Contents

1 Discussion on Optimizer **2**
1.1 What is optimizer? . . . 2
1.2 Importance of optimizer . . . 2
1.3 Terminology . . . 2
1.4 Cost Function . . . 2
1.5 Learning rate . . . 3
1.5.1 Effect of different values for learning rate . . . 4
1.5.2 Is there a better way to determine the learning rate? . . . 4
1.6 Various Optimizers for neural network . . . 4
1.6.1 SGD . . . 5
1.6.2 SGD with Momentum . . . 9
1.6.3 Nesterov accelerated gradient(NAG) . . . 10
1.6.4 Adagrad . . . 11
1.6.5 RMSProp . . . 12
1.6.6 Adam . . . 12
1.7 Performance of various optimizers . . . 13
1.8 Which optimizer should we use? . . . 15

2 Reference **16**

1 Discussion on Optimizer

It is very important to tweak the weights of the neural network during the training process, to make our predictions as correct and optimized as possible. An optimization algorithm finds the value of the parameters(weights) that minimize the error when mapping inputs to outputs. These optimization algorithms or optimizers widely affect the accuracy of the deep learning model. They as well as affect the speed of training of the model.

1.1 What is optimizer?

While training the deep learning model, we need to modify the model's weights and minimize the loss function. An optimizer is a function or an algorithm that modifies the attributes of the neural network, such as weights and learning rate. Thus, it helps in reducing the overall loss and improve the accuracy. The problem of choosing the right weights for the model is a daunting task, as a deep learning model generally consists of millions of parameters. It raises the need to choose a suitable optimization algorithm for a given application.

1.2 Importance of optimizer

One can use different optimizers to update the model's weights and learning rate. However, choosing the best optimizer depends upon the application. One evil thought that comes to mind is to try all the possibilities and choose the one that shows the best results. This might not be a problem initially, but when dealing with hundreds of gigabytes of data, even a single epoch can take a considerable amount of time. So randomly choosing an algorithm is no less than gambling with precious time.

1.3 Terminology

Epoch : The number of times the algorithm runs on the whole training dataset.

Sample : A single row of a dataset.

Batch : It denotes the number of samples to be taken to for updating the model parameters.

Learning rate : It is a parameter that provides the model a scale of how much model weights should be updated.

Cost Function/Loss Function : A cost function is used to calculate the difference between the predicted value and the actual value.

Weights/ Bias : The learnable parameters in a model that controls the signal between two neurons.

1.4 Cost Function

A cost function is a measure of the error in prediction committed by an algorithm. It indicates the difference between the predicted and the actual values for a given dataset. Closer the predicted value to the actual value, the smaller the difference and lower the value of the cost function. Lower the value of the cost function, the better the predictive capability of the model. An ideal value of the cost function is zero. Some of the popular cost functions used in machine learning for applications such as regression, classification, and density approximation
Let's consider the cost function for regression, in which the objective is to learn a mapping function between the predictors (independent variables) and target (dependent variable). If we assume the relationship to be linear, the equation for the predicted value ($\hat{y}_i$) is shown below

$$\hat{y}_i = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + e_i \tag{1}$$

In equation 1, x_1 and x_2 are the two predictors and $\beta_0, \beta_1, \beta_2$ are the model parameters. The algorithm learns(estimates) the values of these parameters during training
In regression, the typical cost function (CF) used is the mean squared error (MSE) cost function. The form of the function is shown below.

$$CF = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{2}$$

In the equation 2, y_i and $\hat{y}_i$ are respectively the actual and predicted values, n is the number of records in the dataset. Replacing $\hat{y}_i$ in the above equation, the cost function can be re-written as shown below

$$CF = \frac{1}{n}\sum_{i=1}^{n}(y_i - (\beta_0 + \beta_1 x_1 + \beta_2 x_2))^2 \tag{3}$$

In the equation 3, it is important to note that the values for $y_i, x_{1i} and x_{2i}$ come from the dataset and cannot be manipulated to minimize the cost function. Only the model parameters $\beta_0, \beta_1, \beta_2$ can be manipulated to minimize the cost function.

1.5 Learning rate

The learning rate is a hyperparameter that controls how much to change the model in response to the estimated error each time the model weights are updated. Choosing the learning rate is challenging as a value too small may result in a long training process that could get stuck, whereas a value too large may result in learning a sub-optimal set of weights too fast or an unstable training process.
The learning rate is the most important hyperparameter when configuring the neural network.Therefore it is vital to know the effects of the learning rate on model performance and to build an intuition about the dynamics of the learning rate on model behavior.Also the learning rate affects how quickly our model can converge to a local minima. Thus getting it right from the get go would mean lesser time for us to train the model.

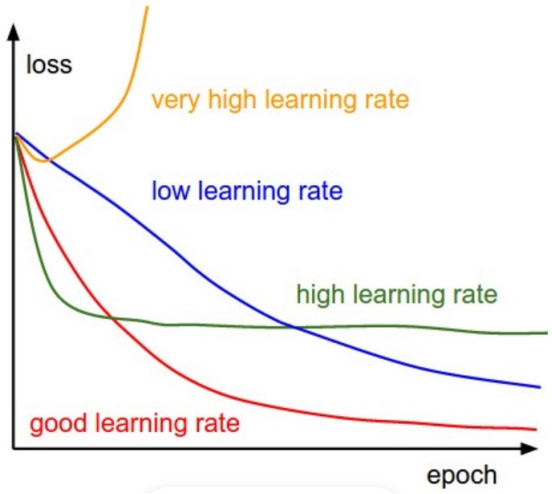


Figure 1: Plot of model's loss versus training epoch obtained for different learning rate

