Gradient Descent vs Adagrad vs Momentum in TensorFlow

Asked 8 years, 9 months ago Modified 5 years, 8 months ago

Viewed 38k times

77

I'm studying *TensorFlow* and how to use it, even if I'm not an expert of neural networks and deep learning (just the basics).

Following tutorials, I don't understand the real and practical differences between the three optimizers for loss. I look at the API and I understand the principles, but my questions are:

1. When is it preferable to use one instead of the others ?

2. Are there important differences to know ?

tensorflow deep-learning

Share

Improve this question

Follow

edited Apr 12, 2018 at 2:56

Stumbler

2,148 ● 8 ● 37 ● 63

asked Mar 22, 2016 at 18:16

Kyrol

3,607 ● 7 ● 35 ● 51

2 There's no theory as to which optimizer is supposed to work better on, say, MNIST, so people try out several ones and pick one that works best for their problem. Gradient Descent is typically the worst of all, Momentum/AdaGrad can be better/worse than the other depending on the dataset – Yaroslav Bulatov Mar 22, 2016 at 18:23

Ok, I need to create a model for image recognition with 4 - 5 classes of recognition. If I use the Imagenet dataset, what are you suggest me ? – Kyrol Mar 22, 2016 at 18:41

AdamOptimizer seems to work well on Imagenet – Yaroslav Bulatov Mar 22, 2016 at 18:42

2 Just to add on what has already been said, the number of hyper parameters required by the optimizer should also be kept in mind when choosing an optimizer. Gradient Descent is slow but you only need to set learning rate. – shekkizh Mar 22, 2016 at 20:46

3 Answers

Sorted by: Highest score (default)

206

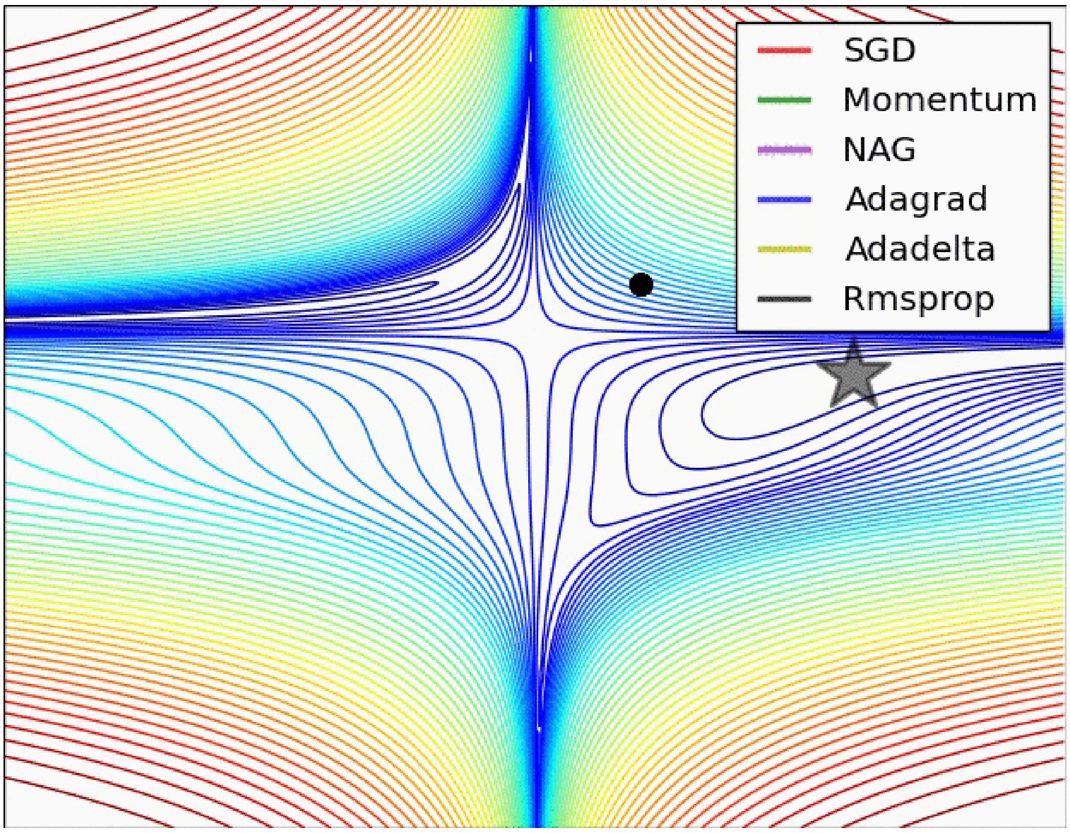
Here is a brief explanation based on my understanding:

