An Empirical Comparison of Optimizers for Machine Learning Models

An in-depth look into optimizers used for machine learning

Rick Wierenga Follow
Dec 4, 2019 · 7 min read

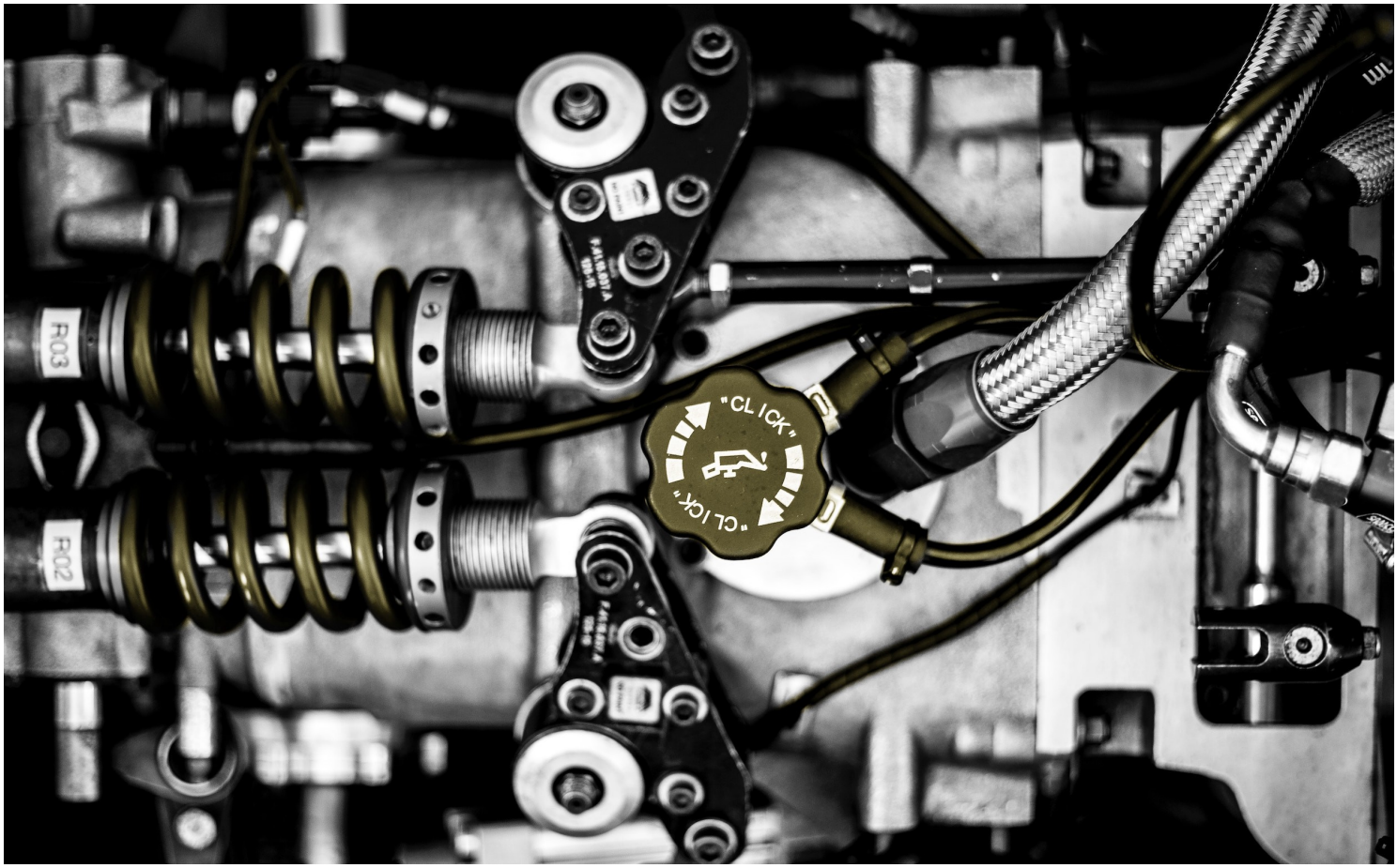
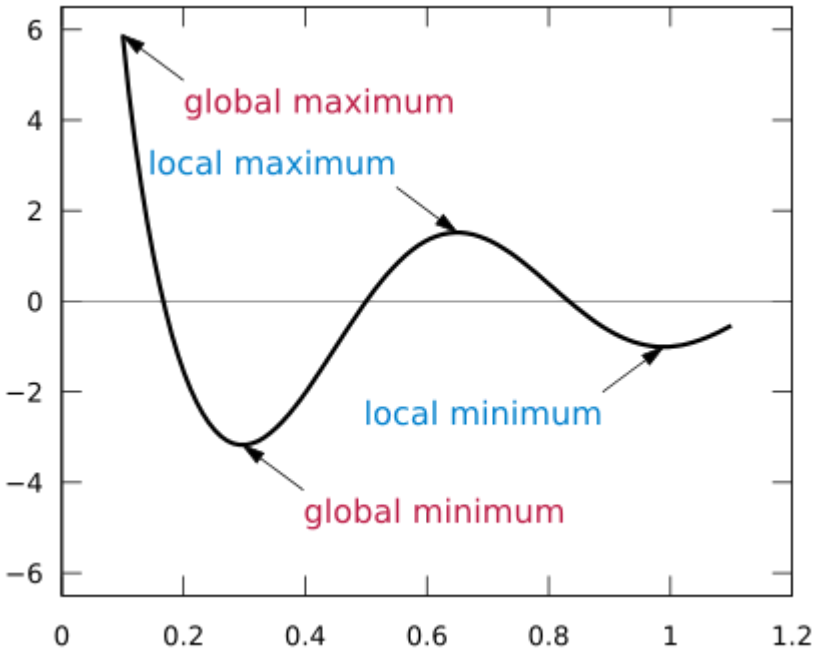


