TFML Report

Siddhant Chaudhary

Abstract

This is the report for a reading project completed during a Machine Learning course in CMI. The title of the paper was *Adam: A method for stochastic optimization.*

Contents

1. Introduction

1.1 Overview

Adam, which stands for *Adaptive Moment Estimation*, is a technique for optimizing stochastic gradient descent. It is one of the more efficient methods to optimize SGD, and is also readily available in most ML packages. In practice, it has been observed that Adam works well and is comparatively better than other optimization techniques like AdaGrad and gradient descent with momentum. Let us now quickly review the mathematical ideas used in Adam.

1.1.1 Moving Averages. Given a sequence $\{a_n\}_{n\in\mathbf{N}}$, one can define the so called *moving average* of the sequence. A *decaying parameter* $\beta > 0$ is picked, and the sequence $\{A_n\}_{n\in\mathbf{N}}$ is defined as follows.

$$A_1 = a_1$$
$$A_n = \beta A_{n-1} + (1-\beta)a_n$$

This is also called the *exponential moving average*, because if we unfold this recurrence, then it can easily be seen that the older terms in the sequences are weighted by powers of β. Adam uses moving averages of the first and second moments of the sequence of gradients obtained at consecutive time steps, which will be made clear in further sections.

1.1.2 Momentum and adaptive step sizes. Adam combines the techniques of *momentum* (another method to optimize gradient descent) and *adaptive weights*, i.e Adam uses a different step size for each learning parameter. So in some sense, Adam adaptively chooses a good increment for each parameter it wants to learn. This makes Adam flexible to cases in which the gradients are *sparse*, i.e cases when the coordinates of the gradient vectors are close to zero.

1.2 Algorithm and Pseudocode

1.2.1 Algorithm Description. As input, Adam takes four parameters: $\alpha > 0$ (a step-size parameter), two decay parameters $\beta_1, \beta_2 \in [0, 1)$, and a parameter $\epsilon > 0$ (which is used to prevent division by zero errors). Good default settings for many tested ML problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta = 0.999$ and $\epsilon = 10^{-8}$.

At each time step t, we are given an objective function $f_t(\theta)$. Here θ lies in the parameter space. One should think of the f_ts as realizations of some stochastic objective function $f(\theta)$ at various time steps. For example, in minibatch gradient descent, few data points are picked randomly, and a corresponding objective is determined out of the data points. Our goal is to output some value of θ which minimizes the objective $f(\theta)$. We assume that f_t is differentiable, and we use the notation g_t to denote the gradient of f_t, i.e $g_t = f_t(\theta_{t-1})$ (where θ_{t-1} is our choice at the previous time step).

The algorithm maintains four estimates: m_t, $\hat{m}_t$, v_t and $\hat{v}_t$. Initially all of these are set to 0. m_t is the moving average of the sequence $\{g_t\}$ of gradients till time step t, and similarly v_t is the moving average of the sequence of second moments $\left\{g_t^2\right\}$ till time step t. Here g_t^2 is the coordinate wise product of g_t with itself. The estimates $\hat{m}_t$ and $\hat{v}_t$ are bias-corrected estimates of m_t and v_t; the bias emerges from the fact that we initialize m_t and v_t to 0. The algorithm then increments the parameter θ_t as follows.

$$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \frac{\hat{m}_t}{\sqrt{v}_t + \epsilon}$$

At the end, the resulting parameter θ_{T+1} is returned, where T denotes the number of time steps the algorithm is run for. For the pseudocode, refer to **Algorithm** 1.

1.2.2 Bias Corrections. As mentioned in the previous discussion, initializing the variables m_t, v_t to zero leads to terms which are biased towards zero. Now we will see how to correct these terms and obtain the biased corrected terms $\hat{m}_t$ and $\hat{v}_t$.

