Gradient Descent, Neural networks and Backpropagation

Enghin Omer

Abstract

1 Gradient Descent

There are often cases when we want to minimize a function C. One way of doing this is to compute the derivatives and try to find the extremum points of C. If C has a small number of variables this is a viable solution, but if C has a large number of variables, as it is often the case for machine learning algorithms, this is not a practical solution anymore.

Let's assume that our function looks like in the Figure 1 and a random point is chosen, the red point. Our goal is to reach the minimum, or the bottom, of that function. So, from our initial position, which direction shall we go to in order to reach the bottom of the valley? Intuitively, the direction is the one that points steepest down. If we go that direction a little bit, then look around and find again the direction that points steepest down and take again a little step into that direction and follow this procedure many times we'll eventually reach the bottom of the valley. But how can we find the direction of steepest descent?

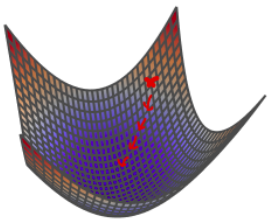

Figure 1: $f(x, y) = x^2 + y^2$

1.1 Directional derivative

To find our desired direction, the notion of directional derivative will be helpful. Let's assume that our function C has 2 variables x and y and we want to know the instantaneous rate of change of C when moving in the direction of $\vec{v} = \begin{bmatrix} v_1 \\ v_2 \end{bmatrix}$. The instantaneous rate of change of C along the x axis is $\frac{\partial C}{\partial x}$ and similarly for y, $\frac{\partial C}{\partial y}$. The vector $\vec{v}$ can be thought of as v_1 units to the x direction and v_2 units to the y direction. Therefore, the directional derivative of C along the $\vec{v}$ direction is $\nabla_{\vec{v}} C = \frac{\partial C}{\partial x} \cdot v_1 + \frac{\partial C}{\partial y} \cdot v_2$, or written more compactly:

$$\nabla_{\vec{v}} C = \nabla C \cdot \vec{v} \tag{1}$$

where $\nabla C = \begin{bmatrix} \frac{\partial C}{\partial x} \\ \frac{\partial C}{\partial y} \end{bmatrix}$. However, this is not the slope of C in the direction of $\vec{v}$. To get the slope the condition $\|\vec{v}\| = 1$ must be satisfied, since $\vec{v}$ can be scaled up and that will reflect in the value of $\nabla_{\vec{v}} C$.

Let's come back to our initial question: If we are at the point w=(a,b), what is $\vec{v}$ so that it points to the steepest descent direction? Well, the instantaneous rate of change of C when moving in the direction of $\vec{v}$ has to be negative and maximum in absolute terms, it is the steepest descent. So, the problem reduces to finding

$$-\max_{\|\vec{v}\|=1} \nabla C(a,b) \cdot \vec{v} = \min_{\|\vec{v}\|=1} \nabla C(a,b) \cdot \vec{v} \tag{2}$$

But $\nabla C(a,b) \cdot \vec{v}$ is the dot product between two vectors and is computed by multiplying the length of the projection of $\vec{v}$ onto $\nabla C(a,b)$ and the length of $\nabla C(a,b)$ (if the projection points to the opposite direction of $\nabla C(a,b)$ then its length has negative value). So, what should $\vec{v}$ be to minimize the product? If you swing $\vec{v}$ in all directions, it's easy to see that the projection of $\vec{v}$ is maximum when $\vec{v}$ points to the same direction as $\nabla C(a,b)$ and the product is minimum when $\vec{v}$ points in the opposite direction of $\nabla C(a,b)$. So, the direction of steepest descent is the opposite direction of $\nabla C(a,b)$. This is the $\vec{v}$ we were looking for!

