48bad937-5bfd-4e38-9e61-337ff69f9fb7)





1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

"""

This script builds and runs a graph with miniflow.

There is no need to change anything to solve this quiz!

However, feel free to play with the network! Can you also

build a network that solves the equation below?

(x + y) + y

"""

from miniflow import \*

x, y = Input(), Input()

f = Add(x, y)

feed\_dict = {x: 10, y: 5}

sorted\_nodes = topological\_sort(feed\_dict)

output = forward\_pass(f, sorted\_nodes)

# NOTE: because topological\_sort set the values for the `Input` nodes we could also access

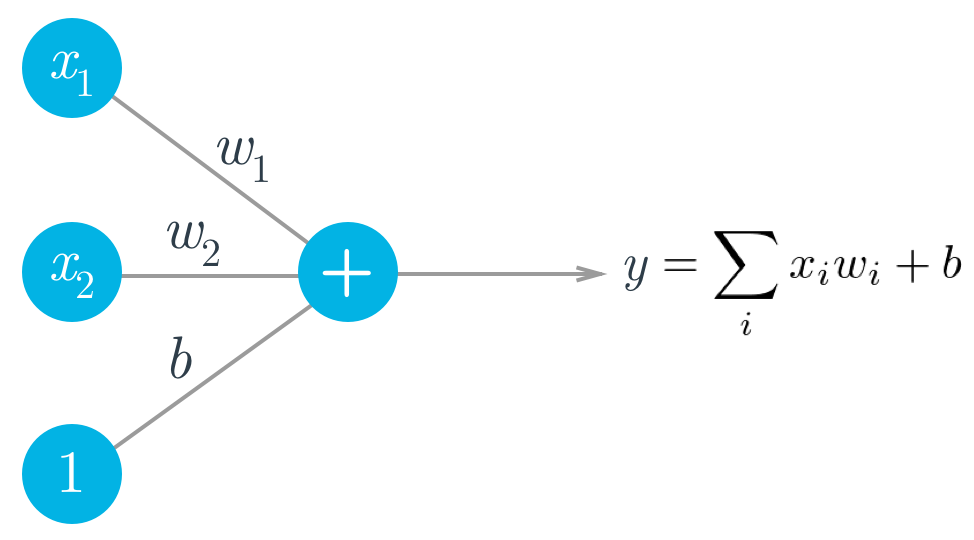
# the value for x with x.value (same goes for y).

print("{} + {} = {} (according to miniflow)".format(feed\_dict[x], feed\_dict[y], output))

**Learning and Loss**

Like MiniFlow in its current state, neural networks take inputs and produce outputs. But unlike MiniFlow in its current state, neural networks can *improve* the accuracy of their outputs over time (it's hard to imagine improving the accuracy of Add over time!). To explore why accuracy matters, I want you to first implement a trickier (and more useful!) node than Add: the Linear node.

**The Linear Function**



Think back to the Intro to Neural Networks lesson. A simple artificial neuron depends on three components:

* inputs, *x*​*i*​​
* weights, *w*​*i*​​
* bias, *b*

The output, *y*, is just the weighted sum of the inputs plus the bias.

Remember, by varying the weights, you can vary the amount of influence any given input has on the output. The learning aspect of neural networks takes place during a process known as backpropagation. In backpropogation, the network modifies the weights to improve the network's output accuracy. You'll be applying all of this shortly.

In this next quiz, you'll try to build a linear neuron that generates an output by applying a simplified version of the weighted sum. Linear should take an list of inbound nodes of length *n*, a list of weights of length *n*, and a bias.

**Instructions**

1. Open nn.py below. Read through the neural network to see the expected output of Linear.
2. Open miniflow.py below. Modify Linear, which is a subclass of Node, to generate an output with*y*=∑*w*​*i*​​*x*​*i*​​+*b*.

(Hint: you could use numpy to solve this quiz if you'd like, but it's possible to solve this with vanilla Python.)

* [**nn.py**](https://classroom.udacity.com/nanodegrees/nd101/parts/2a9dba0b-28eb-4b0e-acfa-bdcf35680d90/modules/269a4aad-9025-4354-a0ea-2623e889540a/lessons/b6deebe4-7f78-4947-b2c6-fc660ca942fb/concepts/96656ffd-8c94-43f4-a962-8d44319f8c9f)
* [**miniflow.py**](https://classroom.udacity.com/nanodegrees/nd101/parts/2a9dba0b-28eb-4b0e-acfa-bdcf35680d90/modules/269a4aad-9025-4354-a0ea-2623e889540a/lessons/b6deebe4-7f78-4947-b2c6-fc660ca942fb/concepts/96656ffd-8c94-43f4-a962-8d44319f8c9f)

NOTE: Here we're using an Input node for more than a scalar.

In the case of weights and inputs the value of the Input node is

actually a python list!

In general, there's no restriction on the values that can be passed to an Input node.

"""

from miniflow import \*

inputs, weights, bias = Input(), Input(), Input()

f = Linear(inputs, weights, bias)

feed\_dict = {

inputs: [6, 14, 3],

weights: [0.5, 0.25, 1.4],

bias: 2

}

graph = topological\_sort(feed\_dict)

output = forward\_pass(f, graph)

print(output) # should be 12.7 with this example

#### Solution to Linear Node

Here's my solution to the last quiz:

**class** **Linear**(Node):

**def** **\_\_init\_\_**(self, inputs, weights, bias):

Node.\_\_init\_\_(self, [inputs, weights, bias])

**def** **forward**(self):

"""

Set self.value to the value of the linear function output.

Your code goes here!

"""