Photo by Sam Loyd on Unsplash

At every point in time during training, a neural network has a certain loss, or error, calculated using a cost function (also referred to as a loss function). This function indicates how 'wrong' the network (parameters) is based on the training or validation

data. Optimally, the loss would be as low as possible. Unfortunately, cost functions are nonconvex — they don't just have one minimum, but many, many local minima.



A non-convex function with one local minimum

To minimize a neural network's loss, an algorithm called backpropagation is used. Backpropagation calculates the derivative of the cost function with respect to the parameters in the neural network. In other words, it finds the "direction" in which to update the parameters so that the model will perform better. This "direction" is called a neural network's **gradient**.

Before updating the model with the gradient, the gradient is multiplied by a learning rate. This yields the actual update on the neural network.. When the learning rate is too high, we might step over the minimum, meaning the model is not as good as it could have been.

But on the other hand, when the learning rate is too low, the optimization process is extremely slow. Another risk of a low learning rate is the fact that the state might end up in a bad local minimum. The model is at a suboptimal state as this point, but it could be much better.

This is where optimizers come in. Most optimizers calculate the learning rate automatically. Optimizers also apply the gradient to the neural network — they make

the network learn. A good optimizer trains models fast, but it also prevents them from getting stuck in a local minimum.

> Optimizers are the engine of machine learning — they make the computer learn.

Over the years, many optimizers have been introduced. In this post, I wanted to explore how they perform, comparatively.

. . .

> Machine learning models are moving closer and closer to edge devices. Fritz AI is here to help with this transition. Explore our suite of developer tools that makes it easy to teach devices to see, hear, sense, and think.

. . .

A quick overview of some popular optimizers

SGD

Stochastic Gradient Descent, or SGD for short, is one of the simplest optimization algorithms. It uses just one static learning rate for all parameters during the entire training phase.

The static learning rate does not imply an equal update after every minibatch. As the optimizers approach an (sub)optimal value, their gradients start to decrease.

AdaGrad

AdaGrad is very similar to SGD. The key difference in design is that AdaGrad uses **Ada**ptive gradients — it has a different learning rate for every single parameter in the neural network. Since not all parameters in a network are equally important, updating them in a different way makes perfect sense.

AdaGrad updates the learning rate for each parameter based on the frequency with which it's updated. Frequently updated parameters are trained very carefully with a low learning rate. Updating these parameters too quickly could result in an irreversible distortion — wasting the parameters usefulness.

Infrequently updated parameters are trained with higher learning rates in an attempt to make them more effective. This risk can be taken, because these parameters are virtually useless in the first place.

Reference: http://jmlr.org/papers/volume12/duchi11a/duchi11a.pdf

RMSProp

AdaGrad has a problem where after a few batches, the learning rates become low — resulting in a long training time. Root Mean Square Propagation (RMSProp), attempts to solve this problem by exponentially decaying the learning rates. This makes RMSProp more volatile.

By using past learning rates, both AdaGrad and RMSProp use momentum for updating. This can be compared to rolling a ball (state of the neural network) down a hill (the graph of a cost function). The longer the ball moves in a certain direction, the faster it goes, giving the ball higher momentum.

The momentum of the ball is useful because it allows the network to 'roll over' local minima — not getting stuck in them.