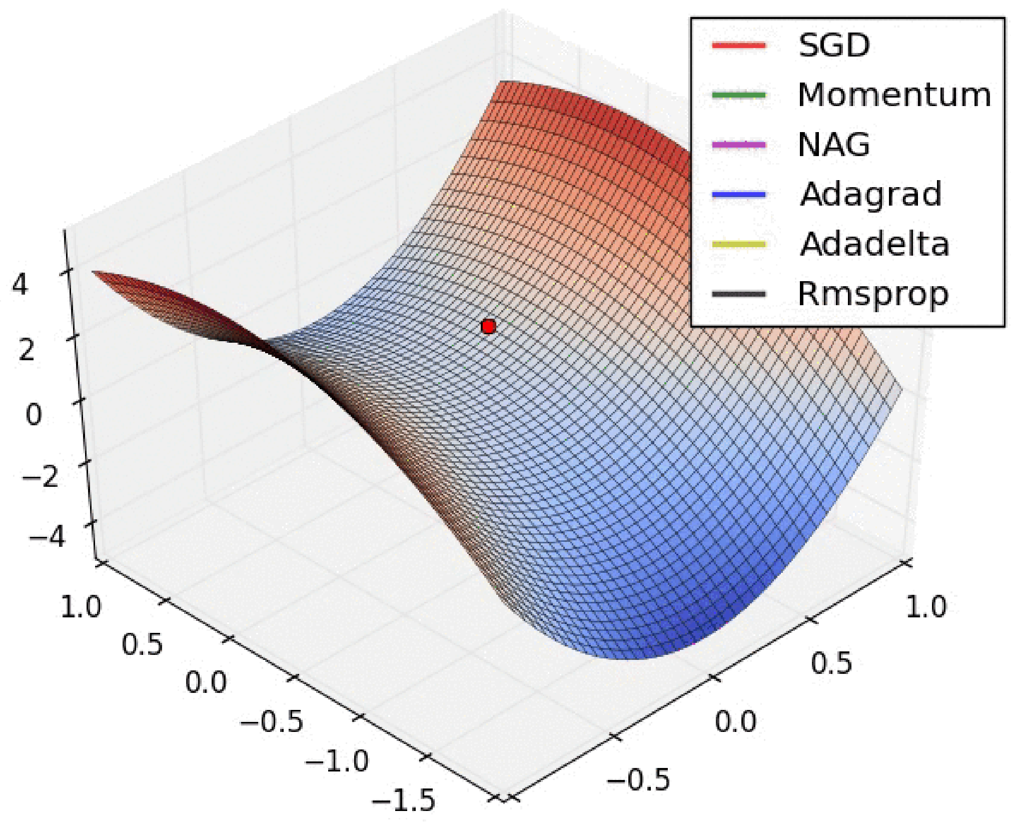
- **momentum** helps SGD to navigate along the relevant directions and softens the oscillations in the irrelevant. It simply adds a fraction of the direction of the previous step to a current step. This achieves amplification of speed in the correct direction and softens oscillation in wrong directions. This fraction is usually in the (0, 1) range. It also makes sense to use adaptive momentum. In the beginning of learning a big momentum will only hinder your progress, so it makes sense to use something like 0.01 and once all the high gradients disappeared you can use a bigger momentum. There is one problem with momentum: when we are very close to the goal, our momentum in most of the cases is very high and it does not know that it should slow down. This can cause it to miss or oscillate around the minima

- **nesterov accelerated gradient** overcomes this problem by starting to slow down early. In momentum we first compute gradient and then make a jump in that direction amplified by whatever momentum we had previously. NAG does the same thing but in another order: at first we make a big jump based on our stored information, and then we calculate the gradient and make a small correction. This seemingly irrelevant change gives significant practical speedups.
- **AdaGrad** or adaptive gradient allows the learning rate to adapt based on parameters. It performs larger updates for infrequent parameters and smaller updates for frequent one. Because of this it is well suited for sparse data (NLP or image recognition). Another advantage is that it basically eliminates the need to tune the learning rate. Each parameter has its own learning rate and due to the peculiarities of the algorithm the learning rate is monotonically decreasing. This causes the biggest problem: at some point of time the learning rate is so small that the system stops learning.

