10 Stochastic Gradient Descent Optimisation Algorithms + Cheatsheet

Stochastic gradient descent optimisation algorithms you should know for deep learning

(I maintain a cheat sheet of these optimisers including RAdam in my blog here.)

Changelog:
5 Jan 2022 — Fix typos
4 May 2020 — Fix typo in Nadam formula in Appendix 2
21 Mar 2020 — Replace V *and* S *with* m *and* v *respectively, update dead links, review the idea of learning rate and gradient components, update intuitions*
6 Oct 2019 — Improve on the idea of EMA of gradients
22 Sep 2019 — Rearrange content order and remove 'evolutionary map'

Gradient descent is an optimisation method for finding the minimum of a function. It is commonly used in deep learning models to update the weights of a neural network through backpropagation.

In this post, I will summarise the common gradient descent optimisation algorithms used in popular deep learning frameworks

(e.g. TensorFlow, Keras, PyTorch). The purpose of this post is to make it easy to read and digest the formulae using consistent nomenclature since there aren't many such summaries out there. At the end of this post is a cheat sheet for your reference.

This post assumes that the reader has some knowledge about gradient descent / stochastic gradient descent.

(For a demo on a linear regression problem using gradient descent optimisers like SGD, momentum and Adam, click here.)

What do stochastic gradient descent optimisers do?

Recall that the vanilla stochastic gradient descent (SGD) updates weights by subtracting the current weight by a factor (i.e. α, the learning rate) of its gradient.

$$w_{\text{new}} = w - \alpha \frac{\partial L}{\partial w}$$

Eqn. 1: The terms in stochastic gradient descent

Variations in this equation are commonly known as stochastic gradient descent optimisers. There are 3 main ways how they differ:

1. **Adapt the "gradient component" ($\partial L/\partial w$)**
   Instead of using only one single gradient like in stochastic vanilla gradient descent to update the weight, take

an *aggregate of multiple gradients*. Specifically, these optimisers use the exponential moving average of gradients.

2. **Adapt the "learning rate component" (α)**
   Instead of keeping a constant learning rate, adapt the learning rate according to the *magnitude* of the gradient(s).

3. **Both (1) and (2)**
   Adapt both the gradient component and the learning rate component.

As you will see later, these optimisers try to improve the amount of information used to update the weights, mainly through using previous (and future) gradients, instead of only the present available gradient.

Below is a table that summarises which "components" are being adapted:

Optimiser	Year	Learning Rate	Gradient
Momentum	1964		✓
AdaGrad	2011	✓	
RMSprop	2012	✓	
Adadelta	2012	✓	
Nesterov	2013		✓
Adam	2014	✓	✓
AdaMax	2015	✓	✓
Nadam	2015	✓	✓
AMSGrad	2018	✓	✓

Fig. 2: Gradient descent optimisers, the year in which the papers were published, and the components they act upon

<change log 22 Sep 2019: removed evolutionary map>

Content

Notations

- t — time step
- w — weight/parameter which we want to update where the subscript t indexes the weight at time step t.
- α — learning rate
- $\partial L/\partial w$ — gradient of L, the loss function to minimise, w.r.t. to w
- I have also standardised the notations and Greek letters used in this post (hence might be different from the papers) so that we can explore how optimisers 'evolve' as we scroll.

1. Stochastic gradient descent

As we have seen earlier, the vanilla SGD updates the current weight using the current gradient $\partial L/\partial w$ multiplied by some factor called the learning rate, α.

$$w_{t+1} = w_t - \alpha \frac{\partial L}{\partial w_t}$$

2. Momentum

Instead of depending only on the current gradient to update the weight, gradient descent with momentum (Polyak, 1964) replaces the current gradient with m ("momentum"), which is an aggregate of gradients. This aggregate is the exponential moving average of current and past gradients (i.e. up to time t). Later in this post, you will see that

this momentum update becomes the standard update for the gradient component for most optimisers.

$$w_{t+1} = w_t - \alpha m_t$$

where

$$m_t = \beta m_{t-1} + (1 - \beta) \frac{\partial L}{\partial w_t}$$

and m initialised to 0.

Common default value:

- $\beta = 0.9$

On the origins of momentum

Note that many articles reference the momentum method to the publication by Ning Qian, 1999. However, the paper titled Sutskever et al. attributed the classical momentum to a much earlier publication by Polyak in 1964, as cited above. (Thank you to James for pointing this out.)

