

Conceptual Understanding of Loss Functions and Optimization Strategies

Part 2

In this part the focus is on the challenges of training a deep neural network given the characteristics of the error function and the limitations of the gradient descent algorithm. Deep neural network (DNN) though theoretically is a universal function approximator, in practice, training a DNN can become quite a challenge.

Before getting into the discussion on the topic in focus, refer to the figure 22 on page 17/20 in part 1. This figure, has not come out well in terms of clarity when converted into PDF which made the contour lines almost invisible. A similar image with better clarity is presented below.

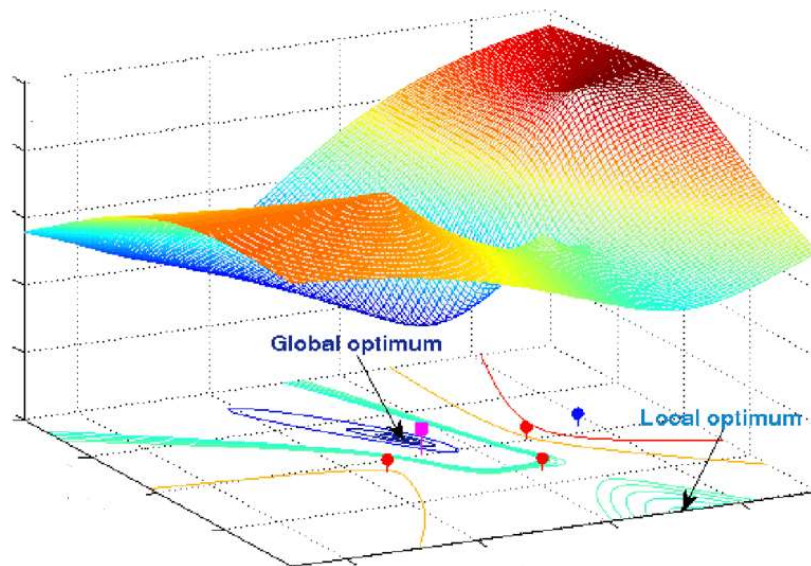


Fig 23

Image Source: https://www.researchgate.net/figure/Non-convex-realization-of-the-localization-objective-function-The-NLS-objective-function_fig2_308307958

In the fig 23 one can see the correspondence between the topology of the error surface in 3d and the contour graph in 2d. The flat regions (upper right red in colour) have the corresponding contour lines separated far from one another. In regions of sharp gradients, the contour lines are tightly packed. One can study the impact of such deviation on the training both from the contour graph point of view or the 3d point of view. However, explaining some behaviour of the algorithm may not be intuitive when visualised in 3d, for e.g., error can increase in subsequent iterations if the learning rate is not set right.

Presence of such deviations from a strictly convex geometry leads to multiple challenges -

Type of deviation	Characteristics	Impact on Gradient Descent	Impact on training
Flat regions	Almost 0 gradient/slope	Change in coefficients at the end of a backprop is very miniscule	Training becomes very slow/ sluggish and may run out of iterations resulting in a sub optimal set of coefficients
Local minimum	Gradient =0 and surrounded by gradients >0	Gradient Descent stops improvising the coefficients	Training stops at a sub optimal level.
Saddle point	Error goes up in one direction and goes down in other	Gradient Descent algorithm oscillates in one direction (direction of increase in error) instead of moving down efficiently	Training takes a greater number of iterations that needed if the Gradient descent oscillated less and moved down

Table 2

Refer to <http://www.denizyuret.com/2015/03/alec-radfords-animations-for.html> for a visual on the way gradient descent and its variants perform under different conditions. The dots represent variants of the gradient descent optimizer. Red one being the vanilla gradient descent. The movement of these dots over time represent the learning process of modifying the coefficients in every iteration. The vertical axis being the error while the two horizontal axes represent the coefficients. The animation shows all the three types of impacts listed in the table above.

Before the year 2006, training a DNN was a difficult task that motivated Yoshua Bengio and Xavier Glorot, two prominent names in the world of deep neural networks, to research the reasons behind DNN failing to learn. They presented their findings in their famous paper <https://scholar.google.com/citations?user=WnkXlkAAAAJ&hl=en>. They discovered multiple reasons that prevented a DNN from learning. The focus of this article is on only the error function related challenges. We will also discuss how the optimizer algorithms are designed to mitigate the risks associated with these challenges emanating from the error functions. To understand this, we have to look at the training step, especially the one in which each neuron gets its share of overall error and the subsequent adjustments to the coefficients.