- **AdaDelta** resolves the problem of monotonically decreasing learning rate in AdaGrad. In AdaGrad the learning rate was calculated approximately as one divided by the sum of square roots. At each stage you add another square root to the sum, which causes denominator to constantly increase. In AdaDelta instead of summing all past square roots it uses sliding window which allows the sum to decrease. **RMSprop** is very similar to AdaDelta
- **Adam** or adaptive momentum is an algorithm similar to AdaDelta. But in addition to storing learning rates for each of the parameters it also stores momentum changes for each of them separately.

A few visualizations:





I would say that SGD, Momentum and Nesterov are inferior than the last 3.

Share Improve this answer

Follow

edited Sep 24, 2018 at 8:53

Oren Milman

474 ● 4 ● 18

answered May 28, 2017 at 9:01

Salvador Dali

222k ● 151 ● 723 ● 762

9 "SGD, Momentum and Nesterov are inferior than the last 3" -- they are not **inferior**, they are **slower**. There is a reason why people use SGD+Momentum for training in papers. SGD+Momentum have higher chances of reaching flat minima. – tastyminerals May 23, 2018 at 10:03

5 @minerals if we are going to be pedantic, then we can't use the word slower as well. Because it in theory it is always possible to find a surface and a starting point at which any

algorithm will perform the best. But in practice this is not that useful. – Salvador Dali May 24, 2018 at 2:47

2 From the comments in the link you provided about AdaDelta: "there is no one clear conclusion of which algorithm to choose and when. All of them perform differently depending on the problem and parametrization. [...] The one 'solid' conclusion that can be drawn here is "to choose whatever works best for your problem" – I'm afraid" – Ricardo Stuven Jun 18, 2018 at 0:25

3 This paper (arxiv.org/abs/1705.08292) discusses how SGD often arrives at better solutions than the adaptive methods. Better in the sense of better validation loss/accuracy (i.e. the model generalizes better) rather than quick training times – Anjum Sayed Sep 16, 2018 at 16:32

1 What did you mean by "Adam or adaptive momentum"? as this answer explains, Adam is short for "adaptive moment estimation", according to the original Adam paper. – Oren Milman Oct 1, 2018 at 16:30

24

Salvador Dali's answer already explains about the differences between some popular methods (i.e. optimizers), but I would try to elaborate on them some more.
(Note that our answers disagree about some points, especially regarding ADAGRAD.)

Classical Momentum (CM) vs Nesterov's Accelerated Gradient (NAG)

