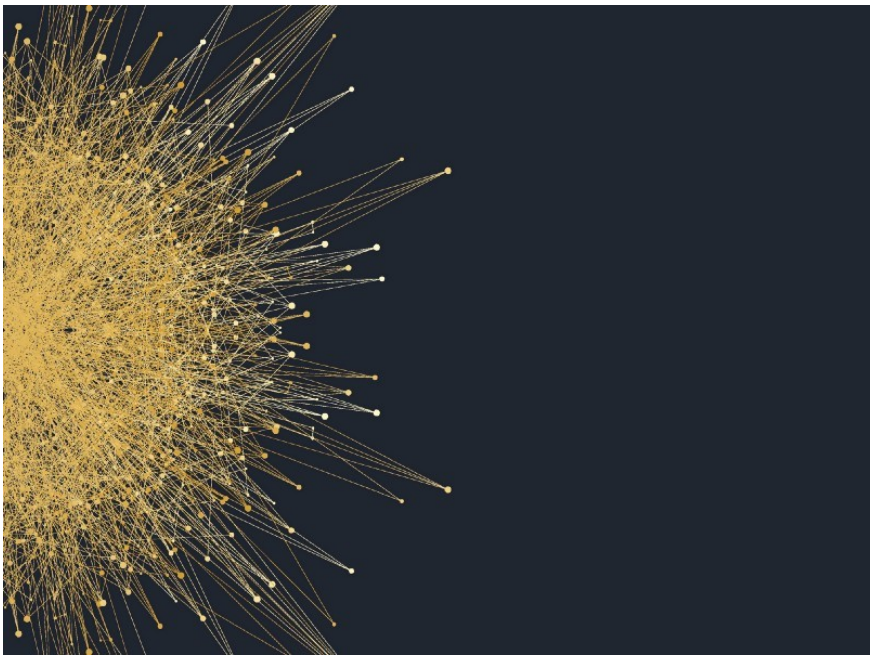
18.



Adam — latest trends in deep learning optimization.

Vitaly Bushaev Follow
Oct 22, 2018 · 16 min read

Adam [1] is an adaptive learning rate optimization algorithm that's been designed specifically for training deep neural networks. First published in 2014, Adam was presented at a very prestigious conference for deep learning practitioners—ICLR 2015. The paper contained some very promising diagrams, showing huge performance gains in terms of speed of training. However, after a while people started noticing, that in some cases Adam actually finds worse solution than stochastic gradient descent. A lot of research has been done to address the problems of Adam.

The algorithms leverages the power of adaptive learning rates methods to find individual learning rates for each parameter. It also has advantages of Adagrad [10], which works really well in settings with sparse gradients, but struggles in non-convex optimization of neural networks, and RMSprop [11], which tackles to resolve some of the problems of Adagrad and works really well in on-line settings. Adam has been raising in popularity exponentially according to 'A Peek at Trends in Machine Learning' article from Andrej Karpathy.

In this post, I first introduce Adam algorithm as presented in the

original paper, and then walk through latest research around it that demonstrates some potential reasons why the algorithms works worse than classic SGD in some areas and provides several solutions, that narrow the gap between SGD and Adam.

Adam

Adam can be looked at as a combination of RMSprop and Stochastic Gradient Descent with momentum. It uses the squared gradients to scale the learning rate like RMSprop and it takes advantage of momentum by using moving average of the gradient instead of gradient itself like SGD with momentum. Let's take a closer look at how it works.

Adam is an adaptive learning rate method, which means, it computes individual learning rates for different parameters. Its name is derived from adaptive moment estimation, and the reason it's called that is because Adam uses estimations of first and second moments of gradient to adapt the learning rate for each weight of the neural network. Now, what is moment ? N-th moment of a random variable is defined as the expected value of that variable to the power of n. More formally:

$$m_n = E\left[X^n\right]$$

m—moment, X -random variable.