Reference: https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

Adam

Adam, adaptive moment estimation, also uses past learning rates like AdaGrad and RMSProp do. However, Adam doesn't stop there—it also uses past gradients to speed up

learning. When Adam moves in a certain direction, it does so with a 'great force'— it's not suddenly going to stop and turn around.

In the ball rolling down the hill analogy, Adam would be a weighty ball.

Reference:

Adam: A Method for Stochastic Optimization

We introduce Adam, an algorithm for first-order gradient-based optimization of stochastic objective functions, based on...

arxiv.org

. . .

. . .

Benchmarking Optimizers

While the above sheds light on the theoretical performance of each optimizer, I wanted to see for myself how they performed.

To get a more accurate insight into the performance, I've assessed each of the following metrics, including the graphs, on an average of 10 runs.

Each optimizer is configured with the default hyperparameters of TensorFlow. SGD has a learning rate of 0.01, and doesn't use momentum.

AdaGrad has an learning rate of 0.001, an initial accumulator value of 0.1, and an epsilon value of 1e-7.

RMSProp uses a learning rate of 0.001, rho is 0.9, no momentum and epsilon is 1e-7.

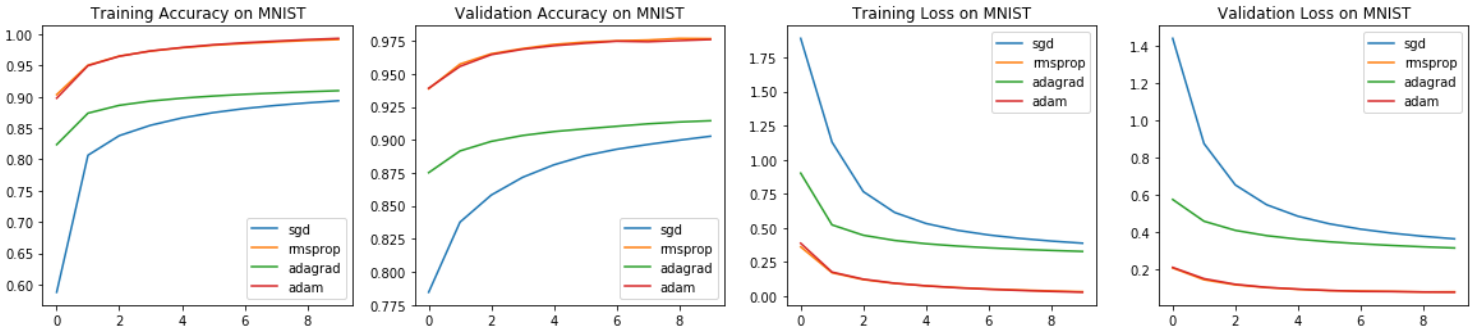
Adam use a learning rate 0.001 as well. Adam's beta parameters were configured to 0.9 and 0.999 respectively. Finally, epsilon=1e-7,

See the full code here.

MNIST

Even though MNIST is a small dataset, and considered to be easily trained on, there were some noticeable differences in performance of the optimizers. SGD was by far the worst optimizer, as can be seen in all graphs. RMSProp and Adam performed extremely similar (nearly identical, in fact) on MNIST.

This can be expected because their key difference is the use of gradients as momentum by Adam, and MNIST is a very small dataset and therefore it also has small gradients, too.



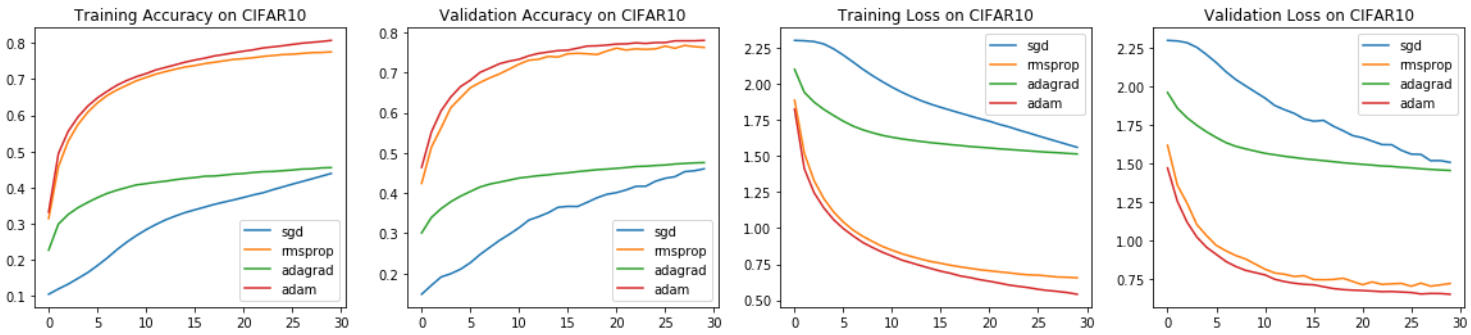
Training performance on MNIST with different optimizers