1.5.1 Effect of different values for learning rate

Learning rate is used to scale the magnitude of parameter updates during gradient descent. The choice of the value for learning rate can impact two things: 1) how fast the algorithm learns and 2) whether the cost function is minimized or not. Figure 2 shows the variation in cost function with a number of iterations/epochs for different learning rates.
It can be seen that for an optimal value of the learning rate, the cost function value is minimized in a few iterations (smaller time).If the learning rate used is lower than the optimal value, the number of iterations/epochs required to minimize the cost function is high (takes longer time). If the learning rate is high, the cost function could saturate at a value higher than the minimum value. If the learning rate is very high, the cost function could continue to increase with iterations/epochs. An optimal learning rate is not easy to find for a given problem. Though getting the right learning is always a challenge, there are some well-researched methods documented to figure out optimal learning rates. Some of these techniques will be discussed in the following sections. In all these techniques the fundamental idea is to vary the learning rate dynamically instead of using a constant learning rate.

1.5.2 Is there a better way to determine the learning rate?

One could estimate a good learning rate by training the model initially with a very low learning rate and increasing it (either linearly or exponentially) at each iteration.If we record the learning at each iteration and plot the learning rate (log) against loss; we will see that as the learning rate increase, there will be a point where the loss stops decreasing and starts to increase. In practice, our learning rate should ideally be somewhere to the left to the lowest point of the graph (as demonstrated in below graph). In this case, 0.001 to 0.01.

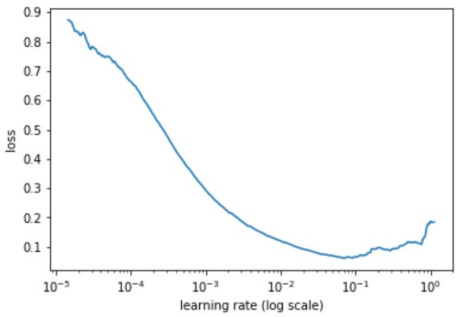


Figure 2: Loss Vs learning rate

1.6 Various Optimizers for neural network

Following are the commonly used optimizers:

- Stochastic Gradient Descent(SGD)
- SGD with Momentum
- Nesterov accelerated gradient
- Adagrad
- RMSProp

- Adam

These optimizers will be explained briefly.

1.6.1 SGD

Stochastic gradient descent is a very popular and common algorithm used in various Machine Learning algorithms, most importantly forms the basis of Neural Networks.

Objective of Gradient Descent:

Gradient, in plain terms means slope or slant of a surface. So gradient descent literally means descending a slope to reach the lowest point on that surface. Let us imagine a two dimensional graph, such as a parabola in the figure below.

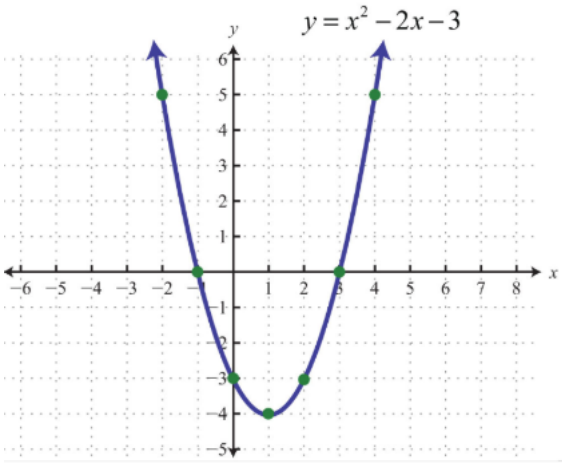


Figure 3: A parabolic function with two dimensions (x,y)

In the above graph, the lowest point on the parabola occurs at x = 1. The objective of gradient descent algorithm is to find the value of **x** such that **y** is minimum. **y** here is termed as the objective function that the gradient descent algorithm operates upon, to descend to the lowest point.

Gradient descent variants

There are three variants of gradient descent, which differ in how much data we use to compute the gradient of the objective function. Depending on the amount of data, we make a trade-off between the accuracy of the parameter update and the time it takes to perform an update.

Batch gradient descent

Vanilla gradient descent, aka batch gradient descent, computes the gradient of the cost function w.r.t. to the parameters θ for the entire training dataset

$$\theta_{new} = \theta_{old} - \eta \nabla_\theta J(\theta) \tag{4}$$

As we need to calculate the gradients for the whole dataset to perform just one update, batch gradient descent can be very slow and is intractable for datasets that don't fit in memory.

In code, batch gradient descent looks something like this:

```
for i in range(nb_epochs):
    params_grad = evaluate_gradient(loss_function, data, params)
    params = params - learning_rate * params_grad
```

Figure 4: Code for batch gradient descent

For a pre-defined number of epochs, we first compute the gradient vector params_grad of the loss function for the whole dataset w.r.t our parameter vector params. Note that state-of-the-art deep learning libraries provide automatic differentiation that efficiently computes the gradient w.r.t. some parameters. If we derive the gradients ourself, then gradient checking is a good idea. We then update our parameters in the opposite direction of the gradients with the learning rate determining how big of an update we perform. Batch gradient descent is guaranteed to converge to the global minimum for convex error surfaces and to a local minimum for non-convex surfaces.

