An overview of gradient descent optimization algorithms 🐦

Note: If you are looking for a review paper, this blog post is also available as an article on arXiv .

Table of contents:

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 , caffe's , and keras' 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 blog post aims at providing you with intuitions towards the behaviour of different algorithms for optimizing gradient descent that will help you put them to use. We are first going to look at the different variants of gradient descent. We will then briefly summarize challenges during training. Subsequently, 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. We will also 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.

Gradient descent is a way to minimize an objective function $J(\theta)$ parameterized by a model's parameters $\theta \in \mathbb{R}^d$ by updating the parameters in the opposite direction of the gradient of the objective function $\nabla_\theta J(\theta)$ w.r.t. to the parameters. The learning rate η 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. If you are unfamiliar with gradient descent, you can find a good introduction on optimizing neural networks here .

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.

Batch gradient descent

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

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta) .$$

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 don't fit in memory. Batch gradient descent also doesn't 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. (See here for some great tips on how to check gradients properly.)

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.

Stochastic gradient descent

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

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i)}; y^{(i)}).$$

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 Image 1.
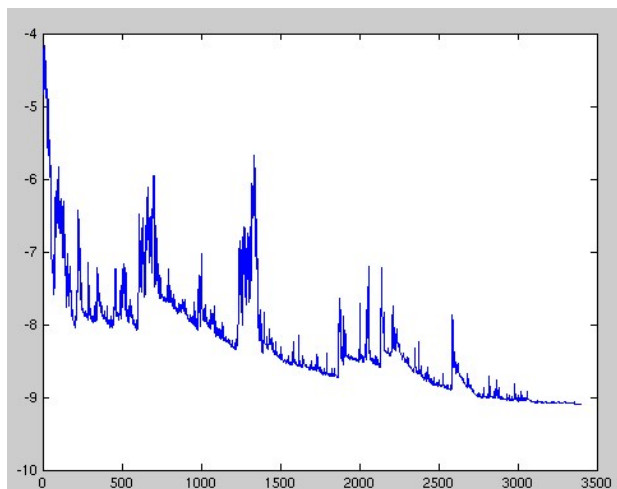

Image 1: SGD fluctuation (Source: Wikipedia)

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 this section.

```
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
```

Mini-batch gradient descent

Mini-batch gradient descent finally takes the best of both worlds and performs an update for every mini-batch of n training examples:

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i:i+n)}; y^{(i:i+n)}) \,.$$

This way, it a) reduces the variance of the parameter updates, which can lead to more stable convergence; and b) can make use of highly optimized matrix optimizations common to state-of-the-art deep learning libraries that make computing the gradient w.r.t. a mini-batch very efficient. Common mini-batch sizes range between 50 and 256, but can vary for different applications. Mini-batch gradient descent is typically the algorithm of choice when training a neural network and the term SGD usually is employed also when mini-batches are used. Note: In modifications of SGD in the rest of this post, we leave out the parameters $x^{(i:i+n)}; y^{(i:i+n)}$ for simplicity.

In code, instead of iterating over examples, we now iterate over mini-batches of size 50:

```python
for i in range(nb_epochs):
  np.random.shuffle(data)
  for batch in get_batches(data, batch_size=50):
    params_grad = evaluate_gradient(loss_function, batch, params)
    params = params - learning_rate * params_grad
```

Challenges

Vanilla mini-batch gradient descent, however, does not guarantee good convergence, but offers a few challenges that need to be addressed:

- Choosing a proper learning rate can be difficult. A learning rate that is too small leads to painfully slow convergence, while a learning rate that is too large can hinder convergence and cause the loss function to fluctuate around the minimum or even to diverge.

- Learning rate schedules [11] try to adjust the learning rate during training by e.g. annealing, i.e. reducing the learning rate according to a pre-defined schedule or when the change in objective between epochs falls below a threshold. These schedules and thresholds, however, have to be defined in advance and are thus unable to adapt to a dataset's characteristics [10].

- Additionally, the same learning rate applies to all parameter updates. If our data is sparse and our features have very different frequencies, we might not want to update all of them to the same extent, but perform a larger update for rarely occurring features.

- Another key challenge of minimizing highly non-convex error functions common for neural networks is avoiding getting trapped in their numerous suboptimal local minima. Dauphin et al. [19] argue that the difficulty arises in fact not from local minima but from saddle points, i.e. points where one dimension slopes up and another slopes down. These saddle points are usually surrounded by a plateau of the same error, which makes it notoriously hard for SGD to escape, as the gradient is close to zero in all dimensions.

Gradient descent optimization algorithms

In the following, we will outline some algorithms that are widely used by the deep learning community to deal with the aforementioned challenges. We will not discuss algorithms that are infeasible to compute in practice for high-dimensional data sets, e.g. second-order methods such as Newton's method .

Momentum

SGD has trouble navigating ravines, i.e. areas where the surface curves much more steeply in one dimension than in another [1], which are common around local optima. In these scenarios, SGD oscillates across the slopes of the ravine while only making hesitant progress along the bottom towards the local optimum as in Image 2.
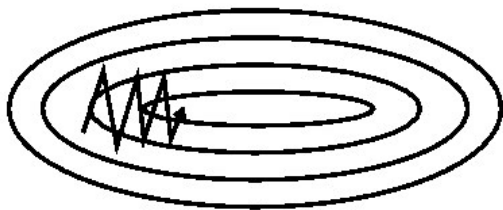

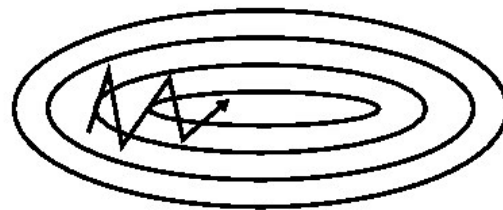Image 2: SGD without momentum   Image 3: SGD with momentum

Momentum [2] is a method that helps accelerate SGD in the relevant direction and dampens oscillations as can be seen in Image 3. It does this by adding a fraction γ of the update vector of the past time step to the current update vector:

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta).$$

$$\theta = \theta - v_t.$$

Note: Some implementations exchange the signs in the equations. The momentum term γ is usually set to 0.9 or a similar value.