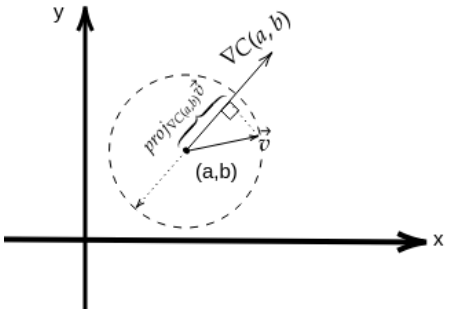

Figure 2: Steepest descent direction

All left to do to approach the minimum is to update until convergence our position w as follows:

$$w \leftarrow w - \eta \cdot \nabla C(w) \tag{3}$$

where η is called the ***learning rate***. This is the ***Gradient Descent*** algorithm. This works not only for two variate functions but for any number of variables.

To make gradient descent work correctly, η has to be carefully chosen. If η is too large the steps we take when updating t will be large and we might "miss" the minimum. On the other hand if η is too small the gradient descent algorithm will work slowly. A limitation of gradient descent algorithm is that it is susceptible to local minimum but many machine learning cost functions are convex and there's no local minimum.

1.2 Batch gradient descent

In machine learning to measure the accuracy of a model a **cost function** is used. One of those functions is the ***Mean Squared Error*** or ***MSE***:

$$C(w,b) = \frac{1}{2m} \sum_x \|\hat{y}(x) - y(x)\|^2 \tag{4}$$

In this equation, w and b denotes the set of all weights and biases in the model, m is the number of training observations, $\hat{y}(x)$ is the value outputted by the model and $y(x)$ is the true prediction of x. C is a sum of squared terms and we want to minimize the value of it, i.e., $C(w) \approx 0$.

To minimize the value of C(w) we can apply **gradient descent** algorithm. To do so, we can start with some initial values for w and then repeatedly update w until it hopefully converges to a value that minimizes C(w) as described above in eq(3). However, there's a drawback with this approach. At each iteration we have to compute the gradient ∇C. But to compute the ∇C we have to compute the gradient ∇C_x for each training example and then average them. When there is a large number of training observation the algorithm may be slow. To speed up things, another approach called *stochastic gradient descent* can be used.

1.3 Stochastic Gradient Descent

The idea of stochastic gradient descent is to use only one training observation at a time to update the parameters value. It works as follow:

1.Randomly 'shuffle' the dataset

2.For i=1...m

$(w,b) \leftarrow (w,b) - \eta \cdot \nabla C_{x^{(i)}}(w,b)$ where (w,b) is the vector containing all the values of weights and biases and

$$C_{x^{(i)}}(w,b) = \frac{1}{2}\|\hat{y}(x^{(i)}) - y(x^{(i)})\|^2 \tag{5}$$

This algorithm doesn't scan all the training observations to update the model parameters, but it tries to fit one training example at a time. Stochastic gradient descent will probably not converge at the global minimum but it can get close enough to be a good approximation.

Another approach is instead of using one training example, as in stochastic gradient descent, or all training examples, as in batch gradient descent, to use some in-between number of observation. This is called ***mini-batch gradient descent***.

Let's say we randomly chose k training examples, $x^{(1)}$ ... $x^{(k)}$. This a *mini-batch*. Having a large enough mini-batch, the average value of gradients in the mini-batch will approximate the one over the entire set of training examples, that is:

$$\frac{\sum_{i=1}^{k} \nabla C_{x^{(i)}}(w,b)}{k} \approx \frac{\sum_{i=1}^{m} \nabla C_{x^{(i)}}(w,b)}{m} = \nabla C(w,b) \tag{6}$$

The algorithm works by randomly picking a mini-batch and updating the parameters using them:

$$(w,b) \leftarrow (w,b) - \frac{\eta}{k} \cdot \sum_{i=1}^{k} \nabla C_{x^{(i)}}(w,b) \tag{7}$$

This is repeated until all the training examples are used, thus an *epoch* of training is completed. After that we can start with a new training epoch following the same procedure.