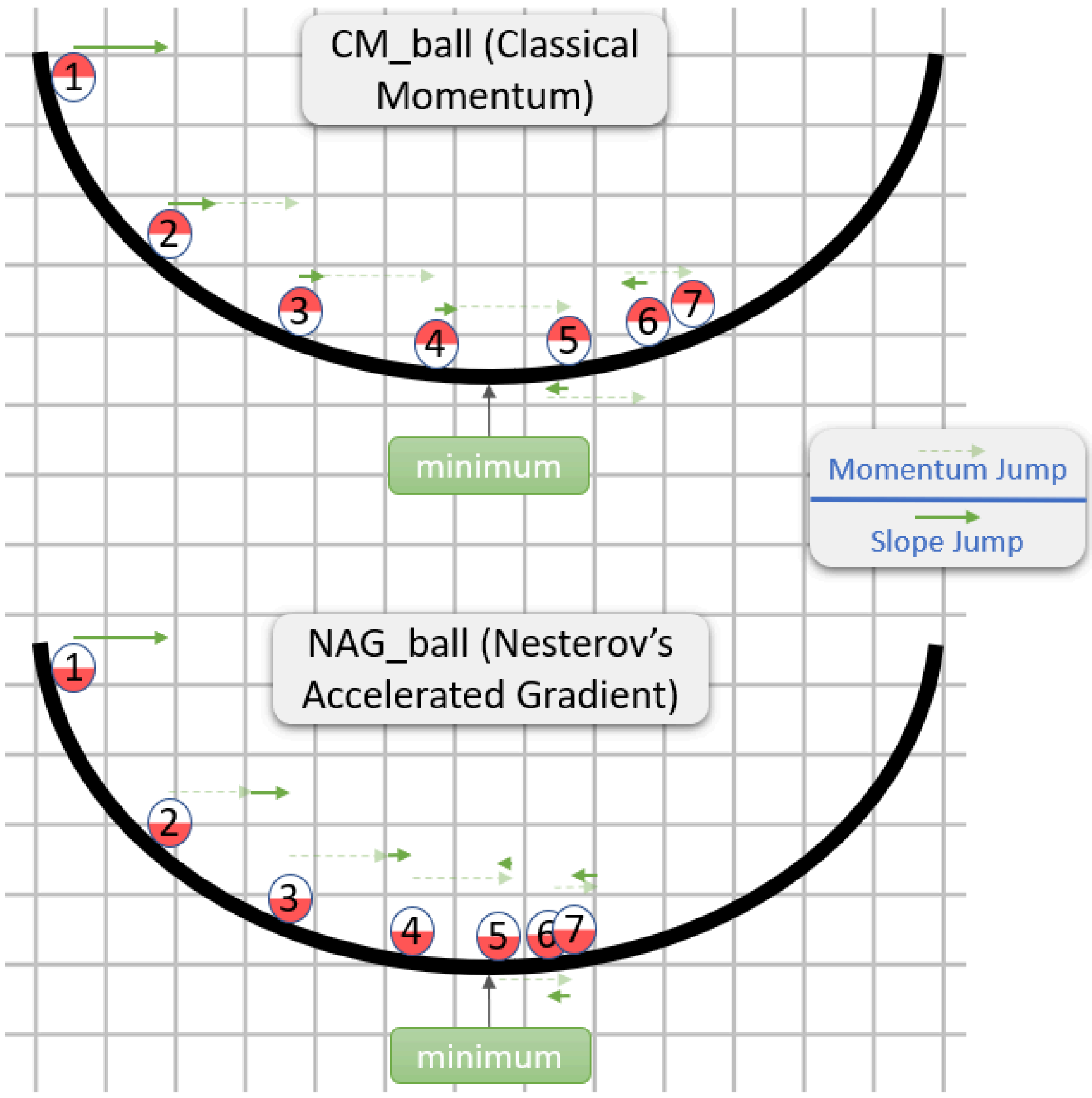
(Mostly based on section 2 in the paper On the importance of initialization and momentum in deep learning.)

Each step in both CM and NAG is actually composed of two sub-steps:

- A momentum sub-step - This is simply a fraction (typically in the range `[0.9,1)`) of the last step.
- A gradient dependent sub-step - This is like the usual step in SGD - it is the product of the learning rate and the vector opposite to the gradient, while the gradient is computed where this sub-step starts from.

CM takes the gradient sub-step first, while NAG takes the momentum sub-step first.

Here is a demonstration from an answer about intuition for CM and NAG:


CM_ball (Classical Momentum)
1
2
3
4
5
6
7
minimum
Momentum Jump
Slope Jump
NAG_ball (Nesterov's Accelerated Gradient)
1
2
3
4
5
6
7
minimum

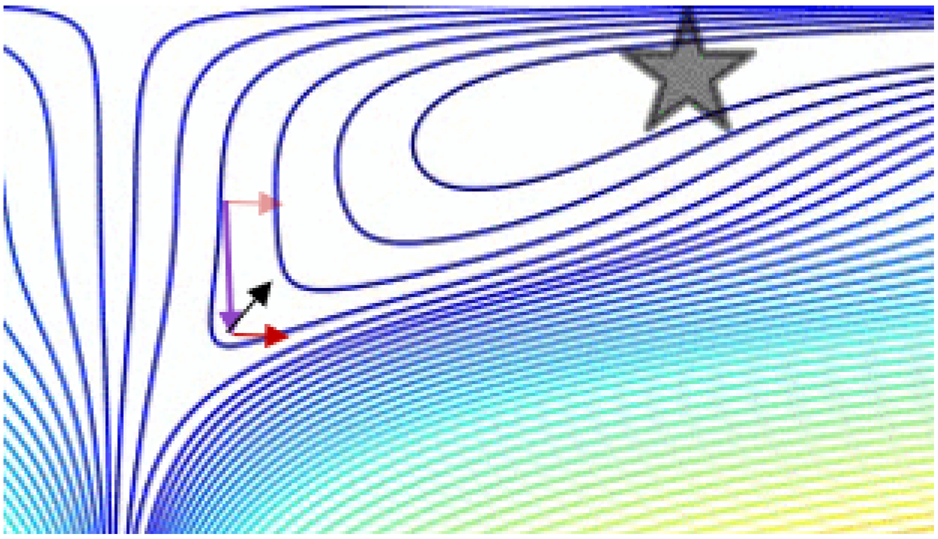
So NAG seems to be better (at least in the image), but why?

