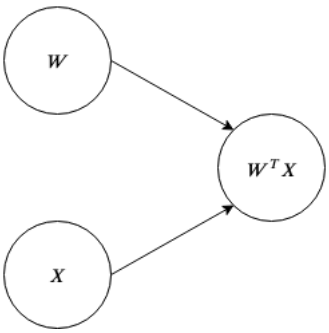
Building A Basic Computational Graph Engine

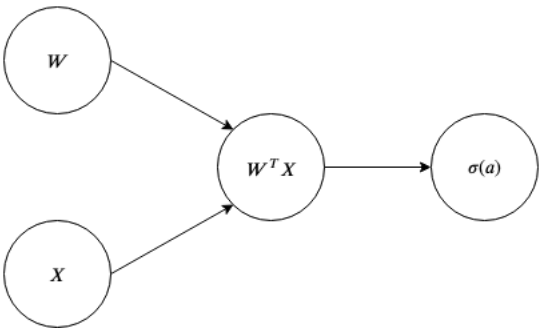
Many deep learning libraries like TensorFlow use graphs to represent the computations involved in neural networks. Not only are these graphs used to compute predictions for a given input to the network but they are also used to backpropagate gradients during the training phase. The main advantage of this graph representation is that each computation can be encapsulated as a node on the graph that only cares about its input and output. This level of abstraction gives you the flexibility to build neural networks of (nearly) arbitrary sizes and shapes (eg. MLPs, CNNs, RNNs, etc.). This blog post will implement a very basic version of a computational graph engine.

Representing Computations as Graphs

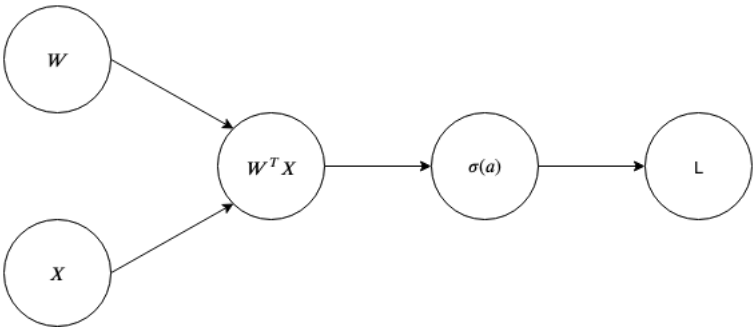
As stated earlier, computations can be thought of as nodes on a graph and the edges between nodes can be thought of as the inputs and outputs to these computations. Below is a graph that computes the dot product between two vectors.



Here the two input vectors are represented as a node and the dot product computation is also represented as a node that takes the vector nodes as inputs. As you can see from this example, not all nodes represent computations. The two input nodes, for example, only store values. We can further extend this example by adding a sigmoid computation to the result of the dot product.



So this computational graph takes the dot product between two inputs and applies the sigmoid function to the result. If we assume that W represents a weight vector and X represents an input feature vector then this computational graph essentially represents logistic regression. And logistic regression uses the cross-entropy loss function which can be expressed as an additional node on the graph.



So ultimately, all that a node in a computational graph needs to worry about is

1. The computation is performs.
2. The nodes that it gets its input from.
3. The nodes it passes its output to.

Based on this we can make the following abstract node class.

```
class Node(object):

    def __init__(self, input_nodes=[]):
        """
        Define the inputs to this node and create variables to hold the output, gradients, and output no
        :param input_nodes: The nodes providing input to this node
        """
        self.input_nodes = input_nodes
        self.output_nodes = []

        for node in self.input_nodes:
            node.output_nodes.append(self)

        self.output = None
        self.gradients = {}

    def compute(self, output=None):
        raise NotImplementedError

    def backpass(self):
        raise NotImplementedError
```

Here you can see the class contains the node's input and output nodes as well as a method to implement whatever the node computes. You can also see a `backpass()` method and `gradient` variable which will be explained later. So, for example, a node representing the sigmoid function would look like

```
class Sigmoid(Node):

    def __init__(self, node):
        Node.__init__(self, [node])

    def compute(self):
        """
        Compute sigmoid activation based on input node.
        """
        input_value = self.input_nodes[0].output
        self.output = 1. / (1. + np.exp(-input_value))
```

A sigmoid function only has one input so there is a single `input_node` in this case.

