Learning in neural networks

We'll use the notation $\boldsymbol{x}$ to denote a training input. We'll denote the corresponding desired output by $y = y(\boldsymbol{x})$, where y can be a n-dimensional vector or a scalar.

We want an algorithm which lets us find a set of *weights* ($\boldsymbol{w}$) and *biases* ($\boldsymbol{b}$) so that the output from the network closely approximates $y(\boldsymbol{x})$ for all training inputs $\boldsymbol{x}$. To quantify how well we're achieving this goal we define a cost function (sometimes referred to as a loss or objective function):

$$C(\boldsymbol{w}, \boldsymbol{b}) = \frac{1}{2n} \sum_{x} (y(x) - \hat{y}(x))^2. \tag{1}$$

Here, $\boldsymbol{w}$ denotes the collection of all weights in the network, $\boldsymbol{b}$ all the biases, n is the total number of training inputs, $\hat{y}(x)$ is the vector of outputs from the network when x is input, and the sum is over all training inputs, x. We'll call C the *quadratic* cost function; also known as the *mean squared error* (MSE).

The $C(\boldsymbol{w}, \boldsymbol{b})$ is a non-negative function, since every term in the sum is non-negative. Furthermore, the cost C becomes small, i.e., $C(\boldsymbol{w}, \boldsymbol{b}) \approx 0$, when $y(x)$ is approximately equal to the network's output, $\hat{y}(x)$, for all training inputs, x. So the training algorithm has done a good job if it can find weights and biases so that $C(\boldsymbol{w}, \boldsymbol{b}) \approx 0$. So the aim of the training algorithm will be to minimize the cost as a function of the weights and biases. In other words, we want to find a set of weights and biases which make the cost as small as possible. This can be done using an algorithm known as *gradient descent*.

Gradient descent

let's think about what happens when we change the network parameters for a small amount $\Delta \boldsymbol{w} \equiv (\Delta w_1, ..., \Delta w_k)^T$ and $\Delta \boldsymbol{b} \equiv (\Delta b_1, ..., \Delta b_l)^T$. Calculus tells us that C changes as follows:

$$\Delta C \approx \frac{\partial C}{\partial w_1} \Delta w_1 + \cdots + \frac{\partial C}{\partial w_k} \Delta w_k + \frac{\partial C}{\partial b_1} \Delta b_1 + \cdots + \frac{\partial C}{\partial b_l} \Delta b_l \tag{2}$$

We'll also define the gradient of C to be the vector of partial derivatives, $(\frac{\partial C}{\partial w_1}, ..., \frac{\partial C}{\partial b_l})$.

$$\Delta C \approx \nabla C \cdot (\Delta \boldsymbol{w} + \Delta \boldsymbol{b}) \tag{3}$$

In this equation ∇C is called the gradient vector: ∇C relates changes in $\boldsymbol{w}$ an $\boldsymbol{b}$ to changes in C. The Equation 3 lets us see how to choose $\Delta \boldsymbol{w}$ and $\Delta \boldsymbol{b}$ so as to make ΔC negative. In particular, suppose we choose

$$\Delta \boldsymbol{w} + \Delta \boldsymbol{b} = -\eta \nabla C, \tag{4}$$

where η is a small positive number known as the *learning rate*. Then Equation 4 tells us that $\Delta C \approx -\eta (\nabla C)^2$, this guarantees that $\Delta C \leq 0$, i.e., C will always decrease if we change the parameters according to Equation 4. That is, we'll use Equation 4 to compute a value for $\Delta \boldsymbol{w}$ and $\Delta \boldsymbol{b}$, then change the $\boldsymbol{w}$ and $\boldsymbol{b}$ accordingly:

$$w_k \to w_k' = w_k - \eta \frac{\partial C}{\partial w_k} \tag{5}$$

$$b_l \to b_l' = b_l - \eta \frac{\partial C}{\partial b_l}. \tag{6}$$

We can think of these as update rules defining the gradient descent algorithm. It gives us a way of repeatedly changing $\boldsymbol{w}$ and $\boldsymbol{b}$ in order to find a minimum of the function C.
To make gradient descent work correctly, we need to choose the learning rate η to be small enough that Equation 3 is a good approximation. At the same time, we don't want η to be too small, since that will make the changes in network's parameters tiny, and thus the gradient descent algorithm will converge very slowly.

The way the gradient descent algorithm works is to repeatedly compute the gradient ∇C, and then to move in the opposite direction. We can visualize it like this:

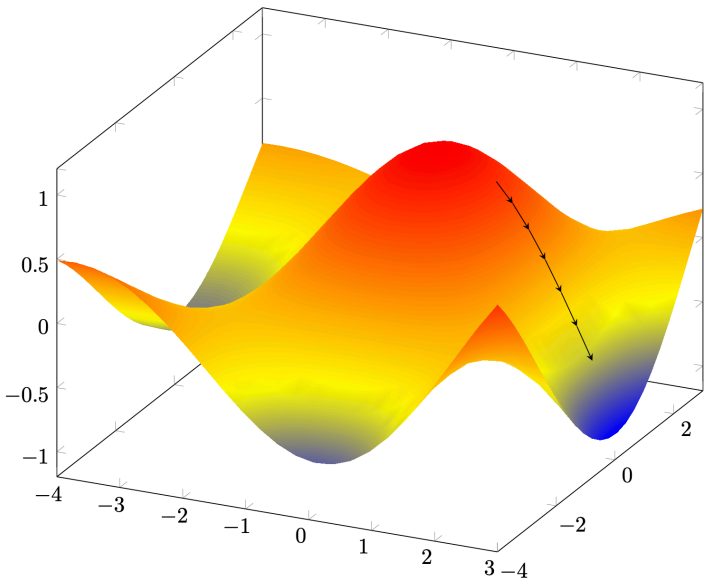

Figure 1: Representation of gradient descent for finding the minimum of a function of two variables.

The cost function in Equation 1 has the form $C = \frac{1}{n}\sum_x C_x$, that is, it's an average over costs C_x for individual training examples. In practice, to compute the gradient ∇C we need to compute the gradients ∇C_x separately for each training input, x, and then take the average, $\nabla C = \frac{1}{n}\sum_x \nabla C_x$. Unfortunately, when the number of training inputs is very large this can take a long time, and learning thus occurs slowly.

An idea called *stochastic gradient descent* (SGD) can be used to speed up learning. In SGD we estimate the gradient ∇C by computing ∇C_x for a small subset of m randomly chosen training inputs known as mini-batch.

$$\nabla C \approx \frac{1}{m}\sum_{j=1}^{m} \nabla C_{X_j}. \tag{7}$$

Stochastic gradient descent works by picking out a randomly chosen mini-batch of training inputs, and training with those,

$$w_k \to w_k' = w_k - \frac{\eta}{m}\sum_j \frac{\partial C_{X_j}}{\partial w_k} \tag{8}$$

$$b_l \to b_l' = b_l - \frac{\eta}{m} \sum_j \frac{\partial C_{X_j}}{\partial b_l}. \tag{9}$$

where the sums are over all the training examples X_j in the current mini-batch. Then we pick out another randomly chosen mini-batch and train with those. And so on, until we've exhausted the training inputs, which is said to complete an epoch of training. At that point we start over with a new training epoch.

The size of a mini-batch is typically much smaller than the entire dataset, leading to faster computations and increased scalability, particularly for large datasets. Furthermore, the random sampling of mini-batches in SGD introduces noise into the optimization process, which can help in escaping local minima and exploring the optimization landscape more thoroughly.

References

https://neuralnetworksanddeeplearning.com/chap1.html#learning_with_gradient_descent