Suppose $g_1, ..., g_t$ are the gradients obtained at the time steps. For simplicity, we can assume that the gradients are *stationary*, i.e they are obtained from the same distribution at each time step. So, for any $1 \leq i, j \leq t$, we have

$$\mathbf{E}\left[g_i\right] = \mathbf{E}\left[g_j\right] \tag{1}$$

For instance, the above assumption is true if one is implementing SGD by sampling a data point uniformly at random from the dataset. Expanding the recurrence $v_t =$

Algorithm 1 Adam

Input: α and $\beta_1, \beta_2 \in [0, 1)$.
Required: $f(\theta)$ (objective) and θ_0 (initial parameter)
$m_0 \leftarrow 0$
$v_0 \leftarrow 0$
$t \leftarrow 0$
while (some convergence critrion on θ_t) **do**
  $t \leftarrow t + 1$
  $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$
  $m_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1) g_t$
  $v_t \leftarrow \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$
  $\hat{m}_t \leftarrow m_t / (1 - \beta_1^t)$
  $\hat{v}_t \leftarrow v_t / (1 - \beta_2^t)$
  $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon)$
end while
return θ_t

$\beta_2 v_{t-1} + (1 - \beta_2) g_t^2$ with the initial condition $v_0 = 0$, we obtain the following formula for v_t.

$$v_t = (1 - \beta_2) \sum_{i=1}^{t} \beta_2^{t-i} g_i^2$$

Ideally we want $\mathbf{E}[v_t] = \mathbf{E}\left[g_t^2\right]$, which is the true expectation of the second moment. Taking expectations on both sides of the above equation and using (1), we obtain the following.

$$\begin{aligned} \mathbf{E}[v_t] &= (1 - \beta_2) \sum_{i=1}^{t} \beta_2^{t-i} \mathbf{E}\left[g_i^2\right] \\ &= \mathbf{E}[g_t]^2 (1 - \beta_2) \sum_{i=1}^{t} \beta_2^{t-i} \\ &= \mathbf{E}\left[g_t^2\right] (1 - \beta_2^t) \end{aligned}$$

So, we see that dividing out by $(1 - \beta_2^t)$ ensures that $\mathbf{E}[\hat{v}_t] = \mathbf{E}\left[g_t^2\right]$, and hence the term $\hat{v}_t$ is free of bias. A similar analysis works for m_t and $\hat{m}_t$ as well. Infact, even if condition (1) is not true, the above analysis is approximately correct, as the values of β_1 and β_2 are typically chosen so that weights assigned to gradients too far in the past are small.

2. Convergence Guarantees and Experiments

2.1 Bound on Regret

2.1.1 The OCO setting. The authors of the paper have analyzed Adam using the Online Convex Optimization (OCO) framework. In this framework, we are given an arbitrary sequence of convex cost functions $f_1(\theta)$, $f_2(\theta)$,..., $f_T(\theta)$. At each time step t, the goal is to pick a parameter θ_t from a convex domain in order to minimize the *regret* $R(T)$, which is defined as follows.

$$R(T) := \sum_{t=1}^{T} [f_t(\theta_t) - f_t(\theta^*)]$$

Above, $\theta^* = \text{argmin}_{\theta \in X} \sum_{t=1}^{T} f_t(\theta)$, where X is the convex domain in question. The authors have shown that Adam has $O(\sqrt{T})$ regret bound, which is comparable to the usual regret bounds for OCO algorithms.

To be more specific, the authors have considered the following conditions.

(1) *Bounded gradients*: for all t and $\theta \in X$, we assume that $||\nabla f_t(\theta)||_2 \leq G$ and $||\nabla f_t(\theta)||_\infty \leq G_\infty$.
(2) *Bounded diameter*: for all $1 \leq i, j \leq T$, we assume that $||\theta_i - \theta_j||_2 \leq D$ and $||\theta_i - \theta_j||_\infty \leq D_\infty$. This will be the case if the *diameter* of the set X w.r.t the $||\cdot||_2$ and $||\cdot||_\infty$ norms is upper bounded.
(3) $\beta_1, \beta_2 \in [0, 1)$ satisfy $\beta_1^2 < \sqrt{\beta_2}$. Here β_1, β_2 are the decay parameters used in Adam.
(4) The step sizes are $\alpha_t = \frac{\alpha}{\sqrt{t}}$.
(5) Instead of using the fixed parameter β_1 at all time steps, the parameter $\beta_{1,t} = \beta_1 \cdot \lambda^{t-1}$ is used at time t, where $\lambda \in (0, 1)$. In simple words, the first moment averaging coefficient decays exponentially with time.

