Introduction to Neural Networks (2)

Gradient descent

We saw in the previous section that training a network corresponds to choosing the parameters, that is, the weights and biases, that minimize the cost function (see Fig. 1). The weights and biases take the form of matrices and vectors, but at this stage it is convenient to imagine them stored as a single vector that we call θ. Generally, we will suppose $\theta \in \mathbb{R}^p$, and write the cost function as $J(\theta)$ to emphasize its dependence on the parameters. So Cost $J : \mathbb{R}^p \to \mathbb{R}$.

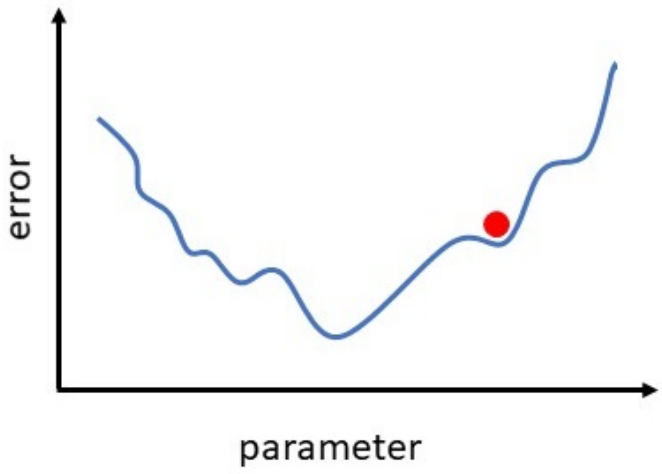


Figure 1: Error hypersurface

We now introduce a classical method in optimization that is often referred to as steepest descent or **gradient descent**. The method proceeds iteratively, computing a sequence of vectors in $\mathbb{R}^p$ with the aim of converging to a vector that minimizes the cost function. Suppose that our current vector is θ. How should we choose a perturbation, $\Delta\theta$, so that the next vector, $\theta + \Delta\theta$, represents an improvement? If $\Delta\theta$ is small, then ignoring terms of order $||\Delta\theta||^2$, a Taylor series expansion gives

$$J(\theta + \Delta\theta) \approx J(\theta) + \sum_{i=1}^{p} \frac{\partial J(\theta)}{\partial \theta_i} \Delta\theta_i$$

Here $\frac{\partial J(\theta)}{\partial \theta_i}$ denotes the partial derivative of the cost function with respect to the i-th weight. For convenience, we will let $\nabla J(\theta) \in \mathbb{R}^p$ denote the vector of partial derivatives, known as

the gradient, so that

$$\nabla J(\theta) = \big(\frac{\partial J(\theta)}{\partial \theta_1}, ..., \frac{\partial J(\theta)}{\partial \theta_p}\big)^\intercal \tag{1}$$

Then,

$$J(\theta + \Delta\theta) \approx J(\theta) + \nabla J(\theta)^\intercal \Delta\theta \tag{2}$$

Our aim is to reduce the value of the cost function. The relation (2) motivates the idea of choosing $\Delta\theta$ to make $\nabla J(\theta)^\intercal \Delta\theta$ as negative as possible. We can address this problem via the Cauchy-Schwarz inequality, which states that for any $f, g \in \mathbb{R}^p$, we have $|f^\intercal g| \leq ||f|| \cdot ||g||$. Moreover, the two sides are equal if and only if f and g are linearly dependent (meaning they are parallel).

So the most negative that $f^\intercal g$ can be is $-||f|| \cdot ||g||$, which happens when $f = -g$. Hence we should choose $\Delta\theta$ to lie in the direction of $-\nabla J(\theta)$. Keeping in mind that (2) is an approximation that is relevant only for small $\Delta\theta$, we will limit ourselves to a small step in that direction. This leads to the update

$$\theta \to \theta - \eta \nabla J(\theta) \tag{3}$$

Here η is small stepsize that, in this context, is known as the **learning rate**. This equation defines the steepest descent method. We choose an initial vector and iterate (3) until some stopping criterion has been met, or until the number of iterations has exceeded the computational budget.

Repeat:

$$\theta_j = \theta_j - \eta \frac{\partial}{\partial \theta_j} J(\theta)$$

$$\text{simultaneously update all} \qquad \theta_j$$

$\eta \in (0, 1]$ denotes the learning parameter.