Essentially, when using momentum, we push a ball down a hill. The ball accumulates momentum as it rolls downhill, becoming faster and faster on the way (until it reaches its terminal velocity if there is air resistance, i.e. $\gamma < 1$). The same thing happens to our parameter updates: The momentum term increases for dimensions whose gradients point in the same directions and reduces updates for dimensions whose gradients change directions. As a result, we gain faster convergence and reduced oscillation.

Nesterov accelerated gradient

However, a ball that rolls down a hill, blindly following the slope, is highly unsatisfactory. We'd like to have a smarter ball, a ball that has a notion of where it is going so that it knows to slow down before the hill slopes up again.

Nesterov accelerated gradient (NAG) [7] is a way to give our momentum term this kind of prescience. We know that we will use our momentum term γv_{t-1} to move the parameters θ. Computing $\theta - \gamma v_{t-1}$ thus gives us an approximation of the next position of the parameters (the gradient is missing for the full update), a rough idea where our parameters are going to be. We can now effectively look ahead by calculating the gradient not w.r.t. to our current parameters θ but w.r.t. the approximate future position of our parameters:

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1}).$$

$$\theta = \theta - v_t.$$

Again, we set the momentum term γ to a value of around 0.9. While Momentum first computes the current gradient (small blue vector in Image 4) and then takes a big jump in the direction of the updated accumulated

gradient (big blue vector), NAG first makes a big jump in the direction of the previous accumulated gradient (brown vector), measures the gradient and then makes a correction (red vector), which results in the complete NAG update (green vector). This anticipatory update prevents us from going too fast and results in increased responsiveness, which has significantly increased the performance of RNNs on a number of tasks [8].
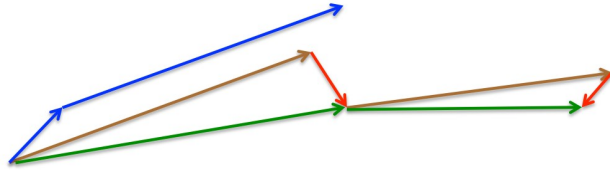

Image 4: Nesterov update (Source: G. Hinton's lecture 6c)

Refer to here for another explanation about the intuitions behind NAG, while Ilya Sutskever gives a more detailed overview in his PhD thesis [9].

Now that we are able to adapt our updates to the slope of our error function and speed up SGD in turn, we would also like to adapt our updates to each individual parameter to perform larger or smaller updates depending on their importance.

Adagrad

Adagrad [3] is an algorithm for gradient-based optimization that does just this: It adapts the learning rate to the parameters, performing larger updates for infrequent and smaller updates for frequent parameters. For this reason, it is well-suited for dealing with sparse data. Dean et al. [4] have found that Adagrad greatly improved the robustness of SGD and used it for training large-scale neural nets at Google, which -- among other things -- learned to recognize cats in Youtube videos . Moreover, Pennington et al. [5] used Adagrad to train GloVe word embeddings, as infrequent words require much larger updates than frequent ones.

Previously, we performed an update for all parameters θ at once as every parameter θ_i used the same learning rate η. As Adagrad uses a different learning rate for every parameter θ_i at every time step t, we first show Adagrad's per-parameter update, which we then vectorize. For brevity, we set $g_{t,i}$ to be the gradient of the objective function w.r.t. to the parameter θ_i at time step t:

$$g_{t,i} = \nabla_\theta J(\theta_i) .$$

The SGD update for every parameter θ_i at each time step t then becomes:

$$\theta_{t+1,i} = \theta_{t,i} - \eta \cdot g_{t,i} .$$

In its update rule, Adagrad modifies the general learning rate η at each time step t for every parameter θ_i based on the past gradients that have been computed for θ_i:

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i} .$$

$G_t \in \mathbb{R}^{d \times d}$ here is a diagonal matrix where each diagonal element i, i is the sum of the squares of the gradients w.r.t. θ_i up to time step t 24 , while ϵ is a smoothing term that avoids division by zero (usually on the order of $1e-8$). Interestingly, without the square root operation, the algorithm performs much worse.

As G_t contains the sum of the squares of the past gradients w.r.t. to all parameters θ along its diagonal, we can now vectorize our implementation by performing an element-wise matrix-vector multiplication $\odot$ between G_t and g_t:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t .$$

One of Adagrad's main benefits is that it eliminates the need to manually tune the learning rate. Most implementations use a default value of 0.01 and leave it at that.

Adagrad's main weakness is its accumulation of the squared gradients in the denominator: Since every added term is positive, the accumulated sum keeps growing during training. This in turn causes the learning rate to shrink and eventually become infinitesimally small, at which point the algorithm is no longer able to acquire additional knowledge. The following algorithms aim to resolve this flaw.

Adadelta

Adadelta [6] is an extension of Adagrad that seeks to reduce its aggressive, monotonically decreasing learning rate. Instead of accumulating all past squared gradients, Adadelta restricts the window of accumulated past gradients to some fixed size w.