Optimizer	Training accuracy	Training loss	Validation accuracy	Validation loss
SGD	89.54%	0.3798	90.27%	0.3657
AdaGrad	91.06%	0.3228	91.45%	0.3167
RMSProp	99.37%	0.0248	97.67%	0.0813
Adam	99.56%	0.0210	97.60%	0.0790

Final results on MNIST

CIFAR10

In a larger dataset such as CIFAR10, the differences between optimizers become even more apparent. SGD has a very steady rate of improvement and would likely benefit from more training. The issue with AdaGrad, where the learning rates increase after a few epochs, is very evident in the following graphs. Adam and RMSProp performed very similar once again, though they diverged towards the last few of epochs.



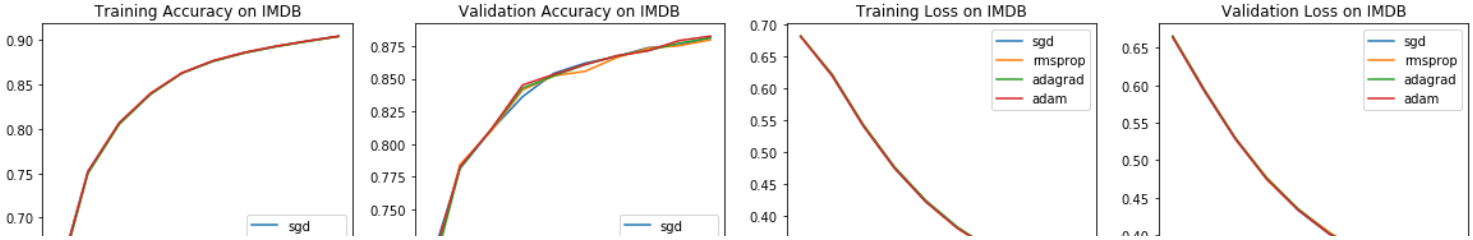
Training performance on CIFAR10 with different optimizers

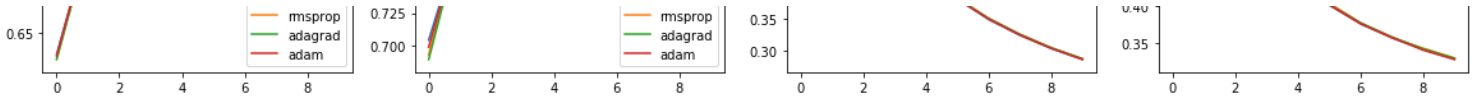
Optimizer	Training accuracy	Training loss	Validation accuracy	Validation loss
SGD	47.02%	1.4931	46.03%	1.5064
AdaGrad	48.56%	1.4463	47.58%	1.4537
RMSProp	85.65%	0.4479	76.23%	0.7176
Adam	92.46%	0.2841	77.97%	0.6477

Final results on CIFAR10

IMDB Reviews

The IMDB reviews dataset (8k subset) is a common natural language processing dataset. It has a vocabulary of only 8000 words, and its goal is to predict the sentiment of a review: positive or negative. Since this dataset is quite small compared to previous datasets mentioned in this article, the results are virtually identical — smart optimizer tricks don't make a difference.





Training performance on IMDB with different optimizers

Optimizer	Training accuracy	Training loss	Validation accuracy	Validation loss
SGD	90.77%	0.2766	87.43%	0.3334
AdaGrad	90.75%	0.2770	87.39%	0.3341
RMSProp	90.79%	0.2771	87.45%	0.3338
Adam	90.86%	0.2755	87.55%	0.3326

Final results on IMDB

Conclusion

Based on these results, you should consider using Adam, or RMSProp, as your optimizer when starting a new deep learning project. Adam got the highest training and validation accuracy on every task described in this post.

. . .

Editor's Note: Heartbeat is a contributor-driven online publication and community dedicated to exploring the emerging intersection of mobile app development and machine learning. We're committed to supporting and inspiring developers and engineers from all walks of life.

Editorially independent, Heartbeat is sponsored and published by Fritz AI, the machine learning platform that helps developers teach devices to see, hear, sense, and think. We pay our contributors, and we don't sell ads.

If you'd like to contribute, head on over to our call for contributors. You can also sign up to receive our weekly newsletters (Deep Learning Weekly and Heartbeat), join us on Slack, and follow Fritz AI on Twitter for all the latest in mobile machine learning.