Stochastic gradient descent

Stochastic gradient descent (SGD) in contrast performs a parameter update for each training example $x^{(i)}$ and label $y^{(i)}$:

$$\theta_{new} = \theta_{old} - \eta \nabla_\theta J(\theta; x^{(i)}; y^{(i)}) \tag{5}$$

Batch gradient descent performs redundant computations for large datasets, as it recomputes gradients for similar examples before each parameter update. SGD does away with this redundancy by performing one update at a time. It is therefore usually much faster and can also be used to learn online. SGD performs frequent updates with a high variance that cause the objective function to fluctuate heavily.**Stochastic**, in plain terms means **random**.

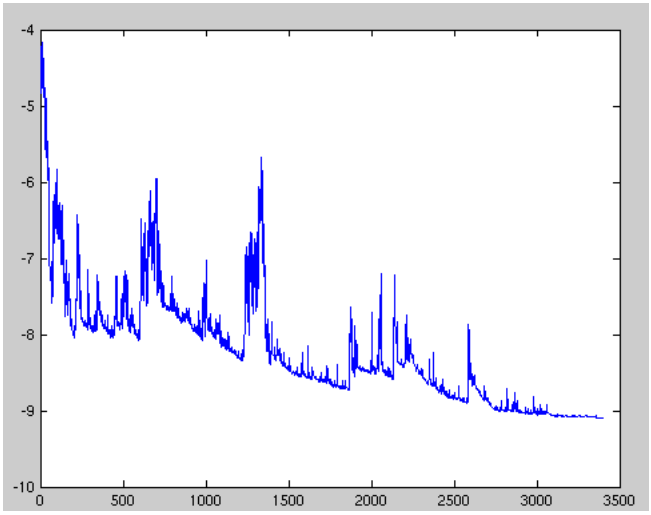


Figure 5: SGD fluctuation(Source:Wikipedia)

While batch gradient descent converges to the minimum of the basin the parameters are placed in, SGD's fluctuation, on the one hand, enables it to jump to new and potentially better local minima. On the other hand, this ultimately complicates convergence to the exact minimum, as SGD will keep overshooting. However, it has been shown that when we slowly decrease the learning rate, SGD shows the same convergence behaviour as batch gradient descent, almost certainly converging to a local or the global minimum for non-convex and convex optimization respectively.

Where can we potentially induce randomness in our gradient descent algorithm

It is while selecting data points at each step to calculate the derivatives. SGD randomly picks one data point from the whole data set at each iteration to reduce the computations enormously.

Its code fragment simply adds a loop over the training examples and evaluates the gradient w.r.t. each example. Note that we shuffle the training data at every epoch.

```
optimizer = optim.SGD(model.parameters(), lr = 0.01)
```

Figure 6: Code for SGD

Mathematics for SGD

In deep learning, the objective function is usually the average of the loss functions for each example in the training dataset. Given a training dataset of n examples, we assume that $f_i(x)$ is the loss function with respect to the training example of index i, where x is the parameter vector. Then we arrive at the objective function

$$f(x) = \frac{1}{n}\sum_{i=1}^{n} f_i(x) \tag{6}$$

The gradient of the objective function at x is computed as

$$\nabla f(x) = \frac{1}{n}\sum_{i=1}^{n} \nabla f_i(x) \tag{7}$$

At each iteration of stochastic gradient descent, we uniformly sample an index $i \in 1, 2, ..., n$ for data examples at random, and compute the gradient $\nabla f_i(x)$ to update x :

$$\theta_{new} = \theta_{old} - \eta \nabla f_i(x) \tag{8}$$

where η is the learning rate. We can see that the computational cost for each iteration drops from $O(n)$ of the gradient descent to the constant $O(1)$. Moreover, we want to emphasize that the stochastic gradient $\nabla f_i(x)$ is an unbiased estimate of the full gradient $\nabla f(x)$ because

$$E_i[\nabla f(x)] = \frac{1}{n}\sum_{i=1}^{n} \nabla f_i(x) \tag{9}$$

This means that, on average, the stochastic gradient is a good estimate of the gradient.

Dynamic Learning Rate

Replacing η with a time dependent learning rate $\eta(t)$ adds to the complexity of controlling convergence of an optimization algorithm. In particular, we need to figure out how rapidly η should decay. If it is too quick, we will stop optimizing prematurely. If we decrease it too slowly, we waste too much time on optimization. The following are a few basic strategies that are used in adjusting η over time

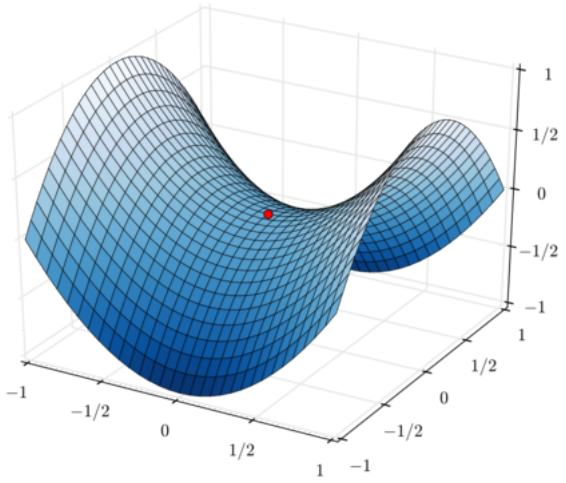
$$\begin{aligned}
&\eta(t) = \eta_i \text{ if } t_i \leq t_{i+1} && \text{piecewise constant} \\
&\eta(t) = \eta_0 . \exp^{-\lambda t} && \text{exponential decay} \\
&\eta(t) = \eta_0 . (\beta t + 1)^{-\alpha} && \text{polynomial decay}
\end{aligned}$$

In the first piecewise constant scenario we decrease the learning rate, e.g., whenever progress in optimization stalls. This is a common strategy for training deep networks. Alternatively we could decrease it much more aggressively by an exponential decay. Unfortunately this often leads to premature stopping before the algorithm has converged. In the case of convex optimization there are a number of proofs that show that this rate is well behaved.