3. AdaGrad

Adaptive gradient, or AdaGrad (Duchi et al., 2011), acts on the learning rate component by dividing the learning rate by the square root of v,

which is the cumulative sum of current and past squared gradients (i.e. up to time *t*). Note that the gradient component remains unchanged like in SGD.

$$w_{t+1} = w_t - \frac{\alpha}{\sqrt{v_t + \epsilon}} \cdot \frac{\partial L}{\partial w_t}$$

where

$$v_t = v_{t-1} + \left[\frac{\partial L}{\partial w_t}\right]^2$$

and v initialised to 0.

Notice that ε is added to the denominator. Keras calls this the *fuzz factor*, a small floating-point value to ensure that we will never have to come across division by zero.

Default values (from Keras):

- $\alpha = 0.01$
- $\varepsilon = 10^{-7}$

4. RMSprop

Root mean square prop or RMSprop (Hinton et al., 2012) is another adaptive learning rate that tries to improve AdaGrad. Instead of taking the cumulative sum of squared gradients like in AdaGrad, we take the exponential moving average (again!) of these gradients. Similar to momentum, we will slowly see that this update becomes the standard update for the learning rate component for most optimisers.

$$w_{t+1} = w_t - \frac{\alpha}{\sqrt{v_t + \epsilon}} \cdot \frac{\partial L}{\partial w_t}$$

where

$$v_t = \beta v_{t-1} + (1 - \beta) \left[\frac{\partial L}{\partial w_t} \right]^2$$

and v initialised to 0.

Default values (from Keras):

- $\alpha = 0.001$
- $\beta = 0.9$ (recommended by the authors of the paper)
- $\varepsilon = 10^{-6}$

5. Adadelta

Like RMSprop, Adadelta (Zeiler, 2012) is also another improvement from AdaGrad, focusing on the learning rate component. Adadelta is probably short for 'adaptive delta', where *delta* here refers to the difference between the current weight and the newly updated weight.

The difference between Adadelta and RMSprop is that Adadelta removes the use of the learning rate parameter completely by replacing it with *D*, the exponential moving average of squared *deltas*.

$$w_{t+1} = w_t - \frac{\sqrt{D_{t-1} + \epsilon}}{\sqrt{v_t + \epsilon}} \cdot \frac{\partial L}{\partial w_t}$$

where

$$D_t = \beta D_{t-1} + (1 - \beta)[\Delta w_t]^2$$

$$v_t = \beta v_{t-1} + (1 - \beta)\left[\frac{\partial L}{\partial w_t}\right]^2$$

with *D* and *v* initialised to 0, and

$$\Delta w_t = w_t - w_{t-1}$$

Default values (from Keras):

- $\beta = 0.95$
- $\varepsilon = 10^{-6}$

6. Nesterov Accelerated Gradient (NAG)

After Polyak had gained his momentum (pun intended 😬), a similar update was implemented using Nesterov Accelerated Gradient (Sutskever et al., 2013). This update utilises m, the exponential moving average of what I would call *projected gradients*.

$$w_{t+1} = w_t - \alpha m_t$$

where

$$m_t = \beta m_{t-1} + (1 - \beta) \frac{\partial L}{\partial w^*}$$

and m initialised to 0.

The last term in the second equation is a projected gradient. This value can be obtained by going 'one step ahead' using the previous velocity (Eqn. 4). This means that for this time step t, we have to carry out another forward propagation before we can finally execute the backpropagation. Here's how it goes:

1. Update the current weight w to a projected weight w^* using the previous velocity.

$$w^* = w_t - \alpha m_{t-1}$$
Eqn. 4

2. Carry out forward propagation, but using this projected weight.

3. Obtain the projected gradient $\partial L/\partial w^*$.

4. Compute V and w accordingly.

Common default value:

- $\beta = 0.9$

On the origins of NAG

Note that the original Nesterov Accelerated Gradient paper (Nesterov, 1983) was not about stochastic gradient descent and did not explicitly use the gradient descent equation. Hence, a more appropriate reference is the above-mentioned publication by Sutskever et al. in 2013, which described NAG's application in stochastic gradient descent. (Again, I'd like to thank James's comment on HackerNews for pointing this out.)

7. Adam

Adaptive moment estimation, or Adam (Kingma & Ba, 2014), is simply a combination of momentum and RMSprop. It acts upon

1. the gradient component by using m, the exponential moving average of gradients (like in momentum), and
2. the learning rate component by dividing the learning rate α by square root of v, the exponential moving average of squared gradients (like in RMSprop).