It can be pretty difficult to grasp that idea for the first time, so if you don't understand it fully, you should still carry on, you'll be able to understand how algorithms works anyway. Note, that gradient of the cost function of neural network can be considered a random variable, since it usually evaluated on some small random batch of data. The first moment is mean, and the second moment is uncentered variance (meaning we don't subtract the mean during variance calculation). We will see later how we use these values, right now, we have to decide on how to get them. To estimates the moments, Adam utilizes exponentially moving averages, computed on the gradient evaluated on a current mini-batch:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$

Moving averages of gradient and squared gradient.

Where m and v are moving averages, g is gradient on current mini-batch, and betas—new introduced hyper-parameters of the algorithm. They have really good default values of 0.9 and 0.999 respectively. Almost no one ever changes these values. The vectors of moving averages are initialized with zeros at the first iteration.

To see how these values correlate with the moment, defined as in first equation, let's take look at expected values of our moving averages. Since m and v are estimates of first and second moments, we want to have the following property:

$$E[m_t] = E[g_t]$$
$$E[v_t] = E[g_t^2]$$

Expected values of the estimators should equal the parameter we're trying to estimate, as it happens, the parameter in our case is also the expected value. If these properties held true, that would mean, that we have **unbiased estimators**. (To learn more about statistical properties of different estimators, refer to Ian Goodfellow's Deep Learning book, Chapter 5 on machine learning basics). Now, we will see that these do not hold true for the our moving averages. Because we initialize averages with zeros, the estimators are biased towards zero. Let's prove that for m (the proof for v would be analogous). To prove that we need to formula for m to the very first gradient. Let's try to unroll a couple values of m to see he pattern we're going to use:

$$m_0 = 0$$
$$m_1 = \beta_1 m_0 + (1-\beta_1) g_1 = (1-\beta_1) g_1$$
$$m_2 = \beta_1 m_1 + (1-\beta_1) g_2 = \beta_1 (1-\beta_1) g_1 + (1-\beta_1) g_2$$
$$m_3 = \beta_1 m_2 + (1-\beta_1) g_3 = \beta_1^2 (1-\beta_1) g_1 + \beta_1 (1-\beta_1) g_2 + (1-\beta_1) g_3$$

As you can see, the 'further' we go expanding the value of m, the less first values of gradients contribute to the overall value, as they get multiplied by smaller and smaller beta. Capturing this patter, we can rewrite the formula for our moving average:

$$m_t = (1-\beta_1) \sum_{i=0}^{t} \beta_1^{t-i} g_i$$

Now, let's take a look at the expected value of m, to see how it relates to the true first moment, so we can correct for the discrepancy of the two :

$$E[m_t] = E[(1-\beta_1) \sum_{i=1}^{t} \beta_1^{t-i} g_i]$$
$$= E[g_i](1-\beta_1) \sum_{i=1}^{t} \beta_1^{t-i} + \zeta$$
$$= E[g_i](1-\beta_1^t) + \zeta$$

Bias correction for the first momentum estimator

In the first row, we use our new formula for moving average to expand m. Next, we approximate g[i] with g[t]. Now we can take it out of sum, since it does not now depend on i. Because the approximation is taking place, the error C emerge in the formula. In the last line we just use the formula for the sum of a finite geometric series. There are two things we should note from that equation.

We have biased estimator. This is not just true for Adam only, the same holds for algorithms, using moving averages (SGD with momentum, RMSprop, etc.).

It won't have much effect unless it's the begging of the training, because the value beta to the power of t is quickly going towards zero.

Now we need to correct the estimator, so that the expected value is the one we want. This step is usually referred to as bias correction. The final formulas for our estimator will be as follows:

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

Bias corrected estimators for the first and second moments.