inputs = self.inbound\_nodes[0].value

weights = self.inbound\_nodes[1].value

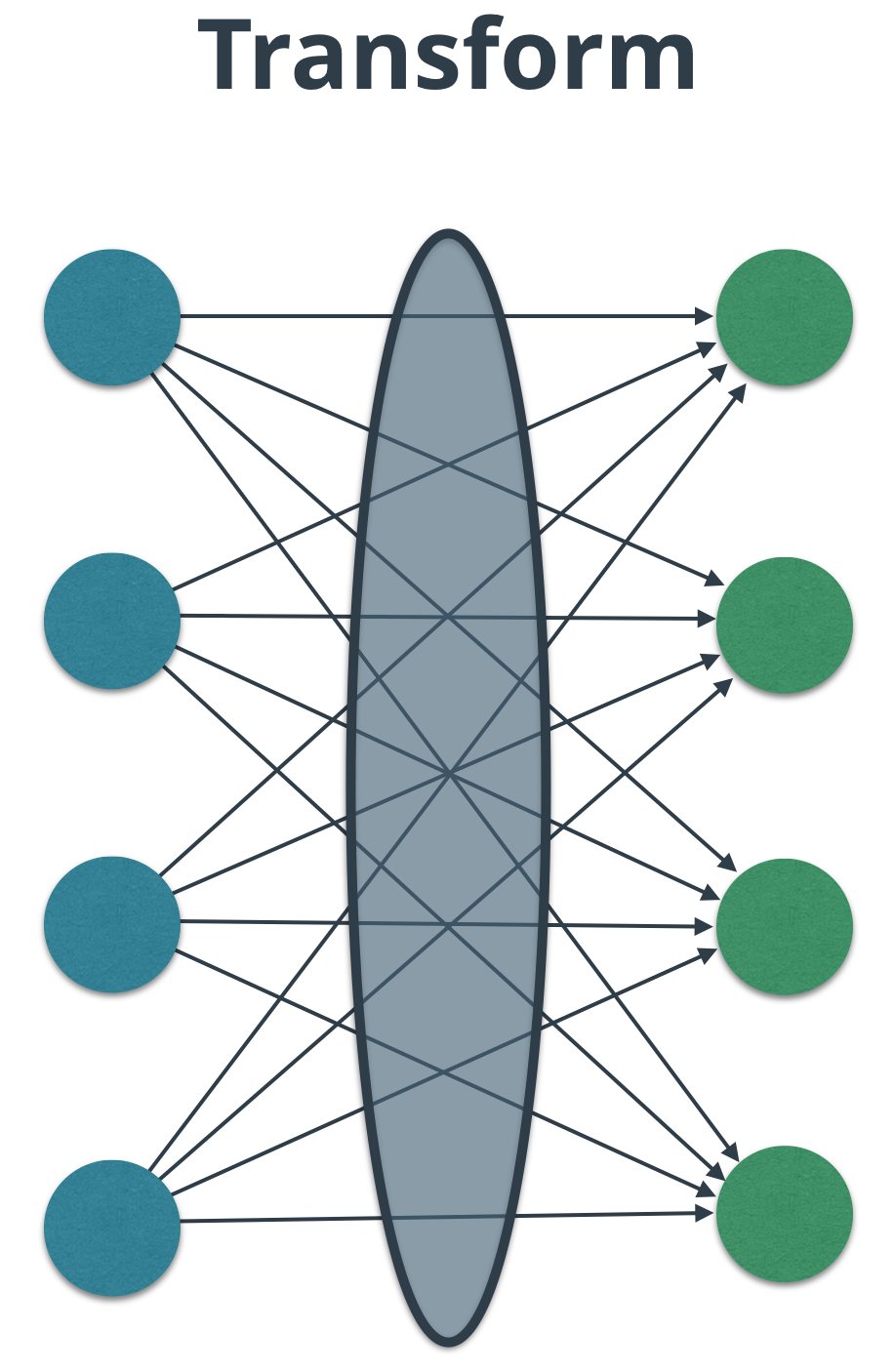
bias = self.inbound\_nodes[2]

self.value = bias.value

**for** x, w **in** zip(inputs, weights):

self.value += x \* w

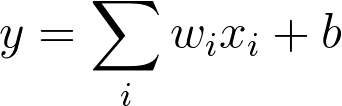
In the solution, I set self.value to the bias and then loop through the inputs and weights, adding each weighted input to self.value. Notice calling .value on self.inbound\_nodes[0] or self.inbound\_nodes[1] gives us a list.



Shift your thinking here to the edges between layers.

[**Linear algebra**](https://www.khanacademy.org/math/linear-algebra) nicely reflects the idea of transforming values between layers in a graph. In fact, the concept of a [**transform**](https://www.khanacademy.org/math/linear-algebra/matrix-transformations/linear-transformations/v/vector-transformations) does exactly what a layer should do - it converts inputs to outputs in many dimensions.

Let's go back to our equation for the output.



Equation (1)

For the rest of this section we'll denote x as X and w as W since they are now matrices, and b is now a vector instead of a scalar.

Consider a Linear node with 1 input and k outputs (mapping 1 input to k outputs). In this context an input/output is synonymous with a feature.

In this case X is a 1 by 1 matrix.

1 by 1 matrix, 1 element.

1 by 1 matrix, 1 element.

W becomes a 1 by k matrix (looks like a row).

A 1 by k weights row matrix.

A 1 by k weights row matrix.

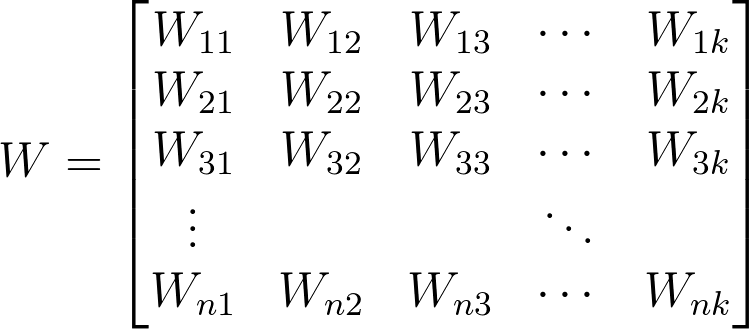
The result of the matrix multiplication of X and W is a 1 by k matrix. Since b is also a 1 by k row matrix (1 bias per output), b is added to the output of the X and W matrix multiplication.

What if we are mapping n inputs to k outputs?

Then X is now a 1 by n matrix and W is a n by k matrix. The result of the matrix multiplication is still a 1 by k matrix so the use of the biases remain the same.

X is now a 1 by n matrix, n inputs/features.

X is now a 1 by n matrix, n inputs/features.



Row matrix of biases, one for each output.

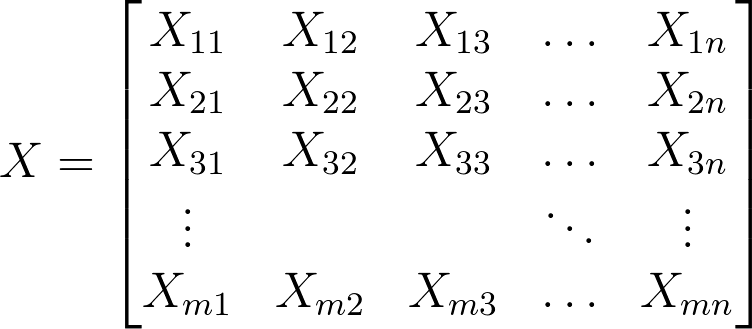
Row matrix of biases, one for each output.

Let's take a look at an example of n inputs. Consider an 28px by 28px greyscale image, as is in the case of images in the [**MNIST dataset**](http://yann.lecun.com/exdb/mnist/), a set of handwritten digits. We can reshape the image such that it's a 1 by 784 matrix, n = 784. Each pixel is an input/feature. Here's an animated example emphasizing a pixel is a feature.

In practice, it's common to feed in multiple data examples in each forward pass rather than just 1. The reasoning for this is the examples can be processed in parallel, resulting in big performance gains. The number of examples is called the batch size. Common numbers for the batch size are 32, 64, 128, 256, 512. Generally, it's the most we can comfortably fit in memory.

What does this mean for X, W and b?

X becomes a m by n matrix and W and b remain the same. The result of the matrix multiplication is now m by k, so the addition of b is [**broadcast**](https://docs.scipy.org/doc/numpy/user/basics.broadcasting.html) over each row.



X is now an m by n matrix. Each row has n inputs/features.

In the context of MNIST each row of X is an image reshaped from 28 by 28 to 1 by 784.

Equation (2)

Equation (2)

Equation (2) can also be viewed as Z = XW + B where B is the biases vector, b, stacked m times as a row. Due to broadcasting it's abbreviated to Z = XW + b.

I want you to rebuild Linear to handle matrices and vectors using the venerable Python math package numpyto make your life easier. numpy is often abbreviated as np, so we'll refer to it as np when referring to code.

I used np.array ([**documentation**](https://docs.scipy.org/doc/numpy/reference/generated/numpy.array.html)) to create the matrices and vectors. You'll want to use np.dot, which functions as matrix multiplication for 2D arrays ([**documentation**](https://docs.scipy.org/doc/numpy/reference/generated/numpy.dot.html)), to multiply the input and weights matrices from Equation (2). It's also worth noting that numpy actually overloads the \_\_add\_\_ operator so you can use it directly with np.array (eg. np.array() + np.array()).

