# Deep Neural Networks As Computational Graphs

## DNNs Don't Need To Be A Black Box

Tyler Elliot Bettilyon
Jun 22, 2018 · 11 min read

*This is the second article in that series, and is associated with our Intro to Deep Learning Github repository where you can find practical examples of many deep learning applications and tactics. You can find the first article in the series here, and the third here, and the fourth here.*

. . .

Lots of people say that neural nets are a "black box" whose successful predictions are impossible to explain. I hate treating anything as a black box, it grinds against my curious nature. It's also not a very helpful mental model — understanding *what* neural nets are and *how* they arrive at the conclusions they do can help practitioners gain insight into using them.

Viewed through the proper lens, neural nets' prediction making capability makes a lot of sense. This article is about taking nets out of their black box by understanding what a neural network really represents. Later in the series we'll explore exactly how they are trained with gradient descent and backpropagation.

At its core, every neural network **represents a single mathematical function.** This means that when you set out to use a neural network for some task, your hypothesis is that there is some mathematical function that will approximate the observed behavior reasonably well. When we train a neural network we are trying to find *one such reasonable approximation.*

Because these functions are often monstrously complex we use graphs to represent them rather than the standard formula notation. These graphs help us organize our thinking about the functions we set out to build and it turns out some graphs work much better than others for particular tasks. A lot of research and development in the neural network space is about inventing new architectures for these graphs, rather than inventing brand new algorithms.

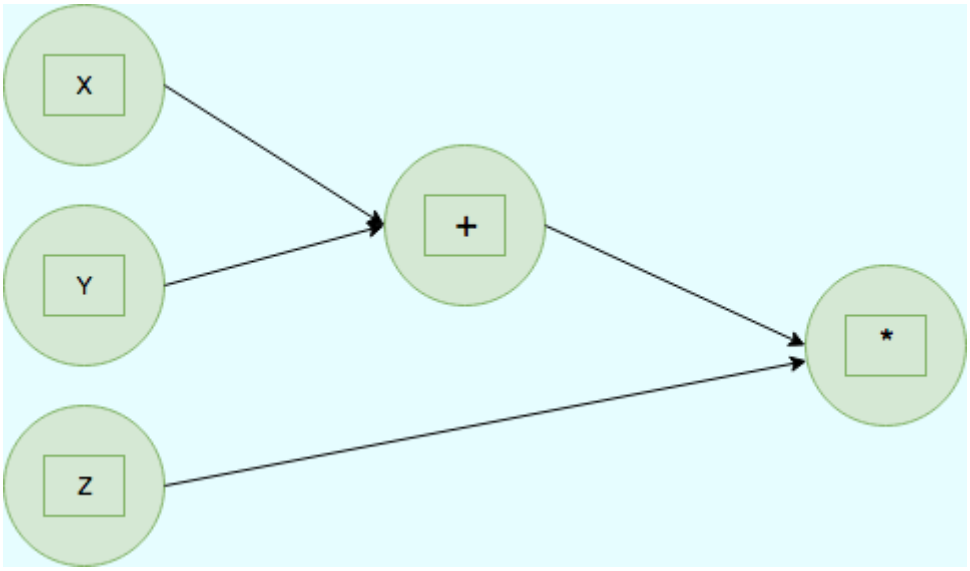
So what is a computational graph and how are they used by neural networks?

## Computational Graphs

A computational graph is a way to represent a math function in the language of graph theory. Recall the premise of graph theory: nodes are connected by edges, and everything in the graph is either a node or an edge.

In a computational graph nodes are either input values or functions for combining values. Edges receive their weights as the data flows through the graph. Outbound edges from an input node are weighted with that input value; outbound nodes from a function node are weighted by combining the weights of the inbound edges using the specified function.

For example, consider the relatively simple expression: $f(x, y, z) = (x + y) * z$. This is how we would represent that function as as computational graph:




There are three input nodes, labeled X, Y, and Z. The two other nodes are function nodes. In a computational graph we generally compose many simple functions into a