The only thing left to do is to use those moving averages to scale learning rate individually for each parameter. The way it's done in Adam is very simple, to perform weight update we do the following:

$$w_t = w_{t-1} - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

Where w is model weights, eta (look like the letter n) is the step size (it can depend on iteration). And that's it, that's the update rule for Adam. For some people it can be easier to understand such concepts in code, so here's possible implementation of Adam in python:

```
for t in range(num_iterations):
    g = compute_gradient(x, y)
    m = beta_1 * m + (1 - beta_1) * g
    v = beta_2 * v + (1 - beta_2) * np.power(g, 2)
    m_hat = m / (1 - np.power(beta_1, t))
```

There are two small variations on Adam that I don't see much in practice, but they're implemented in major deep learning frameworks, so it's worth to briefly mention them.

First one, called **Adamax** was introduced by the authors of Adam in the same paper. The idea with Adamax is to look at the value v as the L2 norm of the individual current and past gradients. We can generalize it to Lp update rule, but it gets pretty unstable for large values of p. But if we use the special case of L-infinity norm, it results in a surprisingly stable and well-performing algorithm. Here's how to implement Adamax with python:

```
for t in range(num_iterations):
    g = compute_gradient(x, y)
    m = beta_1 * m + (1 - beta_1) * g
    m_hat = m / (1 - np.power(beta_1, t))
    v = np.maximum(beta_2 * v, np.abs(g))
```

Second one is a bit harder to understand, called **Nadam** [6]. Nadam was published by Timothy Dozat in the paper 'Incorporating Nesterov Momentum into Adam'. As name suggests the idea is to use Nesterov momentum term for the first moving averages. Let's take a look at update rule of the SGD with momentum:

$$m_t = \beta m_{t-1} + \eta g_t$$
$$w_t = w_{t-1} - m_t = w_{t-1} - \beta m_{t-1} - \eta g_t$$

SGD with momentum update rule

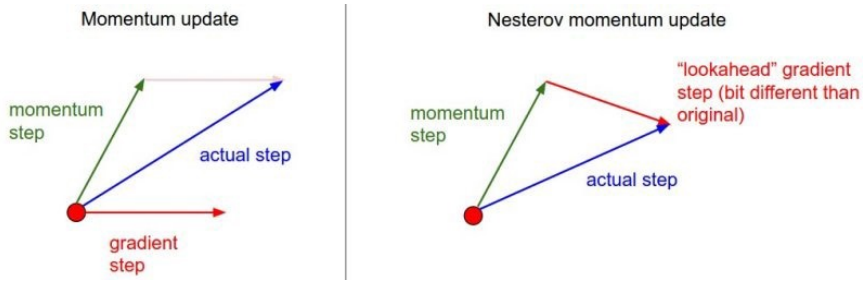
As shown above, the update rule is equivalent to taking a step in the direction of momentum vector and then taking a step in the direction of gradient. However, the momentum step doesn't depend on the current

gradient , so we can get a higher-quality gradient step direction by updating the parameters with the momentum step before computing the gradient. To achieve that, we modify the update as follows:

$$g_t = \nabla f\left(w_{t-1} - \beta m_{t-1}\right)$$
$$m_t = \beta m_{t-1} + \eta g_t$$
$$w_t = w_{t-1} - m_t$$

f—loss function to optimize.

So, with Nesterov accelerated momentum we first make make a big jump in the direction of the previous accumulated gradient and then measure the gradient where we ended up to make a correction. There's a great visualization from cs231n lecture notes:



sourec: cs231n lecture notes.

The same method can be incorporated into Adam, by changing the first moving average to a Nesterov accelerated momentum. One computation trick can be applied here: instead of updating the parameters to make momentum step and changing back again, we can achieve the same effect by applying the momentum step of time step t + 1 only once, during the update of the previous time step t instead of t + 1. Using this trick, the implementation of Nadam may look like this:

```
for t in range(num_iterations):
    g = compute_gradient(x, y)
    m = beta_1 * m + (1 - beta_1) * g
    v = beta_2 * v + (1 - beta_2) * np.power(g, 2)
    m_hat = m / (1 - np.power(beta_1, t)) + (1 - beta_1) * g
```