### Instructions

1. Open nn.py. See how the neural network implements the Linear node.
2. Open miniflow.py. Implement Equation (2) within the forward pass for the Linear node.
3. Test your work!

ere's my solution to the last quiz:

**class** **Linear**(Node):

**def** **\_\_init\_\_**(self, X, W, b):

*# Notice the ordering of the inputs passed to the*

*# Node constructor.*

Node.\_\_init\_\_(self, [X, W, b])

**def** **forward**(self):

X = self.inbound\_nodes[0].value

W = self.inbound\_nodes[1].value

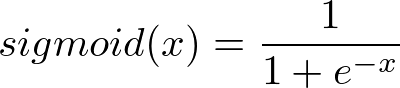
b = self.inbound\_nodes[2].value

self.value = np.dot(X, W) + b

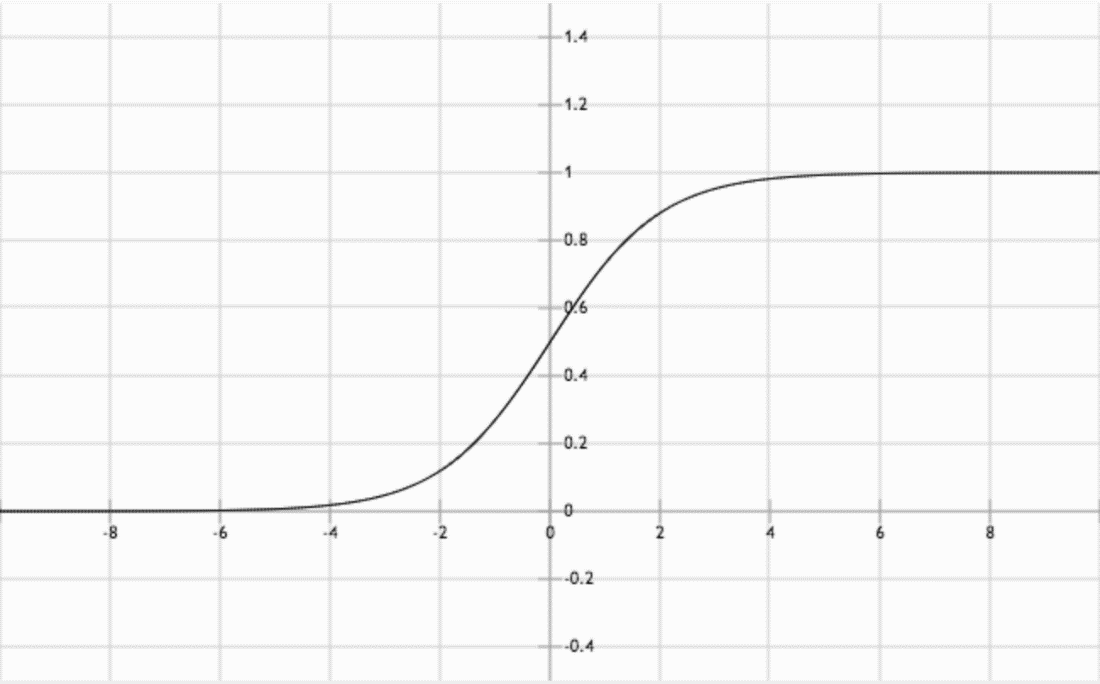
Nothing fancy in my solution. I pulled the value of the X, W and b from their respective inputs. I used np.dot to handle the matrix multiplication.

Neural networks take advantage of alternating transforms and activation functions to better categorize outputs. The sigmoid function is among the most common activation functions.

**Sigmoid Function**



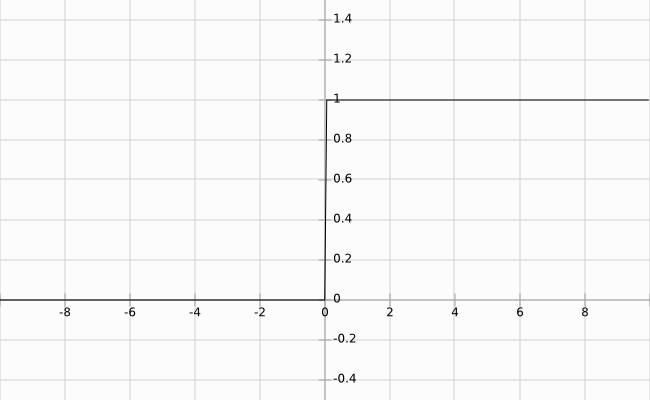
Equation (3)



Graph of the sigmoid function. Notice the "S" shape.

Linear transforms are great for simply *shifting* values, but neural networks often require a more nuanced transform. For instance, one of the original designs for an artificial neuron, [**the perceptron**](https://en.wikipedia.org/wiki/Perceptron), exhibit binary output behavior. Perceptrons compare a weighted input to a threshold. When the weighted input exceeds the threshold, the perceptron is **activated** and outputs 1, otherwise it outputs 0.

You could model a perceptron's behavior as a step function.



Example of a step function (The jump between y = 0 and y = 1 should be instantaneous).

Activation, the idea of binary output behavior, generally makes sense for classification problems. For example, if you ask the network to hypothesize if a handwritten image is a '9', you're effectively asking for a binary output - *yes*, this is a '9', or *no*, this is not a '9'. A step function is the starkest form of a binary output, which is great, but step functions are not continuous and not differentiable, which is *very bad*. Differentiation is what makes gradient descent possible.

The sigmoid function, Equation (3) above, replaces thresholding with a beautiful S-shaped curve (also shown above) that mimics the activation behavior of a perceptron while maintaining continuity, and thus differentiability. As a bonus, the sigmoid function has a very simple derivative that looks remarkably similar to the sigmoid itself.

Equation (4). σ represents Equation (3)

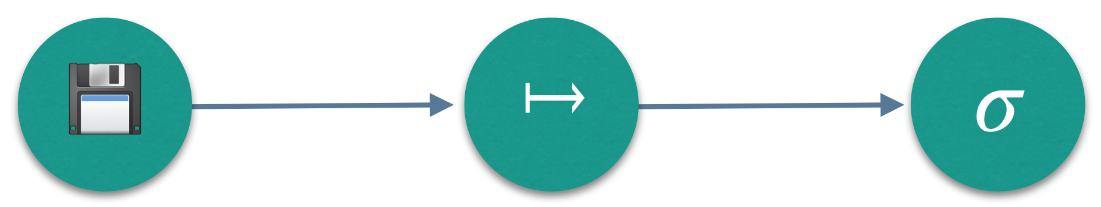
Equation (4). σ represents Equation (3)

Notice that the sigmoid function only has one parameter. Remember that sigmoid is an *activation* function (*non-linearity*), meaning it takes a single input and performs a mathematical operation on it.

Conceptually, the sigmoid function makes decisions. When given weighted features from some data, it indicates whether or not the features contribute to a classification. In that way, a sigmoid activation works well following a linear transformation. As it stands right now with random weights and bias, the sigmoid node's output is also random. The process of learning through backpropagation and gradient descent, which you will implement soon, modifies the weights and bias such that activation of the sigmoid node begins to match expected outputs.

Now that I've given you the equation for the sigmoid function, I want you to add it to the MiniFlow library. To do so, you'll want to use np.exp ([**documentation**](https://docs.scipy.org/doc/numpy/reference/generated/numpy.exp.html)) to make your life much easier.

You'll be using Sigmoid in conjunction with Linear. Here's how it should look:



Inputs > Linear Transform > Sigmoid

**Instructions**

1. Open nn.py to see how the network will use Sigmoid.
2. Open miniflow.py. Modify the forward method of the Sigmoid class to reflect the sigmoid function's behavior.
3. Test your work! Hit "Submit" when your Sigmoid works as expected.

