Deep Learning Training Program

Sasank Chilamkurthy, Qure.ai

May 10, 2017

Calculus: Recap

Let's recap what a derivative is.

Derivative

Derivative of function $f(v)$ ($f'(v)$ or $\frac{df}{dv}$) measures sensitivity of change in $f(v)$ with respect of change in v.

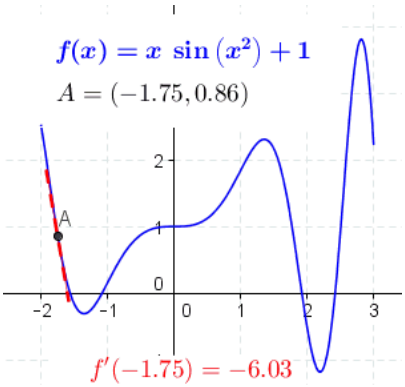

Figure 1: Derivative illustration. Red is for positive v direction and green is for negative v direction. Source.

Direction (i.e sign) of the derivative at a points gives the direction of (greatest) increment of the function at that point.

Gradient

The gradient is a multi-variable generalization of the derivative. It is a vector valued function. For function $f(v_1, v_2, \ldots, v_n)$, gradient is a vector whose components are n partial derivatives of f:

$$\nabla f = \left(\frac{\partial f}{\partial v_1}, \frac{\partial f}{\partial v_2}, \ldots, \frac{\partial f}{\partial v_n}\right)$$

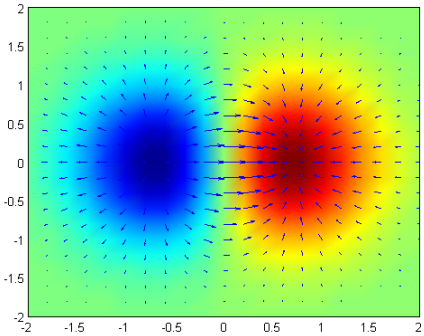

Figure 2: Gradient of the 2D function $f(x, y) = xe^{(x^2+y^2)}$ is plotted as blue arrows over the pseudocolor (red is for high values while blue is for low values) plot of the function. Source.

Similar to derivative, direction of the gradient at a point is the steepest ascent of the function starting from that point.

Optimization

Given a function $C(v_1, v_2, \ldots, v_n)$, how do we optimize it? i.e, find a point which minimizes this function globally.

This is a very generic problem; lot of practical and theoretical problems can be posed in this form. So, there is no general answer.

Gradient Descent

However, for a class of functions called convex functions, there is a simple algorithm which is guaranteed to converge to minima. Convex functions have only one minima. They look something like this:

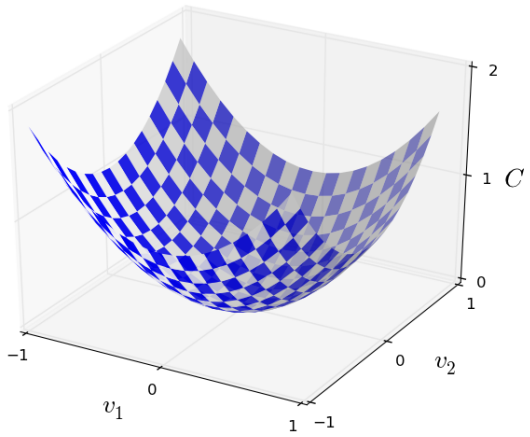

Figure 3: A 2d convex function. Source.

To motivate our algorithm, imagine this function as a valley and a ball is put at a random point on the valley and allowed to roll. Our common sense tells that ball will eventually roll to the bottom of the valley.

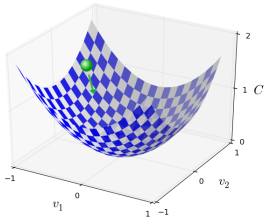

Figure 4: Gradient descent illustration: a ball on a valley. Source.

Let's roughly simulate the motion of the ball! Key observation is that the ball moves in the steepest direction of descent. This is negative [1] of the gradient's direction.

