UE21CS343BB2

Topics in Deep Learning

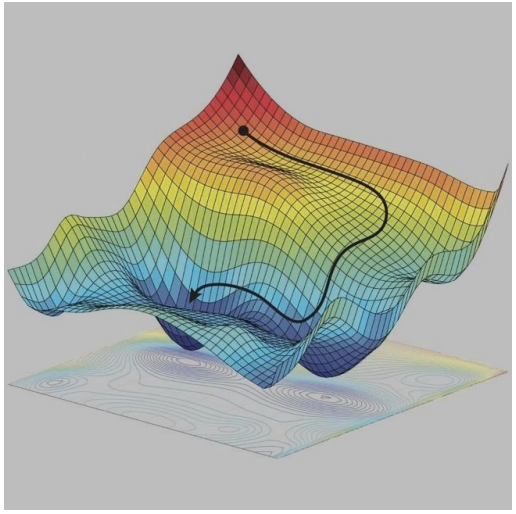
Dr. Shylaja S S
Director of Cloud Computing & Big Data (CCBD), Centre for Data Sciences & Applied Machine Learning (CDSAML)
Department of Computer Science and Engineering
shylaja.sharath@pes.edu

Ack:Anashua Krittika Dastidar, Teaching Assistant

Optimization

Optimization is a process of finding optimal parameters for the model, which significantly reduces the error function.

Optimization algorithms used for training of deep models differ from traditional optimization algorithms in several ways. In most machine learning scenarios we care about some performance measure say , P but we optimise P only indirectly by reducing a different cost function J(θ) in the hope that doing so will improve P.



The following are the optimization algorithms discussed in this course :

- Mini Batch gradient descent
- Stochastic gradient descent
- Gradient descent with momentum
- RMS Prop
- Adam

A Recap of Gradient Descent

Gradient Descent searched the hypothesis space of all possible weight vectors to find the best fit for all training examples. Say we have a cosy function J(w) and we wish to minimise this cost function:

1. We randomly initialize the weight vectors
2. We calculate the loss and update the weight vectors in the direction of steepest descent. The equation to do so is :

$$W_{new} = W - \alpha * \frac{\delta J(W)}{\delta W}$$

1. We repeat till we converge at a minima

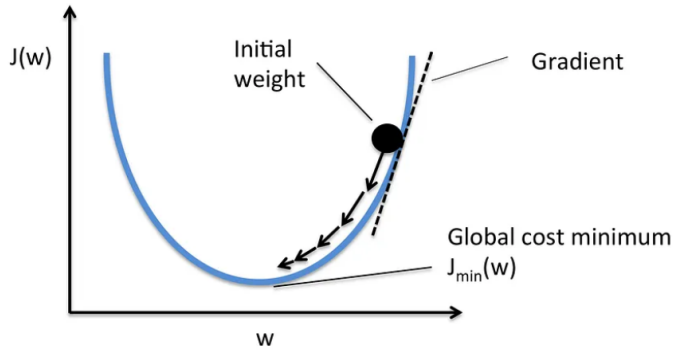
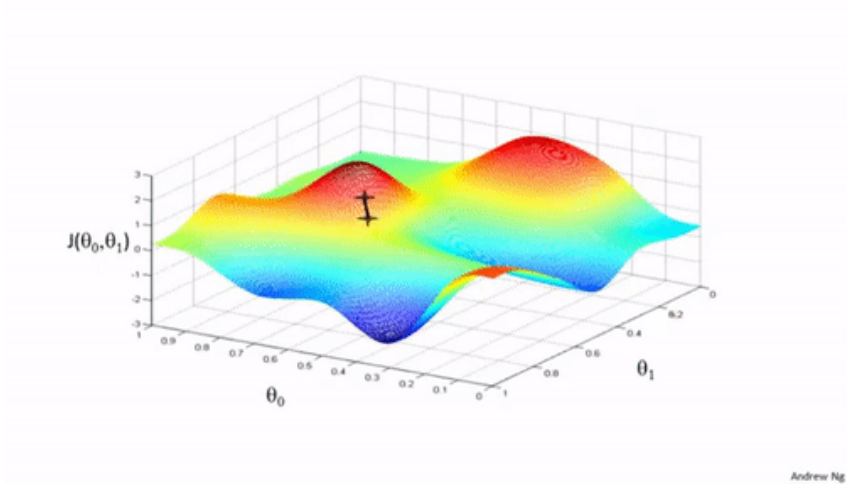