Instead of inefficiently storing w previous squared gradients, the sum of gradients is recursively defined as a decaying average of all past squared gradients. The running average $E[g^2]_t$ at time step t then depends (as a fraction γ similarly to the Momentum term) only on the previous average and the current gradient:

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

We set γ to a similar value as the momentum term, around 0.9. For clarity, we now rewrite our vanilla SGD update in terms of the parameter update vector $\Delta\theta_t$:

$$\Delta\theta_t = -\eta \cdot g_{t,i}.$$

$$\theta_{t+1} = \theta_t + \Delta\theta_t.$$

The parameter update vector of Adagrad that we derived previously thus takes the form:

$$\Delta\theta_t = -\frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t.$$

We now simply replace the diagonal matrix G_t with the decaying average over past squared gradients $E[g^2]_t$:

$$\Delta\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t.$$

As the denominator is just the root mean squared (RMS) error criterion of the gradient, we can replace it with the criterion short-hand:

$$\Delta\theta_t = -\frac{\eta}{RMS[g]_t} g_t.$$

The authors note that the units in this update (as well as in SGD, Momentum, or Adagrad) do not match, i.e. the update should have the same hypothetical units as the parameter. To realize this, they first define another exponentially decaying average, this time not of squared gradients but of squared parameter updates:

$$E[\Delta\theta^2]_t = \gamma E[\Delta\theta^2]_{t-1} + (1-\gamma)\Delta\theta_t^2.$$

The root mean squared error of parameter updates is thus:

$$RMS[\Delta\theta]_t = \sqrt{E[\Delta\theta^2]_t + \epsilon}.$$

Since $RMS[\Delta\theta]_t$ is unknown, we approximate it with the RMS of parameter updates until the previous time step. Replacing the learning rate η in the previous update rule with $RMS[\Delta\theta]_{t-1}$ finally yields the Adadelta update rule:

$$\Delta\theta_t = -\frac{RMS[\Delta\theta]_{t-1}}{RMS[g]_t} g_t.$$

$$\theta_{t+1} = \theta_t + \Delta\theta_t.$$

With Adadelta, we do not even need to set a default learning rate, as it has been eliminated from the update rule.

RMSprop

RMSprop is an unpublished, adaptive learning rate method proposed by Geoff Hinton in Lecture 6e of his Coursera Class .

RMSprop and Adadelta have both been developed independently around the same time stemming from the need to resolve Adagrad's radically diminishing learning rates. RMSprop in fact is identical to the first update vector of Adadelta that we derived above:

$$E[g^2]_t = 0.9E[g^2]_{t-1} + 0.1g_t^2 .$$

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

RMSprop as well divides the learning rate by an exponentially decaying average of squared gradients. Hinton suggests γ to be set to 0.9, while a good default value for the learning rate η is 0.001.

Adam

Adaptive Moment Estimation (Adam) [15] is another method that computes adaptive learning rates for each parameter. In addition to storing an exponentially decaying average of past squared gradients v_t like Adadelta and RMSprop, Adam also keeps an exponentially decaying average of past gradients m_t , similar to momentum:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1)g_t .$$

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

m_t and v_t are estimates of the first moment (the mean) and the second moment (the uncentered variance) of the gradients respectively, hence the name of the method. As m_t and v_t are initialized as vectors of 0's, the authors of Adam observe that they are biased towards zero, especially during the initial time steps, and especially when the decay rates are small (i.e. β_1 and β_2 are close to 1).

They counteract these biases by computing bias-corrected first and second moment estimates:

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

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

They then use these to update the parameters just as we have seen in Adadelta and RMSprop, which yields the Adam update rule:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t .$$

The authors propose default values of 0.9 for β_1 , 0.999 for β_2 , and 10^{-8} for ϵ. They show empirically that Adam works well in practice and compares favorably to other adaptive learning-method algorithms.

Visualization of algorithms

The following two animations (Image credit: Alec Radford) provide some intuitions towards the optimization behaviour of the presented optimization algorithms. Also have a look here for a description of the same images by Karpathy and another concise overview of the algorithms discussed.

In Image 5, we see their behaviour on the contours of a loss surface over time. Note that Adagrad, Adadelta,

and RMSprop almost immediately head off in the right direction and converge similarly fast, while Momentum and NAG are led off-track, evoking the image of a ball rolling down the hill. NAG, however, is quickly able to correct its course due to its increased responsiveness by looking ahead and heads to the minimum.

Image 6 shows the behaviour of the algorithms at a saddle point, i.e. a point where one dimension has a positive slope, while the other dimension has a negative slope, which pose a difficulty for SGD as we mentioned before. Notice here that SGD, Momentum, and NAG find it difficulty to break symmetry, although the two latter eventually manage to escape the saddle point, while Adagrad, RMSprop, and Adadelta quickly head down the negative slope.
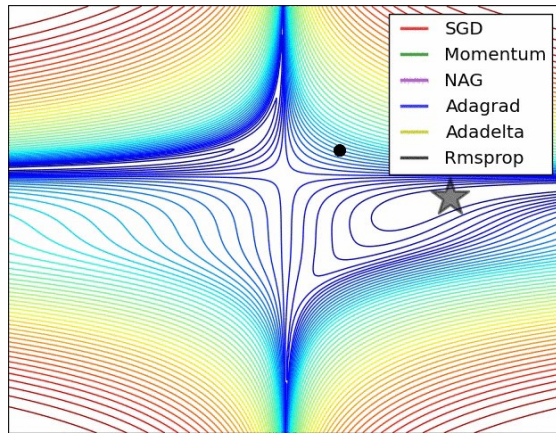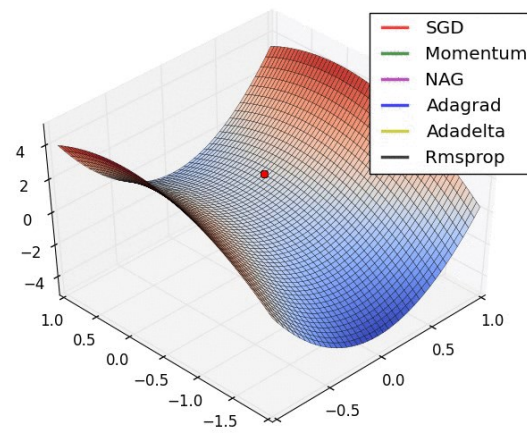