Pros and Cons of SGD

Pros:
- Only a single observation is being processed by the network so it is easier to fit into memory
- May (likely) to reach near the minimum (and begin to oscillate) faster than Batch Gradient Descent on a large dataset
- The frequent updates create plenty of oscillations which can be helpful for getting out of local minimums.

Cons:
- Can veer off in the wrong direction due to frequent updates
- Lose the benefits of vectorization since we process one observation per time
- Frequent updates are computationally expensive due to using all resources for processing one training sample at a time.
- Gradients come from minibatches for this case. So they can be noisy.
- Get stuck if the cost function has a local minima or saddle point as at that point gradiant became 0. For example one can check with the cost function $f(x, y) = x^2 - y^2$. Figure of the cost function is shown below. Red point in the figure indicates the saddle point.



Mini Batch gradient descent

Batch Gradient Descent can be used for smoother curves. SGD can be used when the dataset is large. Batch Gradient Descent converges directly to minima. SGD converges faster for larger datasets. But, since in SGD we use only one example at a time, we cannot implement the vectorized implementation on it. This can slow down the computations. To tackle this problem, a mixture of Batch Gradient Descent and SGD is used.

Here neither we use the dataset all at once nor we use the single example at a time.We use a batch of a fixed number of training examples which is less than the actual dataset and call it a mini-batch. Doing this helps us achieve the advantages of SGD and batch gradient decent. Just like SGD, the average cost over the

epochs in mini-batch gradient descent fluctuates because we are averaging a small number of examples at a time. So, when we are using the mini-batch gradient descent we are updating our parameters frequently as well as we can use vectorized implementation for faster computations.

1.6.2 SGD with Momentum

It always works better than the normal Stochastic Gradient Descent Algorithm. The problem with SGD is that while it tries to reach minima because of the high oscillation we can't increase the learning rate. So it takes time to converge. In this algorithm, we will be using Exponentially Weighted Averages to compute Gradient and used this Gradient to update parameter. In short SGD with Momentum is a stochastic optimization method that adds a momentum term to regular stochastic gradient descent.

$$V_t = \beta V_{t-1} + (1-\beta)\nabla_w L(W, X, y) \tag{10}$$

$$W = W - \eta V_t \tag{11}$$

Where L is loss function, ∇_w is gradient w.r.t weight and η is learning rate.In code, SGD with momentum looks like this:

```
optimizer = optim.SGD(model.parameters(), lr = 0.01, momentum = 0.9)
```

Figure 7: Code for SGD with momentum

Change of learning rate per epoch is shown below(Here EMNIST dataset is taken as dataset)

```
Train Loss: 0.295 | Accuracy: 91.212
Epoch : 1
Testing loss in 100th step is 0.09365995228290558
Test Loss: 0.136 | Accuracy: 96.240
Epoch : 2
Train Epoch: 2 batch-0  Loss: 0.122800 Learning Rate: 0.001
Train Loss: 0.075 | Accuracy: 97.752
Epoch : 2
Testing loss in 100th step is 0.09112879633903503
Test Loss: 0.072 | Accuracy: 98.240
Epoch : 3
Train Epoch: 3 batch-0  Loss: 0.068369 Learning Rate: 0.0001
Train Loss: 0.068 | Accuracy: 97.973
Epoch : 3
Testing loss in 100th step is 0.09299802035093307
Test Loss: 0.070 | Accuracy: 98.260
Epoch : 4
Train Epoch: 4 batch-0  Loss: 0.046422 Learning Rate: 1e-05
Train Loss: 0.068 | Accuracy: 97.927
Epoch : 4
Testing loss in 100th step is 0.09366407990455627
Test Loss: 0.070 | Accuracy: 98.280
```

Figure 8: Changing the learning rate

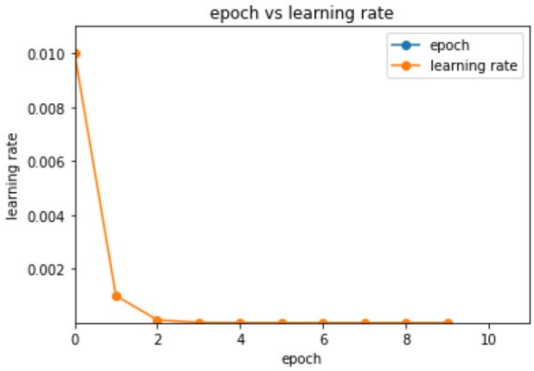


Figure 9: epoch vs learning rate graph

Why momentum works?

With Stochastic Gradient Descent we don't compute the exact derivate of our loss function. Instead, we're estimating it on a small batch. Which means we're not always going in the optimal direction, because our derivatives are 'noisy'.So, exponentially weighed averages can provide us a better estimate which is closer to the actual derivate than our noisy calculations. This is one reason why momentum might work better than classic SGD.
The other reason lies in ravines. Ravine is an area, where the surface curves much more steeply in one dimension than in another. Ravines are common near local minimas in deep learning and SGD has troubles navigating them. SGD will tend to oscillate across the narrow ravine since the negative gradient will point

down one of the steep sides rather than along the ravine towards the optimum. Momentum helps accelerate gradients in the right direction.
Essentially, when using momentum, we push a ball down a hill. The ball accumulates momentum as it rolls downhill, becoming faster and faster on the way (until it reaches its terminal velocity if there is air resistance, i.e. $\beta < 1$). The same thing happens to our parameter updates: The momentum term increases for dimensions whose gradients point in the same directions and reduces updates for dimensions whose gradients change directions. As a result, we gain faster convergence and reduced oscillation.