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

where

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

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

are the bias corrections, and

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) \frac{\partial L}{\partial w_t}$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) \left[\frac{\partial L}{\partial w_t} \right]^2$$

with m and v initialised to 0.

Proposed default values by the authors:

- $\alpha = 0.001$
- $\beta_1 = 0.9$
- $\beta_2 = 0.999$
- $\varepsilon = 10^{-8}$

8. AdaMax

AdaMax (Kingma & Ba, 2015) is an adaptation of the Adam optimiser by the same authors using infinity norms (hence 'max'). m is the exponential moving average of gradients, and v is the exponential moving average of past p-norm of gradients, approximated to the max function as seen below. Refer to the paper for their proof of convergence.

$$w_{t+1} = w_t - \frac{\alpha}{v_t} \cdot \hat{m}_t$$

where

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

is the bias correction for m and

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) \frac{\partial L}{\partial w_t}$$

$$v_t = \max(\beta_2 v_{t-1}, \left| \frac{\partial L}{\partial w_t} \right|)$$

with m and v initialised to 0.

Proposed default values by the authors:

- $\alpha = 0.002$
- $\beta_1 = 0.9$

- $\beta_2 = 0.999$

9. Nadam

Nadam (Dozat, 2015) is an acronym for Nesterov and Adam optimiser. The Nesterov component, however, is a more efficient modification than its original implementation.

First, I'd like to show that the Adam optimiser can also be written as:

$$w_{t+1} = w_t - \frac{\alpha}{\sqrt{\hat{v}_t} + \epsilon}\left(\beta_1 \hat{m}_{t-1} + \frac{1-\beta_1}{1-\beta_1^t} \cdot \frac{\partial L}{\partial w_t}\right)$$

Eqn. 5: Weight update for Adam optimiser

Nadam uses Nesterov to update the gradient one step ahead by replacing the previous *m_hat* in the above equation to the current *m_hat*:

$$w_{t+1} = w_t - \frac{\alpha}{\sqrt{\hat{v}_t} + \epsilon}\left(\beta_1 \hat{m}_t + \frac{1-\beta_1}{1-\beta_1^t} \cdot \frac{\partial L}{\partial w_t}\right)$$

where

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

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

and

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) \frac{\partial L}{\partial w_t}$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) \left[\frac{\partial L}{\partial w_t} \right]^2$$

with m and v initialised to 0.

Default values (taken from Keras):

- $\alpha = 0.002$
- $\beta_1 = 0.9$
- $\beta_2 = 0.999$

- $\varepsilon = 10^{-7}$

10. AMSGrad

Another variant of Adam is the AMSGrad (Reddi et al., 2018). This variant revisits the adaptive learning rate component in Adam and changes it to ensure that the current v is always larger than the v from the previous time step.

$$w_{t+1} = w_t - \frac{\alpha}{\sqrt{\hat{v}_t} + \epsilon} \cdot m_t$$

where

$$\hat{v}_t = \max(\hat{v}_{t-1}, v_t)$$

and

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) \frac{\partial L}{\partial w_t}$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) \left[\frac{\partial L}{\partial w_t} \right]^2$$

with m and v initialised to 0.

Default values (taken from Keras):

- $\alpha = 0.001$
- $\beta_1 = 0.9$
- $\beta_2 = 0.999$
- $\varepsilon = 10^{-7}$

Please reach out to me if something is amiss, or if something in this post can be improved! ✌

Appendix 2: Intuition

Vanilla SGD

$$w_{t+1} = w_t - \alpha \frac{\partial L}{\partial w_t}$$

Momentum

$$w_{t+1} = w_t - \alpha m_t$$
$$m_t = \beta m_{t-1} + (1-\beta)\frac{\partial L}{\partial w_t}$$

Adagrad

$$w_{t+1} = w_t - \frac{\alpha}{\sqrt{v_t + \epsilon}} \cdot \frac{\partial L}{\partial w_t}$$
$$v_t = v_{t-1} + \left[\frac{\partial L}{\partial w_t}\right]^2$$

RMSprop

$$w_{t+1} = w_t - \frac{\alpha}{\sqrt{v_t + \epsilon}} \cdot \frac{\partial L}{\partial w_t}$$
$$v_t = \beta v_{t-1} + (1-\beta)\left[\frac{\partial L}{\partial w_t}\right]^2$$