* [**nn.py**](https://classroom.udacity.com/nanodegrees/nd101/parts/2a9dba0b-28eb-4b0e-acfa-bdcf35680d90/modules/269a4aad-9025-4354-a0ea-2623e889540a/lessons/b6deebe4-7f78-4947-b2c6-fc660ca942fb/concepts/9f421c5e-749c-4de5-ad0c-3812eb34d05d)
* [**miniflow.py**](https://classroom.udacity.com/nanodegrees/nd101/parts/2a9dba0b-28eb-4b0e-acfa-bdcf35680d90/modules/269a4aad-9025-4354-a0ea-2623e889540a/lessons/b6deebe4-7f78-4947-b2c6-fc660ca942fb/concepts/9f421c5e-749c-4de5-ad0c-3812eb34d05d)

This network feeds the output of a linear transform

to the sigmoid function.

Finish implementing the Sigmoid class in miniflow.py!

Feel free to play around with this network, too!

"""

import numpy as np

from miniflow import \*

X, W, b = Input(), Input(), Input()

f = Linear(X, W, b)

g = Sigmoid(f)

X\_ = np.array([[-1., -2.], [-1, -2]])

W\_ = np.array([[2., -3], [2., -3]])

b\_ = np.array([-3., -5])

feed\_dict = {X: X\_, W: W\_, b: b\_}

graph = topological\_sort(feed\_dict)

output = forward\_pass(g, graph)

"""

Output should be:

[[ 1.23394576e-04 9.82013790e-01]

[ 1.23394576e-04 9.82013790e-01]]

"""

print(output)

Here's how I implemented the sigmoid function.

**class** **Sigmoid**(Node):

**def** **\_\_init\_\_**(self, node):

Node.\_\_init\_\_(self, [node])

**def** **\_sigmoid**(self, x):

"""

This method is separate from `forward` because it

will be used with `backward` as well.

`x`: A numpy array-like object.

"""

**return** 1. / (1. + np.exp(-x)) *# the `.` ensures that `1` is a float*

**def** **forward**(self):

input\_value = self.inbound\_nodes[0].value

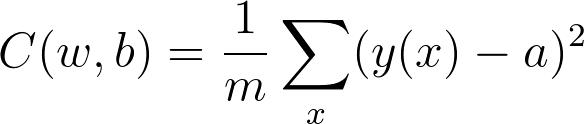
self.value = self.\_sigmoid(input\_value)

It may have seemed strange that \_sigmoid was a separate method. As seen in the derivative of the sigmoid function, Equation (4), the sigmoid function is actually *a part of its own derivative*. Keeping \_sigmoid separate means you won't have to implement it twice for forward and backward propagations.

This is exciting! At this point, you have used weights and biases to compute outputs. And you've used an activation function to categorize the output. As you may recall, neural networks improve the **accuracy** of their outputs by modifying weights and biases in response to training against labeled datasets.

There are many techniques for defining the accuracy of a neural network, all of which center on the network's ability to produce values that come as close as possible to known correct values. People use different names for this accuracy measurement, often terming it **loss** or **cost**. I'll use the term *cost* most often.

For this lab, you will calculate the cost using the mean squared error (MSE). It looks like so:



Equation (5)

Here *w* denotes the collection of all weights in the network, *b* all the biases, *m* is the total number of training examples, *a* is the approximation of *y(x)* by the network, both *a* and *y(x)* are vectors of the same length.

The collection of weights is all the weight matrices flattened into vectors and concatenated to one big vector. The same goes for the collection of biases except they're already vectors so there's no need to flatten them prior to the concatenation.

Here's an example of creating *w* in code:

*# 2 by 2 matrices*

w1 = np.array([[1, 2], [3, 4]])

w2 = np.array([[5, 6], [7, 8]])

*# flatten*

w1\_flat = np.reshape(w1, -1)

w2\_flat = np.reshape(w2, -1)

w = np.concatenate((w1\_flat, w2\_flat))

*# array([1, 2, 3, 4, 5, 6, 7, 8])*

It's a nice way to abstract all the weights and biases used in the neural network and makes some things easier to write as we'll see soon in the upcoming gradient descent sections.

**NOTE:** It's not required you do this in your code! It's just easier to do this talk about the weights and biases as a collective than consider them invidually.

The cost, *C*, depends on the difference between the correct output, *y(x)*, and the network's output, *a*. It's easy to see that no difference between *y(x)* and *a* (for all values of *x*) leads to a cost of 0.

This is the ideal situation, and in fact the learning process revolves around minimizing the cost as much as possible.

I want you to calculate the cost now.

You implemented this network in the forward direction in the last quiz.

As it stands right now, it outputs gibberish. The activation of the sigmoid node means nothing because the network has no labeled output against which to compare. Furthermore, the weights and bias cannot change and learning cannot happen without a cost.

**Instructions**

For this quiz, you will run the forward pass against the network in nn.py. I want you to finish implementing the MSE method so that it calculates the cost from the equation above.

I recommend using the np.square ([**documentation**](https://docs.scipy.org/doc/numpy/reference/generated/numpy.square.html)) method to make your life easier.

1. Check out nn.py to see how MSE will calculate the cost.
2. Open miniflow.py. Finish building MSE.
3. Test your network! See if the cost makes sense given the inputs by playing with nn.py.

Here's how I implemented MSE:

**class** **MSE**(Node):

**def** **\_\_init\_\_**(self, y, a):

"""

The mean squared error cost function.

Should be used as the last node for a network.

"""

*# Call the base class' constructor.*

Node.\_\_init\_\_(self, [y, a])

**def** **forward**(self):

"""

Calculates the mean squared error.

"""

*# NOTE: We reshape these to avoid possible matrix/vector broadcast*

*# errors.*

*#*

*# For example, if we subtract an array of shape (3,) from an array of shape*

*# (3,1) we get an array of shape(3,3) as the result when we want*

*# an array of shape (3,1) instead.*

*#*

*# Making both arrays (3,1) insures the result is (3,1) and does*

*# an elementwise subtraction as expected.*

y = self.inbound\_nodes[0].value.reshape(-1, 1)

a = self.inbound\_nodes[1].value.reshape(-1, 1)

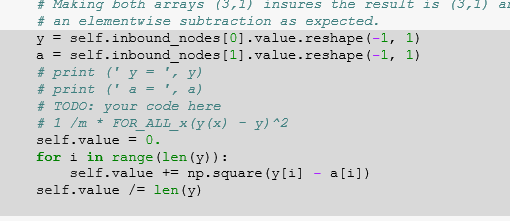
m = self.inbound\_nodes[0].value.shape[0]

diff = y - a

self.value = np.mean(diff\*\*2)

The math behind MSE reflects Equation (5), where *y* is target output and *a* is output computed by the neural network. We then square the difference diff\*\*2, alternatively, this could be np.square(diff). Lastly we need to sum the squared differences and divide by the total number of examples *m*. This can be achieved in with np.mean or (1 /m) \* np.sum(diff\*\*2).

Note the order of y and a doesn't actually matter, we could switch them around (a - y) and get the same value.



**Journey to the Bottom of the Valley**

Here I'll give you a little refresher on gradient descent so we can start training our network with MiniFlow. Remember that our goal is to make our network output as close as possible to the target values by minimizing the cost. You can envision the cost as a hill or mountain and we want to get to the bottom.

Imagine your model parameters are represented by a ball sitting on a hill. Intuitively, we want to push the ball downhill. And that makes sense, but when we're talking about our cost function, how do we know which way is downhill?

