DL in Applied Mathematics

Lecture 4: Stochastic Gradient Descent (SGD)

SGD is a popular optimization algorithm used in machine learning and deep learning for minimizing a loss function, which measures the difference between the observed and predicted values.

Unlike traditional Gradient Descent, which computes the gradient of the loss function using the entire dataset to update model parameters in each iteration, SGD updates the parameters using only a single sample (or a small batch of samples) at each iteration. This approach makes SGD much faster and more scalable to large datasets.

Basic Algorithm

- **Initialize the parameters:** Start with initial values for the parameters of the model you are trying to optimize. These could be coefficients in linear regression or weights in a neural network.
- **Loop until convergence:** Repeat the following steps until the parameters converge to their optimal values or until a specified number of iterations is reached:
- a. **Select a sample:** Randomly pick a single data point (or a small batch of data points) from the training dataset.
- b. **Compute the gradient:** Calculate the gradient of the loss function with respect to each parameter, but only for the selected sample(s). The gradient is a vector that points in the direction of the steepest increase of the loss function.
- c. **Update the parameters:** Adjust the parameters in the opposite direction of the gradient by a small step. The size of the step is determined by the learning rate, a hyperparameter that controls how much we adjust the parameters in response to the estimated error each time the parameters are updated.

- **Mathematical Formulation**

Given a loss function *L* and a learning rate *η*, the parameter update rule for SGD can be expressed as:

$$\theta = \theta - \eta \nabla_{\theta} L\left(\theta ; x^{(i)}, y^{(i)}\right)$$

where *ϑ* represents the parameters of the model,

(*x*(*i*),*y*(*i*)) is a randomly selected sample from the dataset,

∇*ϑL*(*ϑ*;*x*(*i*),*y*(*i*)) is the gradient of the loss function with respect to the parameters for that sample.

Advantages and Disadvantages

Advantages:

- **Efficiency:** SGD is computationally much faster than full-batch gradient descent, especially for large datasets.

- **Online Learning:** It can be used for online learning, where the model needs to be updated as new data arrives.

- **Escape Local Minima:** The inherent noise in the gradient estimation due to sampling can help the algorithm escape local minima.

Disadvantages:

- **Variance:** The stochastic nature of the algorithm can lead to high variance in the parameter updates, causing the objective function to fluctuate heavily.
- **Hyperparameter Tuning:** The learning rate and batch size are critical hyperparameters that need careful tuning.
- **Convergence:** The algorithm may not converge to the exact minimum but will oscillate in a region around the minimum.

Modifications and Improvements

Several variants and improvements of SGD aim to reduce its variance and improve convergence rates, including:

- **Momentum:** Adds a fraction of the previous update to the current update to smooth out the variations.
- **Adagrad, RMSprop, Adam:** These algorithms adapt the learning rate during training to adjust the amount of update for each parameter.
- SGD and its variants are foundational algorithms in the field of machine learning and are crucial for training various models, from simple linear regressors to complex neural networks.

Stochastic Gradient

In previous lectures, we saw that training a network corresponds to choosing the parameters, that is, the weights and biases, that minimize the cost function.

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 ρ

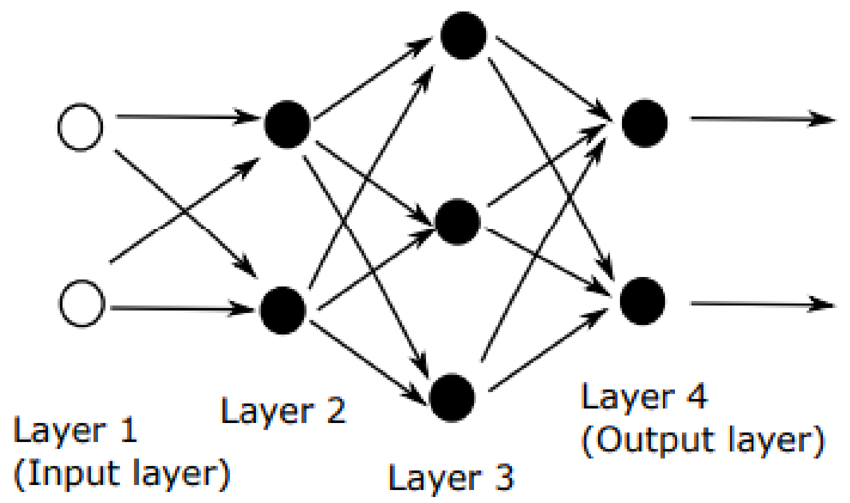

Figure 3: A network with four layers.