2 Neural Networks

A common approach for classification problems is to use the *sigmoid function*.

$$\sigma(z) = \frac{1}{1+e^{-z}} \tag{8}$$

It has weights for each input and an overall bias, b. A very nice property of this function is that $\sigma(w^T \cdot x + b)$ takes values in the interval (0,1), so when its output is greater than 0.5 we can classify the input as belonging to class 1.

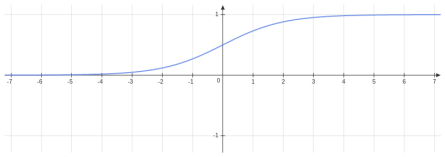

Figure 3: Sigmoid function

However, if we want to use three features including all the quadratic terms we have to compute $\sigma(w_1x_1^2 + w_2x_1x_2 + w_3x_1x_3 + w_4x_2^2 + w_5x_2x_3 + w_6x_3^2)$

which has six features. Here we take all two-element combinations of features. More generally, to compute the number of polynomial terms we can use the combination function with repetition: $\frac{(n+r-1)!}{r!(n-1)!}$. So, if our input is a 100x100 pixel black-and-white picture, the feature size will end up being $\frac{(100+2-1)!}{2!(100-1)!} = 5050$. Obviously, this is not practical and we need another approach.

Neural networks is another option when creating complex models with many features. The architecture of a neural network looks like following.

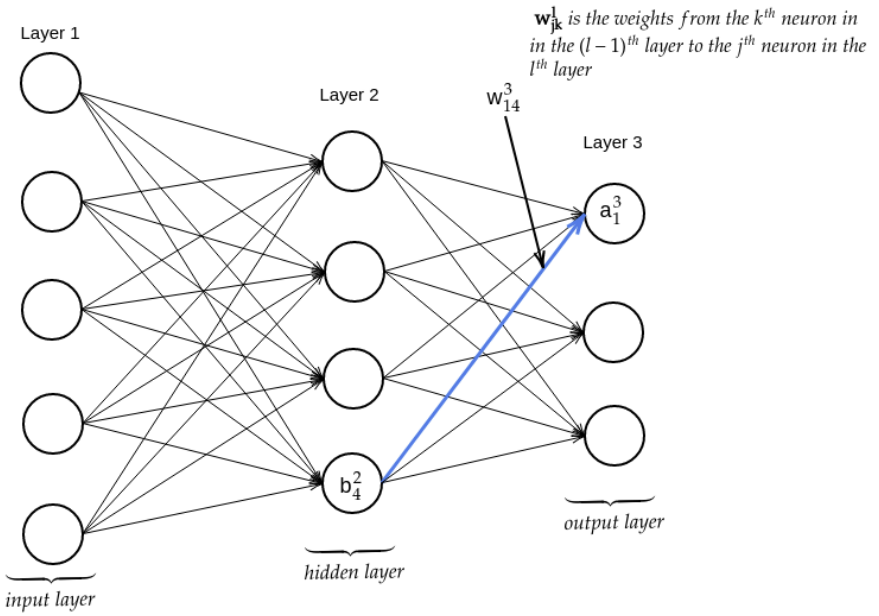

Figure 4: Neural network

A neural network has many layers each layer having a bunch of neurons. The first layer is called the input layer and the neurons within this layer are called *input neurons*. The last layer is called the output layer and the neurons *output neurons*. The layers in the middle are called *hidden layers*. The output of one layer is used as the input for the next layer. These networks are called *feedforward* neural networks.

The sigmoid neuron works as follows. Each input $x_1, x_2 ... x_n$ to the neuron has a corresponding weight $w_1, w_2 ... w_n$. The neuron has also an overall bias b. The output of the neuron is $\sigma(w^T \cdot x + b)$ and this is the activation of the neuron. This output value will serve as input for the next layer.

