Optimization

Table of Contents

[Random Search 2](#_Toc141041546)

[Gradients 3](#_Toc141041547)

[Gradient Descent 3](#_Toc141041548)

[Batch Gradient Descent 4](#_Toc141041549)

[SGD + Momentum 6](#_Toc141041550)

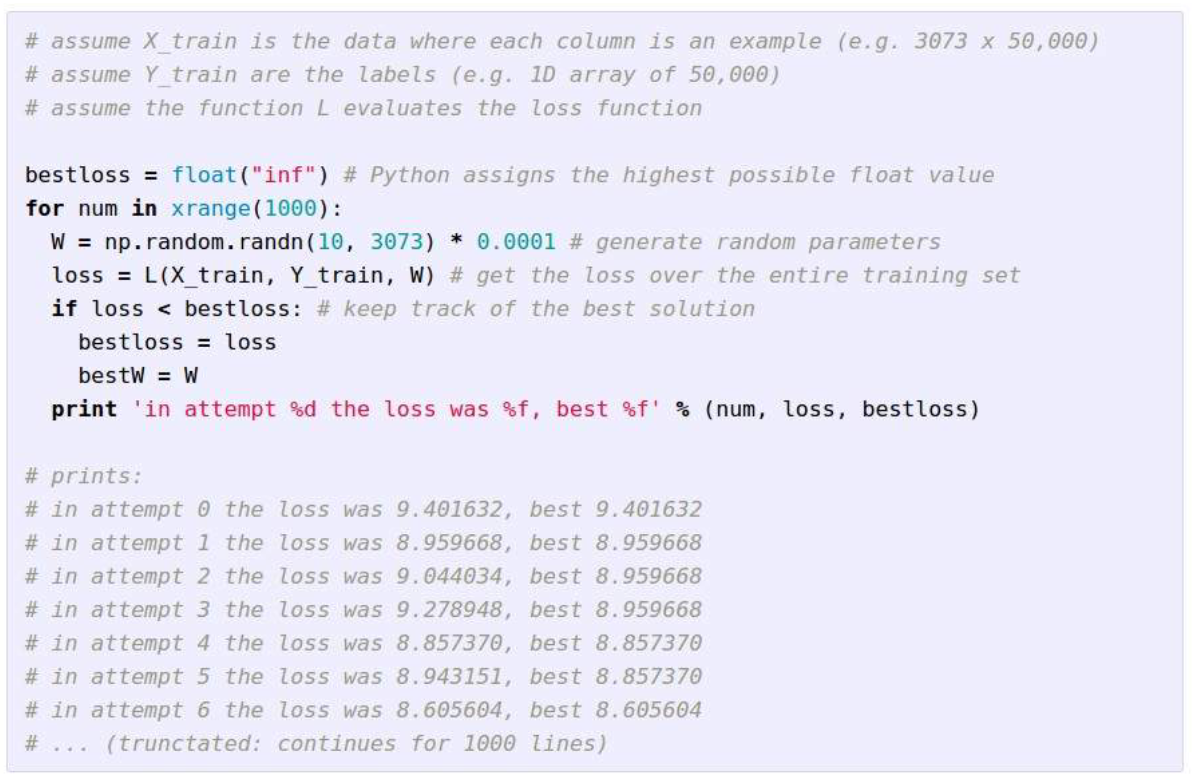
[AdaGrad 7](#_Toc141041551)

[Adam 8](#_Toc141041552)

We have seen how to calculate weights and how to determine if the outcomes produced by a certain set of weights are good or bad. Now we need to see how to actually update the weights so that they improve. This process is called **optimization**.

The process of optimization is a bit like going down a hill. We want to reduce loss, so we have to head in the direction where the loss is the least, i.e., the bottom of the hill. There are several approaches to doing this, each better than the last.

## Random Search

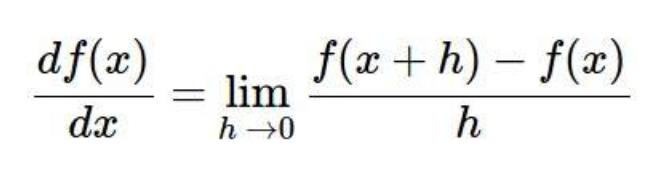


Just…no.

But in reality **random search** gives an accuracy of 15% for the CIFAR dataset which is better than the 10% you would expect (since there are 10 classes). The state of the art is 95% though so this is laughable.

