Introduction to Neural Networks (3)

Automatic differentiation (AD)

A simple example. Suppose we want to calculate the expression:

$$z = xy + \sin(x)$$

What if we're also interested in the derivatives of z? The "obvious" approach is to just find the expression by hand (or using a computer algebra system) and then punch it into the computer. But that assumes we have an explicit form for z. What if all we had was a program?

The important realization that leads to automatic differentiation is the fact that even biggest, most complicated program must be built from a small set of primitive operations such as addition, multiplication, or trigonometric functions. The chain rule allows us to take full advantage of this property.

Forward-mode automatic differentiation

First, we need to think about how a computer would evaluate z via a sequence of primitive operations (multiplication, sine, and addition):

Program A

$$\begin{aligned} x &= ? \\ y &= ? \\ a &= x * y \\ b &= \sin(x) \\ z &= a + b \end{aligned}$$

The question marks indicate that x and y are to be supplied by the user. We were careful to avoid reassigning to the same variable: this way we can treat each assignment as a plain old math equation. Let's try to differentiate each equation with respect to some yet-to-be-given variable t:

$$
\begin{aligned}
\frac{\partial x}{\partial t} &= ? \\
\frac{\partial y}{\partial t} &= ? \\
\frac{\partial a}{\partial t} &= y\frac{\partial x}{\partial t} + x\frac{\partial y}{\partial t} \\
\frac{\partial b}{\partial t} &= \cos(x)\frac{\partial x}{\partial t} \\
\frac{\partial z}{\partial t} &= \frac{\partial a}{\partial t} + \frac{\partial b}{\partial t}
\end{aligned}
$$

If we substitute $t = x$ into above equations, we'd have an algorithm for calculating $\frac{\partial z}{\partial x}$. Alternatively, to get $\frac{\partial z}{\partial y}$, we could just plug in $t = y$ instead.

Now, let's translate the equations back into an ordinary program involving the differential variables $\{dx, dy, ...\}$, which stand for $\{\frac{\partial x}{\partial t}, \frac{\partial y}{\partial t}, ...\}$ respectively:

Program B

$$
\begin{aligned}
dx &= ? \\
dy &= ? \\
da &= y * dx + x * dy \\
db &= \cos(x) * dx \\
dz &= da + db
\end{aligned}
$$

If we substitute $t = x$ into the mathematical equations, what happens to this program? The effect is remarkably simple: we just need to initialize $dx = 1$ and $dy = 0$ as the seed values for the algorithm. Hence, by choosing the seeds $dx = 1$ and $dy = 0$, the variable dz will contain the value of the derivative $\frac{\partial z}{\partial x}$ upon completion of the program. Similarly, if we want $\frac{\partial z}{\partial y}$, we would use the seed $dx = 0$ and $dy = 1$ and the variable dz would contain the value of $\frac{\partial z}{\partial y}$.

To translate using the rules, we simply replace each primitive operation in the original program by its differential analog. The order of the program remains unchanged: if a statement K is evaluated before another statement L, then the differential analog of statement K is still evaluated before the differential analog of statement L. This is forward-mode automatic differentiation.

A careful inspection of Program A and Program B reveals that it is actually possible to interleave the differential calculations with the original calculations:

$$
\begin{aligned}
x &= ? \\
dx &= ? \\
y &= ? \\
dy &= ? \\
a &= x * y \\
da &= y * dx + x * dy \\
b &= \sin(x) \\
db &= \cos(x) * dx \\
z &= a + b \\
dz &= da + db
\end{aligned}
$$

The implementation simplicity of forward-mode AD comes with a big disadvantage, which becomes evident when we want to calculate both $\frac{\partial z}{\partial x}$ and $\frac{\partial z}{\partial y}$. In forward-mode AD, doing so requires seeding with $dx = 1$ and $dy = 0$, running the program, then seeding with $dx = 0$ and $dy = 1$ and running the program again. In effect, the cost of the method scales linearly as $O(n)$ where n is the number of input variables. This would be very costly if we wanted to calculate the gradient of a large complicated function of many variables, which happens surprisingly often in artificial neural network learning.

Reverse-mode automatic differentiation

Let's take a second look at the chain rule we used to derive forward-mode AD:

$$
\begin{aligned}
\frac{\partial w}{\partial t} &= \sum_i \frac{\partial w}{\partial u_i} \frac{\partial u_i}{\partial t} \\
&= \frac{\partial w}{\partial u_1} \frac{\partial u_1}{\partial t} + \frac{\partial w}{\partial u_2} \frac{\partial u_2}{\partial t} + ...
\end{aligned} \tag{1}
$$

To calculate the gradient using forward-mode AD, we had to perform two substitutions: one with $t = x$ and another with $t = y$. This meant we had to run the entire program twice.