Adadelta

$$w_{t+1} = w_t - \frac{\sqrt{D_{t-1} + \epsilon}}{\sqrt{v_t + \epsilon}} \cdot \frac{\partial L}{\partial w_t}$$
$$D_t = \beta D_{t-1} + (1-\beta)[\Delta w_t]^2$$
$$v_t = \beta v_{t-1} + (1-\beta)\left[\frac{\partial L}{\partial w_t}\right]^2$$

Nesterov

$$w_{t+1} = w_t - \alpha m_t$$
$$m_t = \beta m_{t-1} + (1-\beta)\frac{\partial L}{\partial w^*}$$
$$w^* = w_t - \alpha m_{t-1}$$

Adam

$$w_{t+1} = w_t - \frac{\alpha}{\sqrt{\hat{v}_t} + \epsilon} \cdot \hat{m}_t$$
$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$
$$\hat{v}_t = \frac{v_t}{1-\beta_2^t}$$
$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\frac{\partial L}{\partial w_t}$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2)\left[\frac{\partial L}{\partial w_t}\right]^2$$

AdaMax

$$w_{t+1} = w_t - \frac{\alpha}{v_t} \cdot \hat{m}_t$$
$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$
$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\frac{\partial L}{\partial w_t}$$
$$v_t = \max(\beta_2 v_{t-1}, \left|\frac{\partial L}{\partial w_t}\right|)$$

Nadam

$$w_{t+1} = w_t - \frac{\alpha}{\sqrt{\hat{v}_t} + \epsilon}\left(\beta_1 \hat{m}_{t-1} + \frac{1-\beta_1}{1-\beta_1^t} \cdot \frac{\partial L}{\partial w_t}\right)$$
$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$
$$\hat{v}_t = \frac{v_t}{1-\beta_2^t}$$
$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\frac{\partial L}{\partial w_t}$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2)\left[\frac{\partial L}{\partial w_t}\right]^2$$

AMSGrad

$$w_{t+1} = w_t - \frac{\alpha}{\sqrt{\hat{v}_t} + \epsilon} \cdot m_t$$
$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\frac{\partial L}{\partial w_t}$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2)\left[\frac{\partial L}{\partial w_t}\right]^2$$
$$\hat{v}_t = \max(\hat{v}_{t-1}, v_t)$$

Vanilla SGD

$$w_{t+1} = w_t - \alpha \frac{\partial L}{\partial w_t}$$

Momentum

$$w_{t+1} = w_t - \alpha m_t$$
$$m_t = \beta m_{t-1} + (1-\beta)\frac{\partial L}{\partial w_t}$$

Adagrad

$$w_{t+1} = w_t - \frac{\alpha}{\sqrt{v_t + \epsilon}} \cdot \frac{\partial L}{\partial w_t}$$
$$v_t = v_{t-1} + \left[\frac{\partial L}{\partial w_t}\right]^2$$

RMSprop

$$w_{t+1} = w_t - \frac{\alpha}{\sqrt{v_t + \epsilon}} \cdot \frac{\partial L}{\partial w_t}$$
$$v_t = \beta v_{t-1} + (1-\beta)\left[\frac{\partial L}{\partial w_t}\right]^2$$

Adadelta

$$w_{t+1} = w_t - \frac{\sqrt{D_{t-1} + \epsilon}}{\sqrt{v_t + \epsilon}} \cdot \frac{\partial L}{\partial w_t}$$
$$D_t = \beta D_{t-1} + (1-\beta)[\Delta w_t]^2$$
$$v_t = \beta v_{t-1} + (1-\beta)\left[\frac{\partial L}{\partial w_t}\right]^2$$

Nesterov

$$w_{t+1} = w_t - \alpha m_t$$
$$m_t = \beta m_{t-1} + (1-\beta)\frac{\partial L}{\partial w^*}$$
$$w^* = w_t - \alpha m_{t-1}$$

Adam

$$w_{t+1} = w_t - \frac{\alpha}{\sqrt{\hat{v}_t} + \epsilon} \cdot \hat{m}_t$$
$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$
$$\hat{v}_t = \frac{v_t}{1-\beta_2^t}$$
$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\frac{\partial L}{\partial w_t}$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2)\left[\frac{\partial L}{\partial w_t}\right]^2$$

AdaMax

$$w_{t+1} = w_t - \frac{\alpha}{v_t} \cdot \hat{m}_t$$
$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$
$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\frac{\partial L}{\partial w_t}$$
$$v_t = \max(\beta_2 v_{t-1}, \left|\frac{\partial L}{\partial w_t}\right|)$$