Pros and Cons of SGD with momentum

Pros: – Faster convergence than traditional SGD

Cons: – As the ball accelerates down the hill, how do we know that we don't miss the local minima? If the momentum is too much, we will most likely miss the local minima, rolling past it, but then rolling backwards, missing it again. If the momentum is too much, we could just swing back and forward between the local minima.

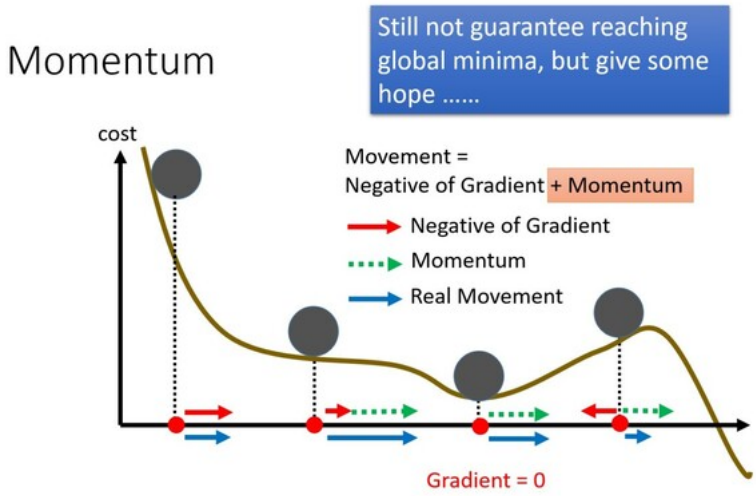


Figure 10: Problem of SGD with momentum

1.6.3 Nesterov accelerated gradient(NAG)

However, a ball that rolls down a hill, blindly following the slope, is highly unsatisfactory. We'd like to have a smarter ball, a ball that has a notion of where it is going so that it knows to slow down before the hill slopes up again.
Nesterov Momentum is a slightly different version of the momentum update that has recently been gaining popularity. In this version we're first looking at a point where current momentum is pointing to and computing gradients from that point. It becomes much clearer when you look at the picture

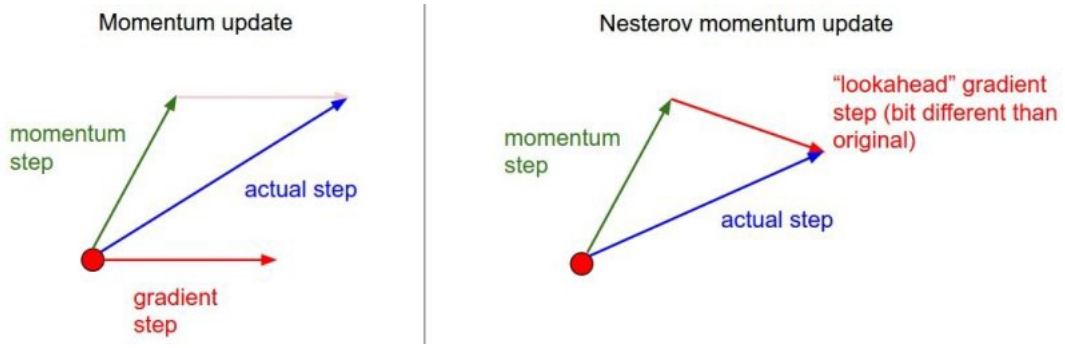


Figure 11: Source:Stanford CS231n class

Mathematically it can be written as follows:

$$V_t = \beta V_{t-1} + (1-\beta)\nabla_w L(W, X, y - \beta V_{t-1}) \tag{12}$$

$$W = W - \eta V_t \tag{13}$$

1.6.4 Adagrad

Adagrad adapts the learning rate to the parameters, performing smaller updates (i.e. low learning rates) for parameters associated with frequently occurring features, and larger updates (i.e. high learning rates) for parameters associated with infrequent features. For this reason, it is well-suited for dealing with sparse data. Dean et al. have found that Adagrad greatly improved the robustness of SGD. Moreover, Pennington et al. used Adagrad to train GloVe word embeddings, as infrequent words require much larger updates than frequent ones.

In its update rule, Adagrad modifies the general learning rate η at each time step t by the formula:

$$\eta' = \frac{\eta}{\sqrt[2]{\alpha_t + \epsilon}} \tag{14}$$

where α_t is the sum of square of previous losses i.e, $\alpha_t = \sum_{i=1}^{t} (\nabla_w L)^2$ and the updated rule become :

$$w_t = w_{t-1} - \frac{\eta}{\sqrt[2]{\alpha_t + \epsilon}} \nabla_w L \tag{15}$$

Here $\nabla_w L = \frac{\delta L}{\delta w_{old}}$ and $(\nabla_w L)^2$ is always non-negetive. So $\alpha_{t+1} \geq \alpha_t$. In code Adagrad looks like below:

```
optimizer = optim.Adagrad(optim_params, args.lr, weight_decay=args.weight_decay)
```

Figure 12: Code of adagrad

Pros and Cons:

Pros :
- No manual tuning of the learning rate required.
- Faster convergence
- More reliable

Cons :
- α_t can become large as the number of iterations will increase and due to this η_t' will decrease at the larger rate. This will make the old weight almost equal to the new weight which may lead to slow convergence.