Luckily, the gradient provides this exact information.

Technically, the gradient actually points uphill, in the direction of **steepest ascent**. But if we put a - sign in front of this value, we get the direction of **steepest descent**, which is what we want.

You'll learn more about the gradient in a moment, but, for now, just think of it as a vector of numbers. Each number represents the amount by which we should adjust a corresponding weight or bias in the neural network. Adjusting all of the weights and biases by the gradient values reduces the cost (or error) of the network.

Got all that?

Great! Now we know where to push the ball. The next thing to consider is how much force should be applied to the *push*. This is known as the *learning rate*, which is an apt name since this value determines how quickly or slowly the neural network learns.

You might be tempted to set a really big learning rate, so the network learns really fast, right?

Be careful! If the value is too large you could overshoot the target and eventually diverge. Yikes!



**Convergence**. This is the ideal behaviour.



**Divergence**. This can happen when the learning rate is too large.

So what is a good learning rate, then?

This is more of a guessing game than anything else but empirically values in the range 0.1 to 0.0001 work well. The range 0.001 to 0.0001 is popular, as 0.1 and 0.01 are sometimes too large.

Here's the formula for gradient descent (pseudocode):

x = x - learning\_rate \* gradient\_of\_x

x is a parameter used by the neural network (i.e. a single weight or bias).

We multiply gradient\_of\_x (the uphill direction) by learning\_rate (the force of the push) and then subtract that from x to make the push go downhill.

Awesome! Time to apply all this in a quiz.

**Setup**

For this quiz you'll complete TODOs in both the f.py and gd.py files.

Tasks:

* Set the learning\_rate in f.py.
* Complete the gradient descent implementation in gradient\_descent\_update function in gd.py.

Notes:

* Setting the learning\_rate to 0.1 should result in x -> 0 and f(x) -> 5 if you've implemented gradient descent correctly.
* Play around with different values for the learning rate. Try very small values, values close to 1, above 1, etc. What happens?
* [**f.py**](https://classroom.udacity.com/nanodegrees/nd101/parts/2a9dba0b-28eb-4b0e-acfa-bdcf35680d90/modules/269a4aad-9025-4354-a0ea-2623e889540a/lessons/b6deebe4-7f78-4947-b2c6-fc660ca942fb/concepts/ce4e18b2-2777-40ca-9514-c659ada3f09a)
* [**gd.py**](https://classroom.udacity.com/nanodegrees/nd101/parts/2a9dba0b-28eb-4b0e-acfa-bdcf35680d90/modules/269a4aad-9025-4354-a0ea-2623e889540a/lessons/b6deebe4-7f78-4947-b2c6-fc660ca942fb/concepts/ce4e18b2-2777-40ca-9514-c659ada3f09a)

from gd import gradient\_descent\_update

def f(x):

"""

Quadratic function.

It's easy to see the minimum value of the function

is 5 when is x=0.

"""

return x\*\*2 + 5

def df(x):

"""

Derivative of `f` with respect to `x`.

"""

return 2\*x

# Random number better 0 and 10,000. Feel free to set x whatever you like.

x = random.randint(0, 10000)

# TODO: Set the learning rate

learning\_rate = ?

epochs = 100

for i in range(epochs+1):

cost = f(x)

gradx = df(x)

print("EPOCH {}: Cost = {:.3f}, x = {:.3f}".format(i, cost, gradx))

x = gradient\_descent\_update(x, gradx, learning\_rate)

### Gradient Descent Solution

**def** **gradient\_descent\_update**(x, gradx, learning\_rate):

"""

Performs a gradient descent update.

"""

x = x - learning\_rate \* gradx

*# Return the new value for x*

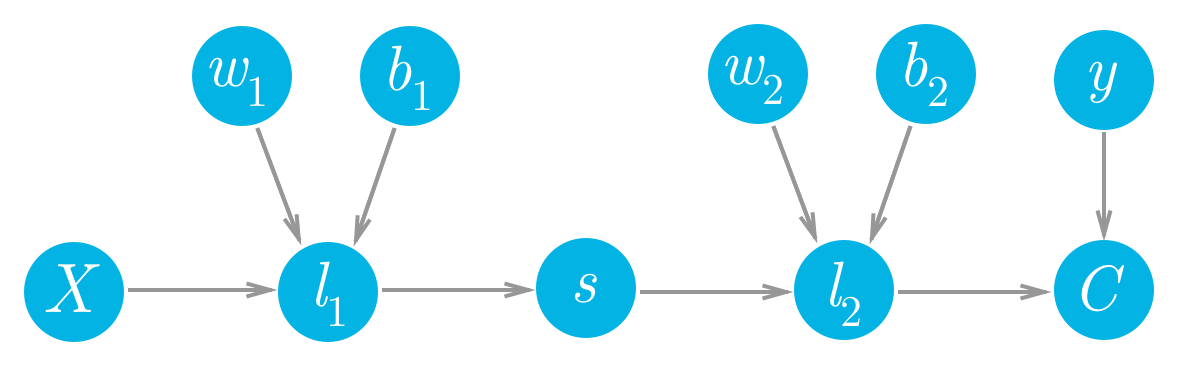
**return** x

We adjust the old x pushing it in the direction of gradx with the force learning\_rate. Subtracting learning\_rate \* gradx. Remember the gradient is initially in the direction of **steepest ascent** so subtracting learning\_rate \* gradx from x turns it into **steepest descent**. You can make sure of this yourself by replacing the subtraction with an addition.

### The Gradient & Backpropagation

Now that we know how to update our weights and biases using the gradient, we need to figure out how to calculate the gradients for all of our nodes. For each node, we'll want to change the values based on the gradient of the cost with respect to the value of that node. In this way, the gradient descent updates we make will eventually converge to the minimum of the cost.

Let's consider a network with a linear node *l*​1​​, a sigmoid node *s*, and another linear node *l*​2​​, followed by an MSE node to calculate the cost, *C*.



Forward pass for a simple two layer network.

Writing this out in MiniFlow, it would look like:

X, y = Input(), Input()

W1, b1 = Input(), Input()

W2, b2 = Input(), Input()

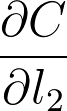
l1 = Linear(X, W1, b1)

s = Sigmoid(l1)

l2 = Linear(s, W2, b2)

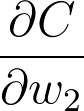
cost = MSE(l2, y)

We can see that each of the values of these nodes flows forwards and eventually produces the cost *C*. For example, the value of the second linear node *l*​2​​ goes into the cost node and determines the value of that node. Accordingly, a change in *l*​2​​ will produce a change in *C*. We can write this relationship between the changes as a gradient,

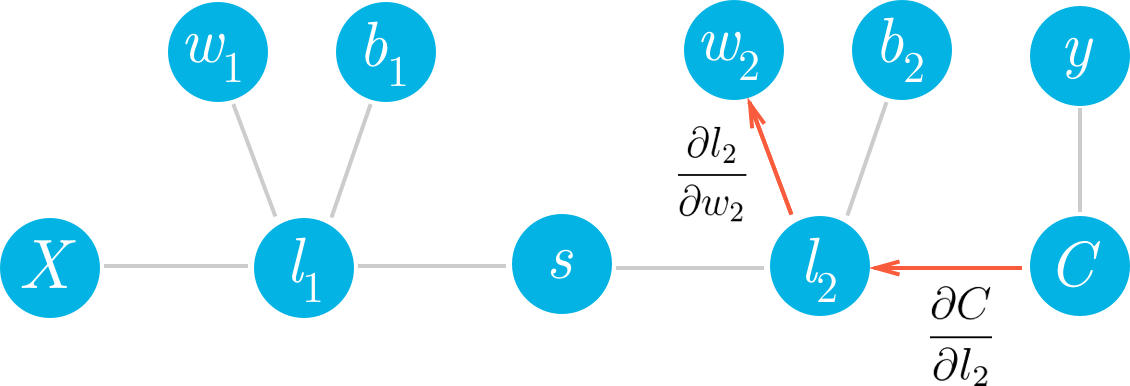


This is what a gradient means, it's a slope, how much you change the cost ∂*C* given a change in *l*​2​​, ∂*l*​2​​. So a node with a larger gradient with respect to the cost is going to contribute a larger change to the cost. In this way, we can assign blame for the cost to each node. The larger the gradient for a node, the more blame it gets for the final cost. And the more blame a node has, the more we'll update it in the gradient descent step.