Great! Let's put together our algorithm:

[1] Gradient gives us steepest direction of *ascent*.

Algorithm 1: Gradient Descent

1: Start at a random point: v
2: **while** v doesn't converge **do**
3: Update v in the direction of steepest descent:

$$v \to v' = v - \eta \nabla C$$

4: **end while**

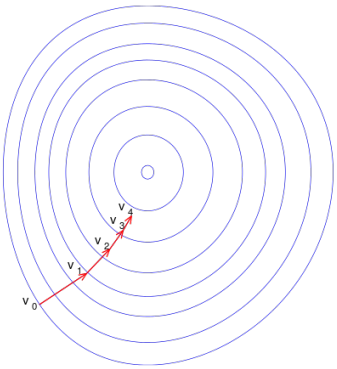

Figure 5: Gradient descent on a series of level sets. Source.

Here η is called *learning rate*. If it is too small, algorithm can be very slow and might take too many iterations to converge. If it is too large, algorithm might not even converge.

Example: Regression

Let's apply the things learnt above on linear regression problem. Here's a recap of linear regression:

Our model is $y(x) = wx + b$. Parameters are $v = (w, b)$ We are given training pairs $(x^1, y^1), (x^2, y^2), \ldots, (x^n, y^n)$ [2].

We want to find w and b which minimize the following cost/loss function:

$$C(w,b) = \frac{1}{n}\sum_{i=1}^{n} C_i(w,b) = \frac{1}{2n}\sum_{i=1}^{n} \|y(x^i) - y^i\|^2$$

where $C_i(w,b) = \frac{1}{2}\|y(x^i) - y^i\|^2 = \frac{1}{2}\|wx^i + b - y^i\|^2$ is the loss of the model for i th training pair.

Let's calculate gradients,

$$\nabla C = \frac{1}{n}\sum_{i=1}^{n} \nabla C_i$$

where ∇C_i is computed using:

$$\nabla C_i = \left(\frac{\partial C_i}{\partial w}, \frac{\partial C_i}{\partial b}\right) = \Big((wx^i + b - y^i)x^i, (wx^i + b - y^i)\Big)$$

Update rule is

$$v \to v' = v - \eta \nabla C = v - \frac{\eta}{n}\sum_{i=1}^{n} \nabla C_i$$

Stochastic Gradient Descent

In the above example, what if we have very large number of samples i.e $n \gg 0$? At every optimization step, we have to compute ∇C_i for each sample $i = 1, 2, \ldots, n$. This can be very time consuming!

Can we approximate ∇C with a very few samples? Yes!

$$\nabla C = \frac{1}{n}\sum_{i=1}^{n} \nabla C_i \approx \frac{1}{m}\sum_{j \in S_m} \nabla C_j$$

[2] Notation clarifiction: Here, superscripts are indices, not powers

where S_m is random subset of size $m \ll n$ of $1, 2, \ldots, n$. It turns out that this approximation, even though estimated only from a small random subset of samples, is good enough for convergence of gradient descent. This subset of data is called *minibatch* and this technique is called *stochastic gradient descent*.

Then stochastic gradient descent works by picking a randomly chosen subset of data and trains (i.e updates v) with gradient approximation computed from them. Next, another subset is picked up and trained with them. And so on, until we've exhausted all the training data, which is said to complete an *epoch* of training. Concretely, following is the stochastic gradient descent algorithm.

Algorithm 2: Stochastic Gradient Descent

1: Start at a random point: v
2: **for** a fixed number of epochs **do**
3: Randomly partition the data into minibatches each of size m
4: **for** For each minibatch S_m **do**
5: Update the parameters using

$$v \to v' = v - \frac{\eta}{m} \sum_{i \in S_m} \nabla C_j$$

6: **end for**
7: **end for**

Neural Networks

With this background, we are ready to start with neural networks.

Perceptron

Perceptron, a type of artificial neuron was developed in the 1950s and 1960s. Today, it's more common to use Rectified Linear Units (ReLUs). Nevertheless, it's worth taking time to understand the older models.

