Notes

Imperial College London

Department of Computing

Machine Learning: Deep Learning II

Author:
W (CID: your college-id number)

Date: January 6, 2020

1 Introduction

From a high level perspective, training a neural network consists of the follow steps:

1. Load data
2. Define function for prediction (Forward Propagation)
3. Define loss function
4. Define training function to minimize loss (Back Propagation)

This lecture focuses on Back Propagation and look into modern methods to speed up the back propagation process.

2 Batch Gradient Descent

There are different ways to do gradient descent:

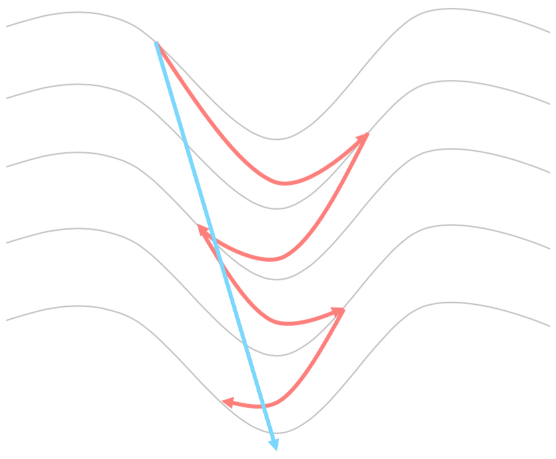
- Full gradient descent: Perform gradient descent on full data set
- Batch gradient descent: Perform gradient descent on batch of data
- Stochastic gradient descent: Perform gradient descent using one data at a time

Full gradient descent does not work when the data set becomes too large. On the other hand stochastic gradient descent is takes too long to reduce the cost function. The medium of the two is **batch gradient descent**.

In batch gradient descent, in each epoch we split the data set into N/B batches of size B. Then for each batch, we find the gradient and update the weights and bias.

3 Momentum

In batch gradient descent, we make small steps in the direction of the gradient until it reaches a local minimum. However, gradient descent struggles in **pathological curvatures** (often described as valleys, trenches or ravines).



Since the surface at the ridge curves is much steeper, the parameters will zigzag back and forth across the valley (red line). This oscillation makes convergence very slow, we want the algorithm to take more straight forwards path towards local minimum (blue line).

Momentum takes advantage of the knowledge accumulated from past gradients to determine the direction to go. It computes the **exponential moving average (EMA)** of the gradients, and use it to update the weights.

In normal gradient descent, weights are updated by

$$W_t \leftarrow W_{t-1} - \alpha \frac{\partial J}{\partial W_{t-1}}$$

In momentum, instead of using the derivatives directly, we take the exponentially weighted averages of the derivatives

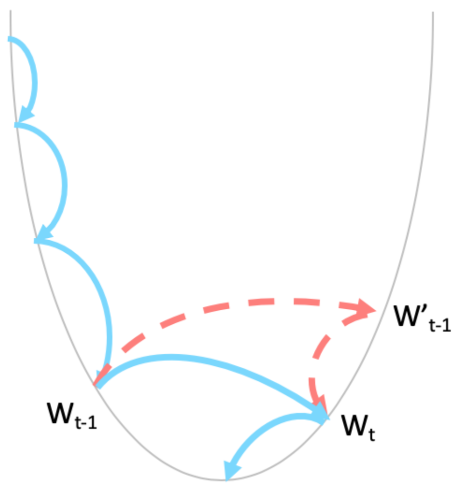
$$\begin{aligned} V_t &\leftarrow \beta V_{t-1} + (1-\beta)\alpha \frac{\partial J}{\partial W_{t-1}} \\ W_t &\leftarrow W_{t-1} - V_t \end{aligned}$$

V_0 is initialized to 0, in each iteration it accumulates the new calculated gradient. β is the hyperparameter momentum, which is a value between 0 to 1 (usually 0.9).