So we can think of a computational graph as a collection of nodes that are connected to eachother in some way. In this case there would be four nodes - the two input nodes, the dot product node and the sigmoid node.

Topological Sort

In the logistic regression example above, it is clear that the nodes must be computed in a certain order. For example, the dot product must be computed before the sigmoid function. In this example it is obvious what the order of computation should be however in more complicated graphs the correct order can be less obvious. We can obtain the correct order of computation by applying a topoligical sort to the collection of nodes in the graph. There are several algorthms for topological sorting, one being *Kahn's algorithm* which is shown below.

```
L ← Empty list that will contain the sorted elements
S ← Set of all nodes with no incoming edge
while S is non-empty do
    remove a node n from S
    add n to tail of L
    for each node m with an edge e from n to m do
        remove edge e from the graph
        if m has no other incoming edges then
            insert m into S
if graph has edges then
    return error   (graph has at least one cycle)
else
    return L   (a topologically sorted order)
```

Once the nodes of the graph are put in sorted order the final output of the graph can be obtained by simply iterating through the list of nodes and calling each one's `compute()` method.

Learning with Graphs

So far we have seen how computational graphs can be used to perform a sequence of arbitrary computations. But we can also use computational graphs to learn parameters within the graph itself. All that is required to do this is a dataset of input and output values, a cost-function, and a computational graph whose nodes perform computations that are differentiable. From this we can train the parameters within the graph using gradient descent. If you recall the gradient descent update formula for a single parameter p is

$$pt+1=pt-\alpha\partial L\partial p$$

Where L is the loss function and α is the learning rate.

Let's stick with our logistic regression computational graph example. In this case, the cost-function is the cross-entropy loss (which can also be represented as a node) shown below.

$$L=\sum(x,z)\in Szlny+(1-z)ln(1-y)$$

Where $z\in\{0,1\}$ is the label and y is the prediction so

$$a=WTX,y=\sigma(a)$$

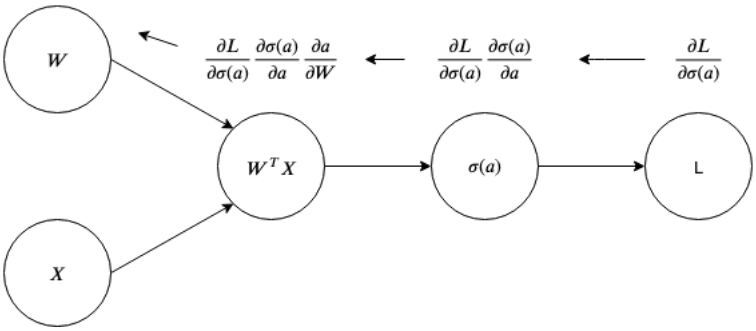
and the parameter we are interested in training is the W vector. Using the chain rule from calculus we can express the partial derivative of the loss function with respect to the element $wi\in W$ as

$$\partial L\partial wi=\partial L\partial y\partial y\partial a\partial a\partial wi$$

Where

$$\partial L\partial y=y-zy(1-y),\partial y\partial a=\sigma(a)(1-\sigma(a)),\partial a\partial wi=xi$$

Using these partial derivatives we can actually incrementally compute the final partial derivative by moving backwards through the graph.



