Tokyo Data Science

Pierre Ouannes

April 27, 2019

Contents

I. Introduction

Risk is expressed as

$$J^*(\theta) = \sum_x \sum_y p_{data}(x,y) L\left(f(x,\theta), y\right)$$

And empirical risk:

$$J(\theta) = \frac{1}{m} \sum_{i=1}^{m} L\left(f(x^{(i)},\theta), y^{(i)}\right)$$

We then expresss the gradients of risk as:

$$g = \nabla_\theta J^*(\theta) = \sum_x \sum_y p_{data}(x,y) \nabla_\theta L\left(f(x,\theta), y\right)$$

And that of the empirical risk:

$$\hat{g} = \nabla_\theta J(\theta) = \frac{1}{m} \sum_{i=1}^{m} \nabla_\theta L\left(f(x^{(i)},\theta), y^{(i)}\right)$$

You can notice that you have the following relation between g and $\hat{g}$:

$$\mathbb{E}[\hat{g}] = g$$

As such, $\hat{g}$ is an unbiased estimator of g.

A. Automatic Differentiation

See talk by Matthew Johnson on Automatic differentiation at the Deep Learning Summer School (DLSS) 2017

If we have y a scalar, x a vector and F a function that is itself made up of several functions compositions that we can write as A, B, C and D, and the following relationship:

$$y = F(x) = D(C(B(A(x))))$$

We can also construct intermediate variables:

$$\begin{aligned} a &= A(x) \\ b &= B(a) \\ c &= C(b) \\ y &= D(c) \end{aligned}$$

The following notation:

$$\frac{\partial a}{\partial x}$$

refers to the derivation of vector a with respect to vector x, and can be written as a Jacobian matrix:

$$\frac{\partial a}{\partial x} = \begin{pmatrix} \frac{\partial a_2}{\partial x_1} & \frac{\partial a_1}{\partial x_2} & \cdots \\ \frac{\partial a_2}{\partial x_1} & \frac{\partial a_2}{\partial x_2} & \cdots \\ \vdots & & \end{pmatrix}$$

Using the chain rule, we have:

$$\nabla_x F(x) = \frac{\partial y}{\partial x} = \frac{\partial y}{\partial c} \cdot \frac{\partial c}{\partial b} \cdot \frac{\partial b}{\partial a} \cdot \frac{\partial a}{\partial x}$$

Each element of this chain rule is a Jacobian matrix, so this is a multiplication of matrices.

Here, x is the parameters of the Neural Network and $F(x)$ refers to the loss function. In a Neural Net, the dimensions are often reduced when we progress

through the net. So A reduces the dimension, B also, and so on. That makes sense: we go from an high-dimension input (like an image) so a low-dimension output (a class).

So in this chain rule, where could we start? At the end, or at the beginning?

The answer is to begin with $\frac{\partial d}{\partial c}$, from the left. That's because of the size of the matrices: there are much smaller than the others: we start from the loss and go backwards, and that's called *back propagation*, or doing a *backward pass*. Indicated in parenthesis is the way to do the computations:

$$\nabla_x F(x) = \frac{\partial y}{\partial x} = \left(\left(\frac{\partial y}{\partial c} \cdot \frac{\partial c}{\partial b}\right) \cdot \frac{\partial b}{\partial a}\right) \cdot \frac{\partial a}{\partial x}$$

Basically, you can see it as back propagating the small matrices sizes instead of forward propagating the big matrices sizes.

For example, let's say $\frac{\partial y}{\partial c}$ is of size 10×100, $\frac{\partial c}{\partial b}$ of size 100×100, $\frac{\partial b}{\partial a}$ of size 100×784, and finally $\frac{\partial a}{\partial x}$ of size 784×784. Then the computation in a forward pass would go as :

$$\begin{aligned}\left[\frac{\partial y}{\partial x}\right] &= [10 \times 100] \cdot ([100 \times 100] \cdot ([100 \times 784] \cdot [784 \times 784])) \\ &= [10 \times 100] \cdot ([100 \times 100] \cdot [100 \times 784]) \\ &= [10 \times 100] \cdot [100 \times 784] \\ &= [10 \times 784]\end{aligned}$$

Whereas doing the computation in a backward pass would more of:

$$\begin{aligned}\left[\frac{\partial y}{\partial x}\right] &= (([10 \times 100] \cdot [100 \times 100]) \cdot [100 \times 784]) \cdot [784 \times 784] \\ &= ([10 \times 100] \cdot [100 \times 784]) \cdot [784 \times 784] \\ &= [10 \times 784] \cdot [784 \times 784] \\ &= [10 \times 784]\end{aligned}$$

So you see that even though you end up with the same 10×784 matrix, you can either get there by doing big matrix calculations (in a forward pass) or smaller ones (in a backward pass).

There are two main Deep Learning frameworks: PyTorch and TensorFlow. Each use different kinds of automatic differentiations engines. PyTorch use *autograd*, while TensorFlow use *graph based differentiation*.

Let's say you have a simple function: $x^2 + abs(x)$. Then we ask autograd: if $x = 7$, what is the gradient? What autograd does is that it replaces the absolute value: $x^2 + x$, and then compute the gradient: $2x + 1$. So with $x = 7$, you get 15.

With graph based differentiation, things are a bit different. This engine does a disjunction of cases: if $x > 0$, the function is $x^2 + x$ so a gradient of $2x + 1$, wheras if $x < 0$ the function would then be $x^2 - x$ so a gradient of $2x - 1$.