Training Cycle

At a very broad level, the training of a DNN can be represented by the following pseudo code –

1. Epoch = N (Number of times entire data will be used)
2. Batches = M (number of batches the file we be read in terms of)
3. For Epoch 1 to N
 - a. For Batch = 1 to M
 - i. Forward propagate Batch thru the network
 - ii. Calculate the Error for the batch
 - iii. Backward propagate error to all neurons in all the layers
 - iv. Adjust the coefficients associated every neuron to reduce the error at that neuron
 - b. Next Batch
4. Next Epoch

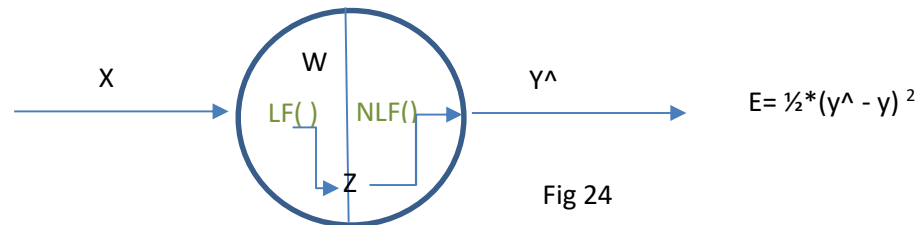
Some points to be noted in this training cycle are –

1. At the beginning of every epoch, the batches will be created randomly. The batches are not some kind of a sequential splitting of data based on the batch size.
2. A batch of data forward propagates thru the DNN resulting in error E.
3. The error is back propagated to assign share of error to each neuron in each layer. This is done using chain equation.
4. The coefficients / weights associated with each neuron are adjusted independently (in silos) such that its share of error is reduced to zero. This results in a new set of weights across the DNN
5. The next batch data forward propagates against the new set of weights left behind by the previous batch.
6. The loop continues until all the epochs are consumed or the optimizer has reached a minimum or early stopping intervention as broken the outer loop.

Bullet point 4 needs special attention.... All the neurons adjust their weights in silo. There is no mechanism built into this logic to assess what impact change in coefficients of one neuron have on the subsequent neuron's performance in the next forward propagation. This has an impact on the training... What is the impact? I leave point for the reader to discover. To discuss impact of error function shapes on training, one needs to understand the concept of the chain equation mentioned in bullet point 3 where lie the reasons for the difficulty in training DNN.

Chain Equation

Let us look at what would happen if we had only one layer and one neuron in the layer.



1. The neuron is initialized with weight W
2. The neuron takes only one input X
3. Linear function (LF) calculates $Z = WX$ (no bias shown)
4. Nonlinear function (NLF) acts on Z to give \hat{y}
5. Error E is difference in prediction (\hat{y}) and ground truth (y) expressed as Squared Error Function
6. Transformation of X to \hat{y} can be shown as $\rightarrow \hat{y} = \text{NLF}(\text{LF}(WX))$

Error E can be expressed as function of W (refer to section 1 and 2 of part 1). Which means, to minimize overall E , W needs to be manipulated and hence we have to explore how error changes (dE) for a unit change in W (dW), i.e., we have to find dE/dW . Since E and W are not directly related, i.e., there are multiple transformations between use of W and generation of E , we have to use chain equation to find $d(E)/d(W)$ as shown below.

$$d(E)/d(W) = d(E)/d(\text{NLF}) * d(\text{NLF}) / d(\text{LF}) * d(\text{LF}) / d(W) \rightarrow \text{Eq. 1 Chain Equation}$$

Since the chain equation is a multiplication of multiple terms, any term going to zero will result in the entire equation collapsing to zero. Which means $d(E)/d(W) \rightarrow 0$. There are two opportunities for the equation to collapse which are $d(\text{NLF})/d(\text{LF})$ and $d(E)/d(\text{NLF})$. This possibility of the chain equation collapsing to zero is technically called the “Vanishing Gradient” problem. The reverse of this too is possible and it is called “Exploding Gradient”. In this article, the focus is on vanishing gradient and that too the one resulting from the error function (shown in blue). Error gradients can also vanish due to the non-linear activation function (shown in red) but is out of scope for this article.

Collapse of the equation / vanishing gradient does not necessarily mean the calculation should go to absolute zero. It also refers to the situation where the $d(E)/d(W)$ becomes so miniscule in magnitude that it is as good as zero. For example, 0.00000034, is as good as 0.

In the discussion of the pseudo code, bullet 3.4, where the coefficients are adjusted, the coefficients are modified using the following formula

1. $W_{\text{new}} = W_{\text{old}} - \eta * d(E)/d(W)$ which is technically shown as ($W_{\text{new}} = W_{\text{old}} - \eta * \nabla J(w)$)
2. If $d(E)/d(W) = 0$ then the expression $\eta * d(E)/d(W)$ will become 0. Hence $W_{\text{new}} = W_{\text{old}} - 0$
3. That means $W_{\text{new}} = W_{\text{old}}$ (the coefficients do not change)

The expression $d(E)/d(W)$ will approach zero when the optimizer is in the flat regions of the error surface such as point A in the figure below.