If we want to update one of the weights with gradient descent, we'll need the gradient of the cost with respect to those weights. Let's see how we can use this framework to find the gradient for the weights in the second layer, *w*​2​​. We want to calculate the gradient of *C* with respect to *w*​2​​:



We can see in the graph that *w*​2​​ is connected to *l*​2​​, so a change in *w*​2​​ is going to create a change in *l*​2​​ which then creates a change in *C*. We can assign blame to *w*​2​​ by sending the cost gradient back through the network. First you have how much *l*​2​​ affected *C*, then how much *w*​2​​ affected *l*​2​​. Multiplying these gradients together gets you the total blame attributed to *w*​2​​.



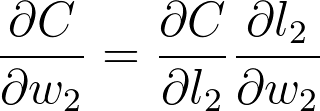
#### Pre-requisites

Below we're getting into the math behind backpropagation which requires multivariable calculus. If you need a refresher, I highly recommend checking out

* [**Khan Academy's lessons on partial derivatives**](https://www.khanacademy.org/math/multivariable-calculus/multivariable-derivatives/partial-derivatives/v/partial-derivatives-introduction)
* Another video on [**gradients**](https://www.khanacademy.org/math/multivariable-calculus/multivariable-derivatives/gradient-and-directional-derivatives/v/gradient)
* And finally, using [**the chain rule**](https://www.khanacademy.org/math/ap-calculus-ab/product-quotient-chain-rules-ab/chain-rule-ab/v/chain-rule-introduction)

#### Continuing on

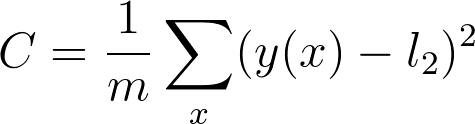
Multiplying these gradients is just an application of the chain rule:



You can see in the graph *w*​2​​, *l*​2​​, and *C* are chained together. Any change in *w*​2​​ will create a change in *l*​2​​ and the size of that change is given by the gradient ∂*l*​2​​/∂*w*​2​​. Now, since *l*​2​​ is changing this will cause a change in the cost *C* and the size of that change is given by the gradient ∂*C*/∂*l*​2​​. You can think of the chain rule similarly to the domino effect, changing something in the network will propagate through it altering other nodes along the way.

If you think of the chain rule as normal fractions, you can see that ∂*l*​2​​ in the denominator and numerator cancel out and you get back ∂*C*/∂*w*​2​​ (although it doesn't exactly work like normal fractions, but it helps to keep track of things.) Okay, let's work out the gradient for *w*​2​​. First, we need to know the gradient for *l*​2​​.

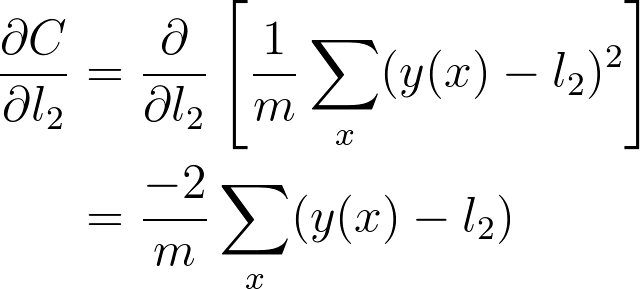
As a reminder [**the cost**](https://classroom.udacity.com/nanodegrees/nd101/parts/2a9dba0b-28eb-4b0e-acfa-bdcf35680d90/modules/269a4aad-9025-4354-a0ea-2623e889540a/lessons/b6deebe4-7f78-4947-b2c6-fc660ca942fb/concepts/60049bb7-bdf9-44bc-a6bc-6c8679b62de7) is

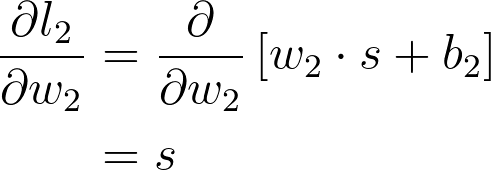


And the value for the second linear node is

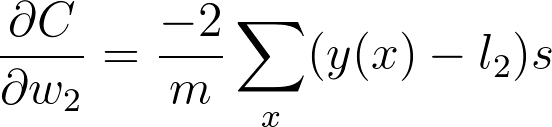
https://d17h27t6h515a5.cloudfront.net/topher/2017/February/589d15a7_l2/l2.png

where *w*​2​​, *s*, and *b*​2​​ are all vectors and *w*​2​​⋅*s* means the dot product of *w*​2​​ and *s*.



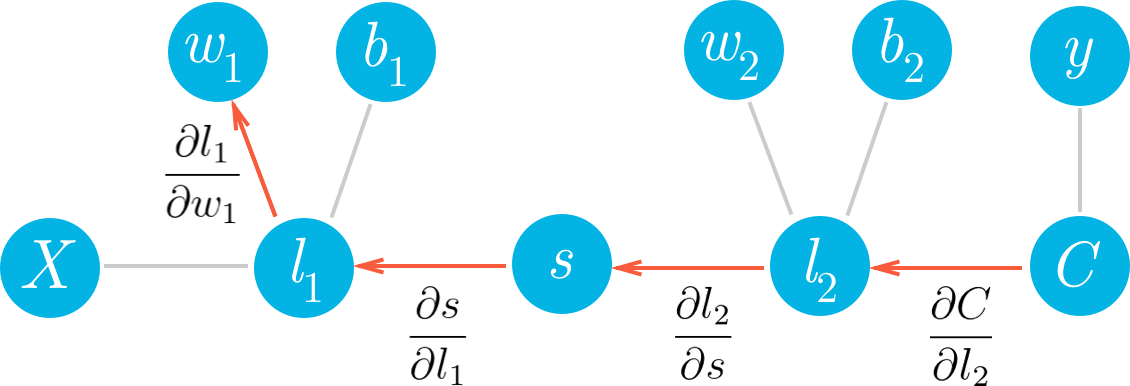


And putting these together, you get the gradient for *w*​2​​

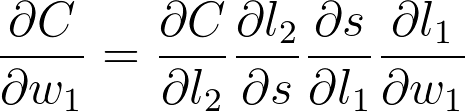


This is the gradient you use in the gradient descent update for *w*​2​​. You can see what we did here, we walked back through the graph and multiplied all the gradients we found along the way.

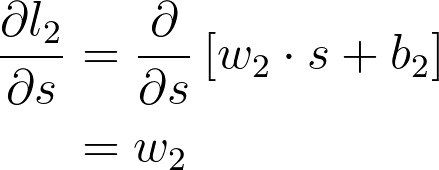
Now, let's go deeper and calculate the gradient for *w*​1​​. Here we use the same method as before, walking backwards through the graph.