In summary, we aim to minimize the cost function

$$\min_\theta \; J(\theta)$$

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

$$\frac{\partial}{\partial \theta_j} J(\theta)$$

Initialization

The input data have to be normalized to have approximately the same range. The biases can be initialized to 0. The weights cannot be initialized with the same values, otherwise, all the neurons of a hidden layer would have the same behavior. Perhaps the only property known with complete certainty is that the initial parameters need to break symmetry between different units. We generally initialize the weights at random: the values $\theta_{ij}^{(l)}$ are i.i.d. Uniform on $[-c, c]$ with possibly $c = 1/\sqrt{N_l}$ where N_l is the size of the hidden layer l. We also sometimes initialize the weights with a normal distribution $N(0, 0.01)$.

Stochastic Gradient

Algorithm for optimization the cost function. When we have a large number of parameters and a large number of training points, computing the gradient vector (1) at every iteration of the steepest descent method (3) can be prohibitively expensive because we have to sum across all training points (for instance in Big Data). A much cheaper alternative is to replace the mean of the individual gradients over all training points by the gradient at a single, randomly chosen, training point. This leads to the simplest form of what is called the stochastic gradient method. A single step may be summarized as

- Choose an integer i uniformly at random from $\{1, ..., n\}$
- update

$$\theta_j = \theta_j - \eta \frac{\partial}{\partial \theta_j} J(\theta; x^{(i)}) \tag{4}$$

Notice we have included $x^{(i)}$ in the notation of $J(\theta; x^{(i)})$ to remark the dependence. In words, at each step, the stochastic gradient method uses one randomly chosen training point to represent the full training set. As the iteration proceeds, the method sees more training points. So there is some hope that this dramatic reduction in cost-per-iteration will be worthwhile overall. We note that, even for very small η, the update (4) is not guaranteed to reduce the overall cost function we have traded the mean for a single sample. Hence, although the phrase stochastic gradient descent is widely used, we prefer to use **stochastic gradient**.

The version of the stochastic gradient method that we introduced in (4) is the simplest from a large range of possibilities. In particular, the index i in (4) was chosen by sampling with replacement after using a training point, it is returned to the training set and is just as likely as any other point to be chosen at the next step. An alternative is to sample without replacement; that is, to cycle through each of the n training points in a random order. Performing n steps in this manner, refered to as completing an epoch, may be summarized as follows:

- Shuffle the integers $\{1, ..., n\}$ into a new order $\{k_1, ..., k_n\}$
- for i in $1 : n$ update $\theta_j = \theta_j - \eta \frac{\partial}{\partial \theta_j} J(\theta; x^{(k_i)})$

If we regard the stochastic gradient method as approximating the mean over all training points by a single sample, then it is natural to consider a compromise where we use a small

sample average. For some $m << n$ we could take steps of the following form.

- Choose m integers $\{k_1, ..., k_m\}$ uniformly at random from $\{1, ..., n\}$
- update $\theta_j = \theta_j - \eta \frac{1}{m} \sum_{i=1}^{m} \frac{\partial}{\partial \theta_j} J(\theta; x^{(k_i)})$

In this iteration, the set $\{x^{(k_i)}\}_{i=1}^{m}$ is known as a **mini-batch**. Because the stochastic gradient method is usually implemented within the context of a very large scale computation, algorithmic choices such as mini-batch size and the form of randomization are often driven by the requirements of high performance computing architectures. Also, it is, of course, possible to vary these choices, along with others, such as the learning rate, dynamically as the training progresses in an attempt to accelerate convergence.

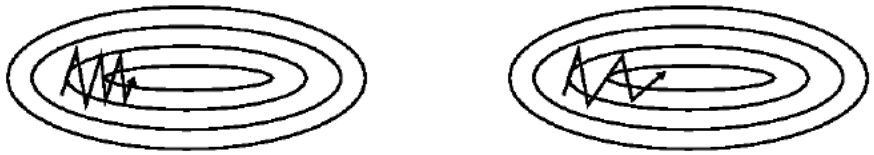
Optimizers

There are a multitude of "tricks of the trade" in fitting or "learning" a neural network, and many of them are connected with gradient descent. Since the choice of the learning rate is delicate and very influent on the convergence of the SGD algorithm, variations of the algorithm have been proposed. They are less sensitive to the learning rate.