The important thing to note is that it doesn't matter when the momentum sub-step comes - it would be the same either way. Therefore, we might as well behave is if the momentum sub-step has already been taken.

Thus, the question actually is: Assuming that the gradient sub-step is taken after the momentum sub-step, should we calculate the gradient sub-step as if it started in the position before or after taking the momentum sub-step?

"After it" seems like the right answer, as generally, the gradient at some point `θ` roughly points you in the direction from `θ` to a minimum (with the relatively right magnitude), while the gradient at some other point is less likely to point you in the direction from `θ` to a minimum (with the relatively right magnitude).

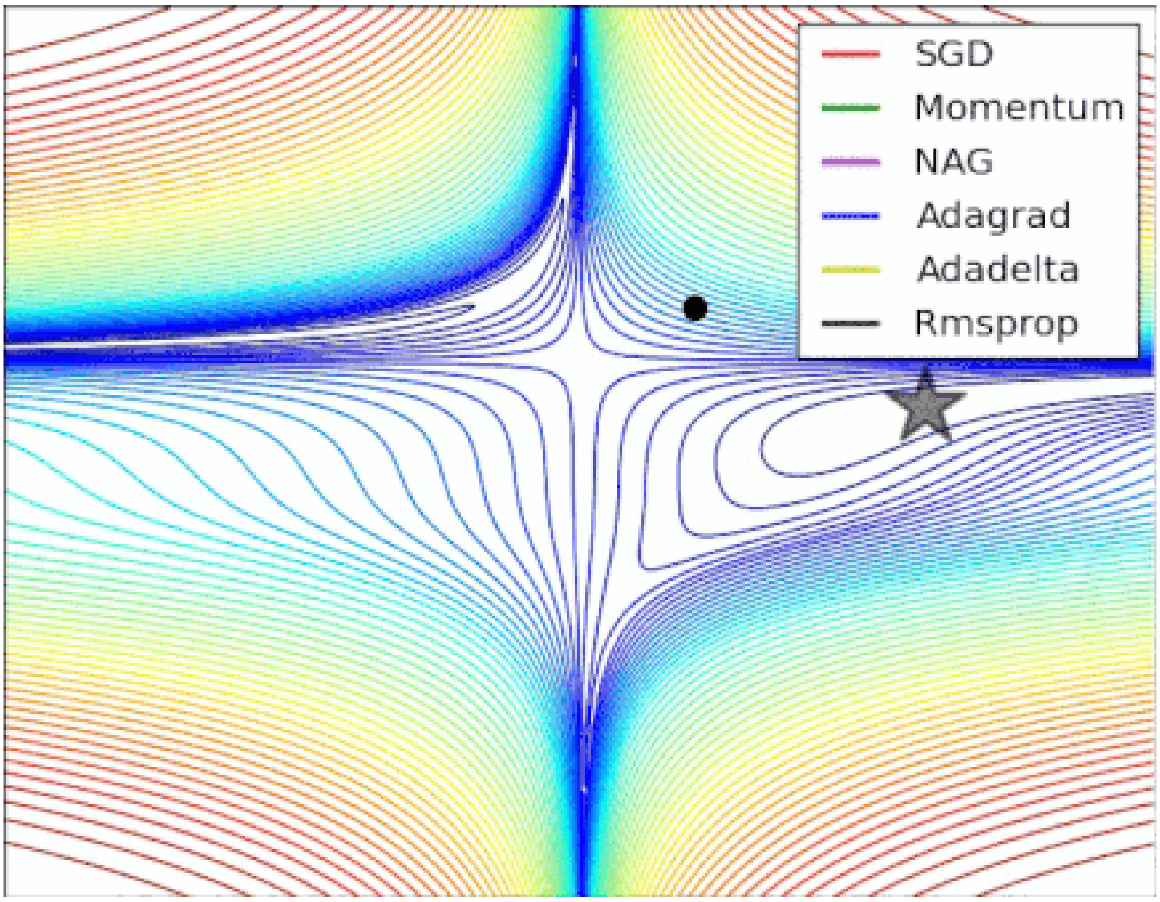
Here is a demonstration (from the gif below):