The weights are denoted w^l_{jk} meaning the weight from the k^{th} neuron in the layer *(l-1)* to the j^{th} neuron in the layer *l*. The b^l_j is the bias of the j^{th} neuron in the l^{th} layer. The activation of the j^{th} neuron in the l^{th} layer is a^l_j.

For example, in a problem of classifying pictures, the input layer may consist of all pixels grayscale taking values between 0.0 and 1.0. To compute

the activation value of the last neuron in layer 2 as in figure 4, we compute $a_4^2 = \sigma(x_1 w_{41}^2 + x_2 w_{42}^2 + x_3 w_{43}^2 + x_4 w_{44}^2 + x_5 w_{45}^2 + b_4^2)$. Then $a_1^3 = \sigma(a_1^2 w_{11}^3 + a_2^2 w_{12}^3 + a_3^2 w_{13}^3 + a_4^2 w_{14}^3 + b_1^3)$.

How can we train a neural network? We can use gradient descent to minimize the cost function. But to perform gradient descent we have to compute $\frac{\partial C}{\partial w}$ and $\frac{\partial C}{\partial b}$. Next, an algorithm that computes these partial derivatives is presented.

3 Backpropagation

3.1 Algorithm

Backpropagation is the algorithm used to compute the gradient of the cost function, that is the partial derivatives $\frac{\partial C}{\partial w_{jk}^l}$ and $\frac{\partial C}{\partial b_j^l}$. To define the cost function we can use eq(4) adapted to neural networks:

$$C(w,b) = \frac{1}{2m} \sum_x \|y(x) - a^L(x)\|^2 \tag{9}$$

where $a^L(x)$ is the vector of activation values for input x. We know that

$$a_j^l = \sum_k \sigma(a_k^{l-1} \cdot w_{jk}^l + b_j^l) \tag{10}$$

where the sum is over the k neurons in the $(l-1)^{th}$ layer.

We can define a *weight matrix* W^l which are the weights connecting to the l^{th} layer and the weight w_{jk}^l corresponds to the entry in W^l with row j and column k. Similarly, we can define a bias vector b^l containing all the biases in layer l, and the vector of activations:

$$a^l = \sigma(W^l a^{l-1} + b^l) \tag{11}$$

It is also useful to compute the intermediate value

$$z_j^l = \sum_k w_{jk}^l a_j^{l-1} + b_j^l \tag{12}$$

or in the vectorized form

$$z^l = w^l a^{l-1} + b^l \tag{13}$$

Obviously,

$$a^l = \sigma(z^l) \tag{14}$$

We also define the *error* of neuron j in layer l, as δ_j^l. We know from gradient descent algorithm that when the partial derivative is 0 or close to zero we approached the minimum otherwise the steeper the slope the more incorrect we are. With this consideration we can define

$$\delta_j^l = \frac{\partial C}{\partial z_j^l} \tag{15}$$

Being equipped with all these tools we can define the error in the output layer as follows:

$$\begin{aligned}\delta_j^L &= \frac{\partial C}{\partial z_j^L} \\ &= \frac{\partial C}{\partial a_j^L}\frac{\partial a_j^L}{\partial z_j^L} \\ &= \frac{\partial C}{\partial a_j^L}\sigma\prime(z_j^L)\end{aligned} \tag{16}$$

where $\sigma\prime(z_j^L) = \sigma(z_j^L)(1-\sigma(z_j^L))$ and if we are using the quadratic function defined in (9) for one example $\frac{\partial C}{\partial a_j^L} = (a_j^L - y_j)$. We can define $\nabla_a C$ the vector containing partial derivatives ∂a_j^L then we can write the vectorized form of (16):

$$\begin{aligned}\delta^L &= \nabla_a C \odot \sigma\prime(z^L) \\ &= (a_j^L - y_j) \odot \sigma\prime(z^L)\end{aligned} \tag{17}$$