However, the chain rule is symmetric: it doesn't care what's in the "numerator" or the "denominator". So let's rewrite the chain rule but turn the derivatives upside down:

$$
\begin{aligned}
\frac{\partial s}{\partial u} &= \sum_i \frac{\partial w_i}{\partial u} \frac{\partial s}{\partial w_i} \\
&= \frac{\partial w_1}{\partial u} \frac{\partial s}{\partial w_1} + \frac{\partial w_2}{\partial u} \frac{\partial s}{\partial w_2} + ...
\end{aligned} \tag{2}
$$

In doing so, we have inverted the input-output roles of the variables. The same naming convention is used here: u for some input variable and w_i for each of the output variables that depend on u. The yet-to-given variable is now called s to highlight the change in position.

In this form, the chain rule could be applied repeatedly to every input variable u, akin to how in forward-mode AD we applied the chain rule repeatedly to every output variable w to get equations below. Therefore, given some t, we expect a program that uses chain rule (2) to be able to compute both $\frac{\partial s}{\partial x}$ and $\frac{\partial s}{\partial y}$ in one go!

So far, this is just a hunch. Let's try it on the example problem

$$\begin{aligned}
\frac{\partial s}{\partial z} &= ? \\
\frac{\partial s}{\partial b} &= \frac{\partial s}{\partial z} \\
\frac{\partial s}{\partial a} &= \frac{\partial s}{\partial z} \\
\frac{\partial s}{\partial y} &= x\frac{\partial s}{\partial a} \\
\frac{\partial s}{\partial x} &= y\frac{\partial s}{\partial a} + \cos(x)\frac{\partial s}{\partial b}
\end{aligned}$$

It can be quite mind-bending because everything seems "backwards": instead of asking what input variables a given output variable depends on, we have to ask what output variables a given input variable can affect. The easiest way to see this visually is by drawing a dependency graph of the expression:

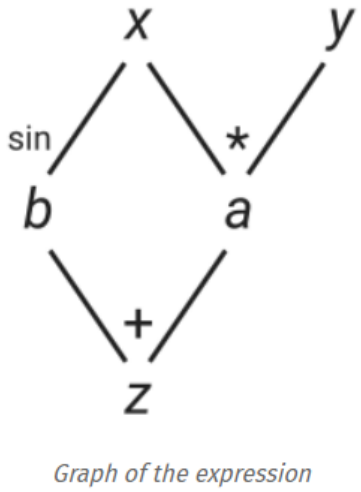

Figure 1: Graph

The graph shows that

- the variable a directly depends on x and y,
- the variable b directly depends on x, and
- the variable z directly depends on a and b.

Or, equivalently:

- the variable b can directly affect z,
- the variable a can directly affect z,
- the variable y can directly affect a, and
- the variable x can directly affect a and b.

Let's now translate the reverse-mode equations into code. As before, we replace the derivatives $\{\frac{\partial s}{\partial z}, \frac{\partial s}{\partial b}, ...\}$ by variables $\{gz, gb, ...\}$, which we call the adjoint variables. This results in:

$$\begin{aligned} gz &= ? \\ gb &= gz \\ ga &= gz \\ gy &= x * ga \\ gx &= y * ga + \cos(x) * gb \end{aligned}$$

Going back to the reverse-mode equations, we see that if we substitute $s = z$, we would obtain the gradient in the last two equations. In the program, this is equivalent to setting $gz = 1$ since gz is just $\frac{\partial s}{\partial z}$. We no longer need to run the program twice! This is reverse-mode automatic differentiation.

There is a trade-off, of course. If we want to calculate the derivative of a different output variable, then we would have to re-run the program again with different seeds, so the cost of reverse-mode AD is $O(m)$ where m is the number of output variables.

Reverse-mode tools are harder to implement, and use more memory, but have a major advantage for machine learning applications. One reverse-mode sweep simultaneously accumulates the derivatives of the output with respect to every quantity in the computation graph. If we give the function multiple inputs, we can still obtain all of the derivatives in a constant times the number of operations used by one function evaluation. However, the forwards-mode procedure would require multiple sweeps through the graph, one sweep for each partial derivative. For a neural network with millions of parameters, reverse-mode differentiation (or backpropagation) is millions of times faster than forwards-mode differentiation! It's easy to dismiss differentiation as "just the chain rule". But reverse-mode differentiation is amazing. We can get a million partial-derivatives, which describe the whole tangent hyperplane of a function, usually for the cost of only a couple of evaluations of the scalar function!

