Adam: a method for stochastic optimization

Czech Technical University in Prague
Faculty of Informatics

Radek Bartyzal

October 21, 2018

Contents

1 Prerequisites

The following text assumes that the reader is familiar with these terms:

- Neural Networks: Chapter 1 of [1]
- Backpropagation: Chapter 2 of [1]

2 Optimization of neural networks

Training of a neural network consists of minimization of its loss function L. An example of such loss function is:

w = weights	b = biases
n = number of inputs	x = one input
a = output of network for x	
$y(x)$ = desired output for x	

$$L(w, b) = \frac{1}{2n} \sum_x \|y(x) - a\|^2$$

By using the backpropagation algorithm we are able to calculate the gradient of the loss function with regards to every parameter. We can use this gradient to update the parameters. They are updated by a taking a small step in the opposite direction of the gradient since that is the direction in which the loss function decreases the most in its value. The size of the step is controlled by a parameter η called *learning rate.* This is a general description of a *Gradient Descent* algorithm, however there are many different ways how to calculate the actual step and we will present the ones most relevant to this work.

3 Gradient Descent variants

There are three variants of the basic gradient descent algorithm and all of them share the following inputs:

$$\begin{aligned}
\theta_0 &\quad \text{Initial parameters} \\
N &\quad \text{Number of training examples} \\
x_i,\ \forall i < N &\quad \text{Training examples} \\
\hat{y}_i,\ \forall i < N &\quad \text{Labels for the training examples} \\
T &\quad \text{Number of epochs}
\end{aligned} \tag{1}$$

- **Batch Gradient Descent (BGD)** shown as Algorithm 1 calculates the step as an average of gradients over all the training examples. This approach unfortunately does not scale well because from a certain point you cannot load the whole dataset into memory making each single update extremely slow. Another problem is that it does not allow *online learning* meaning we cannot simply continue training with newly arrived data points.

Algorithm 1: Batch Gradient Descent

```
for $t \leftarrow 1$ to $T$ do
    $g = 0$
    foreach $i \in dataset$ do
        $g = g + \nabla L(\hat{y}_i, f(\theta_{t-1}, x_i))$
    $g = \frac{1}{N} g$
    $\theta_t = \theta_{t-1} - \eta g$
```

- **Stochastic Gradient Descent (original SGD)** shown as Algorithm 2 updates the parameters with the gradient of every training example sequentially. The training data is shuffled between epochs to add an element of randomness resulting in increased chances of finding a better local minimum. This approach alleviates both mentioned problems of BGD, unfortunately it introduces its own one. Due to the frequent updates, the global loss tends to have a very high variance which may help it escape local minima but it also complicates convergence.

Algorithm 2: Stochastic Gradient Descent

```
$\theta = \theta_0$
for $t \leftarrow 1$ to $T$ do
    $shuffle(dataset)$
    foreach $i \in dataset$ do
        $g = \nabla L(\hat{y}_i, f(\theta, x_i))$
        $\theta = \theta - \eta g$
```

- **Mini-batch Stochastic Gradient Descent (SGD)** described in Algorithm 3 is the logical combination of the two mentioned methods. By updating the parameters after each mini batch of size B the algorithm achieves significantly less variance of the loss while keeping the advantage of frequent updates. It also leverages the fast matrix operations available on current graphical processors. The new hyperparameter B can be selected based on the size of the dataset and available memory to strike a balance between speed and variance. This algorithm is generally referred to as SGD because due to the strong disadvantages of both BGD and original SGD they are very rarely used.

Even though the SGD fixes the mentioned imperfections, it still has two significant problems:

- Finding a good learning rate can prove to be difficult, but we can offload this task to a *learning rate scheduler* that changes it during training based both on elapsed time steps and past performance [2]. Although the adaptable learning rate performs much better than a constant one, the fact that it is identical for all parameters causes problems in situation where each example while having a high dimension has only few non-zero features. Therefore some features occur more frequently than

Algorithm 3: Mini-batch Stochastic Gradient Descent

```
$\theta = \theta_0$
for $t \leftarrow 1$ to $T$ do
    $shuffle(dataset)$
    foreach $mini\_batch \in dataset$ do
        $g = 0$
        foreach $i \in mini\_batch$ do
            $g = g + \nabla L(\hat{y}_i, f(\theta, x_i))$
        $g = \frac{1}{B} g$
        $\theta = \theta - \eta g$
```

others which is not reflected in the applied learning rate. That results in slower updates to the less frequent features.