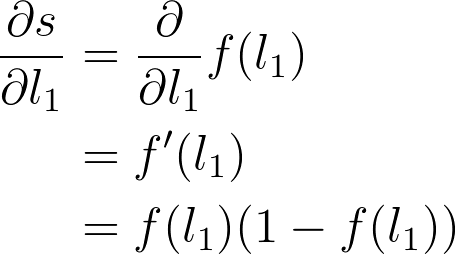
Hopefully it's clear now how to write out the gradient for *w*​1​​ just by looking at the graph. Using the chain rule, we'll write out the gradients for each node going backwards through the graph until we get to *w*​1​​.

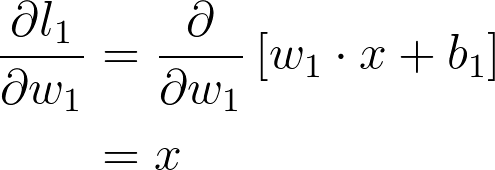


Now we can start calculating each gradient in this expression to get the gradient for *w*​1​​

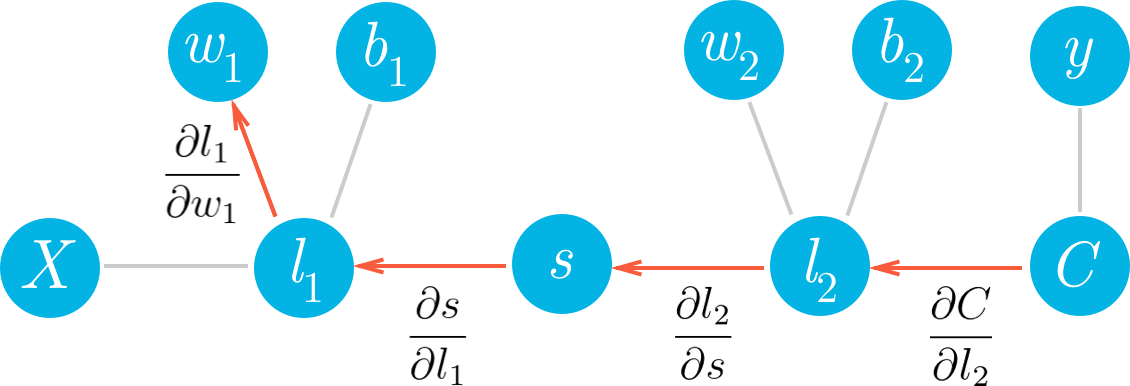


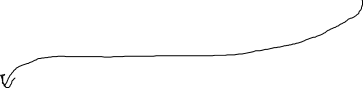
The next part is the gradient of the sigmoid function, *s*=*f*(*l*​1​​). Since we're using the logistic function here, the derivative can be written in terms of the sigmoid itself

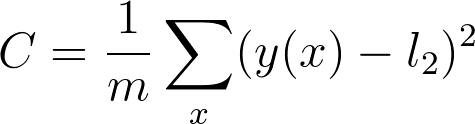


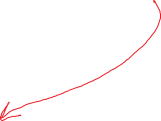
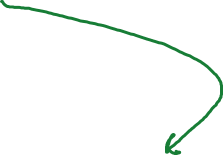
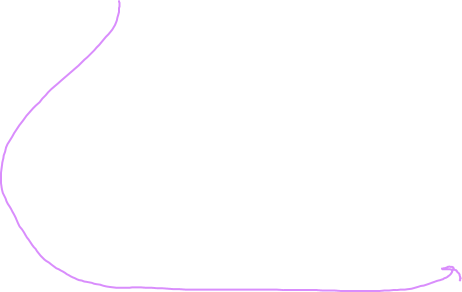


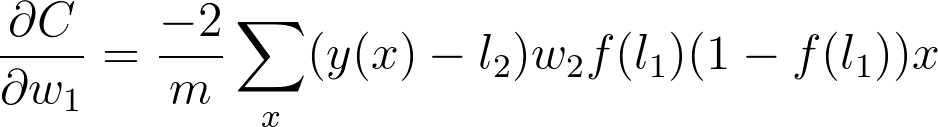
Putting this all together, you get

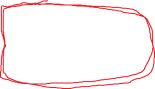












Now we can see a clear pattern. To find the gradient, you just multiply the gradients for all nodes in front of it going backwards from the cost. This is the idea behind **backpropagation**. The gradients are passed backwards through the network and used with gradient descent to update the weights and biases. If a node has multiple outgoing nodes, you just sum up the gradients from each node.

### Implementing in MiniFlow

Let's think about how to translate this into MiniFlow. Looking at the graph, you see that each node gets the cost gradient from it's outbound nodes. For example, the node *l*​1​​ gets ∂*C*/∂*l*​1​​ through the sigmoid node, *s*. Then *l*​1​​ passes on the cost gradient to the weight node *w*​1​​, but multiplied by ∂*l*​1​​/∂*w*​1​​, the gradient of *l*​1​​ with respect to it's input *w*​1​​.



So, each node will pass on the cost gradient to its inbound nodes and each node will get the cost gradient from it's outbound nodes.   
Then, for each node we'll need to calculate a gradient that's the **cost gradient times the gradient of that node with respect to its inputs**. Below I've written out this process for a Linear node.

*# Initialize a partial for each of the inbound\_nodes.*

self.gradients = {n: np.zeros\_like(n.value) **for** n **in** self.inbound\_nodes}

*# Cycle through the outputs. The gradient will change depending*

*# on each output, so the gradients are summed over all outputs.*

**for** n **in** self.outbound\_nodes:

*# Get the partial of the cost with respect to this node.*



grad\_cost = n.gradients[self]



*# Set the partial of the loss with respect to this node's inputs.*

self.gradients[self.inbound\_nodes[0]] += np.dot(grad\_cost, self.inbound\_nodes[1].value.T)



*# Set the partial of the loss with respect to this node's weights.*



self.gradients[self.inbound\_nodes[1]] += np.dot(self.inbound\_nodes[0].value.T, grad\_cost)



*# Set the partial of the loss with respect to this node's bias.*



self.gradients[self.inbound\_nodes[2]] += np.sum(grad\_cost, axis=0, keepdims=**False**)



### New Code

There have been a couple of changes to MiniFlow since we last took it for a spin:

The first being the Node class now has a backward method, as well as a new attribute self.gradients, which is used to store and cache gradients during the backward pass.

**class** **Node**(object):

"""

Base class for nodes in the network.

Arguments:

`inbound\_nodes`: A list of nodes with edges into this node.

"""

**def** **\_\_init\_\_**(self, inbound\_nodes=[]):

"""

Node's constructor (runs when the object is instantiated). Sets

properties that all nodes need.

"""

*# A list of nodes with edges into this node.*

self.inbound\_nodes = inbound\_nodes

*# The eventual value of this node. Set by running*

*# the forward() method.*

self.value = **None**

*# A list of nodes that this node outputs to.*

self.outbound\_nodes = []

*# New property! Keys are the inputs to this node and*

*# their values are the partials of this node with*

*# respect to that input.*

self.gradients = {}

*# Sets this node as an outbound node for all of*

*# this node's inputs.*

**for** node **in** inbound\_nodes:

node.outbound\_nodes.append(self)

**def** **forward**(self):

"""

Every node that uses this class as a base class will

need to define its own `forward` method.

"""

**raise** NotImplementedError

**def** **backward**(self):

"""

Every node that uses this class as a base class will

need to define its own `backward` method.

"""

**raise** NotImplementedError

The second change is to the helper function forward\_pass(). That function has been replaced with forward\_and\_backward().

**def** **forward\_and\_backward**(graph):

"""

Performs a forward pass and a backward pass through a list of sorted nodes.