Backpropagation is the term from the neural network literature for reverse-mode differentiation. Backpropagation is also sometimes used to mean the whole procedure of training a network with a gradient descent method, where the gradients come from reverse-mode differentiation.

Backpropagation

Backpropagation is the algorithm used to compute the gradients of the network. This procedure was developed by several authors in the decade of the 60's but is Paul J. Werbos, (1974) in his thesis when demonstrates the use of this algorithm for ANN. Years later, (David, E. 1986) presents the modern way to apply this technique to ANN, and sets the basis of the algorithm in use today. In this paper, the authors presents a new method capable to change the predictions towards a desired output, they called it the delta rule.

This rule consist in compute the total error for the network and check how the error changes when certain elements from the network changes its value. How do we compute this changes? differentiating the cost function with regard to each element in the network would give us a measure of how much each element is contributing to the total error of the network, this is, computing the gradient of the cost function we can know how the total error changes with regard to each element, and therefore apply the delta rule.

The cost function is an intricate composed function which contains the weights of all layers, the problem now is that the computations of this gradients are not straightforward as in a simple function, a node from a layer is the result of the composition of all the nodes from previous layers. To overcome it, Backpropagation uses the chain rule of differential calculus to compute the gradients of each element in the neural network, it contains two main phases referred to as the forward phase and backward phase:

- Forward Phase: The inputs for a data example are fed into the FNN. This inputs will be propagated through the network for all the neurons in all the layers using the current set of weights, to finally compute the final output.
- Backward Phase: Once the final output and the cost function are computed, we need to compute the gradients for all the weights in all the layers in the FNN to update them in order to reach the minimum. To compute the gradients the chain rule is used, is a backwards a process from output to input, therefore we will start from the output node, compute all the gradients of the previous layer, and so on until we reach the input layer.

Example of backpropagation

We aim to minimice the cost function

$$\min_{\Theta} J(\Theta)$$

In order to use gradient descent, we need to compute $J(\Theta)$ and the partiall derivative terms

$$\frac{\partial}{\partial \theta_{ij}^{(l)}} J(\Theta)$$

We compute $J(\Theta)$. How can we compute the partial derivative terms? Given a **single one training example** (x, y). The cross entropy error for a single example with K independent

targets is given by the sum

$$J(\Theta) = -\sum_{k=1}^{K}\left(y_k \log\left(h_\theta(x)\right)_k + (1-y_k)\log\left(1-(h_\theta(x)\right)_k\right) \tag{3}$$

$$= -\sum_{k=1}^{K}\left(y_k \log a_k^{(3)} + (1-y_k)\log(1-a_k^{(3)})\right) \tag{4}$$

where $y = (y_1, ..., y_K)^\intercal$ is the target vector and $a^{(3)} = (a_1^{(3)}, ..., a_K^{(3)})^\intercal$ is the output vector. In this architecture (see figure 2) the outputs are computed by applying the sigmoid function to the weights sums of the hidden layer activations.