Figure 2: Example of minimizing J(w); (Source: MLextend)



Batch Gradient Descent

Batch gradient descent is a variation of the gradient descent algorithm that calculates the error for each example in the training dataset, but only updates the model after all training examples have been evaluated

Upsides

- Fewer updates to the model means this variant of gradient descent is more computationally efficient than stochastic gradient descent.
- The decreased update frequency results in a more stable error gradient and may result in a more stable convergence on some problems.
- The separation of the calculation of prediction errors and the model update lends the algorithm to parallel processing based implementations.

Downsides

- The more stable error gradient may result in premature convergence of the model to a less optimal set of parameters.
- The updates at the end of the training epoch require the additional complexity of accumulating prediction errors across all training examples.
- Commonly, batch gradient descent is implemented in such a way that it requires the entire training dataset in memory and available to the algorithm.
- Model updates, and in turn training speed, may become very slow for large datasets.

Stochastic Gradient Descent

Stochastic gradient descent, often abbreviated SGD, is a variation of the gradient descent algorithm that calculates the error and updates the model for each example in the training dataset.

Upsides

- The frequent updates immediately give an insight into the performance of the model and the rate of improvement.
- This variant of gradient descent may be the simplest to understand and implement, especially for beginners.
- The increased model update frequency can result in faster learning on some problems.
- The noisy update process can allow the model to avoid local minima (e.g. premature convergence).

Downsides

- Updating the model so frequently is more computationally expensive than other configurations of gradient descent, taking significantly longer to train models on large datasets.
- The frequent updates can result in a noisy gradient signal, which may cause the model parameters and in turn the model error to jump around (have a higher variance over training epochs).
- The noisy learning process down the error gradient can also make it hard for the algorithm to settle on an error minimum for the model.

Mini Batch Gradient Descent

Mini-batch gradient descent is a variation of the gradient descent algorithm that splits the training dataset into small batches that are used to calculate model error and update model coefficients. Mini-batch gradient descent seeks to find a balance between the robustness of stochastic gradient descent and the efficiency of batch gradient descent. It is the most common implementation of gradient descent used in the field of deep learning.

Upsides

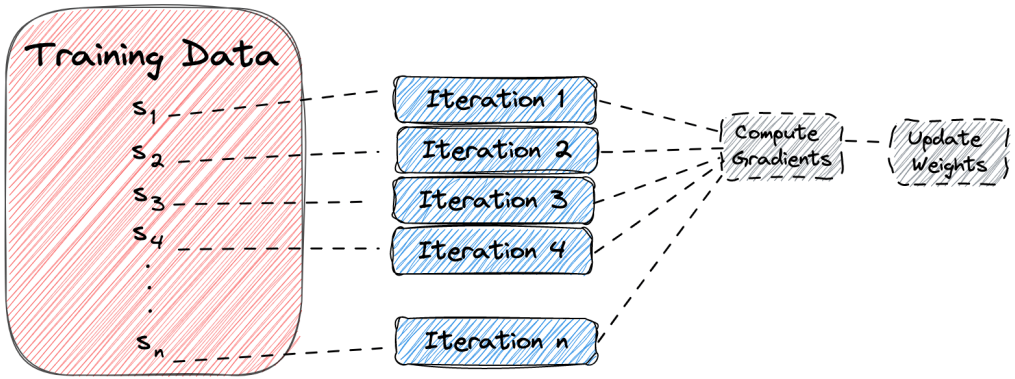
- The model update frequency is higher than batch gradient descent which allows for a more robust convergence, avoiding local minima.
- The batched updates provide a computationally more efficient process than stochastic gradient descent.
- The batching allows both the efficiency of not having all training data in memory and algorithm implementations.