Nadam

$$w_{t+1} = w_t - \frac{\alpha}{\sqrt{\hat{v}_t} + \epsilon}\left(\beta_1 \hat{m}_t + \frac{1-\beta_1}{1-\beta_1^t} \cdot \frac{\partial L}{\partial w_t}\right)$$
$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$
$$\hat{v}_t = \frac{v_t}{1-\beta_2^t}$$
$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\frac{\partial L}{\partial w_t}$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2)\left[\frac{\partial L}{\partial w_t}\right]^2$$

AMSGrad

$$w_{t+1} = w_t - \frac{\alpha}{\sqrt{\hat{v}_t} + \epsilon} \cdot m_t$$
$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\frac{\partial L}{\partial w_t}$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2)\left[\frac{\partial L}{\partial w_t}\right]^2$$
$$\hat{v}_t = \max(\hat{v}_{t-1}, v_t)$$

Gist for the above can be found here. Image generated with QuickLaTeX. (Thank you Ravi for pointing out the typo in Nadam's update.)

Appendix 2: Intuitions

Here I'd like to share with you some intuition why gradient descent optimisers use exponential moving average for the gradient component and root mean square for the learning rate component.

Why take exponential moving average of gradients?

Recall that we need to update the weight, and to do so we need to make use of *some value*. The only value we have is the current gradient, so let's just use this information to update the weight.

But taking only the current gradient value is not enough. We want our updates to be "better guided." And this is achieved through using the previous information about gradients. So let's include previous gradients too, by *aggregating* the current gradient and past gradients.

One way to aggregate these gradients is to take a simple average of all the past and current gradients. But wait, this means each of these gradients are equally weighted. Would that be fair? Maybe. Maybe not.

What we could do is to take the exponential moving average, where past gradient values are given higher weights (importance) than the

current one. Intuitively, discounting the importance given to the current gradient would ensure that the weight update will not be sensitive to the current gradient.

Why divide learning rate by root of exponential average of squared gradients?

The goal of adapting the learning rate is to make the optimiser 'smarter' by dividing the learning rate by the root mean square of multiple gradients. So let's ask ourselves these questions:

1. Why take multiple gradients?
2. Why divide?
3. Why take the root of the exponential moving average of squared gradients?

The first question has been answered from the previous section — on top of the value of the current gradient, we also want to utilise the information from the past gradients.

To answer the second question, firstly, consider a simple case where the average magnitude of the gradients for the past few iterations has been 0.01. Since this value is close to 0, it means that we have been on an approximately flat surface (imagine a section of the 3D loss landscape that is flat). Where we are now, the earth is pretty flat, so we're pretty confident moving about this area. In fact, we want to get

out of this area as fast as possible and look for a downward slope that could possibly lead us to a global minimum. (You might find some articles mentioning that this has the effect of “acceleration.”) Therefore, we want to increase the learning rate component (learn faster) when the magnitude of the gradients is small. To establish this inverse proportion relationship, we take the fixed learning rate α and divide it by the average magnitude of the gradient. This adapted learning rate (which is now a large value) is then multiplied by the gradient component, giving us a large weight update in magnitude (no matter positive or negative).

The intuition holds true too for the converse. Suppose the average of our gradients has been very high, about 2.7. This means we have been on steep slopes. We want to move with caution so we take smaller steps, and this can be achieved by performing the same division.

In the last question, the reason why we take exponential moving average has been apparent to us from the previous section. The reason why we take the square of the gradient is simply that when dealing with the learning rate component, we are concerned with its magnitude. A natural choice to ‘offset’ this is to take its root. There might be math behind this, but let’s just use this intuition to convince ourselves for now.

Appendix 3: *Learning rate schedulers vs. stochastic gradient descent optimisers*

Some of you might ask — what's the difference between learning rate schedulers and stochastic gradient descent optimisers? The main difference between these two is that stochastic gradient descent optimisers adapt the learning rate component by multiplying the learning rate by a factor which is a function of the gradients, whereas learning rate schedulers multiply the learning rate by a factor which is a function of the time step (or even a constant).

References

The respective papers mentioned above for every optimiser

An overview of gradient descent optimization algorithms (ruder.io)

Why Momentum Really Works (distill.pub)

Related Articles on Deep Learning

Animated RNN, LSTM and GRU

Line-by-Line Word2Vec Implementation (on word embeddings)

Step-by-Step Tutorial on Linear Regression with Stochastic Gradient Descent

Counting No. of Parameters in Deep Learning Models

Attn: Illustrated Attention

Illustrated: Self-Attention