Image 5: SGD optimization on loss surface contours   Image 6: SGD optimization on saddle point

As we can see, the adaptive learning-rate methods, i.e. Adagrad, Adadelta, RMSprop, and Adam are most suitable and provide the best convergence for these scenarios.

Which optimizer to use?

So, which optimizer should you now use? If your input data is sparse, then you likely achieve the best results using one of the adaptive learning-rate methods. An additional benefit is that you won't need to tune the learning rate but likely achieve the best results with the default value.

In summary, RMSprop is an extension of Adagrad that deals with its radically diminishing learning rates. It is identical to Adadelta, except that Adadelta uses the RMS of parameter updates in the numinator update rule. Adam, finally, adds bias-correction and momentum to RMSprop. Insofar, RMSprop, Adadelta, and Adam are very similar algorithms that do well in similar circumstances. Kingma et al. [15] show that its bias-correction helps Adam slightly outperform RMSprop towards the end of optimization as gradients become sparser. Insofar, Adam might be the best overall choice.

Interestingly, many recent papers use vanilla SGD without momentum and a simple learning rate annealing schedule. As has been shown, SGD usually achieves to find a minimum, but it might take significantly longer than with some of the optimizers, is much more reliant on a robust initialization and annealing schedule, and may get stuck in saddle points rather than local minima. Consequently, if you care about fast convergence and train a deep or complex neural network, you should choose one of the adaptive learning rate methods.

Parallelizing and distributing SGD

Given the ubiquity of large-scale data solutions and the availability of low-commodity clusters, distributing SGD to speed it up further is an obvious choice.

SGD by itself is inherently sequential: Step-by-step, we progress further towards the minimum. Running it provides good convergence but can be slow particularly on large datasets. In contrast, running SGD asynchronously is faster, but suboptimal communication between workers can lead to poor convergence. Additionally, we can also parallelize SGD on one machine without the need for a large computing cluster. The following are algorithms and architectures that have been proposed to optimize parallelized and distributed

SGD.

Hogwild!

Niu et al. [23] introduce an update scheme called Hogwild! that allows performing SGD updates in parallel on CPUs. Processors are allowed to access shared memory without locking the parameters. This only works if the input data is sparse, as each update will only modify a fraction of all parameters. They show that in this case, the update scheme achieves almost an optimal rate of convergence, as it is unlikely that processors will overwrite useful information.

Downpour SGD

Downpour SGD is an asynchronous variant of SGD that was used by Dean et al. [4] in their DistBelief framework (predecessor to TensorFlow) at Google. It runs multiple replicas of a model in parallel on subsets of the training data. These models send their updates to a parameter server, which is split across many machines. Each machine is responsible for storing and updating a fraction of the model's parameters. However, as replicas don't communicate with each other e.g. by sharing weights or updates, their parameters are continuously at risk of diverging, hindering convergence.

Delay-tolerant Algorithms for SGD

McMahan and Streeter [12] extend AdaGrad to the parallel setting by developing delay-tolerant algorithms that not only adapt to past gradients, but also to the update delays. This has been shown to work well in practice.

TensorFlow

TensorFlow [13] is Google's recently open-sourced framework for the implementation and deployment of large-scale machine learning models. It is based on their experience with DistBelief and is already used internally to perform computations on a large range of mobile devices as well as on large-scale distributed systems. For distributed execution, a computation graph is split into a subgraph for every device and communication takes place using Send/Receive node pairs. However, the open source version of TensorFlow currently does not support distributed functionality (see here). Update 13.04.16: A distributed version of TensorFlow has been released .

Elastic Averaging SGD

Zhang et al. [14] propose Elastic Averaging SGD (EASGD), which links the parameters of the workers of asynchronous SGD with an elastic force, i.e. a center variable stored by the parameter server. This allows the local variables to fluctuate further from the center variable, which in theory allows for more exploration of the parameter space. They show empirically that this increased capacity for exploration leads to improved performance by finding new local optima.

Additional strategies for optimizing SGD

Finally, we introduce additional strategies that can be used alongside any of the previously mentioned algorithms to further improve the performance of SGD. For a great overview of some other common tricks, refer to [22].

Shuffling and Curriculum Learning

Generally, we want to avoid providing the training examples in a meaningful order to our model as this may bias the optimization algorithm. Consequently, it is often a good idea to shuffle the training data after every epoch.

On the other hand, for some cases where we aim to solve progressively harder problems, supplying the training examples in a meaningful order may actually lead to improved performance and better convergence. The method for establishing this meaningful order is called Curriculum Learning [16].

Zaremba and Sutskever [17] were only able to train LSTMs to evaluate simple programs using Curriculum Learning and show that a combined or mixed strategy is better than the naive one, which sorts examples by increasing difficulty.

Batch normalization

To facilitate learning, we typically normalize the initial values of our parameters by initializing them with zero mean and unit variance. As training progresses and we update parameters to different extents, we lose this normalization, which slows down training and amplifies changes as the network becomes deeper.