So how do perceptrons work? A perceptron takes several inputs, $x_1, x_2, \ldots x_n$ and produces a single output:

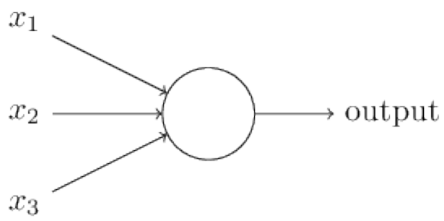

Figure 6: Perceptron Model. Source.

Weights $w_1, w_2, \ldots w_n$ decide the importance of each of the inputs on output $y(x)$. There is also a *threshold* b to decide the output. These are the parameters of the model.

The expression of the output is

$$y(x) = \sigma\left(\sum_j w_j x_j - b\right)$$

where $\sigma(z)$ is step function,

$$\sigma(z) \quad = \quad \begin{cases} 0 & \text{if } z \leq 0 \\ 1 & \text{if } z > 0 \end{cases}$$

Therefore

$$y(x) \quad = \quad \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq b \\ 1 & \text{if } \sum_j w_j x_j > b \end{cases}$$

That's the basic mathematical model. A way you can think about the perceptron is that it's a device that makes decisions by weighing up evidence. An example [3]:

[3] This example is straight from here

Another way perceptrons can be used is to compute the elementary logical functions we usually think of as underlying computation, functions such as `AND`, `OR`, and `NAND`. Check that following perceptron implements `NAND`:

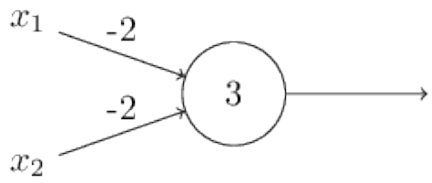

Figure 7: `NAND` implemented by perceptron. Source

If you are familiar with digital logic, you will know that `NAND` gate is a universal gate. That is, any logical computation can be computed using just `NAND` gates. Then, the same property follows for perceptrons.

Multi Layer Perceptrons and Sigmoid

Although perceptron isn't a complete model of human decision-making, the above example illustrates how a perceptron can weigh up different kinds of evidence in order to make decisions. And it should seem plausible that a complex network of perceptrons could make quite subtle decisions

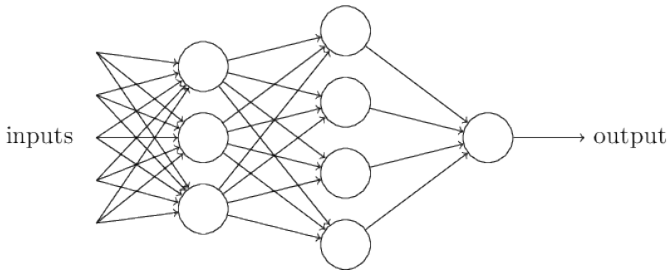

Figure 8: Multi layer perceptron. Source

This network is called *Multi layered perceptron (MLP)*. In this MLP, the first column of perceptrons - what we'll call the first *layer* of perceptrons - is making three very simple decisions, by weighing the

input evidence. What about the perceptrons in the second layer? Each of those perceptrons is making a decision by weighing up the results from the first layer of decision-making. In this way a perceptron in the second layer can make a decision at a more complex and more abstract level than perceptrons in the first layer. And even more complex decisions can be made by the perceptron in the third layer. In this way, a many-layer network of perceptrons can engage in sophisticated decision making.

How do we learn the parameters of the above model? Gradient Descent! [4] However, the network is very discontinuous. In fact, a small change in the weights or bias of any single perceptron in the network can sometimes cause the output of that perceptron to completely flip, say from 0 to 1. This makes it very difficult for gradient descent to converge.

[4] MLP model is far from convex. Therefore, gradient descent is not guaranteed to converge! But it turns out that it works fine with a few tweaks which we describe below.