Downsides

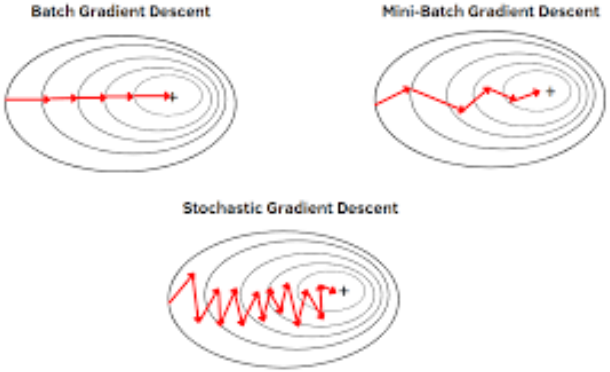
- Mini-batch requires the configuration of an additional “mini-batch size” hyperparameter for the learning algorithm.
- Error information must be accumulated across mini-batches of training examples like batch gradient descent

How to choose the best value for Mini Batch size

- If there is a small training set batch gradient descent is preferred.
- Mini batch size is a hyperparameter and it is a good idea to experiment with a range of values to get the best fit for your model.
- Mini-batch sizes, commonly called “batch sizes” for brevity, are often tuned to an aspect of the computational architecture on which the implementation is being executed. Such as a power of two that fits the memory requirements of the GPU or CPU hardware like 32, 64, 128, 256, and so on.



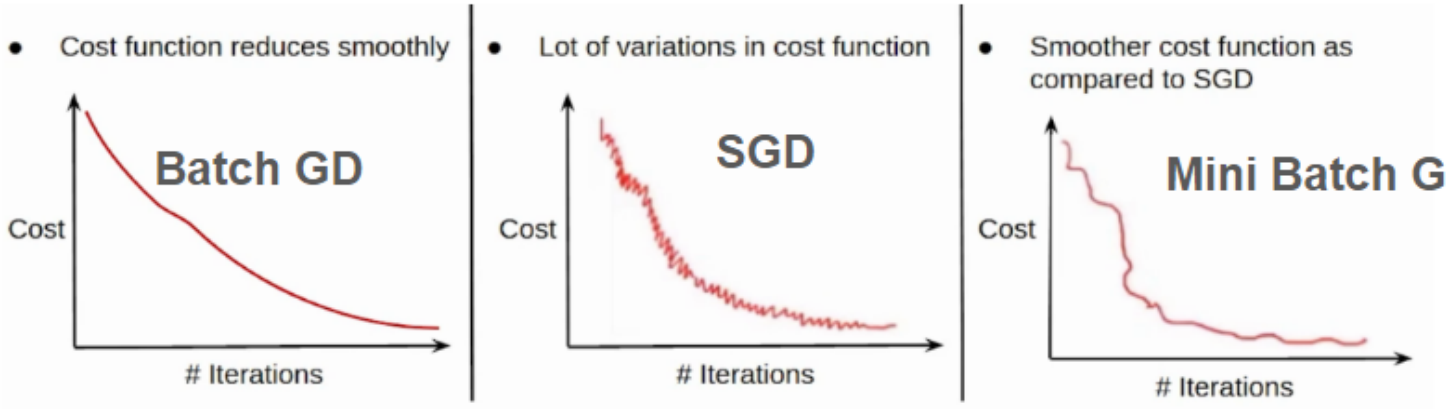
Convergence in different versions of Gradient Descent



⇒ Batch gradient descent takes small steps towards the minima and will converge to a minima.