Batch normalization [18] reestablishes these normalizations for every mini-batch and changes are back-propagated through the operation as well. By making normalization part of the model architecture, we are able to use higher learning rates and pay less attention to the initialization parameters. Batch normalization additionally acts as a regularizer, reducing (and sometimes even eliminating) the need for Dropout.

Early stopping

According to Geoff Hinton: "Early stopping (is) beautiful free lunch" (NIPS 2015 Tutorial slides , slide 63). You should thus always monitor error on a validation set during training and stop (with some patience) if your validation error does not improve enough.

Gradient noise

Neelakantan et al. [21] add noise that follows a Gaussian distribution $N(0, \sigma_t^2)$ to each gradient update:

$$g_{t,i} = g_{t,i} + N(0, \sigma_t^2) \,.$$

They anneal the variance according to the following schedule:

$$\sigma_t^2 = \frac{\eta}{(1+t)^\gamma} \,.$$

They show that adding this noise makes networks more robust to poor initialization and helps training particularly deep and complex networks. They suspect that the added noise gives the model more chances to escape and find new local minima, which are more frequent for deeper models.

Conclusion

In this blog post, we have initially looked at the three variants of gradient descent, among which mini-batch gradient descent is the most popular. We have then investigated algorithms that are most commonly used for optimizing SGD: Momentum, Nesterov accelerated gradient, Adagrad, Adadelta, RMSprop, Adam, as well as different algorithms to optimize asynchronous SGD. Finally, we've considered other strategies to improve SGD such as shuffling and curriculum learning, batch normalization, and early stopping.

I hope that this blog post was able to provide you with some intuitions towards the motivation and the

behaviour of the different optimization algorithms. Are there any obvious algorithms to improve SGD that I've missed? What tricks are you using yourself to facilitate training with SGD? **Let me know in the comments below.**

Acknowledgements

Thanks to Denny Britz and Cesar Salgado for reading drafts of this post and providing suggestions.

Printable version and citation

This blog post is also available as an article on arXiv, in case you want to refer to it later.

In case you found it helpful, consider citing the corresponding arXiv article as:
Sebastian Ruder (2016). An overview of gradient descent optimisation algorithms. arXiv preprint arXiv:1609.04747.

Translations

This blog post has been translated into the following languages:

- Japanese
- Chinese

Update 21.06.16: This post was posted to Hacker News. The discussion provides some interesting pointers to related work and other techniques.

References

1. Sutton, R. S. (1986). Two problems with backpropagation and other steepest-descent learning procedures for networks. Proc. 8th Annual Conf. Cognitive Science Society. ↩

2. Qian, N. (1999). On the momentum term in gradient descent learning algorithms. Neural Networks : The Official Journal of the International Neural Network Society, 12(1), 145–151. http://doi.org/10.1016/S0893-6080(98)00116-6 ↩

3. Duchi, J., Hazan, E., & Singer, Y. (2011). Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. Journal of Machine Learning Research, 12, 2121–2159. Retrieved from http://jmlr.org/papers/v12/duchi11a.html ↩

4. Dean, J., Corrado, G. S., Monga, R., Chen, K., Devin, M., Le, Q. V, … Ng, A. Y. (2012). Large Scale Distributed Deep Networks. NIPS 2012: Neural Information Processing Systems, 1–11. http://doi.org/10.1109/ICDAR.2011.95 ↩

5. Pennington, J., Socher, R., & Manning, C. D. (2014). Glove: Global Vectors for Word Representation. Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, 1532–1543. http://doi.org/10.3115/v1/D14-1162 ↩

6. Zeiler, M. D. (2012). ADADELTA: An Adaptive Learning Rate Method. Retrieved from

http://arxiv.org/abs/1212.5701

7. Nesterov, Y. (1983). A method for unconstrained convex minimization problem with the rate of convergence o(1/k2). Doklady ANSSSR (translated as Soviet.Math.Docl.), vol. 269, pp. 543– 547.

8. Bengio, Y., Boulanger-Lewandowski, N., & Pascanu, R. (2012). Advances in Optimizing Recurrent Networks. Retrieved from http://arxiv.org/abs/1212.0901

9. Sutskever, I. (2013). Training Recurrent neural Networks. PhD Thesis.

10. Darken, C., Chang, J., & Moody, J. (1992). Learning rate schedules for faster stochastic gradient search. Neural Networks for Signal Processing II Proceedings of the 1992 IEEE Workshop, (September), 1–11. http://doi.org/10.1109/NNSP.1992.253713

11. H. Robinds and S. Monro, "A stochastic approximation method," Annals of Mathematical Statistics, vol. 22, pp. 400–407, 1951.

12. Mcmahan, H. B., & Streeter, M. (2014). Delay-Tolerant Algorithms for Asynchronous Distributed Online Learning. Advances in Neural Information Processing Systems (Proceedings of NIPS), 1–9. Retrieved from http://papers.nips.cc/paper/5242-delay-tolerant-algorithms-for-asynchronous-distributed-online-learning.pdf

13. Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., … Zheng, X. (2015). TensorFlow : Large-Scale Machine Learning on Heterogeneous Distributed Systems.

14. Zhang, S., Choromanska, A., & LeCun, Y. (2015). Deep learning with Elastic Averaging SGD. Neural Information Processing Systems Conference (NIPS 2015), 1–24. Retrieved from http://arxiv.org/abs/1412.6651

15. Kingma, D. P., & Ba, J. L. (2015). Adam: a Method for Stochastic Optimization. International Conference on Learning Representations, 1–13.