- The minimum is where the star is, and the curves are contour lines. (For an explanation about contour lines and why they are perpendicular to the gradient, see videos 1 and 2 by the legendary 3Blue1Brown.)
- The (long) purple arrow is the momentum sub-step.
- The transparent red arrow is the gradient sub-step if it starts before the momentum sub-step.
- The black arrow is the gradient sub-step if it starts after the momentum sub-step.
- CM would end up in the target of the dark red arrow.
- NAG would end up in the target of the black arrow.

Note that this argument for why NAG is better is independent of whether the algorithm is close to a minimum.
In general, both NAG and CM often have the problem of accumulating more momentum than is good for them, so whenever they should change direction, they have an embarrassing "response time". The advantage of NAG over CM that we explained doesn't prevent the problem, but only makes the "response time" of NAG less embarrassing (but embarrassing still).

This "response time" problem is beautifully demonstrated in the gif by Alec Radford (which appeared in Salvador Dali's answer):



ADAGRAD

(Mostly based on section 2.2.2 in ADADELTA: An Adaptive Learning Rate Method (the original ADADELTA paper), as I find it much more accessible than Adaptive Subgradient Methods for Online Learning and Stochastic Optimization (the original ADAGRAD paper).)

In SGD, the step is given by `- learning_rate * gradient`, while `learning_rate` is a hyperparameter. ADAGRAD also has a `learning_rate` hyperparameter, but the actual learning rate for each component of the gradient is calculated individually.
The `i`-th component of the `t`-th step is given by:

```
                learning_rate
- --------------------------------------- * gradient_i_t
  norm((gradient_i_1, ..., gradient_i_t))
```

while:

- `gradient_i_k` is the `i`-th component of the gradient in the `k`-th step
- `(gradient_i_1, ..., gradient_i_t)` is a vector with `t` components. This isn't intuitive (at least to me) that constructing such a vector makes sense, but that's what the algorithm does (conceptually).
- `norm(vector)` is the Eucldiean norm (aka `l2` norm) of `vector`, which is our intuitive notion of length of `vector`.
- Confusingly, in ADAGRAD (as well as in some other methods) the expression that is multiplied by

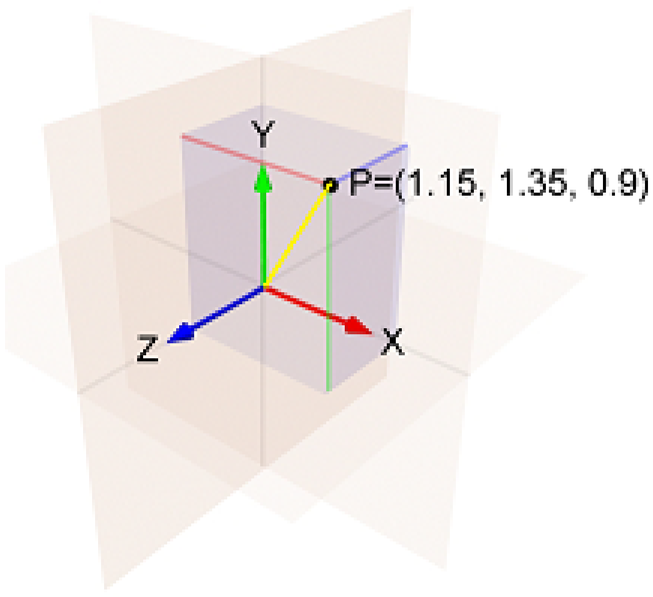
`gradient_i_t` (in this case, `learning_rate / norm(...)`) is often called "the learning rate" (in fact, I called it "the actual learning rate" in the previous paragraph). I guess this is because in SGD the `learning_rate` hyperparameter and this expression are one and the same.

- In a real implementation, some constant would be added to the denominator, to prevent a division by zero.