The example in Figure 3 has a total of 23 weights and biases. So, in that case,$p \in \mathbb{R}^{23}$.

$$\text{Cost} = \frac{1}{N}\sum_{i=1}^{N}\tfrac{1}{2}\|y(x^{\{i\}}) - a^{[L]}(x^{\{i\}})\|_2^2, \qquad (9)$$

Generally, we will suppose $p \in \mathbb{R}^s$, and write the cost function in (9) as Cost(p) to emphasize its dependence on the parameters. So Cost : $\mathbb{R}^s \rightarrow \mathbb{R}$

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 $\mathbb{R}^s$ with the aim of converging to a vector that minimizes the cost function. Suppose that our current vector is p.

How should we choose a perturbation, Δp, so that the next vector, $p + \Delta p$, represents an improvement?

If Δp is small, then ignoring terms of order $\left\|\Delta_p\right\|^2$, a Taylor series expansion gives

$$\text{Cost}(p + \Delta p) \approx \text{Cost}(p) + \sum_{r=1}^{s} \frac{\partial\, \text{Cost}(p)}{\partial\, p_r} \Delta p_r. \qquad (10)$$

Here $\partial\ Cost\ (p)/\partial\ p_r$ denotes the partial derivative of the cost function with respect to the rth parameter.

For convenience, we will let $\nabla\, Cost(p) \in \mathbb{R}^s$ denote the vector of partial derivatives, known as the *gradient*, so that

$$(\nabla \mathrm{Cost}(p))_r = \frac{\partial\, \mathrm{Cost}(p)}{\partial\, p_r}.$$

Then (10) becomes

$$\mathrm{Cost}(p + \Delta p) \approx \mathrm{Cost}(p) + \nabla \mathrm{Cost}(p)^T \Delta p. \qquad (11)$$

Our aim is to reduce the value of the cost function. The relation (11) motivates the idea of choosing Δp to make $\nabla\mathrm{Cost}(p)^T \Delta_p$ as negative as possible.

We can address this problem via the Cauchy–Schwarz inequality, which states that for any $f, g \in \mathbb{R}^S$, we have $|f^T g| \le \| f \|_2 \| g \|_2$, which happens when $f = -g$.

Hence, based on (11), we should choose Δp to lie in the direction $-\nabla$ Cost(p) . Keeping in mind that (11) is an approximation that is relevant only for small Δp , we will limit ourselves to a small step in that direction.

This leads to the update

$$p \to p - \eta \nabla \mathrm{Cost}(p). \tag{12}$$

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 with (12) until some stopping criterion has been met, or until the number of iterations has exceeded the computational budget.

Our cost function (9) involves a sum of individual terms that runs over the training data. It follows that the partial derivative $\nabla\text{Cost}(p)$ is a sum over the training data of individual partial derivatives. More precisely, let

$$C_{x^{\{i\}}} = \frac{1}{2}\|y(x^{\{i\}}) - a^{[L]}(x^{\{i\}})\|_2^2. \tag{13}$$

Then, from (9),

$$\nabla\text{Cost}(p) = \frac{1}{N}\sum_{i=1}^{N} \nabla C_{x^{\{i\}}}(p). \tag{14}$$

When we have a large number of parameters and a large number of training points, computing the gradient vector (14) at every iteration of the steepest descent method (12) can be prohibitively expensive. 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

1. Choose an integer i uniformly at random from {1, 2, 3, . . . , *N*}.
2. Update

$$p \to p - \eta \nabla C_{x^{\{i\}}}(p). \qquad (15)$$

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 (15) 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 (15) is the simplest from a large range of possibilities. In particular, the index i in (15) 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:

1. Shuffle the integers {1, 2, 3, . . . , *N*} into a new order, {k1, k2, k3, . . . , kN }.
2. for i = 1 upto *N*, update

$$p \to p - \eta \nabla C_{x^{\{k_i\}}}(p). \tag{16}$$

If we regard the stochastic gradient method as approximating the mean over all training points in (14) by a single sample, then it is natural to consider a compromise where we use a small sample average. For some $m \ll N$ we could take steps of the following form.

1. Choose m integers, $k_1, k_2, \ldots, k_m$, uniformly at random from {1, 2, 3, . . . , N}.
2. Update

$$p \to p - \eta \frac{1}{m} \sum_{i=1}^{m} \nabla C_{x^{\{k_i\}}}(p). \tag{17}$$

In this iteration, the set $\left\{x^{\{k_i\}}\right\}_{\mathrm{i=1}}^{n}$ is known as a *mini-batch*. There is a *without replacement* alternative where, assuming $N = k_m$ for some K , we split the training set randomly into K distinct mini-batches and cycle through them.

Because the stochastic gradient method is usually implemented within the context of a very large scale computation, algorithmic choices such as minibatch 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.

We are now in a position to apply the stochastic gradient method in order to train an artificial neural network.

Thank you for attention!