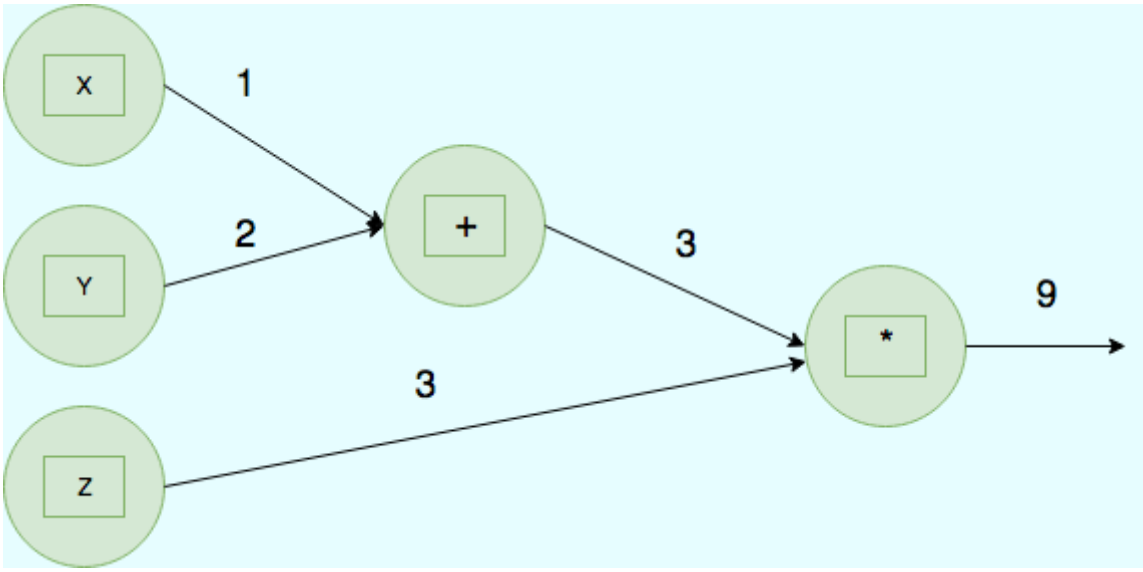
more complex function. We can do composition in mathematical notation as well, but I hope you'll agree the following isn't as clean as the graph above:

```
f(x, y, z) = h(g(x, y) , z)
g(i, j) = i + j
h(p, q) = p*q
```

In both of these notations we can compute the answers to each function separately, provided we do so in the correct order. Before we know the answer to f(x, y, z) first we need the answer to g(x, y) and then h(g(x, y), z). With the mathematical notation we resolve these dependencies by computing the deepest parenthetical first; in computational graphs we have to wait until all the edges pointing *into* a node have a value before computing the output value for that node. Let's look at the example for computing f(1, 2, 3).

```
f(1, 2, 3) = h(g(1, 2), 3)
g(1, 2) = 1 + 2 = 3
f(1, 2, 3) = h(3, 3)
h(3, 3) = 3*3 = 9
f(1, 2, 3) = 9
```

And in the graph, we use the output from each node as the weight of the corresponding edge:




The weight of the outbound edge for the plus (+) node has to be computed before the the multiply (*) node can compute its outbound edge weight.

Either way, graph or function notation, we get the same answer because these are just two ways of expressing the same thing.

In this simple example it might be hard to see the advantage of using a computational graph over function notation. After all, there isn't anything terribly hard to understand about the function f(x, y, z) = (x + y) * z. The advantages become more apparent when we reach the scale of neural networks.

Even relatively "simple" deep neural networks have *hundreds of thousands* of nodes and edges; it's quite common for a neural network to have more than one million edges. Try to imagine the function expression for such a computational graph… can you do it? How much paper would you need to write it all down? This issue of scale is one of the reasons computational graphs are used.

Let's look at one concrete example: suppose we're building a deep neural network for predicting if someone is single or in some sort of relationship; a binary predictor. Furthermore, assume that we've gathered a dataset that tells us four things about a person: their age, gender, what city they live in, and if they are single or in some sort of relationship.

When we say we want to "build a neural network" to make this prediction — we're really saying that we want to find a mathematical function of the form:

```
f(age, gender, city) = predicted_relationship_status
```

Where the output value is 0 if that person is in a relationship, and 1 if that person is not in a relationship.

We are making a huge (and wrong) assumption here that age, gender, and city tell us everything we need to know about whether or not someone is in a relationship. But that's okay — all models are wrong and we can use statistics to find out if this one is *useful* or not. Don't focus on how much this toy model oversimplifies human relationships, focus on what this means for the neural network we want to build.

As an aside, but before we move on: encoding the value for "city" can be tricky. It's not at all clear what the numerical value of "Berkeley" or "Salt Lake City" should be in our mathematical function. Topics such as tokenization and processing categorical data are beyond this article's scope but absolutely worthy of your time if you have not encountered them before. One-hot encoding is a popular tactic.

In fact, a one-hot encoded vector would be used as the output layer for this network as well. The details of using one-hot vectors this way are in the next article in the series.

## Neural Networks as Computational Graphs

I like to think of the architecture of a deep neural network as a template for a function. When we define the architecture of a neural network we're laying out the series of sub-functions and specifying how they should be composed. When we train the neural network we're experimenting with the parameters of these sub-functions. Consider this function as an example:

```
f(x, y) = ax² + bxy + cy²; where a, b, and c are scalars
```

The *component sub-functions* of this function are all of the operators: two squares, two additions, and 4 *multiplications.* The *tunable parameters* of this function are a, b, and c, in neural network parlance these are called **weights**. The inputs to the function are X and Y — we can't tune those values in machine learning because they are the values from the dataset, which we would have (hopefully) gathered earlier in the process.

By changing the values of our weights (a, b, and c) we can dramatically impact the output of the function. On the other hand, regardless of the values of a, b, and c there will always be an $x^2$, a $y^2$ and an xy term — so our function has a limited range of possible configurations.

Here is a computational graph representing this function:



Take a minute to look at this graph. Is it clear how this graph represents our function f(x, y)?