## Gradients

Following the idea of going down a slope, we can use the **gradient** values. For 1 dimension, the **derivative** of a function gives the slope. For multiple dimensions, the gradient is the vector of **partial derivatives** along each dimension.



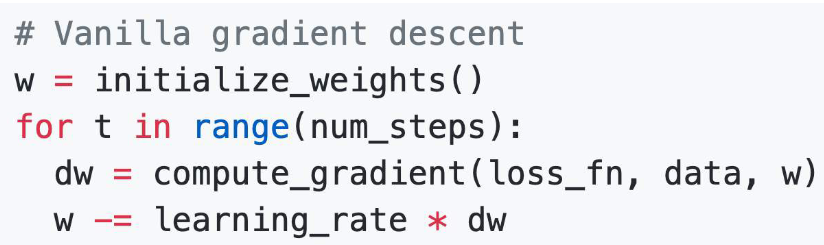
A manual calculation of each of the partial derivatives using the formula above is called the **numeric gradient**. This is slow and only approximates the gradient, but it is easy to calculate.

Another option is to use calculus to calculate the gradient. This is called the **analytic gradient**. This is exact and fast, but error-prone.

In practice, we always use the analytic gradient but check out work using the numeric gradient. This is called the **gradient check**.

## Gradient Descent

**Gradient Descent** is just the process of iteratively calculating the gradient and then updating the weights in the direction of the **negative gradient**, i.e., the direction of the steepest descent.

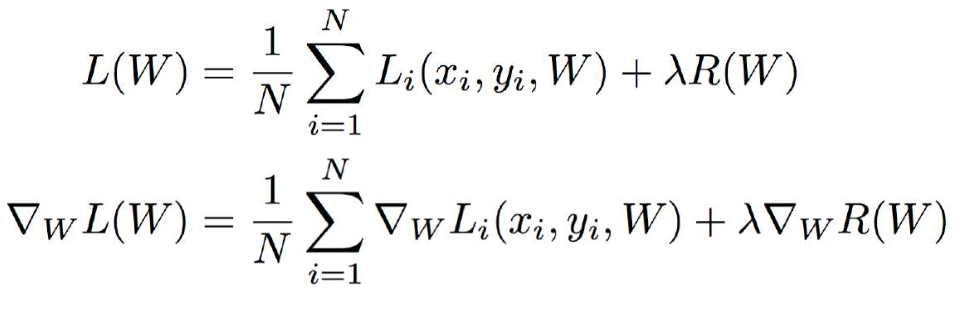


The **hyperparameters** in this case are:

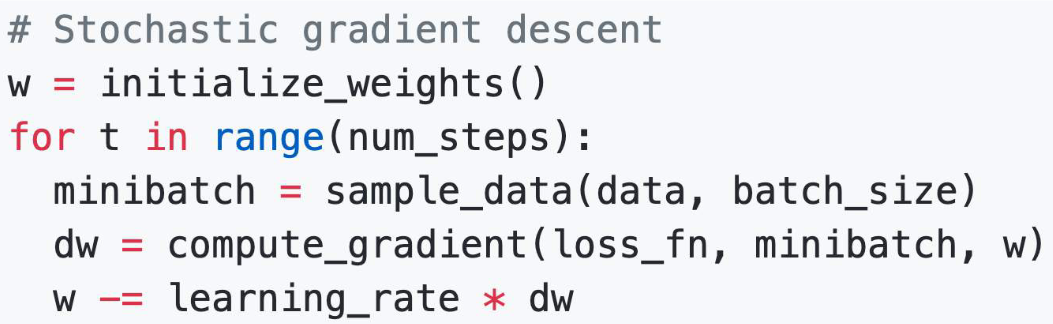
* Weight initialization method (zero or non-zero)
* Number of steps (epochs)
* Learning rate

## Batch Gradient Descent

For vanilla gradient descent, the calculation of loss for large amounts of data is expensive.



