Backpropagation

Introduction

Neuron

A neuron is essentially a big formula. Often it computes a linear combination of inputs, plus a bias, followed by a threshold funtion.

Suppose neuron weights (parameters) are: $w_1, w_2, \ldots, w_N$, and bias as b, and we wish to use ReLU as activation function.

To calculate the neuron output on inputs $x_1, x_2, \ldots, x_N$ we would compute:

$$N_1(\boldsymbol{X}) = ReLU(w_1 * x_1 + w_2 * x_2 + \cdots + w_n * x_n + b)$$

where $\boldsymbol{X} = x_1, \ldots, x_n$, etc.

Layer

A neural network layer is a lot of neurons operating on the same inputs; for example, a layer with 32 neurons would be:

$$L_1(\boldsymbol{X}) = (N_1(\boldsymbol{X}), N_2(\boldsymbol{X}), \ldots, N_{32}(\boldsymbol{X}))$$

Network

Layers are often composed, such as output of one layer goes as input into another. A five layer network might look something like this:

$$output = NN(\boldsymbol{X}) = L_5(L_4(L_3(L_2(L_1(\boldsymbol{X})))))$$

Obviously the inputs-and-outputs of each layer have to be compatible, and the *output* cardinality is determined by the last layer.

Expanding that *output* into the component x_i and w_i would result in a huge formula.

Training

All the weights and biases (the w_is and bs) of all the neurons in a network are called the *parameters*, and are often denoted as $\boldsymbol{\theta}$. Training is the process of updating the parameters to minimize some error/loss function.

For a dataset $\boldsymbol{D}$ of n labeled instances, such as:

$$\boldsymbol{D} = ((\boldsymbol{X_1}, \boldsymbol{y_1}), (\boldsymbol{X_2}, \boldsymbol{y_2}), \ldots, (\boldsymbol{X_n}, \boldsymbol{y_n}))$$

the loss function might look like:

$$Loss(\boldsymbol{\theta}, \boldsymbol{D}) = \sum_{d \in \boldsymbol{D}} (NN_{\boldsymbol{\theta}}(d.\boldsymbol{X}) - d.\boldsymbol{y})^2$$

Note that *Loss* is a big formula. Backpropagation is a mechanism of calculating the derivative of each parameter with respect to this function.

Derivatives

A derivative measures a rate of change with respect to the inputs. It is a function. For example, a function $f(x) = 2x + 7$ has derivative of $f'(x) = 2$, meaning the function grows at a rate that is twice the rate of the input.

Often what we care about in machine learning is the *gradient*, which is the *value* of a derivative evaluated at a particular input. For function $f(x) = 3x^2 + 4x$ the derivative is $f'(x) = 6x + 4$, and the gradient at $x = 2$ is $6 * 2 + 4 = 16$.

Addition

Suppose our function is: $f(a, b) = a + b$.

What is the gradient for a with respect to $f(a, b)$? If we increase a by 1, how much will $f(a, b)$ increase by? Answer: 1.

What is the gradient for b with respect to $f(a, b)$? If we increase b by 1, how much will $f(a, b)$ increase by? Answer: 1.

Multiplication

Suppose our function is: $f(a, b) = a * b$.

What is the gradient for a with respect to $f(a, b)$? If we increase a by 1, how much will $f(a, b)$ increase by? Answer: b.

What is the gradient for b with respect to $f(a, b)$? If we increase b by 1, how much will $f(a, b)$ increase by? Answer: a.

Chain Rule

Wikipedia: As put by George F. Simmons: "If a car travels twice as fast as a bicycle and the bicycle is four times as fast as a walking man, then the car travels $2 \times 4 = 8$ times as fast as the man."

In other words, if we are dealing with gradients, we can just multiply them. For example:

$$a = w_1 * x_1 \tag{1}$$

$$b = w_2 * x_2 \tag{2}$$

$$y = a + b \tag{3}$$

The gradient for a is 1.0, since adding 1 to a will increase y by 1. So we can set $a.grad = 1$. Same reasoning for b.

What is the gradient for w_1? From multiplication section above, the gradient of a with respect to w_1 is x_1. To find out the gradient of y with respect to w_1 we need to multiply all the gradients in the path from y to w_1, or $w_1.grad = x_1 * a.grad$.

To calculate the gradient for every parameter (e.g. w_is, etc.), we need to run the network forward (the *forward-pass*) to calculate all the intermediate values. We then propagate the gradients backwards through the network (the *backprpagation*), to calculate a gradient for every parameter.

Gradient Descent

Gradient descent is an otimization (miminization) process, where we iteratively calculate/estimate the gradient at an input $\boldsymbol{W} = w_1, \ldots, w_n$, and nudge it into direction of negative gradient

$$w_i = w_i + \lambda * -w_i.grad$$

where λ is called the learning rate, and is often set to a small number, such as 0.001 (though controlling the learning rate, such as setting it to a larger number in the beginning and decreasing it towards the end of training is also common).

Autograd

Calculating the gradient can be done manually. What contributed to a recent AI explosion are software packates that support automatic differentiation (or more precisely, automatic gradients: *autograd*).

This is often accomplished by keeping track of the parse-tree for an expression—that way the *parents* of every operation can be kept track of. For example, $a+b$, operation is addition, with parents a and b.

Forward

Given an expression, and values for all variables, we can order topologically order them in a way where parents of every operation are visited before the children are calculated. Then we can simply calculate the *values* of each operation. e.g. if operation is a plus, then add the parents.

Backward

We order all operations such that the descendents of each operation is visited before the parents (reverse of the forward pass). We start off the outout gradient with 1.0. Then for every operation we calculate the gradient for the parents, and multiply by the gradient of the child.

See PyTorch API documentation for more details.