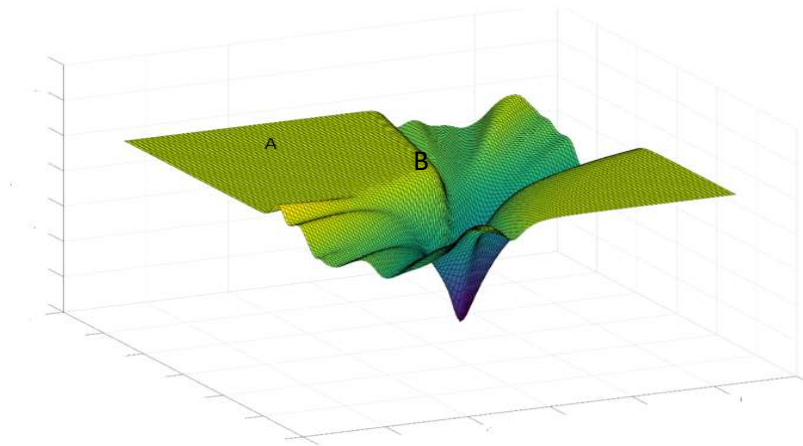


Fig 25

Image source : <https://stats.stackexchange.com/questions/267400/logistic-regression-cost-surface-not-convex>

Modification to the coefficients in any way will result in $d(E)/d(W) \approx 0$ (almost zero assuming there is a small gradient towards the global minimum). As a result, the optimizer algorithm will need a lot of iterations to reach point B on the cusp where there is a fall towards minima. If the $d(E)/d(W)$ becomes exactly zero, the optimizer will stop learning.

Many practitioners developed variants of the gradient descent algorithm to overcome these challenges in training. Other than using those variants of the optimizer, there is a good practice (mini batch) that is recommended by practitioners. This is independent of the optimizer algorithm that will be discussed later.

Mini Batch Gradient Descent

Practitioners in the field of DNN advice that we use small batch size (32, 64, 128) etc. This goes against the grain! All the while we have been of the view that larger the training dataset more generalizable a model is going to be! This is true when working with conventional algorithms such as Logistic Regression, Linear Regression etc. However, in the world of DNN, taking full batch data in one go i.e., doing the epochs with only one full batch often leads to sub-optimal results. Table 3.0 compares the results of a simple shallow neural network trained on Fashion MNIST across different batch size.

Architecture

Layer (type)	Output Shape	Param #
flatten_4 (Flatten)	(None, 784)	0
dense_8 (Dense)	(None, 256)	200960
dense_9 (Dense)	(None, 10)	2570
Total params: 203,530		
Trainable params: 203,530		
Non-trainable params: 0		

Optimizer – RMSProp

Loss function – Categorical Cross Entropy

Learning Rate - 0.001

Batch_Size	Fashion MNIST		
	Epochs		
	5	10	15
32	87.7	87.4	87.9
64	87.7	86.4	87.7
128	86.4	88.5	88.2
60000	59.6	68.75	69.85

Table 3

To understand why this happens, recall that the shape of error functions is influenced by the underlying data distribution too. When we take large batch size such as complete data set, if optimizer gets stuck in a flat region of the corresponding error surface, it has relatively limited options to come out of the situation. When we adopt the mini-batch approach, each minibatch can be imagined to be a stand-alone training file. That means each mini batch has its own corresponding error function. These error functions will not look the same. Which means a point (combination of coefficients) may be in the flat regions of error surface of batch b1 but the same point in the error surface of batch b2 may not be a flat region! The optimizer may find a gradient at that point and use that to find the next best combination of coefficient. This also means that if optimizer gets stuck in a local minimum of error surface of a batch, it may be able to roll down from there in the error surface of next batch! And this continues across epochs where in epoch the batches are created at random.