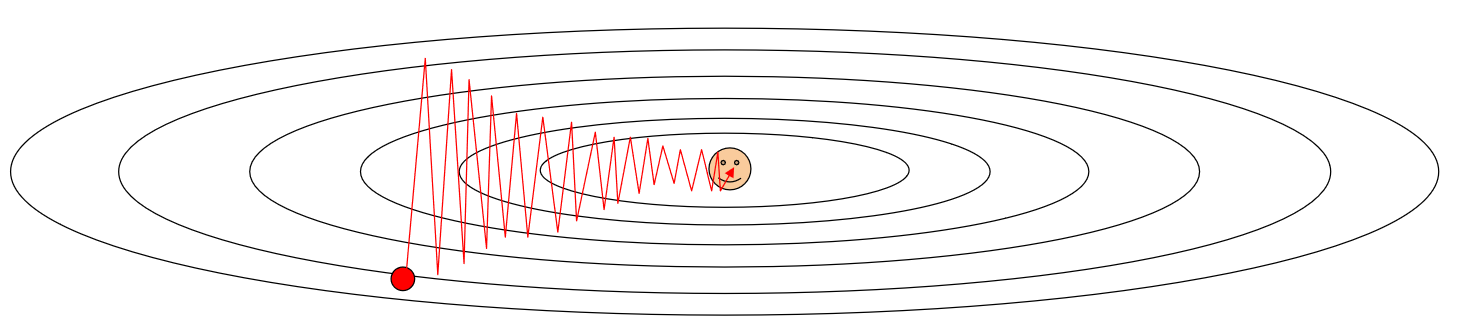
As an alternative, we can use **Stochastic Gradient Decent** or **Mini-Batch Gradient Descent**. In this case, we take a batch of data of limited size and calculate the gradient based on just this batch. We update the weights and then repeat for the rest of the mini batches. It is generally better to take larger batches, but how much data we can process at once depends on the memory available to our machine.



In addition to the previous hyperparameters, SGD adds two more hyperparameters:

* Batch Size
* How the data is sampled

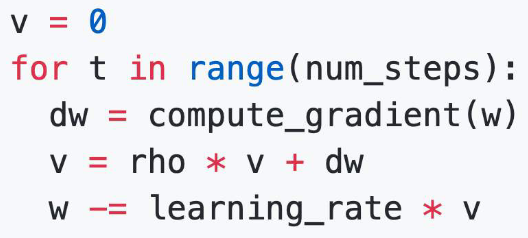
The main issue with SGD is if we have **multiple dimensions** and the loss changes quickly in one dimension but slowly in the other. This will cause the algorithm to **fluctuate** significantly in the direction with the steeper gradient.



Other issues include getting stuck at local minima and saddle points, and noisy gradients due to the fact that the gradients are being calculated based on minibatches and not the entire batch.

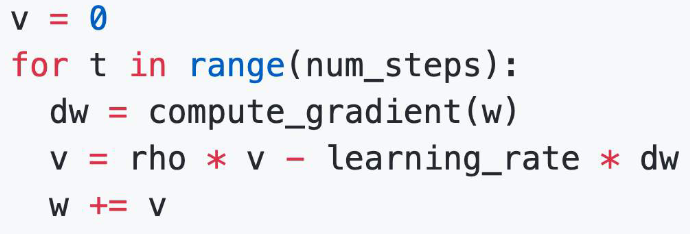
## SGD + Momentum

As a solution to the problems with SGD, **SGD + Momentum** was created. This is an idea that came from basic physics, in that the gradient update should be given a **velocity** and the new value should also depend on the velocity. By doing this, we can make the gradient remember its previous values to some extent.