16. Bengio, Y., Louradour, J., Collobert, R., & Weston, J. (2009). Curriculum learning. Proceedings of the 26th Annual International Conference on Machine Learning, 41–48. http://doi.org/10.1145/1553374.1553380

17. Zaremba, W., & Sutskever, I. (2014). Learning to Execute, 1–25. Retrieved from http://arxiv.org/abs/1410.4615

18. Ioffe, S., & Szegedy, C. (2015). Batch Normalization : Accelerating Deep Network Training by Reducing Internal Covariate Shift. arXiv Preprint arXiv:1502.03167v3.

19. Dauphin, Y., Pascanu, R., Gulcehre, C., Cho, K., Ganguli, S., & Bengio, Y. (2014). Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. arXiv, 1–14. Retrieved from http://arxiv.org/abs/1406.2572

20. Sutskever, I., & Martens, J. (2013). On the importance of initialization and momentum in deep learning. http://doi.org/10.1109/ICASSP.2013.6639346

21. Neelakantan, A., Vilnis, L., Le, Q. V., Sutskever, I., Kaiser, L., Kurach, K., & Martens, J. (2015). Adding Gradient Noise Improves Learning for Very Deep Networks, 1–11. Retrieved from http://arxiv.org/abs/1511.06807

22. LeCun, Y., Bottou, L., Orr, G. B., & Müller, K. R. (1998). Efficient BackProp. Neural Networks: Tricks of the Trade, 1524, 9–50. http://doi.org/10.1007/3-540-49430-8_2 ⮐

23. Niu, F., Recht, B., Christopher, R., & Wright, S. J. (2011). Hogwild! : A Lock-Free Approach to Parallelizing Stochastic Gradient Descent, 1–22. ⮐

24. Duchi et al. [3] give this matrix as an alternative to the full matrix containing the outer products of all previous gradients, as the computation of the matrix square root is infeasible even for a moderate number of parameters $_d$. ⮐

Image credit for cover photo: Karpathy's beautiful loss functions tumblr

---

106 Comments   **Blog**   1 Login ▾

♡ Recommend 33   ⤴ Share   Sort by Best ▾

[Join the discussion…]

DJ Strouse • 3 months ago   — | ⚑
Super helpful even a year out! You should've written the optimization section of Goodfellow, Bengio, & Courville. Please write more!
1 ⌃ | ⌄ • Reply • Share ›

> **Alexander Chrystopher Harringt** → DJ Strouse • 2 months ago   — | ⚑
> I too really found this helpful!
> It provided some great insights into why I experience certain things when using different optimizers.
>
> Great work!
> ⌃ | ⌄ • Reply • Share ›

> **Sebastian Ruder** Mod → DJ Strouse • 2 months ago   — | ⚑
> Thanks a lot, DJ! I'm definitely trying to write more -- particularly with such formidable encouragement.
> ⌃ | ⌄ • Reply • Share ›

justwantanaccount • 3 months ago   — | ⚑
Thank you so much for such a great article!
1 ⌃ | ⌄ • Reply • Share ›

> **Sebastian Ruder** Mod → justwantanaccount • 3 months ago   — | ⚑
> Thanks a lot! :)
> ⌃ | ⌄ • Reply • Share ›

Adam Iou-Jen Liu • 3 months ago   — | ⚑
Very clear explanation, very helpful!
Thank you!
1 ⌃ | ⌄ • Reply • Share ›

> **Sebastian Ruder** Mod → Adam Iou-Jen Liu • 3 months ago   — | ⚑
> Hey Adam,
> Thanks! Glad it was helpful! :)
> ⌃ | ⌄ • Reply • Share ›

Jules Wombat • 4 days ago   — | ⚑
Excellent, I really needed this explanation as a newbie to NN, and the Optimisers setup seemed to bea bit of black art. Many Keras tutorials don't really explain the effects of different Optimisers, which can have a very dramtic effect on NN convergence perfomance. Not sure why SGD is proposed in so many tutorials, without clear guidance on Schedule adding to teh complexity.
⌃ | ⌄ • Reply • Share ›

Сергей Полянских • 21 days ago   — | ⚑
Very good and quite elementary explanation. Thank you

^ | ∨ • Reply • Share ›

Sebastian Ruder `Mod` ➤ Сергей Полянских • 21 days ago
Thanks! Glad you enjoyed it! :)
^ | ∨ • Reply • Share ›

Jennifer Z • 22 days ago
Thank you very much.
^ | ∨ • Reply • Share ›

Sebastian Ruder `Mod` ➤ Jennifer Z • 21 days ago
Thanks! Glad you found it helpful! :)
^ | ∨ • Reply • Share ›

Florin Lucaciu • a month ago
I thought Batch Normalization is on the activations, not the parameters. Am I wrong ?
^ | ∨ • Reply • Share ›

Sebastian Ruder `Mod` ➤ Florin Lucaciu • a month ago
Hey Florin,
From the paper (Section 2, line 1): "We define Internal Covariate Shift as the change in the distribution of network activations due to the change in network parameters during training". So there's a change in the network activations, which is caused by the changing network parameters. In order to mitigate the change in activations, we thus normalize the network parameters.
^ | ∨ • Reply • Share ›

Florin Lucaciu ➤ Sebastian Ruder • a month ago
Actually, I still don`t think this is true. Normalizing parameters just doesn`t make sense (doesn`t this mess with the learning process?) .. or maybe we have different understanding of "parameters".. to me they are the set of weights and biases .. Section 3 Algorithm 1 in the original paper: no change to gamma and beta (the parameters to be learned), only on x-activations of previous layer. Also, according to Keras: "Normalize the activations of the previous layer at each batch, i.e. applies a transformation that maintains the mean activation close to 0 and the activation standard deviation close to 1." - https://keras.io/layers/nor...