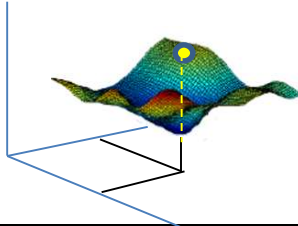
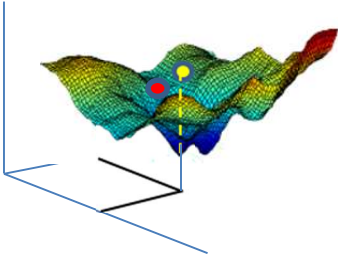
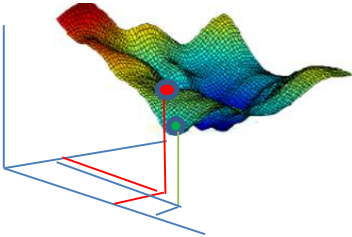
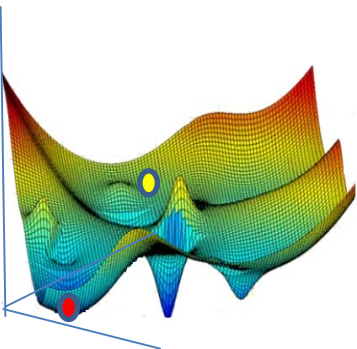
S.No.	Error Surface of Batch no.	Optimizer Situation	Description
1		In the first batch error function, the combination of coefficients indicates the current position that optimizer is working on. The local region around is flat.	If the error surface continues to be this, the optimizer will need many iterations to move towards the cusp with strong gradient
2		The same combination of coefficients in the second batch position the optimizer on error surface with relatively more gradient than batch b1. The yellow point rolls down to red!	The presence of gradient helps optimizer find better combination of coefficients (lower error) shown in red dot
3		The error surface for the third batch has a strong gradient at the combination of coefficients (red dot) found from previous batch. It quickly descends to the local minimum (green dot)	Even though the optimizer is not in local minimum, the next batch error function may not have local minimum at this point. The optimizer will continue to move! Even if given epoch is over, the first batch of next epoch may help it move out of the local minimum
4	 Image source: firsttimeprogrammer.blogspot.co.uk	If the error function represents the entire dataset put together and the optimizer gets stuck in a local minimum (red point), there is no way it can come out of it. If the optimizer gets into the flat region (yellow dot), it may need many epochs to move out the region into a minimum	Practitioners have empirically found that the error function in DNN are plagued by flat regions more than by local minima. The local minima are in close proximity of the global and hence it does not matter too much even if the optimizer is stuck in a local minimum.

Table 4

Note: It is possible that a combination of coefficients in error surface of batch b1 resulting in error E1 may give error E2 in the batch b2 error function where $E1 \neq E2$. E2 may even be greater than E1. But given the mathematical characteristics of the error function (convexity), there will be an overall trend towards lower minima even if the error functions deviate from convexity. Hence, across iterations, the optimizer will move towards coefficients that give lower and lower error.

One can imagine the error functions across all batches and epochs as individual frames in a video and when you run the video, the error function flags which helps the optimizer keep moving in the coefficient space towards the minima.

To get a visual idea of the flagging error surface, pl. refer to <https://www.youtube.com/watch?v=uje3oju3McQ> (this video is made in a different context but serves our purpose too). Watch the portion from 7:41 onwards where it shows a red cone at a particular coordinate. The coordinates are the combination of the coefficients and the cone (assume it to be a point) is amount of error. Each frame of this video can be considered as the error surface of corresponding mini batch. The coordinate remains the same across the batches.

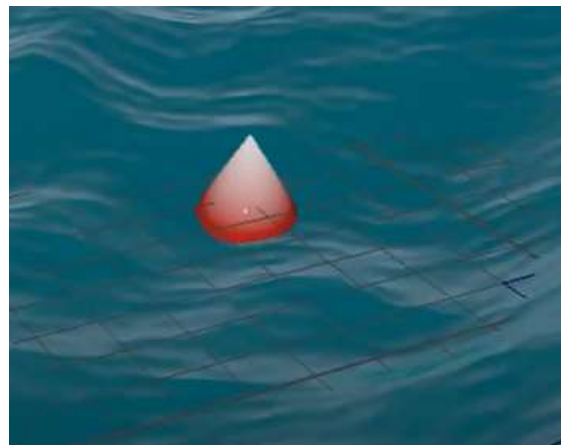


Fig 26

Image Source: <https://www.youtube.com/watch?v=uje3oju3McQ>

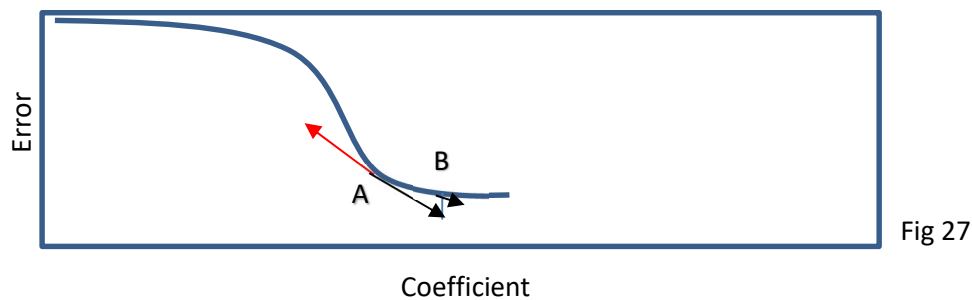
With all the benefits of mini-batch, however, there is still no guarantee that the optimizer will not eventually get stuck in a local minimum and the optimizer runs out of iterations. Usually, it will end up relatively in a better situation than it would have if all data was treated as one single batch.