⇒ While Stochastic gradient descent is noisy and oscillates near the minima . It never actually converges to the minima

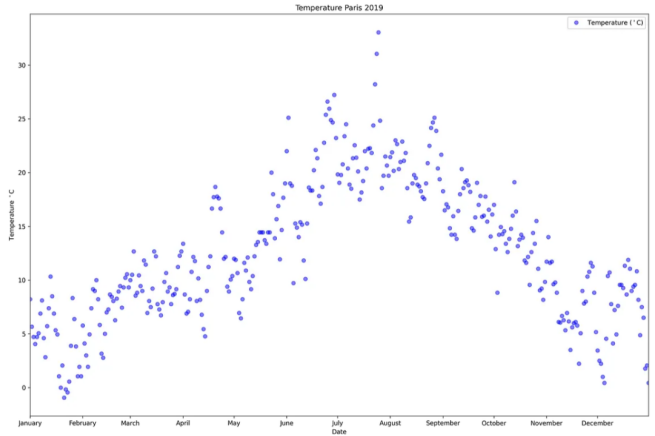
Comparison: Cost function



Exponentially Weighted Averages

The Exponentially Weighted Moving Average (EWMA) is commonly used as a smoothing technique in time series. However, due to several computational advantages (fast, low-memory cost), the EWMA is behind the scenes of many optimization algorithms in deep learning, including Gradient Descent with Momentum, RMSprop, Adam, etc.

Let's understand with an example , say we have the some data for temperatures across multiple days in a city and we wish to approximate the next day's temperature from this data. Thus let the estimated temperature be V_t , let the previous estimate be V_{t-1} , O_t be the temperature for day t and β be a hyperparameter.



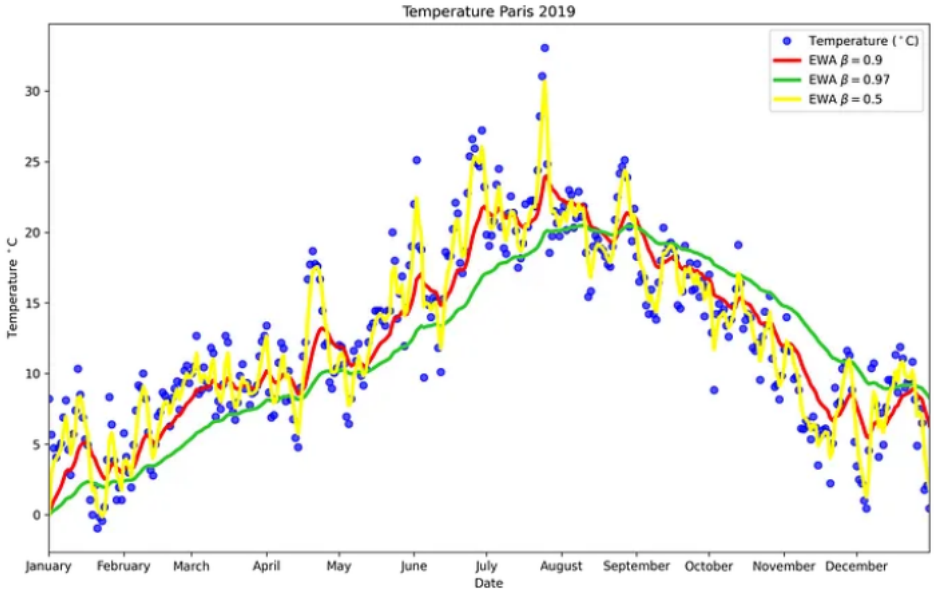
$$V_t = \beta * V_{t-1} + (1- \beta) * O_t$$

Here , β determines how important the previous value is (the trend), and (1-β) how important the current value is.

Exponentially Weighted Averages

Here , V_t is calculated by approximating over 1/ (1-β) days of temperature . Thus if β is chosen to be 0.9 then we approximate over the last 10 days if 0.5 then the last 2 days and so on.

β	Days
0.9	10
0.98	50
0.5	2



Note : As the value of β increases the curve becomes smoother and has less noise.