1.6.5 RMSProp

RMSprop (Root Mean Square Propagation) is a gradient-based optimization technique used in training neural networks. It was proposed by the father of back-propagation, Geoffrey Hinton.It is a derivation of AdaGrad that has different learning rates for each of its parameters (variables). The major difference of RMSProp with AdaGrad is that the gradient is calculated by an exponentially decaying average, instead of the sum of its gradients.

$$\eta' = \frac{\eta}{\sqrt[2]{w_{avg(t)} + \epsilon}} \tag{16}$$

where $w_{avg(t)} = \beta w_{avg(t-1)} + (1-\beta)\sum_{i=1}^{t}(\nabla_w L)^2$ and the updated rule become :

$$w_t = w_{t-1} - \frac{\eta}{\sqrt[2]{w_{avg(t)} + \epsilon}} \nabla_w L \tag{17}$$

```
optimizer = optim.RMSprop(params, opt.learning_rate, opt.optim_alpha,
                          opt.optim_epsilon, weight_decay=opt.weight_decay)
```

Figure 13: Code of RMSProp

Pros of RMSProp:

Pros:
- It is a very robust optimizer which has pseudo-curvature information.
- It can deal with stochastic objectives very nicely, making it applicable to mini batch learning.
- It converges faster than momentum.

1.6.6 Adam

Adaptive Moment Estimation is an algorithm for optimization technique for gradient descent. The method is really efficient when working with large problem involving a lot of data or parameters. It requires less memory and is efficient. Intuitively, it is a combination of the gradient descent with momentum algorithm and the RMSP algorithm. It was first published in 2014, Adam was presented at a very prestigious conference for deep learning practitioners — ICLR 2015. The paper contained some very promising diagrams, showing huge performance gains in terms of speed of training. However, after a while people started noticing, that in some cases Adam actually finds worse solution than stochastic gradient descent. A lot of research has been done to address the problems of Adam.
Mathematically, it follows the rule written bellow:

$$\theta_{t+1} = \theta_t - \frac{\eta.\hat{m}_t}{\sqrt[2]{\hat{v}_t + \epsilon}} \tag{18}$$

where,

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t} \tag{19}$$

$$\hat{v}_t = \frac{v_t}{1-\beta_2^t} \tag{20}$$

$$m_t = (1-\beta_1)\nabla_w L + \beta_1 m_{t-1} \tag{21}$$

$$v_t = (1-\beta_2)(\nabla_w L)^2 + \beta_2 v_{t-1} \tag{22}$$

- ϵ is just a small term preventing division by zero. This term is usually 10^{-8}
- a good default setting of η is 0.001
- Setting first momentum term β_1 as 0.9 and Second momentum term β_2 as 0.999 is a good practise

In code Adam looks like below:

```
optimizer = optim.Adam(params,lr=0.005,betas=(0.9,0.999),eps=1e-08,
                       weight_decay=0,amsgrad=False)
```

Figure 14: Code of Adam

Pros and Cons of Adam

Pros: – The method is too fast and converges rapidly.
– Rectifies vanishing learning rate, high variance.

Cons: – Computationally costly.

1.7 Performance of various optimizers

Building upon the strengths of previous models, Adam optimizer gives much higher performance than the previously used and outperforms them by a big margin into giving an optimized gradient descent. The plot is shown below clearly depicts how Adam Optimizer outperforms the rest of the optimizer by a considerable margin in terms of training cost (low) and performance (high).

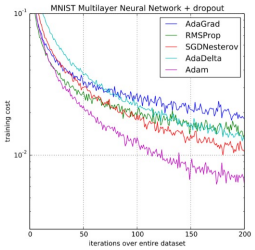


Figure 15: performance of different optimizers(Source:Adam: A Method for Stochastic Optimization)

	Train Accu(SGD)	Val Accurac	Train loss(S	Val loss(SG	Train Acc(S	Val Acc(SG	Train loss(S	Val loss(SG	Train Acc(A	Val Acc(Ad	Train loss(A	Val loss(Ad	Train Acc(I	Val Acc(RM	Train loss(I	Val loss(RM	Train Acc(A	Val Acc(Ad	Train loss(A	Val loss(Adam)
epoch 1	98.226	98.12	0.058374	0.072819	98.054	98.3	0.062634	0.05902	98.872	98.92	0.050782	0.036787	82.496	90.66	2.415281	0.260993	99.244	98.95	0.02526	0.046828
epoch 2	98.252	98.29	0.056298	0.066239	98.424	98.3	0.050695	0.061364	99.352	98.82	0.020423	0.035801	96.032	97.79	0.13352	0.082661	99.386	98.95	0.019694	0.044039
epoch 3	98.29	98.16	0.055473	0.068313	98.598	98.49	0.044283	0.053684	99.432	98.78	0.017447	0.037364	97.288	97.83	0.091748	0.077112	99.456	98.99	0.018251	0.04435
epoch 4	98.394	98.17	0.053264	0.068116	98.768	98.62	0.039527	0.050649	99.47	98.89	0.015871	0.036117	97.874	97.72	0.073728	0.093809	99.482	99.02	0.016265	0.042415
epoch 5	98.306	98.21	0.05248	0.064451	98.804	98.55	0.036508	0.050259	99.506	99	0.014943	0.033331	98.182	98.3	0.062654	0.070632	99.512	98.94	0.016114	0.042205
epoch 6	98.42	98.28	0.050817	0.064675	98.924	98.4	0.034054	0.053404	99.538	99	0.014262	0.032734	98.274	96.62	0.058727	0.100255	99.51	99	0.015682	0.043042
epoch 7	98.522	98.38	0.048641	0.058838	99.012	98.65	0.030314	0.048699	99.578	98.97	0.012887	0.034928	98.48	98.36	0.053123	0.060142	99.582	98.96	0.013385	0.042611
epoch 8	98.514	98.35	0.047479	0.061411	99.052	98.75	0.028586	0.040745	99.598	98.94	0.012628	0.034867	98.478	98.44	0.052545	0.059806	99.554	99.05	0.013436	0.042454
epoch 9	98.466	98.41	0.04819	0.057717	99.146	98.74	0.026376	0.040476	99.606	98.91	0.012119	0.034011	98.62	97.81	0.049553	0.078909	99.55	99.02	0.014268	0.044033
epoch 10	98.538	98.33	0.046269	0.060605	99.184	98.69	0.025392	0.045538	99.634	99.01	0.011529	0.034349	98.74	98.47	0.045622	0.061447	99.612	99.01	0.013009	0.04309