For information on other options for gradient descent (Batch, Minibatch and Stochastic), pl. refer to a beautiful introduction at <https://machinelearningmastery.com/gentle-introduction-mini-batch-gradient-descent-configure-batch-size/>

Momentum to Push Optimizer Out of Flat Regions

Along with the “Learning Rate” (LR) that we dealt with in part 1, “Momentum” plays a critical role in mitigating the risk of optimizer getting stuck in the flat region and the local minimum of the error function. In simple gradient descent approach, the optimizer calculates gradient at each step. It forgets what the gradient was in the previous step. Suppose a part of the error function looks as shown in fig 27 below the optimizer is at point A from where it moves to point B based on the gradient.

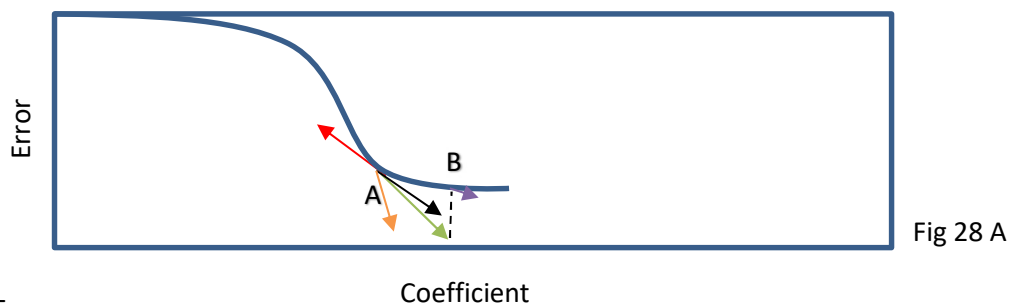
At point B, the gradient is almost zero! (shown as a very small black arrow). As a result, the optimizer gets sluggish. The fact that there was a strong gradient in the previous step is forgotten!



To add to the problem, the learning rate (unless it is set to 1 which means it has no effect) when multiplied with the very tiny gradient at B will make the problem even more severe for the optimizer. It is like being not only on a flat (almost zero gradient) ground but also in knee deep snow (that makes the progress slow).

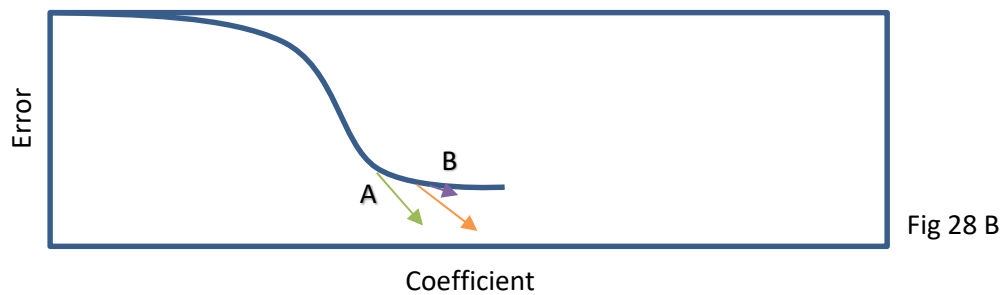
What we experience when we go down a slope is not this. Even on the flat ground the effect of the slope we left behind, takes us forward. In language of Physics this is what is called inertia. Technically the accumulated momentum from the past positions, should also influence the rate of forward movement, it is not just the slope at a given point. Which means, even if slope at a point is zero, if the optimizer had relatively large gradients before reaching the point, it could continue to move even in flat zero gradient region

In SGD, moving average of the past gradients for each coefficient (only one coefficient is shown in figure 27) is used to update the coefficients at current step. But how much of the past gradients should be accumulated to update the current coefficients? For this we use another hyperparameter (β), which ranges from 0 to 1. Small β results in small contribution from past while a large one means the opposite.



At point A –

1. Orange arrow is the result of past accumulated gradient
2. black arrow is the result of current gradient at A
3. Green arrow is sum of the orange and black vector. This represents momentum.
4. Following the green arrow brings the optimizer to point B (dotted vertical line)
5. Hence, the new coefficient value is the value at B



At point B (current position)–

1. the small purple arrow is the current gradient at B. This does not help much in moving on the flat surface fast
2. To the purple arrow, add momentum from the past. Let the green arrow represent the past accumulated momentum
3. The result is the orange arrow which takes the optimizer much further than the current gradient.

Introducing the concept of momentum into the optimizer functioning helps move out of the flat regions and saddle points relatively faster than without it. It also helps converge faster on steep slopes even with small learning rates. However, it can become a nuisance near the minima when the error function has sharp curvatures. Momentum, in such a case, can cause oscillations around the minimum (beat the purpose of learning rate!). This led to creation of another approach called the Nesterov Momentum.