$$a_k^{(3)} = \frac{1}{1+e^{-z_k^{(3)}}} \tag{5}$$

$$z_k^{(3)} = \sum_j a_j^{(2)}\theta_{kj}^{(2)} \tag{6}$$

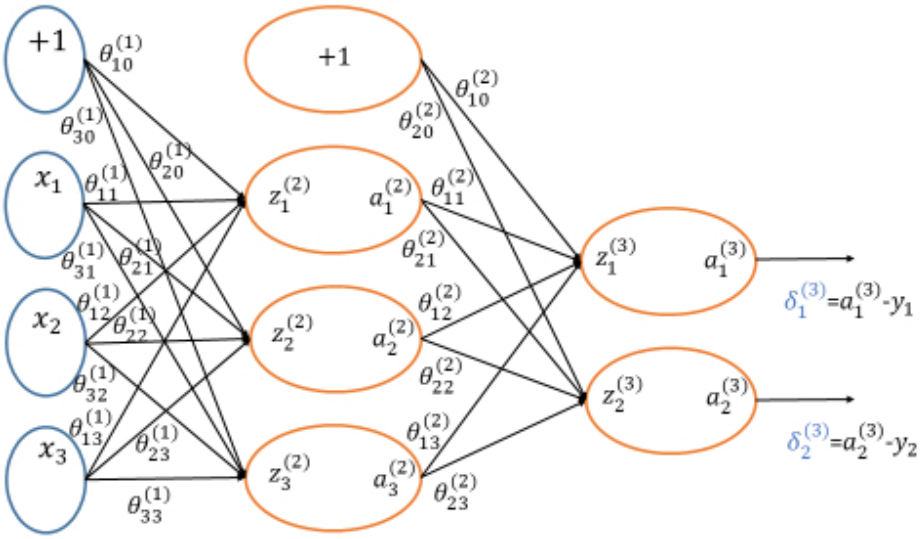

Figure 2: Backpropagation

We can compute the derivative of the error with respect to each weight connecting the hidden units to the output units using the chain rule.

$$\frac{\partial J}{\partial \theta_{kj}^{(2)}} = \frac{\partial J}{\partial a_k^{(3)}}\frac{\partial a_k^{(3)}}{\partial z_k^{(3)}}\frac{\partial z_k^{(3)}}{\partial \theta_{kj}^{(2)}}$$

Examining each factor in turn,

$$\frac{\partial J}{\partial a_k^{(3)}} = -\frac{y_k}{a_k^{(3)}} + \frac{1-y_k}{1-a_k^{(3)}} \tag{7}$$

$$= \frac{a_k^{(3)} - y_k}{a_k^{(3)}(1-a_k^{(3)})} \tag{8}$$

$$\frac{\partial a_k^{(3)}}{\partial z_k^{(3)}} = a_k^{(3)}(1-a_k^{(3)}) \tag{9}$$

$$\frac{\partial z_k^{(3)}}{\partial \theta_{kj}^{(2)}} = a_j^{(2)} \tag{10}$$

Combining things back together,

$$\frac{\partial J}{\partial z_k^{(3)}} = a_k^{(3)} - y_k$$

and

$$\frac{\partial J}{\partial \theta_{kj}^{(2)}} = (a_k^{(3)} - y_k)a_j^{(2)}$$

The above gives us the gradients of the cost with respect to the weights in the last layer of the network, but computing the gradients with respect to the weights in lower layers of the network (i.e. connecting the inputs to the hidden layer units) requires another application of the chain rule. This is the backpropagation algorithm.

It is useful to calculate the quantity $\frac{\partial J}{\partial z_j^{(2)}}$ where j indexes the hidden units,

$$z_j^{(2)} = \sum_s a_s^{(1)}\theta_{js}^{(1)} = \sum_s x_s\theta_{js}^{(1)}$$

is the weigthed input at hidden unit j, and

$$a_j^{(2)} = \frac{1}{1+e^{-z_j^{(2)}}}$$

is the activation at unit j.

We have

$$\frac{\partial J}{\partial z_j^{(2)}} = \sum_k^K \frac{\partial J}{\partial z_k^{(3)}} \frac{\partial z_k^{(3)}}{\partial a_j^{(2)}} \frac{\partial a_j^{(2)}}{\partial z_j^{(2)}} \tag{11}$$

$$= \sum_k^K ((a_k^{(3)} - y_k))(\theta_{kj}^{(2)})(a_j^{(2)}(1-a_j^{(2)})) \tag{12}$$

Then a weight $\theta_{js}^{(1)}$ connecting input unit j to hidden unit s has gradient

$$\frac{\partial J}{\partial \theta_{js}^{(1)}} = \frac{\partial J}{\partial z_j^{(2)}} \frac{\partial z_j^{(2)}}{\partial \theta_{js}^{(1)}} \tag{13}$$

$$= \sum_k^K ((a_k^{(3)} - y_k))(\theta_{kj}^{(2)})(a_j^{(2)}(1 - a_j^{(2)}))(x_s) \tag{14}$$

$$= a_j^{(2)}(1 - a_j^{(2)}) \Big(\sum_k^K (a_k^{(3)} - y_k)\theta_{kj}^{(2)} \Big) x_s \tag{15}$$

By recursively computing the gradient of the error with respect to the activity of each neuron, we can compute the gradients for all weights in a network.

References

- Aggarwal, Charu C. Neural networks and deep learning. Berlin, Germany. Springer, 2018.
- https://alexander-schiendorfer.github.io/2020/02/16/automatic-differentiation.html
- https://rufflewind.com/2016-12-30/reverse-mode-automatic-differentiation