Figure 16: Performance report of various optimizer on MNIST DATASET

Now we show some plot of accuracy and loss for train and validation set applied on MNIST dataset.

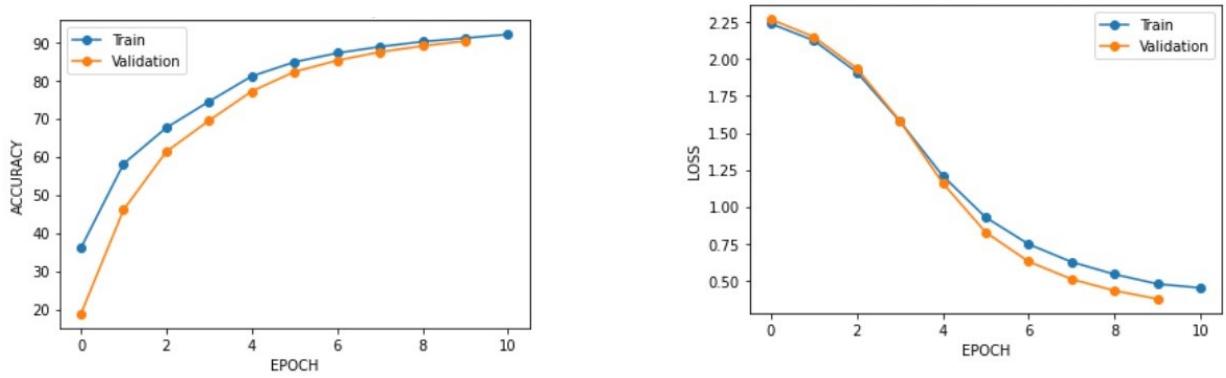


Figure 17: Plot of Accuracy and loss of model trained using SGD optimizer

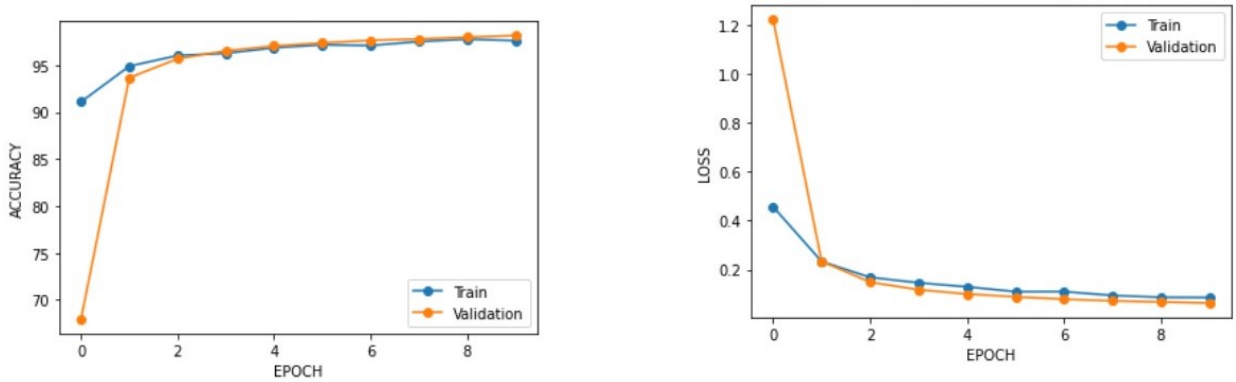


Figure 18: Plot of Accuracy and loss of model trained using SGD with momentum optimizer