3.1 Nesterov Momentum

Nesterov Momentum is a slightly different version of the momentum update and can be summarized in two steps

1. Make a jump to the point where the current momentum is pointing to
2. Compute the gradient at that point and make correction



First we calculate the look-ahead value

$$W'_{t-1} = W_{t-1} - \beta V_{t-1}$$

We calculate the gradient at this point and use it to update V. Then use this V to update the weights

$$\begin{aligned} V_t &\leftarrow \beta V_{t-1} + \alpha \frac{\partial J}{\partial W'_{t-1}} \\ W_t &\leftarrow W_{t-1} - V_t \end{aligned}$$

We can reformulate the equations and express everything in terms of W'

$$\begin{aligned} W_t &\leftarrow W_{t-1} - V_t \\ W'_t + \beta V_t &\leftarrow W'_{t-1} + \beta V_{t-1} - V_t \\ W'_t &\leftarrow W'_{t-1} + \beta V_{t-1} - (1+\beta) V_t \\ W'_t &\leftarrow W'_{t-1} + V_t - \alpha \frac{\partial J}{\partial W'_{t-1}} - (1+\beta) V_t \\ W'_t &\leftarrow W'_{t-1} - \beta V_t - \alpha \frac{\partial J}{\partial W'_{t-1}} \end{aligned}$$

We can now express Nesterov momentum update entirely with look-ahead W'. Since the non look-ahead values are no longer needed, we can treat W as if they are the look-ahead values, and update the weights as follow

$$\begin{aligned} V_t &\leftarrow \beta V_{t-1} + \alpha \frac{\partial J}{\partial W_{t-1}} \\ W_t &\leftarrow W_{t-1} - \beta V_t - \alpha \frac{\partial J}{\partial W_{t-1}} \end{aligned}$$

4 Learning Rate Techniques

In standard gradient descent, the learning rate α is held constant throughout training. If the learning rate is too low, the algorithm takes too long to converge. If the learning rate is too high, the algorithm will oscillate and becomes unstable.

We want to have high learning rate initially to make big steps towards the goal, but a small learning rate for fine tuning when it is near the optimum.

4.1 Variable Learning Rate

Using variable learning rates, the learning rate decreases with time or iterations.

- Step Decay: Reduce the learning rate by some factor k every few epochs.
- Exponential Decay: $\alpha = \alpha_0 e^{-kt}$
- $1/t$ Decay: $\alpha = \frac{\alpha_0}{(1+kt)}$

In the above equations, k is a hyperparameter and t is the number of iterations.

4.2 Adaptive Learning Rate

The dependence of the cost on each parameter is not the same, some parameters affect the cost more than others. Adaptive learning rates updates each individual parameter to perform larger or smaller updates depending on their importance.

4.2.1 AdaGrad

AdaGrad introduces a new variable *cache*, and each parameter has its own *cache*. If W is a $M \times N$ matrix, then we create a $M \times N$ *cache* for W.

$$cache \leftarrow cache + (\nabla_W J)^2$$
$$W \leftarrow W - \alpha \frac{\nabla_W J}{\sqrt{cache + \epsilon}}$$

Where $\nabla_W J$ is the derivative of loss w.r.t W and $\epsilon \approx 10^{-9}$ to prevent division by 0. The *cache* is accumulating the square of gradients and so it is always positive.

If a parameter has accumulated large gradients, the cache will be large and will have smaller learning rate.

If a parameter has accumulated small gradients, the cache will be small and will have higher learning rate.

4.2.2 RMSProps

AdaGrad decreases the learning rate too aggressively. RMSProps introduces a hyperparameter $\beta \approx 0.99$ to slow down the increase of *cache*, hence slow down the decrease of the learning rate.

$$cache \leftarrow \beta \times cache + (1 - \beta)(\nabla_W J)^2$$
$$W \leftarrow W - \alpha \frac{\nabla_W J}{\sqrt{cache + \epsilon}}$$

There is no standard for the initial value of *cache*. Some libraries initialize it to 0, others initialize it to 1.

4.2.3 Adam Optimization

Adam is a new modern adaptive learning technique that combines the benefit of AdaGrad and RMSProp.