E.g. if:

- The `i`-th component of the gradient in the first step is `1.15`
- The `i`-th component of the gradient in the second step is `1.35`
- The `i`-th component of the gradient in the third step is `0.9`

Then the norm of `(1.15, 1.35, 0.9)` is the length of the yellow line, which is: `sqrt(1.15^2 + 1.35^2 + 0.9^2) = 1.989`.
And so the `i`-th component of the third step is: `- learning_rate / 1.989 * 0.9`



Note two things about the `i`-th component of the step:

1. It is proportional to `learning_rate`.
2. In the calculations of it, the norm is increasing, and so the learning rate is decreasing.

This means that ADAGRAD is sensitive to the choice of the hyperparameter `learning_rate`.
In addition, it might be that after some time the steps become so small, that ADAGRAD virtually gets stuck.

ADADELTA and RMSProp

From the ADADELTA paper:

> The idea presented in this paper was derived from ADAGRAD in order to improve upon the two main drawbacks of the method: 1) the continual decay of learning rates throughout training, and

> 2) the need for a manually selected global learning rate.

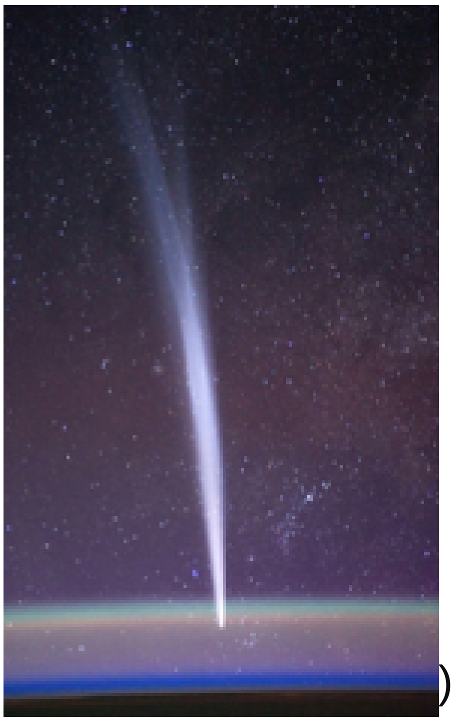
The paper then explains an improvement that is meant to tackle the first drawback:

> Instead of accumulating the sum of squared gradients over all time, we restricted the window of past gradients that are accumulated to be some fixed size `w` [...]. This ensures that learning continues to make progress even after many iterations of updates have been done. Since storing `w` previous squared gradients is inefficient, our methods implements this accumulation as an exponentially decaying average of the squared gradients.

By "exponentially decaying average of the squared gradients" the paper means that for each `i` we compute a weighted average of all of the squared `i`-th components of all of the gradients that were calculated. The weight of each squared `i`-th component is bigger than the weight of the squared `i`-th component in the previous step.

This is an approximation of a window of size `w` because the weights in earlier steps are very small.

(When I think about an exponentially decaying average, I like to visualize a comet's trail, which becomes dimmer and dimmer as it gets further from the comet:

)

If you make only this change to ADAGRAD, then you will get RMSProp, which is a method that was proposed by Geoff Hinton in Lecture 6e of his Coursera Class.

So in RMSProp, the `i`-th component of the `t`-th step is given by:

```
                    learning_rate
- --------------------------------------------------
* gradient_i_t
  sqrt(exp_decay_avg_of_squared_grads_i + epsilon)
```

while:

- `epsilon` is a hyperparameter that prevents a division by zero.
- `exp_decay_avg_of_squared_grads_i` is an exponentially decaying average of the squared `i`-th

components of all of the gradients calculated (including `gradient_i_t`).

But as aforementioned, ADADELTA also aims to get rid of the `learning_rate` hyperparameter, so there must be more stuff going on in it.

In ADADELTA, the `i`-th component of the `t`-th step is given by:

```
  sqrt(exp_decay_avg_of_squared_steps_i + epsilon)
- ------------------------------------------------
* gradient_i_t
  sqrt(exp_decay_avg_of_squared_grads_i + epsilon)
```

while `exp_decay_avg_of_squared_steps_i` is an exponentially decaying average of the squared `i`-th components of all of the steps calculated (until the `t-1`-th step).
`sqrt(exp_decay_avg_of_squared_steps_i + epsilon)` is somewhat similar to momentum, and according to the paper, it "acts as an acceleration term". (The paper also gives another reason for why it was added, but my answer is already too long, so if you are curious, check out section 3.2.)

Adam

(Mostly based on Adam: A Method for Stochastic Optimization, the original Adam paper.)