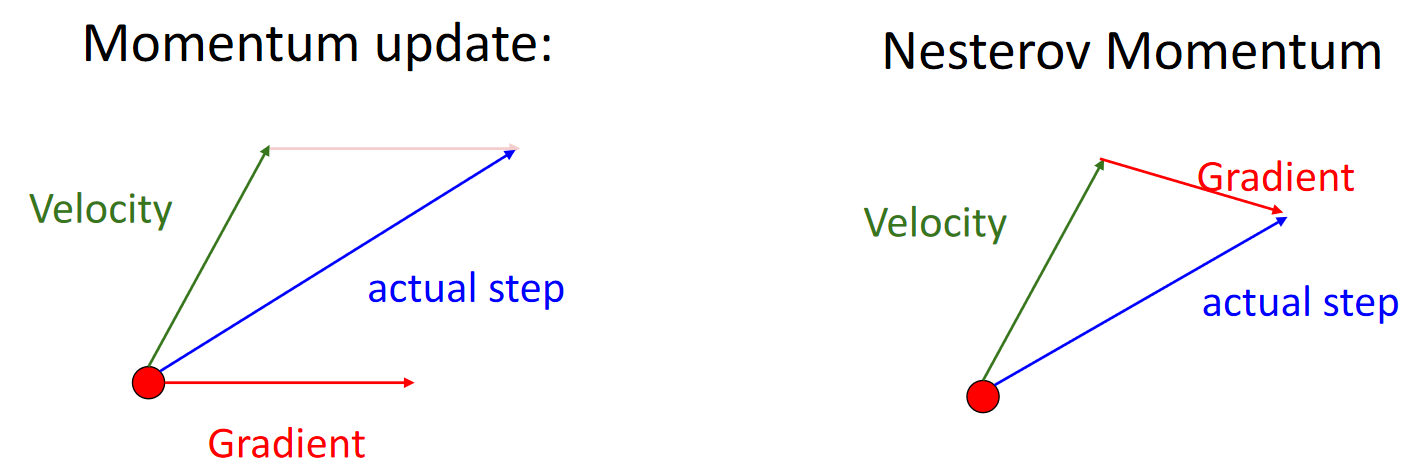
Here, is the **velocity** and a hyperparameter called **friction**, typically set to or .

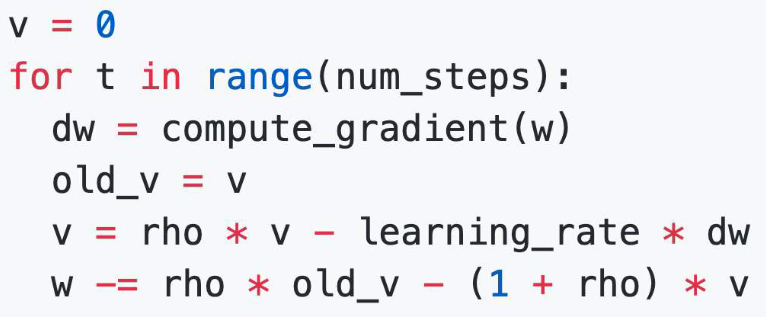
An alternative way of writing the same equations is shown below.



SGD + Momentum is both less likely to get stuck at local minima or saddle points and will have less noise.

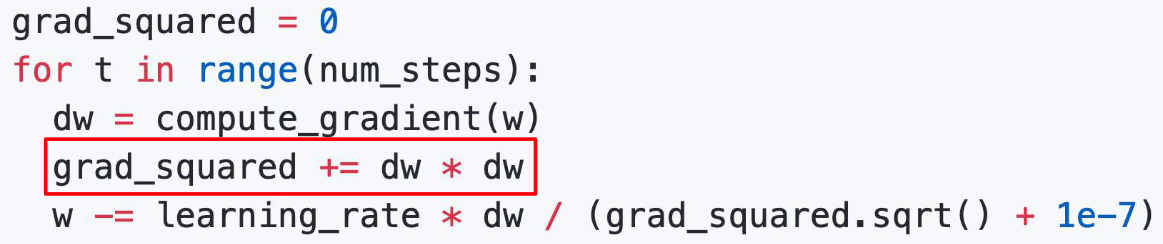
Yet another form of SGD + Momentum is called **Nesterov Momentum**, which mixes the gradient that would have been found had we just followed the velocity and mixes it in with the actual gradient.



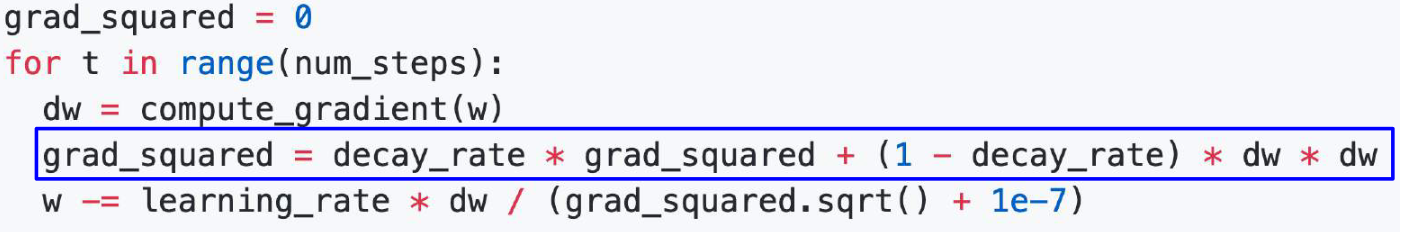


## AdaGrad

SGD + Momentum has the issue of overshooting the minima. **AdaGrad** solves this. It stands for adaptive gradient, and it adds an **element-wise scaling** of the gradients based on the historical sum of squares in each dimension. Simply put, it will increase the learning rate in the direction which has a flat gradient and decrease the learning rate in the direction which has a steep gradient.