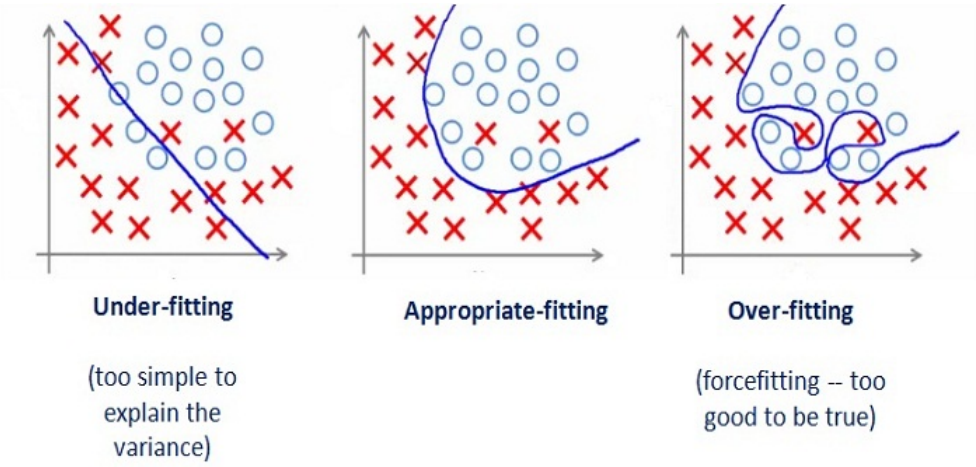
Even though you have to compute the differentiation every times with autograd, whereas you just have the path with graph based differentiation, the speed of the two engines is similar (autograd is a bit slower but not by much). On the other hand, the whole graph can be huge for complex neural networks. So as often, it's a tradeoff between storage and efficiency. Graph-based differentiation allow you to "precompile" the network and makes for faster differentiation, but at the cost of storage.

Good exercise: cs321n's homework of designing a Neural Net in NumPy.

Note: these days, PyTorch also uses Graph based diff, and TF also uses autograd.

B. Overfitting

Underfitting vs overfitting:



How to avoid overfitting? For example, let's say you have a polynomial:

$$P(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{20} x^{20}$$

You can try to add something to the loss fonction that constrains the size of the coefficient, something like:

$$R(\mathbf{a}) = a_0^2 + a_1^2 + \cdots + a_{20}^2 = \sum_{i=0}^{20} a_i^2$$

Then we can add R to the loss to try to avoid overfitting:

$$Loss + \alpha \underbrace{\sum_{i=0}^{20} a_i^2}_{\text{Regularization}}$$

α is an hyperparameter that regulates how much regularization we want to add.

Is it possible to overfit while doing linear regression? Let's say you have:

$$y = w_0 + w_1 x_1$$

Then you probably won't overfit. However, if you have a linear regression with a lot of variables:

$$y = w_0 + \sum_{i=1}^{2000} w_i x_i$$

Then you may overfit. One example of that may be a survey.

a. Multivariate regression

Multivariate regression is something like:

$$\hat{y}(w, x) = w_0 + w_1 x_1 + \cdots + w_p x_p$$

x_i are called *regressors* in statistics, and *features* in Machine Learning. In a neural networks, w_i for $i > 0$ would be called the *weights*, and w_0 the *bias* and is rather written as b.

To make the notation simpler, we'll rather write:

$$\hat{y}(w, x) = w_0 x_0 + w_1 x_1 + \cdots + w_p x_p$$

With $x_0 = 1$. It's exactly the same thing, but it's not simpler to write for example with a single sum:

$$\hat{y}(w, x) = \sum_{i=0}^{p} w_i x_i \quad \text{instead of} \quad \hat{y}(w, x) = w_0 + \sum_{i=1}^{p} w_i x_i$$

We can write the loss as:

$$Loss = \frac{1}{m} \sum_{i=1}^{m} \left(y^{(i)} - \sum_{j=1}^{p} w_j x_j \right)^2$$

We can also right it as:

$$Loss = \frac{1}{m} \sum_{i=1}^{m} \left(y^{(i)} - \mathbf{w} \cdot \mathbf{x} \right)^2$$

With $\mathbf{x} = (x_1, \ldots, x_p)$ and $\mathbf{w} = (w_1, \ldots, w_p)$.

Finally, we can also right is as:

$$Loss = \frac{1}{m} \left\| X \cdot \mathbf{w} - \mathbf{y} \right\|_2^2$$

In this expression, $\mathbf{w}$ is the vector from before but now in a column, to make the matrix multiplication work, while X is:

$$X = \begin{bmatrix} x_1^{(1)} & x_2^{(1)} & \ldots & x_p^{(1)} \\ x_1^{(2)} & x_2^{(2)} & \ldots & x_p^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ x_1^{(m)} & x_2^{(m)} & \ldots & x_p^{(m)} \end{bmatrix}$$

So it's a $m \times p$ matrix, with each column containing a given observation.

What is $\|\cdot\|_2$? It's the L^2 norm, defined as, for a vector v:

$$\|v\|_2 = (v_1^2 + v_2^2 + \ldots v_p^2)^{1/2}$$

More generally, L^n norm is defined as:

$$\|v\|_n = (v_1^n + v_2^n + \ldots v_p^n)^{1/n}$$