To implement the momentum along with learning rate in vanilla gradient descent –

```
tf.keras.optimizers.SGD(learning_rate=0.1, momentum=0.9)
```

Nesterov Momentum

Also known as Nestorov Accelerated Gradient (NAG) is a variant of gradient descent with momentum. Introduced by Y.Nesterov, in this approach, the optimizer looks ahead and takes into account the gradient in future to decide the direction of change. This helps the optimizer converge relatively faster than when using only momentum. The key idea is that instead of using gradient at the current location, use the gradient at the location the optimizer will be and use a part of it to fix the direction. Note: if the gradient in future position (where current momentum will take) is zero, that could be a flat region or a local minimum, the momentum may help the optimizer move on.

Using gradient from future position helps minimize the overshooting problem. As discussed earlier, if the error function is sharp (strong gradients near the minimum), optimizer with momentum can find itself on the other side of the minimum, may be ever at a higher error level than at the current position.

The Nesterov algorithm can be understood in four steps -

1. At the current point A, calculate the momentum from the past
2. As per the momentum (blue arrow) the optimizer should be at B
3. Find the result of gradient at B (red arrow)
4. Adjust the direction obtained in step 2 with the future gradient of step 3 (vector addition of the blue and red arrow)
5. Move in the resultant direction (shown in green) to reach at C

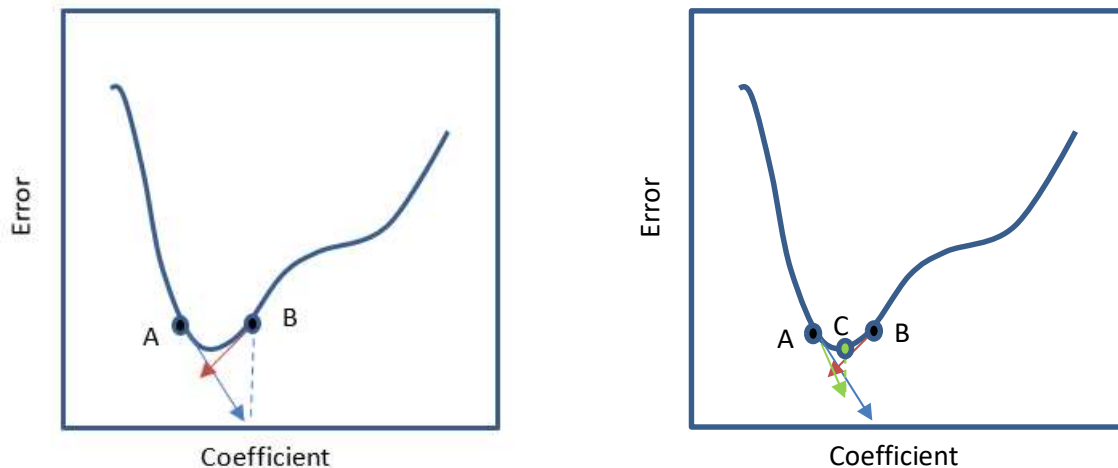


Fig 29

Nesterov, taking into account future gradient reduces the chances of oscillation around the minimum. The calculations to find the direction to go into will be done on scaled values of the momentum and gradients (not shown in the figure). The point to be noted is that the optimizer does not use the gradient at a point. For e.g., at point A, it takes scaled momentum (from previous steps) to estimate where it will reach (point B). From B take the scaled gradient. Gradient at current location A is not used!

```
tf.keras.optimizers.SGD( learning_rate=0.01, momentum=0.0, nesterov=True)
```

Both Momentum and Nesterov Momentum approach make use of the past and future gradients respectively for estimating the next set of coefficients and both are designed to help optimizer come out of flat region and reduce overshooting respectively. But all the while the optimizer is using the same learning rate across all coefficients. Why should that be? Why can't the learning rate be different for different coefficients?