How do we overcome this? What is the source of this discontinuity? Remember that output of perceptron is given by $y(x) = \sigma\left(\sum_j w_j x_j - b\right)$ where $\sigma(z)$ is step function

$$\sigma(z) = \begin{cases} 0 & \text{if } z \leq 0 \\ 1 & \text{if } z > 0 \end{cases}$$

This $\sigma(z)$ is the source of discontinuity. Can we replace step function with a smoother version of it?

Check out the following function:

$$\sigma(z) = \frac{1}{1+e^{-z}}.$$

If you graph it, it's quite clear that this function is smoothed out version of a step function. This function is called *sigmoid*.

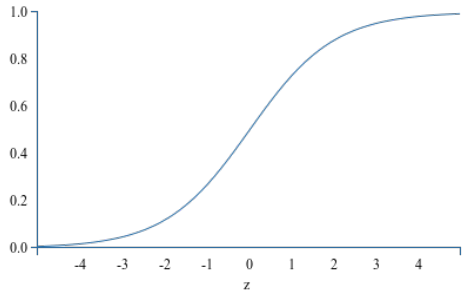

Figure 9: Sigmoid function. When z is large and positive, Then $e^{-z} \approx 0$ and so $\sigma(z) \approx 1$. Suppose on the other hand that z is very negative. Then $e^{-z} \to \infty$, and $\sigma(z) \approx 0$. Source.

With the *sigmoid neuron*, gradient descent usually converges. Before computing gradients for gradient descent, we need to discuss loss functions[5] and activation functions.

[5] I use terms loss function and cost function interchangeably.

Activation and Loss Functions

By now, you have seen that general form of a artificial neuron is $y(x) = \sigma\left(w^T x + b\right)$. I have rewritten sum $\sum_j w_j x_j$ as dot product

$w^T x$ and changed the sign of b. . Here the function $\sigma(z)$ is called *activation function*. So far, we have seen two different activations:

1. Step function
2. Sigmoid function

ReLU:

Let me introduce another activation function, *rectifier* or *rectified linear unit* (ReLU):

$$\sigma(z) = \max(0, z)$$

Its graph looks like this:

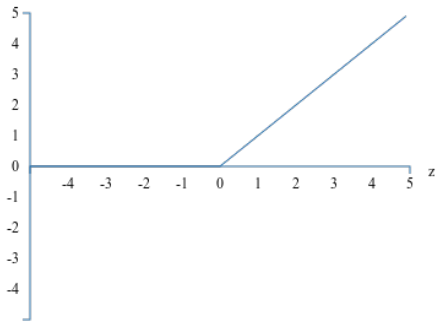

Figure 10: ReLU. Source.

Because of technical reasons [6], ReLUs are preferred activation functions these days. You will almost never see sigmoid activation function in modern deep neural networks.

[6] More specifically, because of vanishing and exploding gradients problem. Read more here

Loss functions:

Let's discuss loss/cost functions now. We will make two assumptions about our cost function:

1. The cost function can be written as an average over cost functions C_i for individual training examples, (x^i, y^i). i.e, $C = \frac{1}{n}\sum_{i=1}^{n} C_i$
2. Cost can be written as a function of the outputs from the neural network. i.e, $C_i = L(y(x^i), y^i)$ where $y(x)$ is the output from the network.

In the case of regression, we used L_2 loss, $L(o, y) = \frac{1}{2}\|o - y\|^2$. We could have also used L_1 loss, $L(o, y) = \|o - y\|_{L_1}$.

Cross Entropy Loss:

What if we have a classification problem? What loss do we use? Consider the following MLP for digit classification:

Here, we have output o of size 10 and a target class y. We could have used $L_1(o, e_y)$ or $L_2(o, e_y)$ loss where e_y is yth unit vector. But it turns out that this doesn't work very well.