Let's now try to compute the error of any neuron. In particular

$$\begin{aligned}\delta_j^l &= \frac{\partial C}{\partial z_j^l} \\ &= \sum_k \frac{\partial C}{\partial z_k^{l+1}}\frac{\partial z_k^{l+1}}{\partial z_k^l} \\ &= \sum_k \delta_k^{l+1}\frac{\partial z_k^{l+1}}{\partial z_k^l}\end{aligned} \tag{18}$$

but

$$\begin{aligned}z_k^{l+1} &= \sum_j w_{kj}^{l+1}a_j^l + b_k^{l+1} \\ &= \sum_j w_{kj}^{l+1}\sigma\prime(z_j^l) + b_k^{l+1}\end{aligned} \tag{19}$$

Therefore, we have

$$\frac{\partial z_k^{l+1}}{\partial z_k^l} = w_{kj}^{l+1}\sigma\prime(z_j^l) \tag{20}$$

substituting in (18) we get:

$$\delta_j^l = \sum_k w_{kj}^{l+1} \delta_k^{l+1} \sigma\prime(z_j^l) \tag{21}$$

or the vectorized form:

$$\delta^l = ((W^{l+1})^T \delta^{l+1}) \odot \sigma\prime(z^l) \tag{22}$$

So, using this formula, if we know δ^l we can compute δ^{l-1}, δ^{l-2} etc. That is we *backpropagate* the error, thus the name of the algorithm. We already saw in (17) how to compute δ^L.

Let's try now to compute $\frac{\partial C}{\partial b_j^l}$

$$\begin{aligned} \frac{\partial C}{\partial b_j^l} &= \frac{\partial C}{\partial z_j^l} \frac{\partial z_j^l}{\partial b_j^l} \\ &= \delta_j^l \frac{\partial((\sum_k w_{jk}^l a_j^{l-1}) + b_j^l)}{\partial b_j^l} \\ &= \delta_j^l \end{aligned} \tag{23}$$

The second equality results from using (15) and (12). Equation (23) shows that $\frac{\partial C}{\partial b_j^l} = \delta_j^l$. This is great news since we already saw how to compute δ_j^l for any l.

Similarly, we can compute

$$\begin{aligned} \frac{\partial C}{\partial w_{jk}^l} &= \frac{\partial C}{\partial z_j^l} \frac{\partial z_j^l}{\partial b_{jk}^l} \\ &= \delta_j^l a_k^{l-1} \end{aligned} \tag{24}$$

Equations (23) and (24) shows how to compute the gradient of the cost function.

3.2 Cross-entropy

In the previous section I described the backpropagation algorithm using the quadratic cost function (9). Another cost function used for classification problems is the ***Cross-entropy*** function.

$$C = -\frac{1}{m} \sum_x \sum_j [y_j \ln a_j^L + (1 - y_j) \ln(1 - a_j^L)] \tag{25}$$

Let's break this function into pieces and see why it makes sense. If y=1 then the second term cancels out and $-\ln a$ remains. If we look at the graph of this function we see that as a approaches 1 the value of the function approaches 0 and closer a is to 0 the function goes to infinity. That is the closer a is to the true value of y the smaller is the cost. Similarly for y=0. The remaining term is $-\ln(1-a)$. As a approaches 1 the value of the function approaches infinity and closer a is to 0 the function goes toward zero. The sum over j

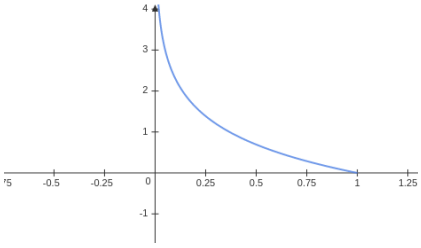

Figure 5: $-\ln a$

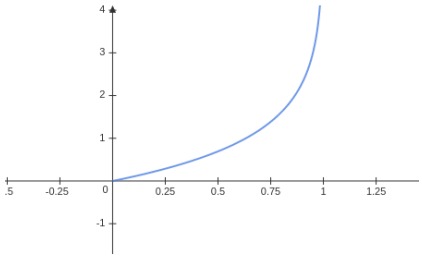

Figure 6: $-\ln(1-a)$

in the Cross-entropy function means the sum is over all neurons in the output layer.