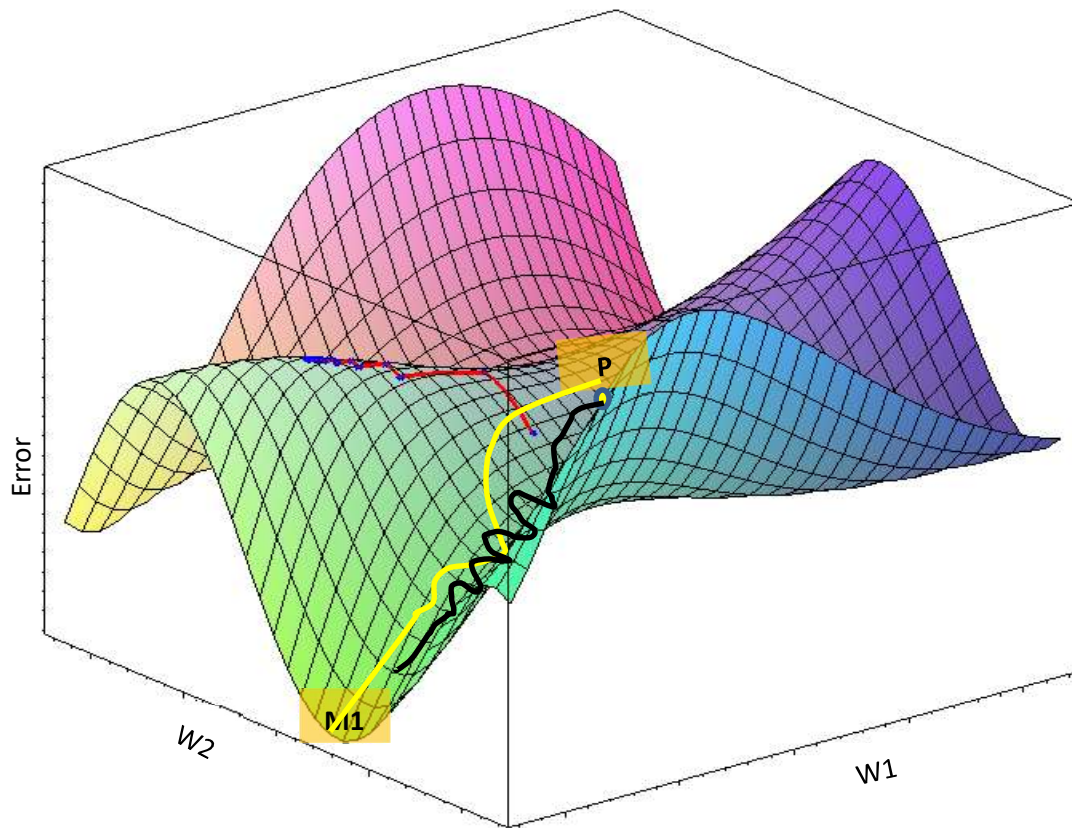


Fig 30

Image Source : <https://commons.wikimedia.org/w/index.php?curid=521422>

At point P, the gradient of the function w.r.t. coefficient w_1 is very flat relative to the gradient of the function at the same point w.r.t. coefficient w_2 . Given the shape, the optimizer will move from P towards minimum M1. If it uses the same learning rate in both directions, it will take much long compared to if it takes large steps in w_1 direction and small one in w_2 . This is likely to result in a path as shown in bright yellow instead of the black one where learning rate is same in both directions. The bright yellow path will reach minima much faster than the black one due to initial oscillations (if the picture above represents the motion study for time t).

There are several algorithms that adopt the idea of using learning rates tailored w.r.t. each coefficient based on the gradients in the direction of the coefficients. The names of these algorithms begin with the word “Ada”, standing for “adaptive”.

Adaptive Learning Rate

Adagrad (Adaptive Gradient Learning)

It helps perform learning rate decay on a per coefficient basis. It does this in the following steps –

1. Square the gradient at the current step
2. Add the squared gradient to a running sum
3. Divide the gradient by the running sum

$S \leftarrow S + (d(E)/d(W))^2$	# Step 1 and 2
$W = W - (\eta * d(E)/d(W)) / \sqrt{S}$	# Step 3

Since squared gradient is added to the running sum in all the steps, the running sum gets larger and larger as squared term can only be positive. As a result, because the running sum is used to divide a gradient at each step, the resulting learning rate gets smaller and smaller in every step.

This is similar to learning rate decay with the only difference that here the learning rate decay is done based on the error surface topology experienced by the optimizer. Not like the learning rate decay that is fixed somewhat arbitrarily.

```
tf.keras.optimizers.Adagrad(
    learning_rate=0.001,
    initial_accumulator_value=0.1,
    epsilon=1e-07,
    name="Adagrad",
```

Because Adagrad automatically computes learning rate for every coefficient on the fly, the learning rate used to initialize the training is not critical. Usually, it is set to a small number such as 0.01 and let the optimizer to estimate it at every step in the gradient descent.

The problem with this approach is that the denominator becomes larger and larger with no limits and as a result the learning step approaches zero! Recall the chain equation of which learning rate is a part. It may so happen that the entire chain equation collapses close to zero and that makes the training very sluggish.

Adadelta

Instead of summing up all the squared gradients since the beginning of the training, this method keeps a decaying sum of the gradients. The old gradients (found in the beginning of the training) decay.

This is like maintaining a stack of limited size holding all previous gradients in the chronological order. New gradients push the oldest gradient out of the stack. Associated with each position / index in the stack is weights, the earliest entry with lowest weight. Each gradient is multiplied with respective weight and summed up. This way, the running sum is most influenced by recent gradient though the past gradients also have some influence.

This algorithm requires a hyper parameter γ (gamma). A large value of gamma will allow gradients from long back to have influence on the learning rate. A small value means less influence of the old gradients. This parameter is often set to .9.