This isn't *technically* a neural network, but it's very close in all the ways that count. It's a graph that represents a function; we could use it to predict some kinds of trends; and we could train it using gradient descent and backpropagation if we had a dataset that mapped two inputs to an output. This particular computational graph will be good at modeling some quadratic trends involving exactly 2 variables, but bad at modeling anything else.

In this example, training the network would amount to changing the weights until we find some combination of a, b, and c that causes the function to work well as a predictor for our dataset. If you're familiar with linear regression, this should feel similar to tuning the weights of the linear expression.

This graph is still quite simple compared to even the simplest neural networks that are used in practice, but the main idea — that a, b, and c can be adjusted to improve the model's performance — remains the same.

The reason this neural network would not be used in practice is that it isn't very *flexible*. This function only has 3 parameters to tune: a, b, and c. Making matters worse, we've only given ourselves room for 2 features per input (x and y).

Fortunately, we can easily solve this problem by using *more complex functions* and allowing for *more complex input.* Huzzah!

Recall two facts about deep neural networks:

1. DNNs are a special kind of graph, a "computational graph".
2. DNNs are made up of a series of "fully connected" layers of nodes.

"Fully connected" means that the output from each node in the first layer becomes one of the inputs for *every node* in the second layer. In a computational graph the edges are the output values of functions — so in a fully connected layer the output for each sub-function is used as one of the inputs for each of the sub-functions in the next layer. But, what are those functions?

The function performed by each node in the neural net is called a **transfer function** (which is also called the **activation function**). There are two steps in every transfer function. First, all of the input values are combined in some way, usually this is a weighted sum. Second, a "nonlinear" function is applied to that sum; this second function might change from layer to layer within a single neural network.

Popular nonlinear functions for this second step are tanh, log, max(0, x) (called Rectified Linear Unit, or ReLU), and the sigmoid function. At the time of this writing, ReLU is the most popular choice of nonlinearity, but things change quickly.

If we zoom in on a neural network, we'd notice that each "node" in the network was actually 2 nodes in our computational graph:

Each node is actually multiple component functions.

In this case, the **transfer function** is a sum followed by a sigmoid. Typically, all the nodes in a layer have the same transfer and activation function. Indeed it is common for all the layers in the same network to use the same activation functions, though it is not a requirement by any means.

The last sources of complexity in our neural network are **biases** and **weights.** Every incoming edge has a unique **weight,** the output value from the previous node is multiplied by this weight *before* it is given to the transfer function. Each transfer function also has a single **bias** which is added *before* the nonlinearity has been applied. Lets zoom in one more time:

In this diagram we can see that each input to the sum is first **weighted** via multiplication *then* it is summed. The bias is added to that sum as well, and finally the total is sent to our nonlinear function (sigmoid in this case). These weights and biases are the parameters that are ultimately fine-tuned during training.

In the previous example, I said we didn't have enough flexibility because we only had 3 parameters to fine tune. So just how many parameters are there in a deep neural network for us to tune?

If we define a neural net to predict binary classification (in/not in a relationship) with 2 hidden layers each with 512 nodes and an input vector with 20 features we will have 20*512 + 512*512 + 512*2 = 273,408 weights that we can fine tune plus 1024 biases (one for each node in the hidden layers). This is a "simple" neural network. "Complex" neural networks frequently have several million tunable weights and biases.

This extraordinary flexibility is what allows neural nets to find and model complex relationships. It's also why they require **lots** of data to train. Using backpropagation and gradient descent we can purposely change the millions of weights until the output becomes more correct, but because we're doing calculations involving millions of variables it takes a lot of time and a lot of data to find the right combination of weights and biases.

While they are sometimes called a "black box", neural networks are really just a way of representing very complex mathematical functions. The neural nets we build are particularly useful functions *because* they have so many parameters that can be fine tuned. The result of the fine tuning is that rich complexities between different components of the input can be plucked out of the noise.

Ultimately, the "architecture" of our computational graph will have a big impact on how well our network can perform. Questions like: how many nodes per layer, which activation functions are used at each layer, and how many layers to use, are the subject of research and might change dramatically from neural network to neural network. The architecture will depend on the type of prediction being made and the kind of data being fed into the system — just like we shouldn't use a linear function to model parabolic data, we shouldn't use *any* neural net to solve *every* problem.

In the next article in this series I'm going to build and examine a few "simple" neural networks using the Keras library. I'll be working through the "hello world" of machine learning — classifying handwritten digits using the MNIST dataset. My goal in that article is to elucidate how different neural net architectures impact training time and performance of the model.

After an interlude into practical-land we'll return to the theory to discuss two important topics: gradient descent and backpropagation. See you then!

Part 3: Classifying MNIST Digits With Different Neural Network Architectures

. . .

This article is produced by *Teb's Lab. To learn more about my journey to graduate school, and learn what I'm learning about machine learning, genetics, and bioinformatics sign up for the Weekly Lab Report, read The Once and Future Student, become a patron on Patreon, or just follow me here on Medium.*