Adam is short for Adaptive Moment Estimation (see this answer for an explanation about the name).
The `i`-th component of the `t`-th step is given by:

```
                    learning_rate
- ----------------------------------------------
* exp_decay_avg_of_grads_i
  sqrt(exp_decay_avg_of_squared_grads_i) + epsilon
```

while:

- `exp_decay_avg_of_grads_i` is an exponentially decaying average of the `i`-th components of all of the gradients calculated (including `gradient_i_t`).
- Actually, both `exp_decay_avg_of_grads_i` and `exp_decay_avg_of_squared_grads_i` are also corrected to account for a bias toward `0` (for more about that, see section 3 in the paper, and also an answer in stats.stackexchange).

Note that Adam uses an exponentially decaying average of the `i`-th components of the gradients where most SGD methods use the `i`-th component of the current gradient. This causes Adam to behave like "a heavy ball with friction", as explained in the paper GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equilibrium.
See this answer for more about how Adam's momentum-like behavior is different from the usual momentum-like behavior.

Share Improve this answer Follow

edited Oct 5, 2018 at 12:47

answered Sep 24, 2018 at 6:56

Oren Milman
474 ● 4 ● 18

3 Excellent Answer!! – Failed Scientist Sep 24, 2018 at 7:21

1 Highly underrated answer; deserves more upvotes. Also thanks for pointing people to 3Blue1Brown. He's a beautiful beautiful teacher, animator, and mathematical thinker
– Nathan majicvr.com Jun 17, 2019 at 17:32

5

Let's boil it down to a couple of simple question:

Which optimizer would give me the best result/accuracy?

There is no silver bullet. Some optimizers for your task would probably work better than the others. There is no way to tell beforehand, you have to try a few to find the best one. Good news is that the results of different optimizers would probably be close to each other. You have to find the best hyperparameters for any single optimizer you choose, though.

Which optimizer should I use right now?

Maybe, take AdamOptimizer and run it for learning_rate 0.001 and 0.0001. If you want better results, try running for other learning rates. Or try other optimizers and tune their hyperparameters.

Long story

There are a few aspects to consider when choosing your optimizer:

- Easy of use (i.e. how fast you can find parameters that work for you);
- Convergence speed (basic as SGD or faster as any other);
- Memory footprint (typically between 0 and x2 sizes of your model);
- Relation to other parts of the training process.

Plain SGD is the bare minimum that can be done: it simply multiplies the gradients by the learning rate and adds the result to the weights. SGD has a number of beautiful qualities: it has only 1 hyperparameter; it does not need any additional memory; it has minimal effect on the other parts of training. It also has 2 drawbacks: it might be too sensitive to learning rate choice and training can take longer than with other methods.

From these drawbacks of plain SGD we can see what are the more complicated update rules (optimizers) are for: we sacrifice a part of our memory to achieve faster training and, possibly, simplify the choice of hyperparameters.

Memory overhead is typically non-significant and can be ignored. Unless the model is extremely large, or you are training on GTX760, or fighting for ImageNet leadership. Simpler methods like momentum or Nesterov accelerated gradient need 1.0 or less of model size (size of the model hyperparameters). Second order methods (Adam, might need twice as much memory and computation.

Convergence speed-wise pretty much anything is better than SGD and anything else is hard to compare. One note might be that AdamOptimizer is good at starting training almost immediately, without a warm-up.

I consider **easy-of-use** to be the most important in the choice of an optimizer. Different optimizers have a different number of hyperparameters and have a different sensibility to them. I consider Adam the most simple of all readily-available ones. You typically need to check 2-4 learning_rates between `0.001` and `0.0001` to figure out if the model converges nicely. For comparison for SGD (and momentum) I typically try `[0.1, 0.01, ... 10e-5]`. Adam has 2 more hyperparameters that rarely have to be changed.

Relation between optimizer and other parts of training. Hyperparameter tuning typically involves selecting `{learning_rate, weight_decay, batch_size, droupout_rate}` simultaneously. All of them are interrelated and each can be viewed as a form of model regularization. One, for example, has to pay close attention if, exactly, weight_decay or L2-norm is used and possibly choose `AdamWOptimizer` instead of `AdamOptimizer`.

Share Improve this answer

Follow

answered Apr 12, 2019 at 17:38

y.selivonchyk

9,850 ● 8 ● 62 ● 79