I`m writing a report and the theory section includes BN, that`s why I wanted to be sure. Thank you for your answer !
^ | ∨ • Reply • Share ›

Melwin Jose • 2 months ago
I liked the way you explained the optimizations using easy to understand examples. Keep writing ;)
^ | ∨ • Reply • Share ›

Sebastian Ruder `Mod` ➤ Melwin Jose • 2 months ago
Hey Melwin, thanks a lot! Really appreciate it! :)
^ | ∨ • Reply • Share ›

Charles • 3 months ago
Hi,
Here a good description of some available algorithm.
Just a small question about the adam method. I understood that g_t and g_t^2 are vectors (because they are gradients)
So m_t and v_t are also vector.
But how do you evaluate $\dfrac{m_t}{\sqrt{v_t}+\epsilon}$?

Thank's for your answer
^ | ∨ • Reply • Share ›

Charles ➤ Charles • 3 months ago
Ok, I found it! It's element-wise operations!
May be it will be usefull for someone!
^ | ∨ • Reply • Share ›

Siddarth Sampangi • 4 months ago
Hi Sebastian,

First of all, thanks for the great post. It's really useful to understand the different optimizers on an intuitive level and in comparison with each other.

I was able to follow you until your description of Adadelta. What I'm getting caught up on is the step where you switch root(E[g^2]_t) to RMS[g_t]. I've read the paper, but the authors go through that step with even less of an explanation than you haha. To me, that change makes it look like we're going from a formulation where past gradients are heavily discounted due to the \rho value to a formulation where we just take the RMS of all past gradients.

I'm actually also unsure about the E[g^2]_t term. Is this supposed to be read as "the expected value of the square of the gradient at timestep t"?

∧ | ∨ · Reply · Share ›

Siddarth Sampangi → Siddarth Sampangi · 4 months ago

So I think I was overcomplicating things. They called it RMS simply because of the act of squaring the gradient, adding it to the E[g^2]_t-1 term (after weighting with \rho) and taking the square root.

∧ | ∨ · Reply · Share ›

Sebastian Ruder Mod → Siddarth Sampangi · 4 months ago

Hey Siddarth,
Sorry for my late reply. Yep, that's correct. It's designated as RMS simply because the gradients are squared and we take the square root.

∧ | ∨ · Reply · Share ›

Tianxiang Gao · 5 months ago

Very good review. Compact and comprehensive. How did you get image 5 and 6? It looks so cool. i hope I can get such figures in my work.

∧ | ∨ · Reply · Share ›

Sebastian Ruder Mod → Tianxiang Gao · 5 months ago

Thanks! The images are actually created by Alec Radford as attributed in the text. They're very cool indeed. :)

∧ | ∨ · Reply · Share ›

Cassandra · 5 months ago

Fantastic article, thank you!

I believe there is a typo in the Nesterov update description (the correction vectors are "red" not "green").

∧ | ∨ · Reply · Share ›

Sebastian Ruder Mod → Cassandra · 5 months ago

Hey Cassandra, thank you! You're right. I've made a correction to make this clearer.

∧ | ∨ · Reply · Share ›

Nate · 5 months ago

Great article! Sebastian, after reading your article it is clear to me how ADAM/RMSprop/Adadelta work to handle the issue of the squared gradients in the denominator when using Adagrad. However, I am training a deep+wide neural network (regression), using tensorflow, with both continuous and categorical variables and have found that using ANY of the adaptive learning optimizers (ADAM/RMSprop/Adadelta) works very well in minimizing the loss but my prediction accuracy gets lower and lower with each step. Adagrad has a decreasing loss but with an increasing prediction accuracy with each step. The loss optimization with the adaptive learning optimizer is much better than Adagrad, but it definitely does not correspond to a higher/increasing accuracy training. I have attached a picture of the loss. Purple = Adagrad, Green and Orange = ADAM, Light Blue = Adadelta. Purple is the only one that has an increasing accuracy with each step...both of the others have a decreasing accuracy with each step. Why? Is this a property of the adaptive methods? I would love to have the Orange loss curve with an increasing prediction accuracy!!

see more

∧ | ∨ · Reply · Share ›

Sebastian Ruder Mod → Nate · 5 months ago

Hey Nate, that's an interesting question. I assume you're training some sort of a recommender system as you're using a deep+wide neural network. It seems to me that the issue you're facing seems to have less to do with the behaviour of the optimizers (they minimize the training loss, as they should) and more to do with

the correlation between your objective function (MSE?) and your evaluation metric (accuracy). Often, these don't correlate that well, e.g. lower log-likelihood values don't always equate to higher BLEU scores in MT. It seems strange, though, that this behaviour differs in your case across different optimizers. It would be helpful if you would plot your training accuracy on the same plot as your training loss so we can see how the accuracy develops exactly. Additionally, you might want to try swapping your objective function with another one (e.g. MSE vs. cross-entropy) and see if that changes things in any way.

⌃ | ⌄ · Reply · Share ›

Nate → Sebastian Ruder · 5 months ago

Sebastian - You make a great point. I had previously laid out the tensorflow graphs with included specifying the cost function for the optimizer. Recently, however, I switched to using tensorflow.contrib.learn. I had assumed that my code below was optimizing the mean squared error - but I realized, after reading your reply, that I really have no clue what it is optimizing! I will have to do some digging to 1. find out what is the default loss/objective function and 2. how to change it, if it is not applicable to my case. I will return with accuracy plots...unless you have any suggestions on where to specify objective function tensor below?