However with large values of β as we are averaging over a larger window the formula adapts more slowly to changes in data as high weightage is given to older values

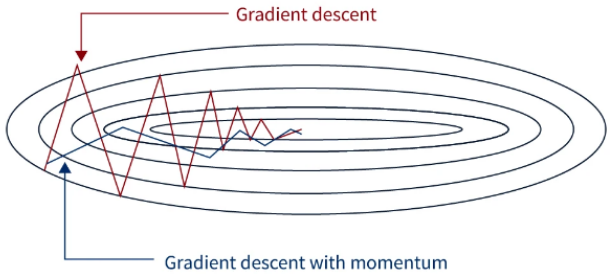
Gradient Descent with Momentum

In this method we compute the exponentially weighted average of the gradients and use this new value to calculate the weights instead

A problem with the gradient descent algorithm is that the progression of the search can bounce around the search space based on the gradient. For example, the search may progress downhill towards the minima, but during this progression, it may move in another direction, even uphill, depending on the gradient of specific points (sets of parameters) encountered during the search.

This can slow down the progress of the search, especially for those optimization problems where the broader trend or shape of the search space is more useful than specific gradients along the way.

One approach to this problem is to add history to the parameter update equation based on the gradient encountered in the previous updates



Gradient Descent with Momentum

1. First we calculate dW (where w is the weight) for the current mini batch

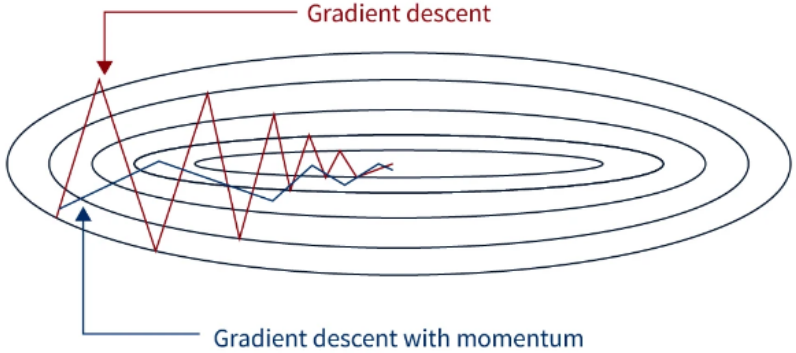
1. Then we compute

$$V_{dw} = \beta * V_{dw} + (1-\beta) * dw$$

1. Then we update our weights as

$$W_{new} = W - \alpha * V_{dw}$$

Using a very high value for β we can dampen out the oscillations which arise in gradient descent !



Gradient Descent with Momentum

Upsides:

1. Momentum has the effect of damping down the change in the gradient and, in turn, the step size with each new point in the search space.

1. Momentum is most useful in optimization problems where the objective function has a large amount of curvature (e.g. changes a lot), meaning that the gradient may change a lot over relatively small regions of the search space.

1. It is also helpful when the gradient is estimated, such as from a simulation, and may be noisy, e.g. when the gradient has a high variance.

1. Finally, momentum is helpful when the search space is flat or nearly flat, e.g. zero gradient. The momentum allows the search to progress in the same direction as before the flat spot and helpfully cross the flat region.

Downsides :

1. It can overshoot the global minimum and converge to a local minimum instead.
2. Another disadvantage is that the momentum term can cause the optimization process to oscillate around the global minimum.