If the above conditions hold, the authors have shown that Adam has $O(dG_\infty\sqrt{T})$ regret bound, where d is the dimension of the space where the data points are. In fact, the authors have shown the following stronger bound on regret.

$$R(T) \leq \frac{D^2}{2\alpha(1-\beta_1)} \sum_{t=1}^{d} \sqrt{T\hat{v}_{T,i}} + \frac{\alpha(1+\beta_1)G_\infty}{(1-\beta_1)\sqrt{1-\beta_2}(1-\gamma)^2} \sum_{i=1}^{d} ||g_{1:T}, i||_2 + \sum_{i=1}^{d} \frac{D_\infty^2 G_\infty \sqrt{1-\beta_2}}{2\alpha(1-\beta_1)(1-\lambda)^2}$$

Above, $\hat{v}_{T,i}$ denotes the i^{th} coordinate of $\hat{v}_T$, and for each $1 \leq i \leq d$, the vector $g_{1:T} \in \mathbf{R}^T$ is defined to be the T-dimensional vector containing the i^{th} coordinates of all the gradients till time T, i.e

$$g_{1:T,i} = (g_{1,i}, g_{2,i}, \cdots, g_{T,i})$$

It can be shown that the RHS of the above bound is $O(dG_\infty\sqrt{T})$. Moreover, in case of sparse gradients, the above bound is much stronger than $O(dG_\infty\sqrt{T})$.

2.2 Experiment: A comparison with AdaGrad

2.2.1 Overview of the experiment. We conducted a simple test in TensorFlow to compare the performance of Adam and AdaGrad in terms of the rate of convergence in training accuracies on the MNIST and the Fashion MNIST datasets. The problem is a multi-class classification problem on these datasets.

To do this, we use the usual method of *softmax classification*, which is a generalisation of logistic regression to multiple classes. We train a simple neural network whose output is a probability distribution $a = (a_1, \cdots, a_{10})$ on the 10 labels. Each coordinate of a represents the probability that a given input has that particular label. The loss function we used is the standard *cross entropy loss* using one-hot encoding of the labels. So, if (x, y) is a datapoint, with $y = (y_1, ..., y_{10})$ being the one-hot encoding of x's label, then

$$L_{(x,y)}(a) = -\sum_{k=1}^{10} y_i \log(a_i)$$

2.2.2 Experiment on MNIST. For the MNIST dataset, our neural network was a simple 4 layered netword. The first three layers had 128 nodes each (with the identity

activation), and the last layer had 10 nodes with softmax activation. Both Adam and AdaGrad were trained on MNIST with 10 iterations and batch size of 128.

Adam was trained with $\alpha = 0.001$ (learning rate), $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-7}$. AdaGrad was trained with learning rate $\alpha = 0.001$ and $\epsilon = 10^{-7}$. Over the 10 iterations, the accuracies were as follows (look at the table).

T	Adam	AdaGrad
1	0.8899	0.6364
2	0.9143	0.8318
3	0.9160	0.8602
4	0.9199	0.8751
5	0.9195	0.8836
6	0.9196	0.8889
7	0.9207	0.8935
8	0.9221	0.8963
9	0.9223	0.8982
10	0.9228	0.9000

2.2.3 Experiment on Fashion MNIST. For confirmation, we tried the same experiment on the Fashion MNIST dataset as well. Here, we chose an even simpler network: a two layered network, in which the first layer had 128 nodes with ReLu activation, and the second (and last) layer had 10 nodes with softmax activation. The hyperparameters were the same. The training was done for $T = 50$ steps. And again, Adam outperformed AdaGrad. After 50 iterations, Adam ended up with an accuracy of 0.9606, while AdaGrad ended up with an accuracy of just 0.8434.

2.2.4 Code. The notebooks for the tests are available at this link: https://github.com/codetalker7/adam-vs-adagrad. Look at `main-test.ipynb` for the first test, and `main-test-fashion-mnist.ipynb` for the second test.