Here's what I return from my 'build_estimator' function:

```
return learn.DNNLinearCombinedRegressor(
model_dir=model_dir,
linear_feature_columns=wide_columns,
linear_optimizer=tf.train.AdamOptimizer(epsilon=0.01),
dnn_feature_columns=deep_columns,
dnn_hidden_units=[hidden_units_l1,hidden_units_l2],
dnn_optimizer=tf.train.AdamOptimizer(epsilon=0.01),
enable_centered_bias=False
)
```

⌃ | ⌄ · Reply · Share ›

Alek · 5 months ago

Very useful article, thank you!

⌃ | ⌄ · Reply · Share ›

Sebastian Ruder Mod → Alek · 5 months ago

Thank you! :)

⌃ | ⌄ · Reply · Share ›

御宅暴君 · 6 months ago

What kind of data is sparse? Does data of a image accounted? What about audio data? text data? etc. Thank you!

⌃ | ⌄ · Reply · Share ›

Sebastian Ruder Mod → 御宅暴君 · 6 months ago

Hey, good question! Data is sparse in general, if it has many dimensions (e.g. 10,000) but only few dimensions have non-zero entries. This is typically true for text data: If you represent a document as a bag-of-words vector, the vector has the dimensionality of the size of the vocabulary, but only has non-zero entries for the words that are in the document. As far as I know, image data is typically not sparse, as you have pixel values for every data point in your image. I'm not too familiar with audio data, but I assume it depends if you choose to analyze wave lengths directly or other parts of the audio input. In real-world scenarios, data also is often sparse, as for your task you might have many attributes or features available, but for each user you only have information about a few of those. Overcoming sparsity in this context is one of the central challenges of recommender systems.

1 ⌃ | ⌄ · Reply · Share ›

御宅暴君 · 6 months ago

Helpful post! But what does w.r.t. mean? Thank you!

⌃ | ⌄ · Reply · Share ›

Sebastian Ruder Mod → 御宅暴君 · 6 months ago

Thanks and good question! w.r.t. means "with regard to".

1 ⌃ | ⌄ · Reply · Share ›

Yingwei Zhou → 御宅暴君 · 6 months ago

I like your name hahaha....

⌃ | ⌄ · Reply · Share ›

Roman Kvasnyj · 6 months ago

Wow! Great summary of algorithms, thank you! But why no visualization of Adam?

^ | ∨ • Reply • Share ›

Sebastian Ruder `Mod` → Roman Kvasnyj • 6 months ago

Thanks! A visualization of Adam might be added in the future.

^ | ∨ • Reply • Share ›

delta1epsilon • 6 months ago

Awesome post!

^ | ∨ • Reply • Share ›

Sebastian Ruder `Mod` → delta1epsilon • 6 months ago

Thanks! :)

^ | ∨ • Reply • Share ›

Aruna Vallurupalli • 7 months ago

Thank you for this excellent write-up, Sebastian! I found both the blog and the comments very informative.

^ | ∨ • Reply • Share ›

Sebastian Ruder `Mod` → Aruna Vallurupalli • 6 months ago

Thank you!

^ | ∨ • Reply • Share ›

孫以恆 • 7 months ago

Really a great job there. Thanks so much!

^ | ∨ • Reply • Share ›

Sebastian Ruder `Mod` → 孫以恆 • 7 months ago

Thank you!

^ | ∨ • Reply • Share ›

Wei Wen • 7 months ago

Great blog! I have a question regarding the adaptive learning rate: why state of art deep neural networks choose to use fixed learning rate (with momentum) step by step, including alexnet vggnet googlenet and resnet? I tried adadelta on alexnet with base rate of 0.01 but the accuracy is 0.26 after 200000 iterations. Thanks!

^ | ∨ • Reply • Share ›

Sebastian Ruder `Mod` → Wei Wen • 6 months ago

It seems that for a large number of epochs, SGD might do better than Adadelta and Adagrad for image classification (e.g. compare your results to the results in Figure 1 a) in http://openreview.net/pdf?i.... Using Adam instead might improve results.

^ | ∨ • Reply • Share ›

Jaromír Janisch • 8 months ago

I appreciate the effort put in writing this article. It's been very helpful, thank you.

^ | ∨ • Reply • Share ›

Sebastian Ruder `Mod` → Jaromír Janisch • 8 months ago

Thanks. :)

^ | ∨ • Reply • Share ›

Marco Baioletti • 8 months ago

Very good survey. There exists a printable version of this paper ? Thanks

^ | ∨ • Reply • Share ›

Sebastian Ruder `Mod` → Marco Baioletti • 8 months ago

The printable version is now available on arXiv here: http://arxiv.org/abs/1609.0...

^ | ∨ • Reply • Share ›

Load more comments

ALSO ON BLOG

Transfer Learning - Machine Learning's Next Frontier

9 comments • 2 months ago

Tim Martin — Very informative post! I would just offer

On word embeddings - Part 1

21 comments • a year ago

Sebastian Ruder — No, word embeddings cannot directly replace languages in MT or other tasks, as they

one criticism on where you touch on psychology: "Toddlers only need to be told once what a dog is in …

only approximate the language modelling objective. …

Lisbon Machine Learning Summer School Highlights

3 comments • 9 months ago•

Vivek Kumar — Got it.Thank you.

An overview of gradient descent optimization algorithms

2 comments • a year ago•

Sebastian Ruder — Hey Liam,Thanks for asking. Sure, feel free to translate the post. It would be great if you could link back to the original and let me know once …