In practice, the cross-entropy cost seems to be most often used instead of the quadratic cost function. We'll see in a moment why that's the case.

We presented earlier the backpropagation algorithm. How does it change for this new function? Equation (16) gave us a way of computing the error in output layer and we also saw how it would look like for the quadratic cost function. Let's now calculate it for the cross-entropy function.

$$\begin{aligned}
\delta_j^L &= \frac{\partial C}{\partial z_j^L} \\
&= \frac{\partial C}{\partial a_j^L}\frac{\partial a_j^L}{\partial z_j^L} \\
&= \frac{\partial C}{\partial a_j^L}\sigma\prime(z_j^L) \\
&= -(\frac{y}{\sigma(z_j^L)} - \frac{1-y}{1-\sigma(z_j^L)})\sigma\prime(z_j^L) \\
&= -(\frac{y-\sigma(z_j^L)y - \sigma(z_j^L) + \sigma(z_j^L)y}{\sigma(z_j^L)(1-\sigma(z_j^L))})\sigma\prime(z_j^L) \\
&= (\frac{\sigma(z_j^L)-y}{\sigma(z_j^L)(1-\sigma(z_j^L))})\sigma\prime(z_j^L) \\
&= (\frac{\sigma(z_j^L)-y}{\sigma\prime(z_j^L)})\sigma\prime(z_j^L) \\
&= \sigma(z_j^L) - y \\
&= a_j^L - y
\end{aligned} \tag{26}$$

It seems that the only difference with the quadratic function is that we got rid of the $\sigma\prime(z_j^L)$ term. How important is this? If we look at Fig(3) we can

notice that when the sigmoid function approaches 0 or 1 the graph is flattening, thus $\sigma\prime(x)$ is getting very close to 0. For example if an output layer neuron will output a value very close to 0 then $\sigma\prime(z_j^L)$ will be very close to 0 thus, δ_j^L as well. But if the true value is 1, the value of δ_j^L being close to 0 will make the partial derivatives of the corresponding weights very small and so the learning will be very slow and many iterations will be required. But when using the cross-entropy function the $\sigma\prime(z_j^L)$ disappears and the learning can be faster.

The way of computing $\frac{\partial C}{\partial w_{jk}^l}$ and $\frac{\partial C}{\partial b_j^l}$ remains the same as described in the previous section, the only difference being the way of computing δ_j^L.

3.3 Regularization

Regularization is a way of overcoming overfitting. There are many techniques of achieving regularization but here I'll describe only the *L2 regularization*. The idea of L2 regularization is to penalize large value of weights by adding a regularization term. Then, the regularized cross-entropy becomes:

$$\begin{aligned} C &= -\frac{1}{m}\sum_x\sum_j[y_j \ln a_j^L + (1-y_j)\ln(1-a_j^L)] + \frac{\alpha}{2m}\sum_w w^2 \\ &= C_0 + \frac{\alpha}{2m}\sum_w w^2 \end{aligned} \tag{27}$$

The biases are not included in the regularization term, so the partial derivatives with respect to the biases do not change and the gradient descent update rule doesn't change:

$$b = b - \eta\frac{\partial C}{\partial b} \tag{28}$$

The update rule for the weights becomes:

$$\begin{aligned} w &= w - \eta\frac{\partial C_0}{\partial w} - \frac{\eta\alpha}{m}w \\ &= (1-\frac{\eta\alpha}{m})w - \eta\frac{\partial C_0}{\partial w} \end{aligned} \tag{29}$$

As you can notice, the first term $1-\frac{\eta\alpha}{n}$ shrinks the weight. This is also called *weight decay*.