Root Mean Square or RMS prop

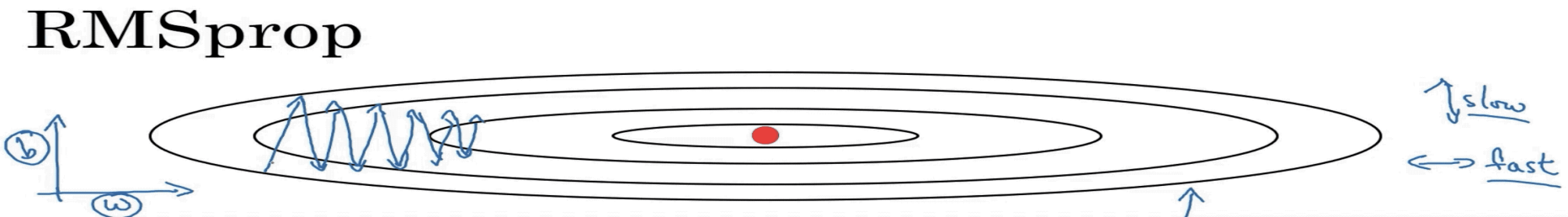
We know that while implementing gradient descent we end up with lots of oscillations
Let the horizontal direction be w and the vertical direction be b. In this case we wish to speed up learning in the w direction and slow down learning in the b direction.
for iteration t:

1. Calculate the derivative *d*W for the current mini batch ‘
2. Calculate an exponentially weighted average of the squares of derivatives

$$S_{dW} = \beta_2 * S_{dW} + (1 - \beta_2) * (dW)^2$$
$$S_{db} = \beta_2 * S_{db} + (1 - \beta_2) * (db)^2$$

NOTE : β_2 and β are separate hyperparameters

1. Then we update the weights as: $W_{new} = W - \alpha * \frac{dW}{\sqrt{SdW}}$ and $b_{new} = b - \alpha * \frac{db}{\sqrt{Sdb}}$



RMS prop intuition

- In this example we wish to dampen out oscillations in the b direction and to learn faster in the x direction.
- Thus as S_{dW} is a small number while S_{db} is a large number.

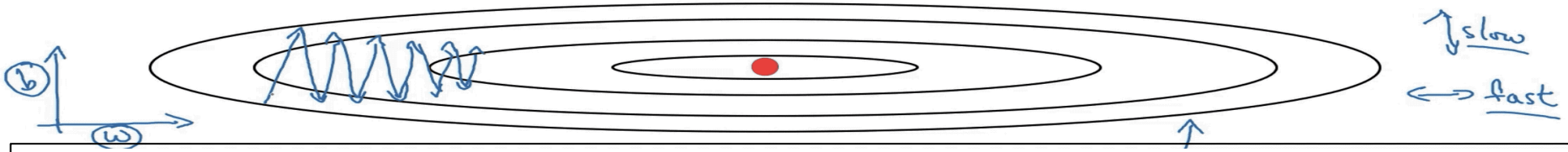
$S_{dW} = \beta_2 * S_{dW} + (1 - \beta_2) * (dW)^2$ ⇐ SMALL

$S_{db} = \beta_2 * S_{db} + (1 - \beta_2) * (db)^2$ ⇐ LARGE

$$W_{new} = W - \alpha * \frac{dW}{\sqrt{SdW}} \quad \text{and} \quad b_{new} = b - \alpha * \frac{db}{\sqrt{Sdb}}$$

- Thus by dividing by a small S_{dW} we can increase the magnitude of update in the W direction and while dividing with a large S_{db} we can dampen the oscillations in the b direction.

RMSprop



Note : if S_{dW} is very close to zero in order to avoid the W term becoming very large we add a small term ϵ to the denominator

$$W_{new} = W - \alpha * \frac{dW}{\sqrt{SdW} + \epsilon}$$

RMS Prop

Upsides:

1. Faster convergence: RMSprop can converge faster than SGD by adapting the learning rate based on the magnitude of the gradients for each parameter.

2. Robustness to different learning rates: RMSprop is more robust to different learning rates for different parameters, which can be helpful in deep learning models with many parameters.