- The loss function of a deep neural network is undoubtedly very complex which brings many challenges to optimization. The most profound difficulty in optimizing such a high dimensional non-convex function is however believed to stem from an extensive number of saddle points surrounded by large plateaus with high error [3]. This is where the loss function increases in some dimensions while it is constant or decreasing in other dimensions. A simple three dimensional example can be a slowly descending valley with steep slopes on both sides. The problem stems from the small decrease in value in the one dimension that actually leads to a minimum. An intuitive explanation of what will happen is that the SGD will keep jumping across the valley while moving very slowly in the desired direction.

The following methods alleviate one or both of the described issues [4].

4 AdaGrad

AdaGrad is an adaptive gradient method attempting to solve the issue of some features appearing less frequently than others [12]. The general idea is for the learner to give larger weight to infrequent features when they appear.

This is implemented by taking note of the past gradient updates and using the sum of the squared past gradients to divide the actual learning rate.

We start with a gradient of the loss function L with respect to a parameter i at time t:

$$g_{t,i} = \nabla_\theta L(\theta_{t,i}) \tag{2}$$

The classic mini-batch SGD update would look like this:

$$\theta^{SGD}_{t+1,i} = \theta_{t,i} - \eta g_{t,i} \tag{3}$$

However the AdaGrad update leverages the past gradient update information to adaptively change the learning rate for each of the parameters separately:

$$\theta^{AdaGrad}_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{\sum_{\tau=1}^{t} g^2_{\tau,i} + \epsilon}} g_{t,i} \tag{4}$$

The epsilon is used to prevent division by an extremely small number. Even though this new update rule nicely adapts to each parameter, their learning rates keep getting smaller with increasing time steps. This is caused by the sum of squared gradients that can only increase with time resulting in a smaller and smaller effective learning rate, possibly reaching zero and stopping the training entirely. Another issue is the sensitivity to the initial setting of the learning rate. If the gradients are too large at the beginning, the parameter updates will be small for the rest of the training [5].

5 RMSProp and AdaDelta

Both RMSProp and AdaDelta have been invented around the same time to solve the mentioned issue of AdaGrad's diminishing learning rate. The RMSProp has been introduced by G. Hinton in his course at University of Toronto [11]. It is slightly simpler than AdaDelta [5] while using the same idea which is why we will discuss it here.

The central idea is to use a decaying running average of past squared gradients representing gradients from a certain time window instead of using all of them. This ensures that the training will not slow down after a large number of updates.

The running average of past gradients can be effectively calculated as:

$$\mathbb{E}[g^2]_t = \gamma \mathbb{E}[g^2]_{t-1} + (1-\gamma) g_t^2 \tag{5}$$

Then we just replace the summation term in the AdaGrad update rule with this running average estimate:

$$\theta_{t+1}^{RMSProp} = \theta_t - \frac{\eta}{\sqrt{\mathbb{E}[g^2]_t + \epsilon}} g_t \tag{6}$$

The decay rate γ is recommended to be set to 0.9 while a good default learning rate is 0.001. These methods are generally much less sensitive to a different initial learning rate making hyper-parameter tuning easier [5].

6 Adam

The Adam (adaptive moment estimator) method is an improvement of the previously mentioned RMSProp with an addition of an estimate of the first order momentum [6].

Just as the RMSProp the Adam calculates the decaying running average of the squared past gradients, here called v_t. It also calculates the running average of the gradients themselves, called m_t the same way. The m_t and v_t estimate the first and second order moments of the gradient corresponding to the mean and the uncentered variance.

