An overview of gradient descent optimization algorithms

Sebastian Ruder
Insight Centre for Data Analytics, NUI Galway
Aylien Ltd., Dublin
ruder.sebastian@gmail.com

Abstract

Gradient descent optimization algorithms, while increasingly popular, are often used as black-box optimizers, as practical explanations of their strengths and weaknesses are hard to come by. This article aims to provide the reader with intuitions with regard to the behaviour of different algorithms that will allow her to put them to use. In the course of this overview, we look at different variants of gradient descent, summarize challenges, introduce the most common optimization algorithms, review architectures in a parallel and distributed setting, and investigate additional strategies for optimizing gradient descent.

1 Introduction

Gradient descent is one of the most popular algorithms to perform optimization and by far the most common way to optimize neural networks. At the same time, every state-of-the-art Deep Learning library contains implementations of various algorithms to optimize gradient descent (e.g. lasagne's[2], caffe's[3], and keras'[4] documentation). These algorithms, however, are often used as black-box optimizers, as practical explanations of their strengths and weaknesses are hard to come by.

This article aims at providing the reader with intuitions with regard to the behaviour of different algorithms for optimizing gradient descent that will help her put them to use. In Section 2, we are first going to look at the different variants of gradient descent. We will then briefly summarize challenges during training in Section 3. Subsequently, in Section 4, we will introduce the most common optimization algorithms by showing their motivation to resolve these challenges and how this leads to the derivation of their update rules. Afterwards, in Section 5, we will take a short look at algorithms and architectures to optimize gradient descent in a parallel and distributed setting. Finally, we will consider additional strategies that are helpful for optimizing gradient descent in Section 6.

Gradient descent is a way to minimize an objective function $J(\ )$ parameterized by a model's parameters $\mathbb{R}^d$ by updating the parameters in the opposite direction of the gradient of the objective function $J(\ )$ w.r.t. to the parameters. The learning rate determines the size of the steps we take to reach a (local) minimum. In other words, we follow the direction of the slope of the surface created by the objective function downhill until we reach a valley.[5]

This paper originally appeared as a blog post at http://sebastianruder.com/optimizing-gradient-descent/index.html on 19 January 2016.

[2] http://lasagne.readthedocs.org/en/latest/modules/updates.html

[3] http://caffe.berkeleyvision.org/tutorial/solver.html

[4] http://keras.io/optimizers/

[5] If you are unfamiliar with gradient descent, you can find a good introduction on optimizing neural networks at http://cs231n.github.io/optimization-1/.

2 Gradient descent variants

There are three variants of gradient descent, which differ in how much data we use to compute the gradient of the objective function. Depending on the amount of data, we make a trade-off between the accuracy of the parameter update and the time it takes to perform an update.

2.1 Batch gradient descent

Vanilla gradient descent, aka batch gradient descent, computes the gradient of the cost function w.r.t. to the parameters for the entire training dataset:

$$= - \cdot J( ) \tag{1}$$

As we need to calculate the gradients for the whole dataset to perform just *one* update, batch gradient descent can be very slow and is intractable for datasets that do not fit in memory. Batch gradient descent also does not allow us to update our model *online*, i.e. with new examples on-the-fly.

In code, batch gradient descent looks something like this:

```
for i in range(nb_epochs):
  params_grad = evaluate_gradient(loss_function, data, params)
  params = params - learning_rate * params_grad
```

For a pre-defined number of epochs, we first compute the gradient vector `params_grad` of the loss function for the whole dataset w.r.t. our parameter vector `params`. Note that state-of-the-art deep learning libraries provide automatic differentiation that efficiently computes the gradient w.r.t. some parameters. If you derive the gradients yourself, then gradient checking is a good idea.[6]

We then update our parameters in the direction of the gradients with the learning rate determining how big of an update we perform. Batch gradient descent is guaranteed to converge to the global minimum for convex error surfaces and to a local minimum for non-convex surfaces.

2.2 Stochastic gradient descent

Stochastic gradient descent (SGD) in contrast performs a parameter update for *each* training example $x^{(i)}$ and label $y^{(i)}$:

$$= - \cdot J(; x^{(i)}; y^{(i)}) \tag{2}$$

Batch gradient descent performs redundant computations for large datasets, as it recomputes gradients for similar examples before each parameter update. SGD does away with this redundancy by performing one update at a time. It is therefore usually much faster and can also be used to learn online. SGD performs frequent updates with a high variance that cause the objective function to fluctuate heavily as in Figure 1.

While batch gradient descent converges to the minimum of the basin the parameters are placed in, SGD's fluctuation, on the one hand, enables it to jump to new and potentially better local minima. On the other hand, this ultimately complicates convergence to the exact minimum, as SGD will keep overshooting. However, it has been shown that when we slowly decrease the learning rate, SGD shows the same convergence behaviour as batch gradient descent, almost certainly converging to a local or the global minimum for non-convex and convex optimization respectively. Its code fragment simply adds a loop over the training examples and evaluates the gradient w.r.t. each example. Note that we shuffle the training data at every epoch as explained in Section 6.1.

```
for i in range(nb_epochs):
  np.random.shuffle(data)
  for example in data:
    params_grad = evaluate_gradient(loss_function, example, params)
    params = params - learning_rate * params_grad
```

[6] Refer to `http://cs231n.github.io/neural-networks-3/` for some great tips on how to check gradients properly.