3. Adaptive learning rate: RMSprop adaptively scales the learning rate based on the history of the squared gradients, which can improve the convergence speed and stability of the optimization process

Downsides:

1. RMSProp can sometimes lead to slow convergence.
2. It is sensitive to the learning rate.

Adam optimizer

Adam or Adaptive moment estimation optimizer is a combination of gradient descent with momentum and the RMSprop optimizer

1. First we initialize $V_{dW} = 0$,$V_{db} = 0$,$S_{dW} = 0$,$S_{db} = 0$
2. for iteration t :
   a. Compute *d*W and *d*b for the current mini batch
   b. Then compute :
      i. A momentum like update

$$V_{dw} = \beta_1 * V_{dw} + (1-\beta_1) * dW$$
$$V_{db} = \beta_1 * V_{db} + (1-\beta_1) * db$$

      i. A RMSprop like update

$$S_{dW} = \beta_2 * S_{dW} + (1 - \beta_2) * (dW)^2$$
$$S_{db} = \beta_2 * S_{db} + (1 - \beta_2) * (db)^2$$

Note :Adam optimization applies bias correction to the first and second moment estimates to ensure that they are unbiased estimates of the true values.

So, the bias corrected terms:

$$V_{dW}^{C} = \frac{V_{dW}}{1-\beta_1^t} \qquad V_{db}^{C} = \frac{V_{db}}{1-\beta_1^t}$$

$$S_{dW}^{C} = \frac{V_{dW}}{1-\beta_2^t} \qquad S_{db}^{C} = \frac{V_{db}}{1-\beta_2^t}$$

Adam optimizer

Thus the weights will be updated as :

$$W_{new} = W - \alpha * \frac{V_{dW}^{C}}{\sqrt{S_{dW}^{C}} + \epsilon} \quad \text{and} \quad b_{new} = b - \alpha * \frac{V_{db}^{C}}{\sqrt{S_{db}^{C}} + \epsilon}$$

Adam Configuration Parameters

- **alpha**. Also referred to as the learning rate or step size. The proportion that weights are updated (e.g. 0.001). Larger values (e.g. 0.3) results in faster initial learning before the rate is updated. Smaller values (e.g. 1.0E-5) slow learning right down during training
- **beta1**. The exponential decay rate for the first moment estimates (e.g. 0.9).
- **beta2**. The exponential decay rate for the second-moment estimates (e.g. 0.999). This value should be set close to 1.0 on problems with a sparse gradient (e.g. NLP and computer vision problems).
- **epsilon**. Is a very small number to prevent any division by zero in the implementation (e.g. 10E-8).

Further, learning rate decay can also be used with Adam.

Adam optimizer

Advantages :

1. **Adaptive Learning Rates**: Unlike fixed learning rate methods like SGD, Adam optimization provides adaptive learning rates for each parameter based on the history of gradients. This allows the optimizer to converge faster and more accurately, especially in high-dimensional parameter spaces.

1. **Momentum**: Adam optimization uses momentum to smooth out fluctuations in the optimization process, which can help the optimizer avoid local minima and saddle points.

1. **Bias Correction**: Adam optimization applies bias correction to the first and second moment estimates to ensure that they are unbiased estimates of the true values.

1. **Robustness**: Adam optimization is relatively robust to hyperparameter choices and works well across a wide range of deep learning architectures.

Adam optimizer

Disadvantages :

Memory intensive: Adam needs to store moving averages of past gradients for each parameter during training and hence it requires more memory than some other optimization algorithms, particularly when dealing with very large neural networks or extensive datasets.

Slower convergence in some cases: While Adam usually converges quickly, it might converge to flawed solutions in some cases or tasks. In such scenarios, other optimization algorithms like *SGD (stochastic gradient descent)* with momentum or *Nesterov accelerated gradient (NAG)* may perform better.

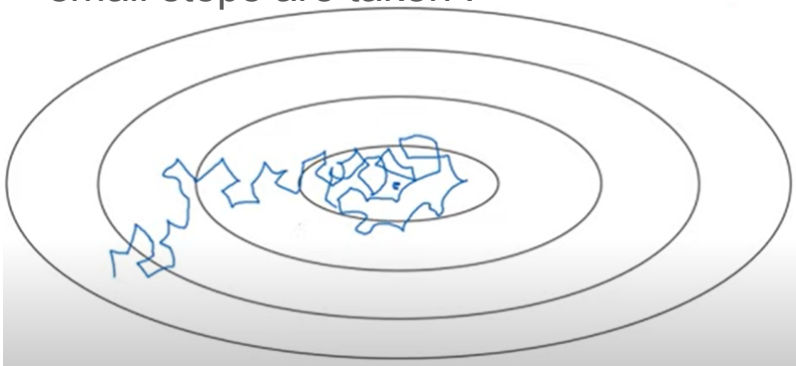
Hyperparameter sensitivity: Although Adam is less sensitive to hyperparameter choices than some other algorithms, it still has hyperparameters like the learning rate, beta1, and beta2. Choosing inappropriate values for these hyperparameters could impact the performance of the algorithm.