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t \\ v_t &= \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \end{aligned} \tag{7}$$

The m_t and v_t are unfortunately biased toward zero at the start of training due to their initialization to zero vectors. To correct that bias the Adam algorithm divides the moment estimates with a time sensitive term approaching 1 with increasing number of time steps:

$$\begin{aligned} \hat{m_t} &= \frac{m_t}{(1-\beta_1^t)} \\ \hat{v_t} &= \frac{v_t}{(1-\beta_2^t)} \end{aligned} \tag{8}$$

These bias corrected moment estimates are then used in the actual update rule similarly to the RMSProp and AdaDelta algorithms:

$$\theta_{t+1}^{Adam} = \theta_t - \frac{\eta}{\sqrt{v_t} + \epsilon} m_t \tag{9}$$

The recommended default values for the hyper-parameters are $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$. The positive aspects of the second order moment dividing the learning rate have been explained in the previous sections. The addition of the first order moment can be understood as a momentum term. It aims to solve the problem SGD has with getting out of saddle points. The effects of momentum can be presented on the example with the slowly descending valley given at the end of Section 3.

If we imagine a ball without momentum in such valley, it will keep going up the opposing sides while slowly moving through the valley. If we add momentum to the ball, it will dampen its oscillation while increasing the speed of movement in the direction of consistent decrease in function value. We can compare that to adding weight to the ball.

Transferring the example to the effects on optimization, the updates in the dimensions where the gradient directions keep changing will be smaller while the updates in the dimensions that are consistently going a certain direction will become incrementally larger.

A follow-up research into Adam has uncovered several convergence issues and possible areas of improvement which led to the introduction of new versions such as:

- **NAdam:** Adapt the momentum term with the Nesterov accelerated gradient method [7].
- **AdamW:** Claims to fix the weight decay calculation[8].
- **AmsGrad:** Finds errors in the proof of Adam's convergence and claims to improve it by introducing AmsGrad algorithm [9].

To the contrary of the numerous papers claiming to improve the original Adam algorithm, experimental results show that it in many cases works better or at least as well as its newer variants. While some of them look promising, there is no single variant dominating others at all tested optimization tasks which is why we choose to use the default Adam for all of our experiments [10].

References

[1] Nielsen, Michael A. Neural networks and deep learning. Vol. 25. USA: Determination press, 2015.

[2] Darken, Christian, Joseph Chang, and John Moody. ”Learning rate schedules for faster stochastic gradient search.” Neural Networks for Signal Processing [1992] II., Proceedings of the 1992 IEEE-SP Workshop. IEEE, 1992.

[3] Dauphin, Yann N., et al. ”Identifying and attacking the saddle point problem in high-dimensional non-convex optimization.” Advances in neural information processing systems. 2014.

[4] Ruder, Sebastian. ”An overview of gradient descent optimization algorithms.” arXiv preprint arXiv:1609.04747 (2016).

[5] Zeiler, Matthew D. ”ADADELTA: an adaptive learning rate method.” arXiv preprint arXiv:1212.5701 (2012).

[6] Kingma, Diederik P., and Jimmy Ba. ”Adam: A method for stochastic optimization.” arXiv preprint arXiv:1412.6980 (2014).

[7] Dozat, Timothy. ”Incorporating nesterov momentum into adam.” (2016).

[8] Loshchilov, Ilya, and Frank Hutter. ”Fixing weight decay regularization in adam.” arXiv preprint arXiv:1711.05101 (2017).

[9] Reddi, Sashank J., Satyen Kale, and Sanjiv Kumar. ”On the convergence of adam and beyond.” (2018).

[10] FastAI’s experimental results of Adam variants [online]. `http://www.fast.ai/2018/07/02/adam-weight-decay/`, [Online; accessed 2018-October-21].

[11] G. Hinton’s lecture introducing RMSProp [online]. `http://www.cs.toronto.edu/œtijmen/csc321/slides/lecture_slides_lec6.pdf`, [Online; accessed 2018-October-21].

[12] Duchi, John, Elad Hazan, and Yoram Singer. ”Adaptive subgradient methods for online learning and stochastic optimization.” Journal of Machine Learning Research 12.Jul (2011): 2121-2159.