Instead, We will consider the outputs as a probability distribution over 10 classes and use what is called a cross entropy loss:

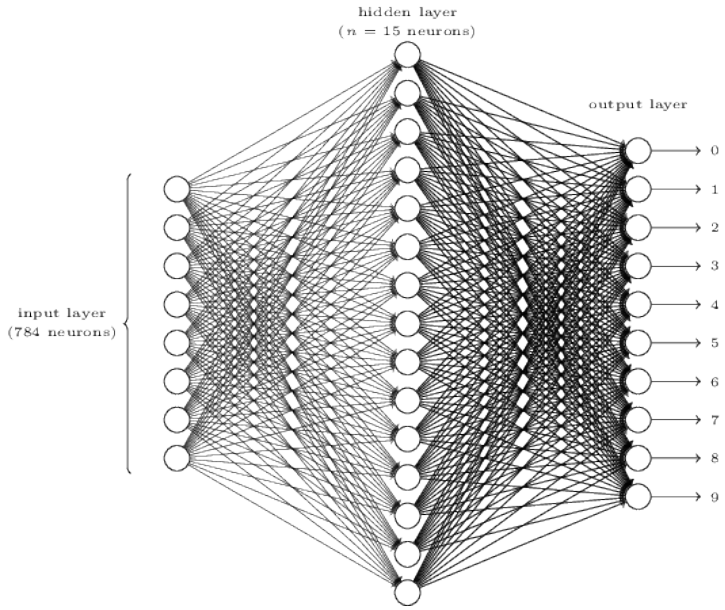

Figure 11: MLP for digit classification. Source.

$$L(o, y) = -\log(o_y)$$

To understand this function, realize that o_y is the output probability of the target class y. Minimizing negative of log of this probability, maximizes the probability. Also, $L \geq 0$ because $\log(x) < 0$ for $x \in [0, 1]$.

Softmax Activation:

If we use cross entropy loss, we cannot use sigmoids as activations of output layer because sigmoids do not guarantee a probability distribution. Although each component of the output is in $[0, 1]$, they need not add up to 1.

We therefore use an activation layer called *softmax*. According to this function, the activation a_j of the jth output neuron is

$$a_j = \frac{e^{z_j}}{\sum_k e^{z_k}}$$

where in the denominator we sum over all the output neurons. This expression may seem opaque if you are not familiar with it. Observe the following:

1. Activations are positive: $a_j \geq 0$
2. Activations sum to 1: $\sum_j a_j = 1$
3. If you increase z_m keeping others constant, a_m increases. Other activations decrease to ensure sum remains 1.
4. If z_m is much larger than the others, $a_m \approx 1$ and $a_k \approx 0$ for $k \neq m$.

Therefore softmax is a probability distribution which behaves like smooth version of `argmax`.

Backpropogation

We have so far discussed the model component of neural networks. We haven't yet discussed how we learn the parameters of the networks.

As expected, we will use stochastic gradient descent. For this, we need gradients of C_i, loss for the ith training example. Computation of this quantity, ∇C_i is slightly involved. Let's start with writing the expression for C_i. Let's represent all the parameters of the network with θ:

$$C_i(\theta) = L\left(y(x^i, \theta), y^i\right)$$

Let's break the above function into composition of functions (or layers).

$$C = y_n \circ \ y_{n-1} \circ \cdots \circ y_1$$

Here, ith [7] function takes in input u_i and outputs

[7] In particular, $C = o_n = y_n(u_n) = L(u_n, x_i)$

$$o_i = y_i(u_i, \theta_i) \tag{1}$$

where θ_i are learnable parameters of this function. Since output of $i-1$th layer is fed to ith layer as input,

$$u_i = o_{i-1} \tag{2}$$

We require $\nabla C = \left(\frac{\partial C}{\partial \theta_1}, \frac{\partial C}{\partial \theta_2}, \ldots, \frac{\partial C}{\partial \theta_n}\right)$. Therefore, we need to compute, for $j = 1, 2, \ldots n$

$$\frac{\partial C}{\partial \theta_j} = \frac{\partial o_n}{\partial \theta_j}$$

To compute this quantity, we will compute generic [8]:

[8] You will see ahead why this quantity is useful.

$$\frac{\partial o_i}{\partial \theta_j}$$

Before getting started, let's write down the chain rule. Chain rule is the underlying operation of our algorithm.