Learning Rate Decay

Learning rate decay is a technique used in machine learning models to train modern neural networks. It involves starting with a large learning rate and then gradually reducing it until local minima is obtained.

⇒ Say, we are implementing mini batch gradient descent with a relatively small batch size and the gradient takes noisy steps as shown below. It also does not converge at the minima but oscillates around the minimum value.

⇒ In such a situation slowly reducing the learning rate (α) value as we approach the minima could be advantageous as the final value would oscillate closer to the minima if small steps are taken .



This could be done by setting :

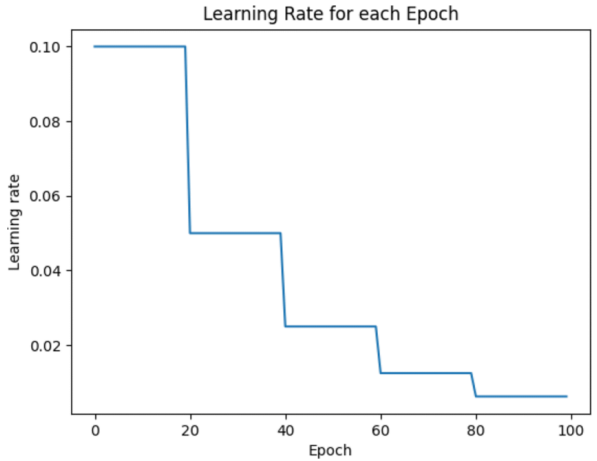
$$\alpha = \frac{1}{1 + \text{decay rate} * \text{epoch number}} * \alpha_0$$

Here decay rate is yet another hyperparameter while is α_0 the initial learning rate value.

Other learning rate decay methods

1. Exponential decay: $\alpha = decay\ rate^{epoch\ number} * \alpha_0$

2. Epoch number Based decay: $\alpha = \frac{constant}{\sqrt{epoch\ number}} * \alpha_0$



3. Discrete Step decay : In this method learning rate is decreased in some discrete steps after every certain interval of time , for example you are reducing learning rate to its half after every 10 secs..

4. Manual decay : In this method practitioners manually examine the performance of algorithm and decrease the learning rate manually day by day or hour by hour etc.

Resources

- https://www.scaler.com/topics/momentum-based-gradient-descent/
- https://medium.com/mlearning-ai/exponentially-weighted-average-5eed00181a09
- https://medium.com/nerd-for-tech/optimizers-in-machine-learning-f1a9c549f8b4
- https://www.analyticsvidhya.com/blog/2021/03/variants-of-gradient-descent-algorithm/
- https://machinelearningmastery.com/gradient-descent-with-momentum-from-scratch/
- https://machinelearningmastery.com/gentle-introduction-mini-batch-gradient-descent-configure-batch-size/
- https://youtube.com/playlist?list=PLkDaE6sCZn6Hn0vK8co82zjQtt3T2Nkqc&si=7Z_y1LQXzn7fZF1I
- https://arxiv.org/pdf/1412.6980.pdf

UE21CS343BB2

Topics in Deep Learning

Dr. Shylaja S S
Director of Cloud Computing & Big Data (CCBD), Centre for Data Sciences & Applied Machine Learning (CDSAML)
Department of Computer Science and Engineering
shylaja.sharath@pes.edu

Ack:Anashua Krittika Dastidar, Teaching Assistant