Adam calculates an exponential moving average of the gradient and the squared gradient. $\beta_1 \approx 0.9$ and $\beta_2 \approx 0.99$ controls the decay rate of these moving averages.

$$cache_{mean} \leftarrow \beta_1 \times cache_{mean} + (1 - \beta_1)(\nabla_W J)$$
$$cache_{variance} \leftarrow \beta_2 \times cache_{variance} + (1 - \beta_2)(\nabla_W J)^2$$

Since $cache_{mean}$ and $cache_{variance}$ are initially set to 0, Adam uses bias-correction estimates to overcome the problem with biased towards 0. At iteration t, we update the *cache* as above, and correct these estimates by

$$\hat{cache}_{mean} = \frac{cache_{mean}}{(\beta_1)^t}$$
$$\hat{cache}_{variance} = \frac{cache_{variance}}{(\beta_2)^t}$$

Finally we update the weights using the bias corrected *cache*.

$$W \leftarrow W - \alpha \frac{\hat{cache}_{mean}}{\sqrt{\hat{cache}_{variance} + \epsilon}}$$

5 Tuning Hyperparameters

Hyperparamters are settings that can be tuned to control the behaviour of a machine learning algorithm. Examples of hyperparameters include

- Learning rate / Decay Rate
- Momentum
- Regularization
- Hidden layer size
- Number of hidden layers

In practice we can use **Cross Validation** to find the best configuration. First we split the data set into K folds, then for each parameter set we train the model K times, each time leaving a different fold out of the training set and using it as a validation set. At the end, we calculate a score averaged across all K tests. Finally, we choose the set of parameters with the highest average score.

5.1 Grid Search

Grid search is an exhaustive search. For each hyperparameter, we define a set of values we want to try then try every possible combination using cross validation. For example:

$$\alpha = [0.1, 0.01, 0.001, 0.0001]$$
$$\beta = [1, 0.1, 0.01, 0.0001]$$
$$\lambda = [1, 0.1, 0.01]$$

Then the parameter sets are

$$(\alpha = 0.1, \beta = 1, \lambda = 1), (\alpha = 0.1, \beta = 1, \lambda = 0.1), (\alpha = 0.1, \beta = 1, \lambda = 0.01),$$
$$(\alpha = 0.1, \beta = 0.1, \lambda = 1), (\alpha = 0.1, \beta = 0.1, \lambda = 0.1)$$
$$\ldots\ldots$$

5.2 Random Search

In random search, we move randomly in the hyperparameter space until the score improves. First we define a set of initial parameters and the maximum number of iterations. In each iteration, we choose a new hyperparameter set by adding or subtracting a certain amount to the current hyperparameter. If the new hyperparameter set performs better, we move to this new setting and continue the search.

In general we want to generate random numbers in a **logarithmic scale**. For hyperparemters like learning rate we want to try values such as 0.1, 0.01, 0.001. and for hyperparameters such as decay rate we want to try values such as 0.9, 0.99, 0.999.

If we use uniform sampling $U(0, 0.1)$ we get many numbers between 0.01 and 0.1, while the other numbers are under represented.

To solve the problem, we can sample uniformly $X \sim U(-7, -1)$ and define the learning rate as 10^x. This way we can get an even distribution for every 10^{th} power.

Similarly, we can sample uniformly $X \sim U(-7, -1)$ and define decay rate as $1 - 10^x$

6 Weight Initialization

The basic form of neural is a composite function

$$y = f \circ g \circ h \circ ... \circ (x)$$

To find the derivative of y w.r.t w at a layer requires chain rule

$$\frac{\partial y}{\partial w} = \frac{\partial y}{\partial f}\frac{\partial f}{\partial g}\frac{\partial g}{\partial h}...$$

Vanishing Gradient Problem arises from sigmoid activation functions. When we look at the derivative of sigmoid, all outputs are between $0 < z \leq 0.25$. Multiplying many of these derivatives approaches zero, prohibiting the weights from updating when training a deep neural network.

Exploding Gradient Problem happens when the derivatives are too large, multiplying all these derivatives approaches infinity, as a result all weights will be infinity and the neural network cannot predict results.

Therefore weights cannot be too small or too big during initialization.

Bias can be initialized to 0.

6.1 Initializing Weights for Logistic Regression

We do not want to initialize the weights to constants because then each hidden unit represents the same feature, this is as if there is only one neuron in each hidden layer.