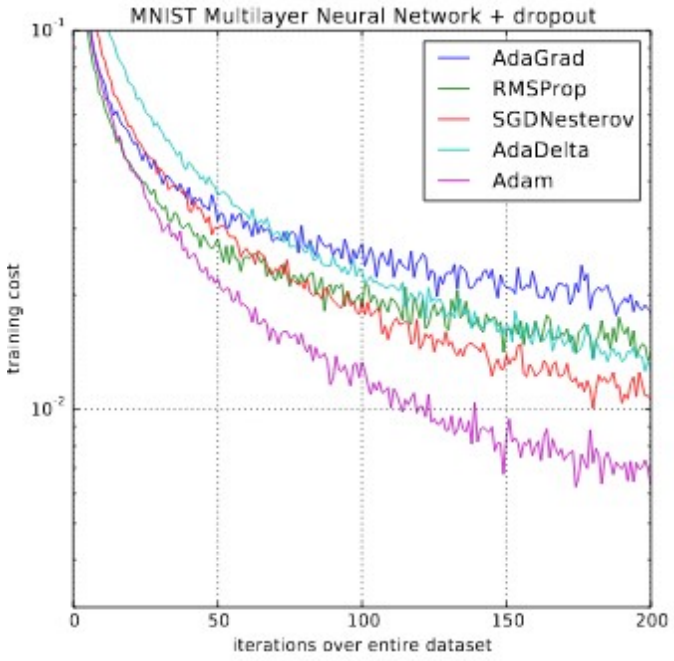
Properties of Adam

Here I list some of the properties of Adam, for proof that these are true refer to the paper.

> Actual step size taken by the Adam in each iteration is approximately bounded the step size hyper-parameter. This property add intuitive understanding to previous unintuitive learning rate hyper-parameter.
>
> Step size of Adam update rule is invariant to the magnitude of the gradient, which helps a lot when going through areas with tiny gradients (such as saddle points or ravines). In these areas SGD struggles to quickly navigate through them.
>
> Adam was designed to combine the advantages of Adagrad, which works well with sparse gradients, and RMSprop, which works well in on-line settings. Having both of these enables us to use Adam for broader range of tasks. Adam can also be looked at as the combination of RMSprop and SGD with momentum.

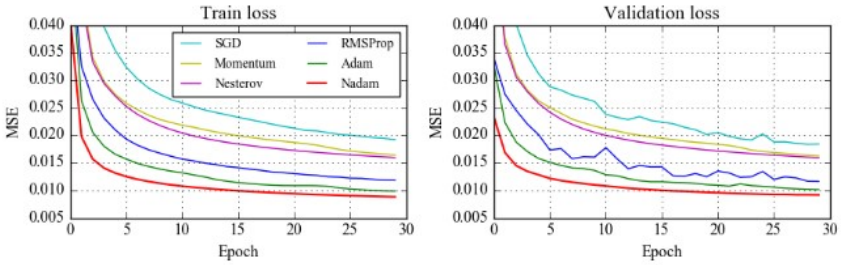
Problems with Adam

When Adam was first introduced, people got very excited about its power. Paper contained some very optimistic charts, showing huge performance gains in terms of speed of training:



source: original Adam paper

Then, Nadam paper presented diagrams that showed even better results:



source: Nadam paper

However, after a while people started noticing that despite superior training time, Adam in some areas does not converge to an optimal solution, so for some tasks (such as image classification on popular CIFAR datasets) state-of-the-art results are still only achieved by applying SGD with momentum. More than that Wilson et. al [9] showed in their paper 'The marginal value of adaptive gradient methods in machine learning' that adaptive methods (such as Adam or Adadelta) do not generalize as well as SGD with momentum when tested on a diverse set of deep learning tasks, discouraging people to use popular optimization algorithms. A lot of research has been done since to analyze the poor generalization of Adam trying to get it to close the gap with SGD.