For function f(x, y),

$$\frac{\partial f}{\partial t} = \frac{\partial f}{\partial x} * \frac{\partial x}{\partial t} + \frac{\partial f}{\partial y} * \frac{\partial y}{\partial t} \tag{3}$$

If $j > i$,

$$\frac{\partial o_i}{\partial \theta_j} = 0 \tag{4}$$

because output of the functions in back doesn't depend on parameters of layer in the front.

If $j = i$, using equation (1) and the fact that u_i and θ_i are independent,

$$\frac{\partial o_i}{\partial \theta_i} = \frac{\partial y_i}{\partial \theta_i} \tag{5}$$

$\frac{\partial y_i}{\partial \theta_i}$ is a computable quantity which depends on the form of function y_i.

If $j < i$, using equation (1), (2), chain rule (3),

$$\begin{aligned}
\frac{\partial o_i}{\partial \theta_j} &= \frac{\partial y_i}{\partial \theta_j} \\
&= \frac{\partial y_i}{\partial u_i} * \frac{\partial u_i}{\partial \theta_j} + \frac{\partial y_i}{\partial \theta_i} * \frac{\partial \theta_i}{\partial \theta_j} \\
&= \frac{\partial y_i}{\partial u_i} * \frac{\partial u_i}{\partial \theta_j} \\
&= \frac{\partial y_i}{\partial u_i} * \frac{\partial o_{i-1}}{\partial \theta_j}
\end{aligned}$$

Therefore,

$$\frac{\partial o_i}{\partial \theta_j} = \frac{\partial y_i}{\partial u_i} * \frac{\partial o_{i-1}}{\partial \theta_j} \tag{6}$$

Like $\frac{\partial y_i}{\partial \theta_i}$, $\frac{\partial y_i}{\partial u_i}$ is a computable quantity which depends on y_i.

Let's put everything together and compute the required quantity [9]:

[9] Apply (6) repetitively

$$\begin{aligned}
\frac{\partial C}{\partial \theta_j} &= \frac{\partial o_n}{\partial \theta_j} \\
&= \frac{\partial y_n}{\partial u_n} * \frac{\partial o_{n-1}}{\partial \theta_j} \\
&= \frac{\partial y_n}{\partial u_n} * \frac{\partial y_{n-1}}{\partial u_{n-1}} * \frac{\partial o_{n-2}}{\partial \theta_j} \\
&\vdots \\
&= \frac{\partial y_n}{\partial u_n} * \frac{\partial y_{n-1}}{\partial u_{n-1}} * \cdots * \frac{\partial y_{j-1}}{\partial u_{j-1}} * \frac{\partial o_j}{\partial \theta_j} \\
&= \frac{\partial y_n}{\partial u_n} * \frac{\partial y_{n-1}}{\partial u_{n-1}} * \cdots * \frac{\partial y_{j-1}}{\partial u_{j-1}} * \frac{\partial y_j}{\partial \theta_j}
\end{aligned}$$

Now, algorithm to compute gradients ∇C, i.e. $\frac{\partial C}{\partial \theta_j}$ for all j is fairly clear.

Algorithm 3: Back Propogation

1: {Forward pass}
2: Set $u_0 = x$
3: **for** $i = 1, \ldots n$ **do**
4: Store $u_i = y_i(u_{i-1}, \theta_i)$
5: **end for**
6: {Backward pass}
7: Set `buffer` $= 1$
8: **for** $j = n, n-1, \ldots 1$ **do**
9: Store $\frac{\partial C}{\partial \theta_j} = \frac{\partial y_j}{\partial \theta_j} *$ `buffer`
10: Update `buffer` $= \frac{\partial y_j}{\partial u_j} *$ `buffer`
11: **end for**
12: **return** $\left(\frac{\partial C}{\partial \theta_1}, \frac{\partial C}{\partial \theta_2}, \ldots, \frac{\partial C}{\partial \theta_n}\right)$.

Although this derivation is for scalar functions, it will work with vector functions with a few modifications.

Deep Networks