In the simplest case logistic regression

$$\begin{aligned} z &= w_1x_1 + w_2x_2 + \ldots + w_Dx_D \\ var(z) &= var(w_1)var(x_1) + var(w_2)var(x_2) + \ldots + var(w_D)var(x_D) \\ var(z) &= var(w_1) + var(w_2) + \ldots + var(w_D) \\ var(z) &= D \times var(w) \end{aligned}$$

Notice that $var(X)$ is 1 since we normalized the input data. To get output with variance of 1

$$var(w) = \frac{1}{D}$$

To achieve this we can first initialize the weights randomly using standard normal distribution, then multiply each value by $1/\sqrt{D}$.

6.2 Initializing Weights for a Layer

Given a neural network layer with M inputs and N outputs, we can initialize the weights using the following methods.

- Create random weights with standard normal distribution then scale each weight by 0.01
- Create random weights with standard normal distribution then scale each weight by $\sqrt{2/(M+N)}$. (tanh)
- Create random weights with standard normal distribution then scale each weight by $\sqrt{1/M}$. (tanh)
- Create random weights with standard normal distribution then scale each weight by $\sqrt{2/M}$. (ReLU)
- Glorot Uniform: Select weights from uniform distribution $U\left(-\sqrt{6/(M+N)}, \sqrt{6/(M+N)}\right)$
- LeCun Uniform: Select weights from uniform distribution $U\left(-\sqrt{3/M}, \sqrt{3/M}\right)$

7 Regularization

7.1 Dropout

In **ensemble learning**, we train multiple prediction models and then make predictions by taking the average prediction from all the models. This often leads to better accuracy than we just used one model.

Dropout regularization emulates ensemble learning by ignoring random nodes for each iteration during training. Given a neural network with N nodes, and each node can have two states: keep or drop. This is like training 2^N neural network with different configurations.

Dropout prevents over fitting since it forces the neural network to not rely on a single feature.

7.1.1 Implementing Dropout

For each layer l, we define p_l as the probability of keeping a node during training. p is usually 0.8 for the input layer and 0.5 for hidden layers.

During training, in each iteration the output of a node in layer l is multiplied by 0 with probability p_l.

During prediction, all the nodes are used. However we have to multiply the output of all the nodes by their respective p_l. This is to compensate for the missing activation during training.

7.2 Noise Injection

One way to prevent over fitting is to add noise during training. The most common method is to add **Gaussian noise** to the input variables, but is it also valid to add noise to the weights.

Gaussian noise has a normal distribution with 0 mean and small variance σ^2. σ is a hyperparameter when training the model.

8 Batch Normalization

In deep learning we often normalize data to have mean 0 and variance 1 because this is the region where the activation function is most dynamic. Batch normalization is used in **batch gradient descent**.

8.1 Training

In batch normalization, instead of normalizing the data in the input level, we normalize the data in every level of the neural network.

In a regular layer:

$$Z = XW + b$$
$$A = f(Z)$$

In batch normalization, we normalized Z before passing it to the activation function. $\bar{Z}$ and σ_Z is the mean and standard deviation of the **batch** respectively.

$$Z = XW$$

$$Z_{normalized} = \gamma\left(\frac{z_i - \bar{Z}}{\sigma_Z + \epsilon}\right) + \beta$$
$$A = f(Z_{normalized})$$

After we normalized Z, we scale it with parameters γ and β. The reason behind this is because we are unsure if standardization is always a good choice. Therefore we will use gradient descent to update γ and β to achieve what is best for the neural network.

Notice the bias term b becomes redundant since we are already adding a bias term β in the normalization step.

Epsilon is added to prevent division by 0.

8.2 Prediction

During prediction we cannot normalize the data with only one sample as with batch gradient descent. Instead we will use the mean and standard deviation of the **whole data set** as $\bar{Z}$ and σ_Z.

$$Z = XW$$
$$Z_{normalized} = \gamma\left(\frac{z_i - \bar{Z}}{\sigma_Z + \epsilon}\right) + \beta$$
$$A = f(Z_{normalized})$$

If the data set is small we can calculate the mean and standard deviation directly. Alternatively we can keep track of the mean and standard deviation of each batch and calculate the exponentially moving average

$$\mu = decay \times \mu + (1 - decay)\mu_{batch}$$
$$\sigma = decay \times \sigma + (1 - decay)\sigma_{batch}$$