Nitish Shirish Keskar and Richard Socher in their paper 'Improving Generalization Performance by Switching from Adam to SGD' [5] also showed that by switching to SGD during training training they've been able to obtain better generalization power than when using Adam alone. They proposed a simple fix which uses a very simple idea. They've noticed that in earlier stages of training Adam still outperforms SGD but later the learning saturates. They proposed simple strategy which they called **SWATS** in which they start training deep neural network with Adam but then switch to SGD when certain criteria hits. They managed to achieve results comparable to SGD with momentum.

On the convergence of Adam

One big thing with figuring out what's wrong with Adam was analyzing it's convergence. The authors proved that Adam converges to the global minimum in the convex settings in their original paper, however, several papers later found out that their proof contained a few mistakes. Block et. al [7] claimed that they have spotted errors in the original convergence analysis, but still proved that the algorithm converges and provided proof in their paper. Another recent article from Google employees was presented at ICLR 2018 and even won best paper award. To go deeper to their paper I should first describe the framework used by Adam authors for proving that it converges for convex functions.

In 2003 Martin Zinkevich introduced Online Convex Programming problem [8]. In the presented settings, we have a sequence of convex functions c1, c2, etc (Loss function executed in ith mini-batch in the case of deep learning optimization). The algorithm, that solves the problem (Adam) in each timestamp t chooses a point x[t] (parameters of the model) and then receives the loss function c for the current timestamp. This setting translates to a lot of real world problems, for examples read the introduction of the paper. For understanding how good the algorithm works, the value of regret of the algorithm after T rounds is defined as follows:

$$R(T)=\sum_{t=1}^{T}\left[c_t(w_t)-c_t(w_t^*)\right]$$

Regret of the algorithm in the online convex programming

where R is regret, c is the loss function on tth mini batch, w is vector of model parameters (weights), and w star is optimal value of weight vector. Our goal is to prove that the regret of algorithm is R(T) = O(T) or less, which means that on average the model converges to an optimal solution. Martin Zinkevich in his paper proved that gradient descent converges to optimal solutions in this setting, using the property of the convex functions:

$$c_t(x)\geq(\nabla c_t(x_t))(x-x_t)+c_t(x_t)$$

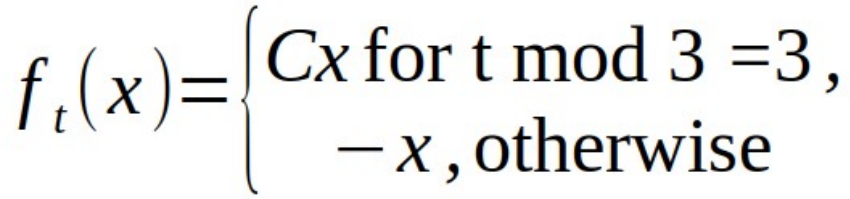
Well-known property of convex functions.

The same approach and framework used Adam authors to prove that their algorithm converges to an optimal solutions. Reddi et al. [3] spotted several mistakes in their proof, the main one lying in the value, which appears in both Adam and Improving Adam's proof of convergence papers:

$$\frac{V_{t+1}}{\eta_{t+1}}-\frac{V_t}{\eta_t}$$

Where V is defined as an abstract function that scales learning rate for parameters which differs for each individual algorithms. For Adam it's the moving averages of past squared gradients, for Adagrad it's the sum of all past and current gradients, for SGD it's just 1. The authors found that in order for proof to work, this value has to be positive. It's easy to see, that for SGD and Adagrad it's always positive, however, for Adam(or RMSprop), the value of V can act unexpectedly. They also

presented an example in which Adam fails to converge:



Adam fails on this sequence

For this sequence, it's easy to see that the optimal solution is x = -1, however, how authors show, Adam converges to highly sub-optimal value of x = 1. The algorithm obtains the large gradient C once every 3 steps, and while the other 2 steps it observes the gradient -1 , which moves the algorithm in the wrong direction. Since values of step size are often decreasing over time, they proposed a fix of keeping the maximum of values V and use it instead of the moving average to update parameters. The resulting algorithm is called **Amsgrad.** We can confirm their experiment with this short notebook I created, which shows different algorithms converge on the function sequence defined above.

```
for t in range(num_iterations):
    g = compute_gradient(x, y)
    m = beta_1 * m + (1 - beta_1) * g
    v = beta_2 * v + (1 - beta_2) * np.power(g, 2)
    v_hat = np.maximum(v, v_hat)
```

Amsgrad without bias correction

How much does it help in practice with real-world data ? Sadly, I haven't seen one case where it would help get better results than Adam. Filip Korzeniowski in his post describes experiments with Amsgrad, which show similar results to Adam. Sylvain Gugger and Jeremy Howard in their post show that in their experiments Amsgrad actually performs even worse that Adam. Some reviewers of the paper also pointed out that the issue may lie not in Adam itself but in framework, which I described above, for convergence analysis, which does not allow for much hyper-parameter tuning.

Weight decay with Adam

One paper that actually turned out to help Adam is 'Fixing Weight Decay Regularization in Adam' [4] by Ilya Loshchilov and Frank Hutter. This paper contains a lot of contributions and insights into Adam and weight decay. First, they show that despite common belief L2 regularization is not the same as weight decay, though it is equivalent for stochastic gradient descent. The way weight decay was introduced back in 1988 is:

$$w_{t+1} = (1 - \lambda) w_t - \eta \nabla f_t(w_t)$$

Where lambda is weight decay hyper parameter to tune. I changed notation a little bit to stay consistent with the rest of the post. As defined above, weight decay is applied in the last step, when making the weight update, penalizing large weights. The way it's been traditionally implemented for SGD is through L2 regularization in which we modify the cost function to contain the L2 norm of the weight vector:

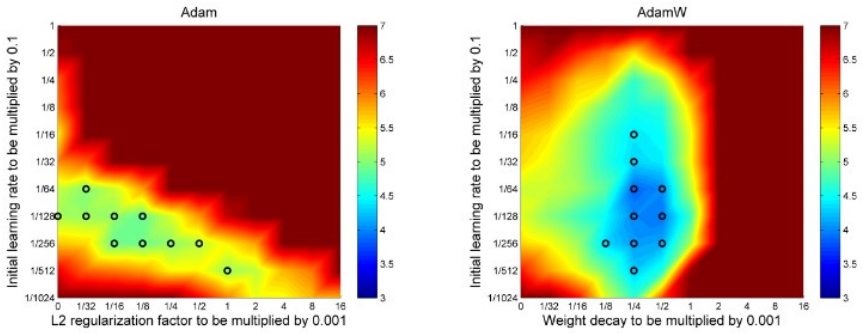
$$f_t^{\text{reg}}(w_t) = f_t(w_t) + \frac{\lambda}{2} \|w_t\|_2^2$$

Historically, stochastic gradient descent methods inherited this way of implementing the weight decay regularization and so did Adam. However, L2 regularization is not equivalent to weight decay for Adam. When using L2 regularization the penalty we use for large weights gets scaled by moving average of the past and current squared gradients and therefore weights with large typical gradient magnitude are regularized by a smaller relative amount than other weights. In contrast, weight decay regularizes all weights by the same factor. To use weight decay with Adam we need to modify the update rule as follows:

$$w_t = w_{t-1} - \eta\left(\frac{\hat{m}}{\sqrt{\hat{v}_t} + \epsilon} + \lambda w_{t-1}\right)$$

Adam update rule with weight decay

Having show that these types of regularization differ for Adam, authors continue to show how well it works with both of them. The difference in results is shown very well with the diagram from the paper:



The Top-1 test error of ResNet on CIFAR-10 measured after 100 epochs

These diagrams show relation between learning rate and regularization method. The color represent high low the test error is for this pair of hyper parameters. As we can see above not only Adam with weight decay gets much lower test error it actually helps in decoupling learning rate and regularization hyper-parameter. On the left picture we can the that if we change of the parameters, say learning rate, then in order to achieve optimal point again we'd need to change L2 factor as well, showing that these two parameters are interdependent. This dependency contributes to the fact hyper-parameter tuning is a very difficult task sometimes. On the right picture we can see that as long as we stay in some range of optimal values for one the parameter, we can change another one independently.