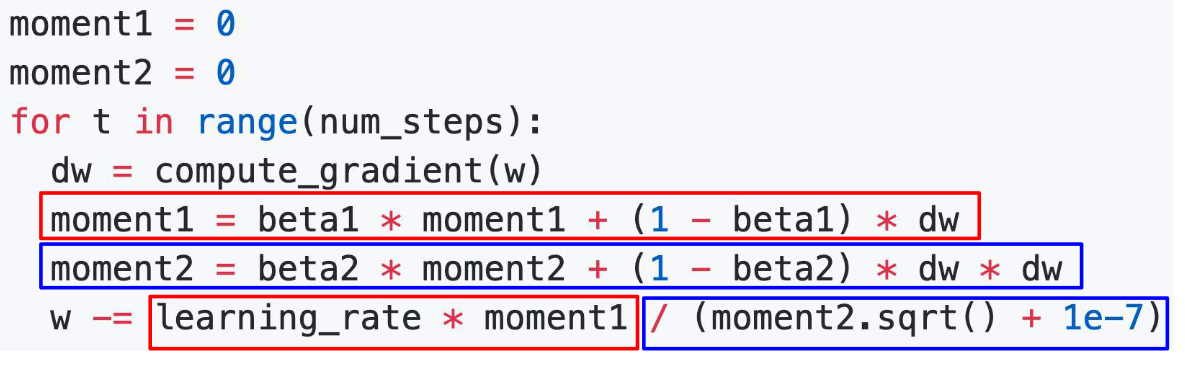


**RMSProp** is an alternative form of AdaGrad which adds a decay to the running squared gradient. In this way, it is comparable to the friction term used in SGD + Momentum. Only a fraction of the old values is retained, as determined by a new hyperparameter, **decay rate**.



## Adam

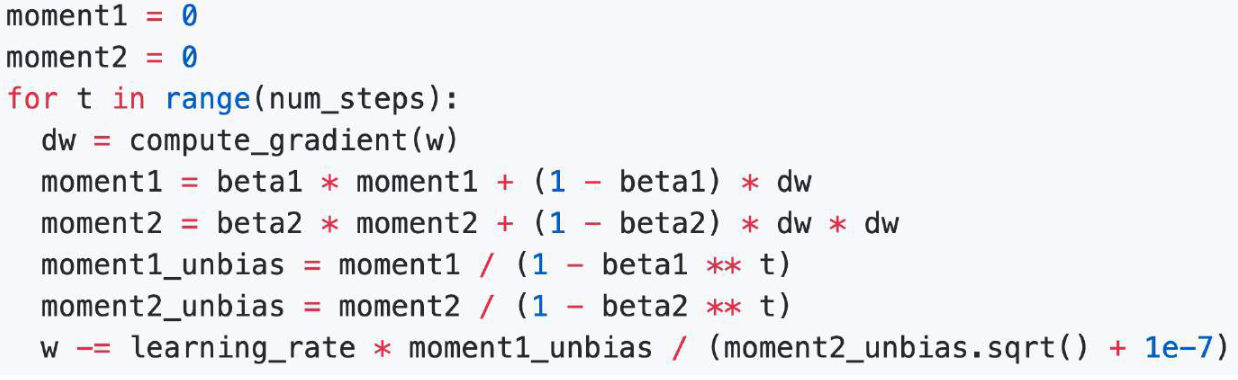
The **Adam** optimizer combines RMSProp with Momentum, thus achieving the best of both worlds.



Here, and are both hyperparameters.

The above algorithm has one drawback. The moment values both start at 0. Having high beta values will result in the moment values remaining close to 0. This in turn will result in a large gradient step at the very beginning of our training, which can make the model biased.

The Adam Optimizer actually provides a way to correct this bias. The moment values are modified to have less effect in the first few steps. Their effect is slowly increased as the values adjust to better estimates.



The above combination has resulted in the Adam optimizer being one of the most widely used optimizers in the entire domain of machine learning.