We start at the final node which is the cross-entropy cost-function and here we can compute ∂L∂y and store the result in that node's `gradients` variable. Then we can travel backwards to the sigmoid node where we can compute ∂y∂a and combining this with the previous gradient we get the product ∂L∂y∂y∂a which we then store in that node's `gradients` variable. Then finally we can travel backwards once more to the dot product node where we can compute ∂a∂wi and combine this with the previous gradients to arrive at the final result of ∂L∂y∂y∂a∂a∂wi.

So just as we travelled forwards through the graph to compute the output, we can travel backwards through the graph to compute the partial derivatives. And just like in the forwards pass, the backwards pass can be computed in an encapsulated way where all a node needs to be aware of is it's derivative and the nodes it is connected to. This is where the `backpass()` method comes in which computes the partial derivative with respect to that node. So for the sigmoid function it would look like

```
class Sigmoid(Node):

    def __init__(self, node):
        Node.__init__(self, [node])

    def compute(self):
        """
        Compute sigmoid activation based on input node.
        """
        input_value = self.input_nodes[0].output
        self.output = 1. / (1. + np.exp(-input_value))

    def backpass(self):
        """
        Backpropagate gradients to input node.
        """
        # clear gradients for input nodes
        self.gradients = {n: np.zeros_like(n.output) for n in self.input_nodes}

        # backpropagate gradients to input nodes which is also a function of gradients from output nodes
        for node in self.output_nodes:
            grad_cost = node.gradients[self]
            sigmoid = self.output
            self.gradients[self.input_nodes[0]] += sigmoid * (1 - sigmoid) * grad_cost
```

As you can see, the method is computing the sigmoid derivative and multiplying it with the derivative from the node that it is connected to in the forward direction (in our example that would be the cross-entropy node).

Now we are finally able to implement the gradient descent update.

```
    def update(self, training_nodes):
        """
        For nodes with trainable weights, update the weights based on previously computed gradients.
        :param training_nodes: Nodes with trainable weights.
        """
        for node in training_nodes:
```

```
            partial = node.gradients[node]
            node.output -= self.learning_rate * partial
```

This function iterates through the nodes with trainable parameters and uses their stored gradients to update their values using a given learning rate.

Building a Network

Below is a trimmed version of a feedforward neural network implementation for regression. You can find the full code here.

```
n_features = X_.shape[1]
n_hidden = 10
W1_ = np.random.randn(n_features, n_hidden)
b1_ = np.zeros(n_hidden)
W2_ = np.random.randn(n_hidden, 1)
b2_ = np.zeros(1)

X, y = Input(), Input()
W1, b1 = Input(), Input()
W2, b2 = Input(), Input()

l1 = Linear(X, W1, b1)
s1 = Sigmoid(l1)
l2 = Linear(s1, W2, b2)
cost = MSE(y, l2)

feed_dict = {
    X: X_,
    y: y_,
    W1: W1_,
    b1: b1_,
    W2: W2_,
    b2: b2_
}

epochs = 20
m = X_.shape[0]
batch_size = 11
steps_per_epoch = m // batch_size

graph = Graph(feed_dict)
sgd = SGD(1e-2)
trainables = [W1, b1, W2, b2]

for i in range(epochs):
    loss = 0
    for j in range(steps_per_epoch):

        # sample minibatch
        X_batch, y_batch = resample(X_, y_, n_samples=batch_size)

        X.output = X_batch
        y.output = y_batch

        graph.compute_gradients()
        sgd.update(trainables)

        loss += graph.loss()

    print("Epoch: {}, Loss: {:.3f}".format(i + 1, loss / steps_per_epoch))
```

The weights, biases, inputs, and cost-functions are all implemented as nodes in a computational graph as was described in this blog post. The particular optimization

algorithm used here is stochastic gradient descent where at each iteration a minibatch of training examples are sampled from the training data set and a forward and backward pass are made through the network (`graph.compute_gradients()`) which computes the gradients for each node. Then the `sgd.update(trainables)` function actually updates the weights based on the gradients.

Thank you for reading and you can check out the full computational graph implementation on github.

References