Another parameter represented by ϵ (Epsilon) which is used to prevent divide by zero error.

```
tf.keras.optimizers.Adadelta(
    learning_rate=0.001, rho=0.95, epsilon=1e-07, name="Adadelta", **kwargs)
```

RMSProp

Very similar to Adadelta, it uses “root-mean-squared” (RMS) operation, to determine the adjustment that is propagated. Following steps outline the RMSProp logic

1. $S \leftarrow \beta S + (1 - \beta) \left(\frac{d(E)}{d(W)} * \frac{d(E)}{d(W)} \right)$ # Assign weights β and $(1 - \beta)$ to previous S and current gradient respectively
2. $W \leftarrow W - \eta * \frac{d(E)}{d(W)} / \sqrt{S + \epsilon}$
3. The decay rate β is usually set to .9.

Unlike AdaGrad which slows down when it comes across continuous steep gradients (never converging to the minimum), the RMSProp fixes this by accumulating only the gradients from recent past (as opposed to AdaGrad which accumulates all the gradients from history).

```
opt = tf.keras.optimizers.RMSprop()
m = tf.keras.models.Sequential([tf.keras.layers.Dense(10)])
m.compile(opt, loss='mse')
```

Adam (Adaptive Moment Estimation)

This algorithm combines the ideas of Momentum optimization and RMSProp. Squaring the gradients before summing as is done by the algorithms discussed earlier, means we lose track of negative gradients. That’s useful information to have as we noticed in the Nesterov algorithm. To prevent this loss of information, this algorithm maintains two lists of the gradients, one without squaring them and the other which is a squared version (as seen in other algorithms above).

1. $M \leftarrow \beta_1 M + (1 - \beta_1) * \frac{d(E)}{d(W)}$ # Momentum
2. $S \leftarrow \beta_2 S + (1 - \beta_2) * \frac{d(E)}{d(W)} * \frac{d(E)}{d(W)}$ # Sum of Square of Gradients
3. $M \leftarrow M / (1 - \beta_1^T)$ # Decay of momentum across iterations T
4. $S \leftarrow S / (1 - \beta_2^T)$ # Decay of Sum of Squared Gradients across iterations T
5. $W \leftarrow W - (\eta * M / \sqrt{S + \epsilon})$ # Update the coefficients

There are many other variants of gradient descent algorithm (ref :<https://keras.io/api/optimizers/>) all of which are designed to address the different challenges posed by the deviations in the error function from convexity in training a deep neural network. Which one of these is the best? There is no one single variant that gives the best performance in all situations. We have to iterate through the different methods to find which suits our needs. However, the vanilla gradient descent with a learning rate and momentum is generally not considered suitable. Keras provides RMSProp as the default optimizer. Still, one needs to evaluate the other modern options too.

Summary

1. The objective of this article (part 1 and part 2) was
 2. to introduce the very important concept of error functions deviating from the convex shape that mathematics guarantees them to be.
 3. The adverse impact of the deviations from convexity on the optimizer algorithm's ability to learn the coefficients that represent the patterns of relation between target / predicted variable and independent/ predictor variables.
 4. The concept was discussed in the context of DNN where the chain equation during back propagation could collapse to zero or be almost zero leading to vanishing gradient problem.
 5. Importance of mini-batch Vs full batch in addressing the challenges
 6. Various scientists working in DNN developed alternatives to the batch gradient descent to enable the optimizer handle the challenges in the error function.
- All the variants of the optimizer help mitigate the challenges during training though none of them guarantee that the optimizer will find global minimum.
-

References –

1. https://www.researchgate.net/figure/Non-convex-realization-of-the-localization-objective-function-The-NLS-objective-function_fig2_308307958
2. <https://scholar.google.com/citations?user=WnkXlkAAAAJ&hl=en>.
3. <https://stats.stackexchange.com/questions/267400/logistic-regression-cost-surface-not-convex>
4. <https://www.youtube.com/watch?v=uje3oju3McQ>
5. <https://machinelearningmastery.com/gentle-introduction-mini-batch-gradient-descent-configure-batch-size/>
6. <https://keras.io/api/optimizers/>

Books Referred:

1. Hands-On Machine Learning with Scikit-Learn & Tensorflow (Aurelien Geron)
2. Deep Learning From Basics to Practice volume 1 & 2 (Andrew Glassner)
3. Deep Learning Illustrated (Jon Krohn)

Other blogs – (All on LinkedIn):

1. Long Short Term Memory (LSTM) - The Way I Understand It
2. Recurrent Neural Networks - The Way I Understand It
3. Convolution & Pooling - The Way I Understand it
4. Principal Component Analysis for All
5. PCA Code with Pitfalls
6. What I Do With PairPlot Analysis
7. Null Hypothesis and P Values... Introduction for Budding Data Scientists
8. Layman Introduction to Bias Variance Errors
9. Conceptual Understanding of the Loss Functions part 1 and part 2