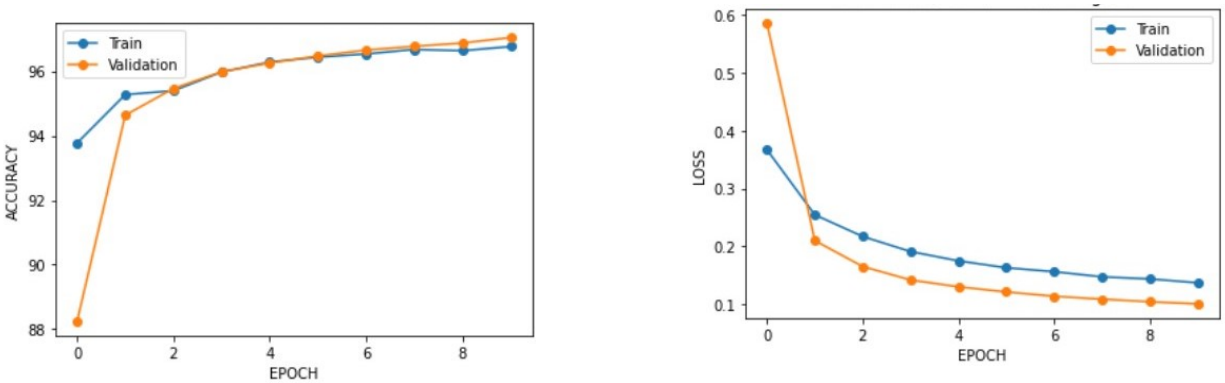


Figure 19: Plot of Accuracy and loss of model trained using Adagrad optimizer

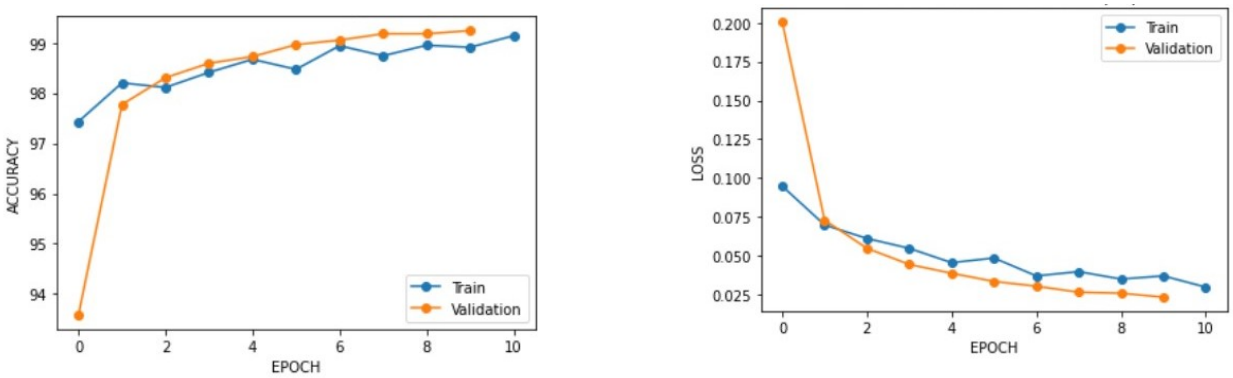


Figure 20: Plot of Accuracy and loss of model trained using RMSProp optimizer

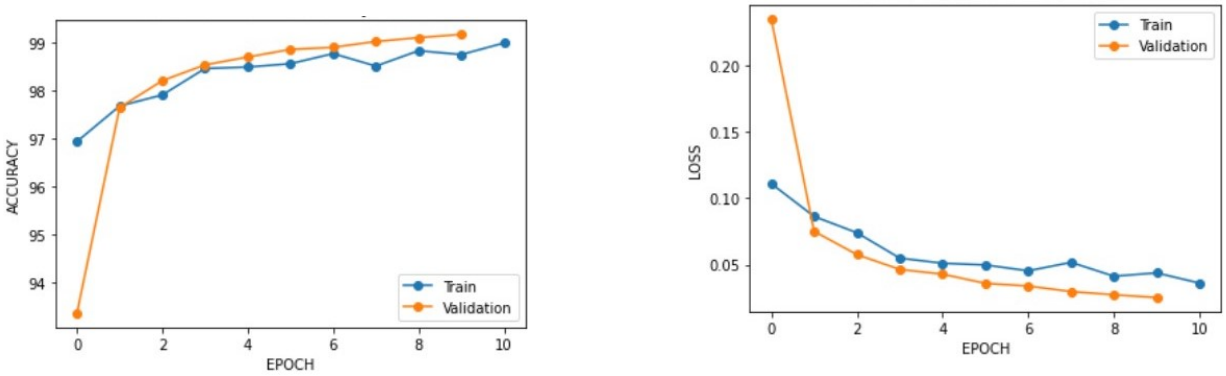


Figure 21: Plot of Accuracy and loss of model trained using Adam optimizer

1.8 Which optimizer should we use?

There is currently no consensus on this point. Schaul et al.(2014)presented a valuable comparison of a large number of optimization algorithms across a wide range of learning tasks. While the results suggest that the family of algorithms with adaptive learning rates like RMSProp performed fairly robustly, no single best algorithm has emerged.
Currently, the most popular optimization algorithms actively in use include SGD, SGD with momentum, RMSProp and Adam. The choice of which algorithm to use, at this point, seems to depend largely on the user's familiarity with the algorithm (for ease of hyperparameter tuning). But we can observe the followings in time of choosing optimizer:

- SGD algorithm is stuck at a saddle point. So SGD algorithm can only be used for shallow networks.
- AdaGrad algorithm can be used for sparse data.
- Momentum and NAG work well for most cases but is slower.
- Adam is the fastest algorithm to converge to minima.

2 Reference

1. Deep Learning by Ian Goodfellow and Yoshua Bengio and Aaron Courville
2. Adam: A Method for Stochastic Optimization by Diederik P. Kingma, Jimmy Ba
3. towardsdatascience.com
4. A Comprehensive Guide on Deep Learning Optimizers - Blog by Ayush Gupta
5. medium.com/mlearning-ai
6. machinelearningmastery.com
7. A Blog on an overview of gradient descent optimization algorithms
8. A Blog on Optimizers
9. geeksforgeeks
10. Pytorch Documentation