- Vanilla Gradient Descent.
- Stochastic Gradient Descent.
    - SGD
    - Tendency to oscillate especially on steep error surfaces. Noisy or small gradients may also be problematic.
- Momentum based Learning Algorithms
    - Take a running average by incorporating the previous update in the current change as if there is a momentum due to previous updates.
    - New hyperparameter that determines how quickly the contributions of previous gradients exponentially decay.
    - SGD with momentum.
- Adaptive Gradient based Learning Algorithms
    - Optimization algorithms that individually adapts the learning rates of model parameters. The parameters with the largest partial derivative of the loss have a rapid decrease in their learning rate, while parameters with small partial derivatives have a relatively small decrease in their learning rate
    - Adagrad, Rmsprop
- Momentum and Adaptive Gradient based Learning Algorithms
    - Adam

Momentum

SGD has trouble navigating ravines, i.e. areas where the surface curves much more steeply in one dimension than in another, which are common around local optima. In these scenarios, SGD oscillates across the slopes of the ravine while only making hesitant progress along the bottom towards the local optimum as in Figure 2a



(a) SGD without momentum

(b) SGD with momentum

Figure 2: SGD with momentum

Momentum is a method that helps accelerate SGD in the relevant direction and dampens oscillations as can be seen in Figure 2b. It does this by adding a fraction of the update vector of the past time step to the current update vector

$$\begin{aligned} v_t &= \gamma v_{t-1} + \eta \nabla J(\theta) \\ \theta &= \theta - v_t \end{aligned}$$

The momentum term γ is usually set to 0.9 or a similar value.

Essentially, when using momentum, we push a ball down a hill. The ball accumulates momentum as it rolls downhill, becoming faster and faster on the way (until it reaches its terminal velocity, if there is air resistance, i.e. $\gamma < 1$). The same thing happens to our parameter updates: The momentum term increases for dimensions whose gradients point in the same directions and reduces updates for dimensions whose gradients change directions. As a result, we gain faster convergence and reduced oscillation.

Adaptative gradient (Adagrad)

Adagrad is an algorithm for gradient-based optimization that does just this: It adapts the learning rate to the parameters, performing larger updates for infrequent and smaller updates for frequent parameters. For this reason, it is well-suited for dealing with sparse data (Adagrad was used to train GloVe word embeddings, as infrequent words require much larger updates than frequent ones.)

Previously, we performed an update for all parameters θ at once as every parameter θ_i used the same learning rate η. As Adagrad uses a different learning rate for every parameter θ_i at every step t, we first show Adagrad's per-parameter update, which we then vectorize. For

brevity, we set $g_{t,i}$ to be the gradient of the objective function w.r.t. to the parameter θ_i at iteration step t:

$$g_{t,i} = \frac{\partial}{\partial \theta_i} J(\theta^t)$$

The SGD update for every parameter θ_i at each iteration step t then becomes:

$$\theta_{t+1,i} = \theta_{t,i} - \eta g_{t,i}$$

In its update rule, Adagrad modifies the general learning rate η at each step t for every parameter θ_i based on the past gradients that have been computed for θ_i:

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} g_{t,i}$$

$G_{t,ii}$ is a $\mathbb{R}^{p \times p}$ here is a diagonal matrix where each diagonal element i; i is the sum of the squares of the gradients w.r.t. θ_i up to step t, while ϵ is a smoothing term that avoids division by zero (usually on the order of 1e-8). Interestingly, without the square root operation, the algorithm performs much worse.

As G_t contains the sum of the squares of the past gradients w.r.t. to all parameters θ along its diagonal, we can now vectorize our implementation by performing an element-wise matrix-vector multiplication $\odot$ between G_t and g_t:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t$$

One of Adagrad's main benefits is that it eliminates the need to manually tune the learning rate. Most implementations use a default value of 0.01 and leave it at that. Adagrad's main weakness is its accumulation of the squared gradients in the denominator: Since every added term is positive, the accumulated sum keeps growing during training. This in turn causes the learning rate to shrink and eventually become infinitesimally small, at which point the algorithm is no longer able to acquire additional knowledge. RMSprop, Adam algorithms aim to resolve this flaw.

References

Aggarwal, Charu C. Neural networks and deep learning. Berlin, Germany. Springer, 2018.