Arguments:

`graph`: The result of calling `topological\_sort`.

"""

*# Forward pass*

**for** n **in** graph:

n.forward()

*# Backward pass*

*# see: https://docs.python.org/2.3/whatsnew/section-slices.html*

**for** n **in** graph[::-1]:

n.backward()

### Setup

Here's the derivative of the *sigmoid* function w.r.t *x*:

*sigmoid*(*x*)=1/(1+*exp*(−*x*))

​∂*x*​​∂*sigmoid*​​=*sigmoid*(*x*)∗(1−*sigmoid*(*x*))

* Complete the implementation of backpropagation for the Sigmoid node by finishing the backwardmethod in miniflow.py.
* The backward methods for all other nodes have already been implemented. Taking a look at them might be helpful.

Here's my solution to the last quiz.

**class** **Sigmoid**(Node):

"""

Represents a node that performs the sigmoid activation function.

"""

**def** **\_\_init\_\_**(self, node):

*# The base class constructor.*

Node.\_\_init\_\_(self, [node])

**def** **\_sigmoid**(self, x):

"""

This method is separate from `forward` because it

will be used with `backward` as well.

`x`: A numpy array-like object.

"""

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

**def** **forward**(self):

"""

Perform the sigmoid function and set the value.

"""

input\_value = self.inbound\_nodes[0].value

self.value = self.\_sigmoid(input\_value)

**def** **backward**(self):

"""

Calculates the gradient using the derivative of

the sigmoid function.

"""

*# Initialize the gradients to 0.*

self.gradients = {n: np.zeros\_like(n.value) **for** n **in** self.inbound\_nodes}

*# Sum the derivative with respect to the input over all the outputs.*

**for** n **in** self.outbound\_nodes:

grad\_cost = n.gradients[self]

sigmoid = self.value

self.gradients[self.inbound\_nodes[0]] += sigmoid \* (1 - sigmoid) \* grad\_cost

The backward method sums the derivative (it's a normal derivative when there's only one variable) with respect to the only input over all the output nodes. The last line implements the derivative, ​∂*x*​​∂*sigmoid*​​​∂*sigmoid*​​∂*cost*​​.

Replacing the math expression with code:

​∂*x*​​∂*sigmoid*​​ is sigmoid \* (1 - sigmoid) and ​∂*sigmoid*​​∂*cost*​​ is grad\_cost.

Now that you have the gradient of the cost with respect to each input (the return value from forward\_and\_backward()) your network can start learning! To do so, you will implement a technique called **Stochastic Gradient Descent**.

**Stochastic Gradient Descent**

Stochastic Gradient Descent (SGD) is a version of Gradient Descent where on each forward pass a batch of data is randomly sampled from total dataset. Remember when we talked about the batch size earlier? That's the size of the batch. Ideally, the entire dataset would be fed into the neural network on each forward pass, but in practice, it's not practical due to memory constraints. SGD is an approximation of Gradient Descent, the more batches processed by the neural network, the better the approximation.

A naïve implementation of SGD involves:

1. Randomly sample a batch of data from the total dataset.
2. Running the network forward and backward to calculate the gradient (with data from (1)).
3. Apply the gradient descent update.
4. Repeat steps 1-3 until convergence or the loop is stopped by another mechanism (i.e. the number of epochs).

If all goes well, the network's loss should generally trend downwards, indicating more useful weights and biases over time.

So far, MiniFlow can already do step 2. In the following quiz, steps 1 and 4 are already implemented. It will be your job to implement step 3.

As a reminder, here's the gradient descent update equation, where *α* represents the learning rate:

*x*=*x*−*α*∗​∂*x*​​∂*cost*​​

We're also going to use an actual dataset for this quiz, the [**Boston Housing dataset**](https://archive.ics.uci.edu/ml/datasets/Housing). After training the network will be able to predict prices of Boston housing!

Boston's Back Bay

By Robbie Shade (Flickr: Boston's Back Bay) [CC BY 2.0 (http://creativecommons.org/licenses/by/2.0)], via Wikimedia Commons

Boston's Back Bay

By Robbie Shade (Flickr: Boston's Back Bay) [CC BY 2.0 ([**http://creativecommons.org/licenses/by/2.0**](http://creativecommons.org/licenses/by/2.0))], via Wikimedia Commons

Each example in the dataset is a description of a house in the Boston suburbs, the description consists of 13 numerical values (features). Each example also has an associated price. With SGD, we're going to minimize the MSE between the actual price and the price predicted by the neural network based on the features.

If all goes well the output should look something like this:

When the batch size is 11:

Total number of examples = 506

Epoch: 1, Loss: 140.256

Epoch: 2, Loss: 34.570

Epoch: 3, Loss: 27.501

Epoch: 4, Loss: 25.343

Epoch: 5, Loss: 20.421

Epoch: 6, Loss: 17.600

Epoch: 7, Loss: 18.176

Epoch: 8, Loss: 16.812

Epoch: 9, Loss: 15.531

Epoch: 10, Loss: 16.429

When the batch size is the same as the total number of examples (batch is the whole dataset):

Total number of examples = 506

Epoch: 1, Loss: 646.134

Epoch: 2, Loss: 587.867

Epoch: 3, Loss: 510.707

Epoch: 4, Loss: 446.558

Epoch: 5, Loss: 407.695

Epoch: 6, Loss: 324.440

Epoch: 7, Loss: 295.542

Epoch: 8, Loss: 251.599

Epoch: 9, Loss: 219.888

Epoch: 10, Loss: 216.155

Notice the *cost* or *loss* trending towards 0.

**Instructions**

1. Open nn.py. See how the network runs with this new architecture.
2. Find the sgd\_update method in miniflow.py and implement SGD.
3. Test your network! Does your loss decrease with more epochs?

Note! The virtual machines on which we run your code have time limits. If your network takes more than 10 seconds to run, you will get a timeout error. Keep this in mind as you play with the number of epochs.

Here's my solution to the last quiz.

**def** **sgd\_update**(trainables, learning\_rate=1e-2):

"""

Updates the value of each trainable with SGD.

Arguments:

`trainables`: A list of `Input` nodes representing weights/biases.

`learning\_rate`: The learning rate.

"""

*# Performs SGD*

*#*

*# Loop over the trainables*

**for** t **in** trainables:

*# Change the trainable's value by subtracting the learning rate*

*# multiplied by the partial of the cost with respect to this*

*# trainable.*

partial = t.gradients[t]

t.value -= learning\_rate \* partial

Take a look at the last few lines:

*# Performs SGD*

*#*

*# Loop over the trainables*

**for** t **in** trainables:

*# Change the trainable's value by subtracting the learning rate*

*# multiplied by the partial of the cost with respect to this*

*# trainable.*

partial = t.gradients[t]

t.value -= learning\_rate \* partial

There are two keys steps. First, the partial of the cost (*C*) with respect to the trainable t is accessed.

partial = t.gradients[t]

Second, the value of the trainable is updated according to Equation (12).

t.value -= learning\_rate \* partial

This is done for all trainables.

Equation (12)

Equation (12)

With that, the loss decreases on the next pass through the network.

I'm putting the same quiz below again. If you haven't already, set the number of epochs to something like 1000 and watch as the loss decreases!

Congratulations on making it to the end of this lab! Building a neural network from scratch is no small task. You should be proud!

MiniFlow has the makings of becoming a powerful deep learning tool. It is entirely possible to classify something like the [**MNIST**](http://yann.lecun.com/exdb/mnist/) database with MiniFlow.

I'll leave it as an exercise for you to finish MiniFlow from here.