Another contribution by the author of the paper shows that optimal value to use for weight decay actually depends on number of iteration during training. To deal with this fact they proposed a simple adaptive formula for setting weight decay:

$$\lambda = \lambda_{norm} \sqrt{\frac{b}{BT}}$$

where b is batch size, B is the total number of training points per epoch and T is the total number of epochs. This replaces the lambda hyper-parameter lambda by the new one lambda normalized.

The authors didn't even stop there, after fixing weight decay they tried to apply the learning rate schedule with warm restarts with new version of Adam. Warm restarts helped a great deal for stochastic gradient descent, I talk more about it in my post 'Improving the way we work with learning rate'. But previously Adam was a lot behind SGD. With new weight decay Adam got much better results with restarts, but it's still not as good as SGDR.

ND-Adam

One more attempt at fixing Adam, that I haven't seen much in practice is proposed by Zhang et. al in their paper 'Normalized Direction-preserving Adam' [2]. The paper notices two problems with Adam that may cause worse generalization:

> The updates of SGD lie in the span of historical gradients, whereas it is not the case for Adam. This difference has also been observed in already mentioned paper [9].
>
> Second, while the magnitudes of Adam parameter updates are invariant to descaling of the gradient, the effect of the updates on the same overall network function still varies with the magnitudes of parameters.

To address these problems the authors propose the algorithm they call Normalized direction-preserving Adam. The algorithms tweaks Adam in the following ways. First, instead of estimating the average gradient magnitude for each individual parameter, it estimates the average squared L2 norm of the gradient vector. Since now V is a scalar value and M is the vector in the same direction as W, the direction of the update is the negative direction of m and thus is in the span of the historical gradients of w. For the second the algorithms before using

gradient projects it onto the unit sphere and then after the update, the weights get normalized by their norm. For more details follow their paper.

Conclusion

Adam is definitely one of the best optimization algorithms for deep learning and its popularity is growing very fast. While people have noticed some problems with using Adam in certain areas, researches continue to work on solutions to bring Adam results to be on par with SGD with momentum.

References

Diederik P. Kingma and Jimmy Lei Ba. Adam : A method for stochastic optimization. 2014. arXiv:1412.6980v9

Zijun Zhang et al. Normalized direction-preserving Adam. 2017. arXiv:1709.04546v2

Sashank J. Reddi, Satyen Kale, Sanjiv Kumar. On the Convergence of Adam and Beyond. 2018.

Ilya Loshchilov, Frank Hutter. Fixing Weight Decay Regularization in Adam. 2017. arXiv:1711.05101v2

Nitish Shirish Keskar, Richard Socher. Improving Generalization Performance by Switching from Adam to SGD. 2017 arXiv:1712.07628v1

Timothy Dozat. Incorporating Nesterov momentum into Adam. 2016.

Sebastian Bock, Josef Goppold, Martin Weiß. An improvement of the convergence proof of the ADAM-Optimizer. 2018. arXiv:1804.10587v1

Martin Zinkevich. Online Convex Programming and Generalized Infinitesimal Gradient Ascent. 2003.

Ashia C. Wilson, Rebecca Roelofs, Mitchell Stern, Nathan Srebro, Benjamin Recht. The Marginal Value of Adaptive Gradient Methods in Machine Learning. 2017. arXiv:1705.08292v2

John Duchi, Elad Hazan, and Yoram Singer. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. Journal of Machine Learning Research, 12:2121–2159, 2011.

Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. COURSERA: neural networks for machine learning, 4(2):26–31, 2012.