

[Subscribe to KDnuggets](#)



[Submit a blog](#)  
[Win a Reward!](#)




- [Blog](#)
- [Opinions](#)
- [Tutorials](#)
- [Top stories](#)
- [Courses](#)
- [Datasets](#)
- [Education: Online](#)
- [Certificates](#)
- [Events / Meetings](#)
- [Jobs](#)
- [Software](#)
- [Webinars](#)



Everything you need  
to know about data prep  
Data Preparation **for Dummies**



GET EBOOK »

[Everything you need to know about data prep. Get \*Data Preparation for Dummies\* ebook](#)

[Submit a blog](#) to KDnuggets -- [Top Blogs Win A Reward](#)

Topics: [AI](#) | [Data Science](#) | [Data Visualization](#) | [Deep Learning](#) | [Machine Learning](#) | [NLP](#) | [Python](#) | [R](#) | [Statistics](#)

[KDnuggets Home](#) » [News](#) » [2020](#) » [Dec](#) » [Tutorials, Overviews](#) » Optimization Algorithms in Neural Networks ( [20:n48](#) )

## Optimization Algorithms in Neural Networks

[<= Previous post](#)

[Next post =>](#)



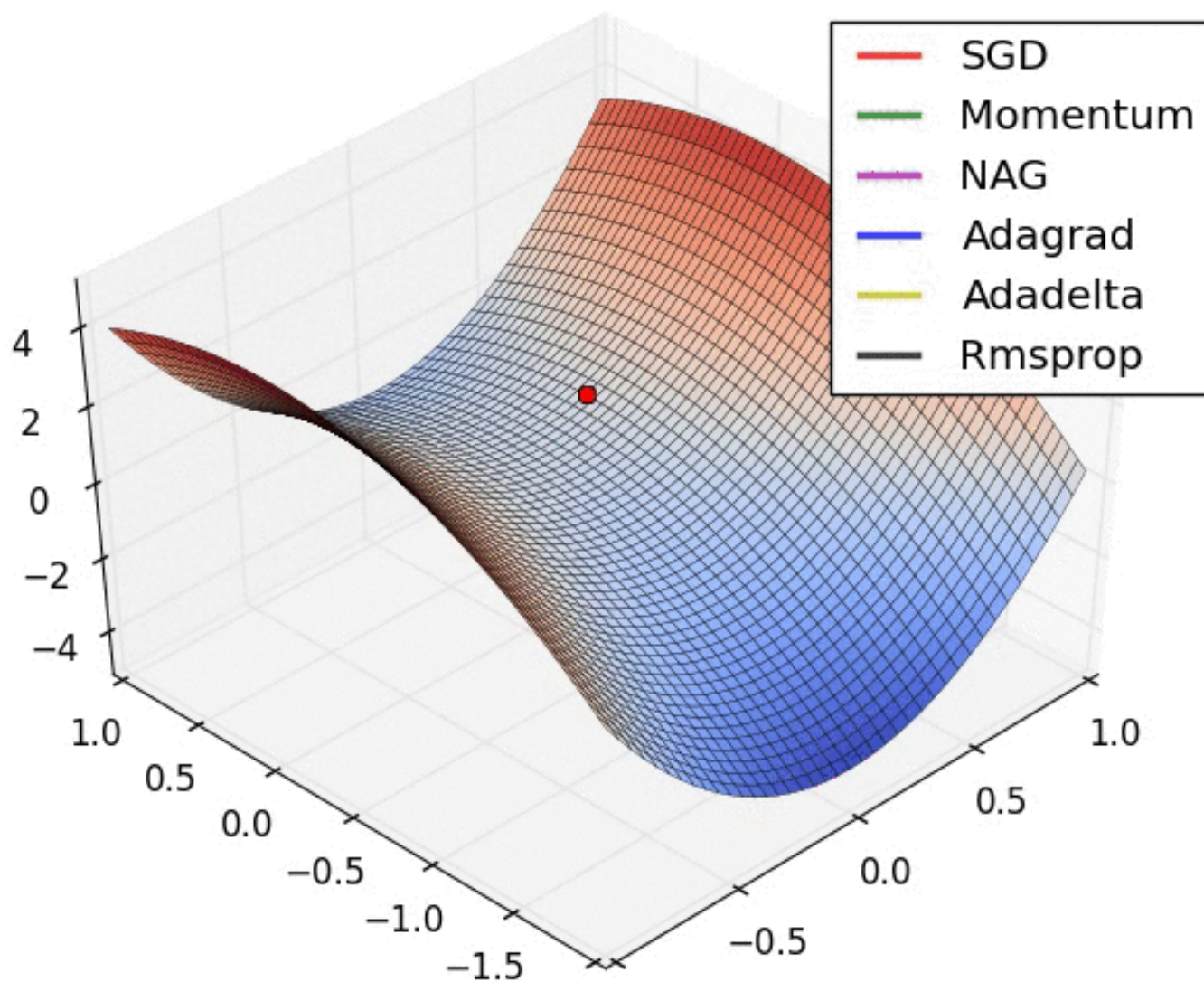
Tags: [Gradient Descent](#), [Neural Networks](#), [Optimization](#)

This article presents an overview of some of the most used optimizers while training a neural network.

SAS - the only Leader  
8 years running  
for DS and ML  
Read Gartner Report

By [Nagesh Singh Chauhan](#), Data Science Enthusiast.

[comments](#)



[Credits](#)

## Introduction

In deep learning, we have the concept of loss, which tells us how poorly the model is performing at that current instant. Now we need to use this loss to **train** our network such that it performs better. Essentially what we need to do is to take the loss and try to **minimize** it, because a lower loss means our model is going to perform better. The process of minimizing (or maximizing) any mathematical expression is called **optimization**.

Optimizers are algorithms or methods used to change the attributes of the neural network such as **weights** and **learning rate** to reduce the losses. Optimizers are used to solve optimization problems by minimizing the function.

## How do Optimizers work?

For a useful mental model, you can think of a hiker trying to get down a mountain with a blindfold on. It's impossible to know which direction to go in, but there's one thing she can know: if she's going down (making progress) or going up (losing progress). Eventually, if she keeps taking steps that lead her downwards, she'll reach the base.

Similarly, it's impossible to know what your model's weights should be right from the start. But with some trial and error based on the loss function

(whether the hiker is descending), you can end up getting there eventually.

How you should change your weights or learning rates of your neural network to reduce the losses is defined by the optimizers you use. Optimization algorithms are responsible for reducing the losses and to provide the most accurate results possible.

Various optimizers are researched within the last few couples of years each having its advantages and disadvantages. Read the entire article to understand the working, advantages, and disadvantages of the algorithms.

We'll learn about different types of optimizers and how they exactly work to minimize the loss function.

1. Gradient Descent
2. Stochastic Gradient Descent (SGD)
3. Mini Batch Stochastic Gradient Descent (MB-SGD)
4. SGD with momentum
5. Nesterov Accelerated Gradient (NAG)
6. Adaptive Gradient (AdaGrad)
7. AdaDelta
8. RMSprop
9. Adam

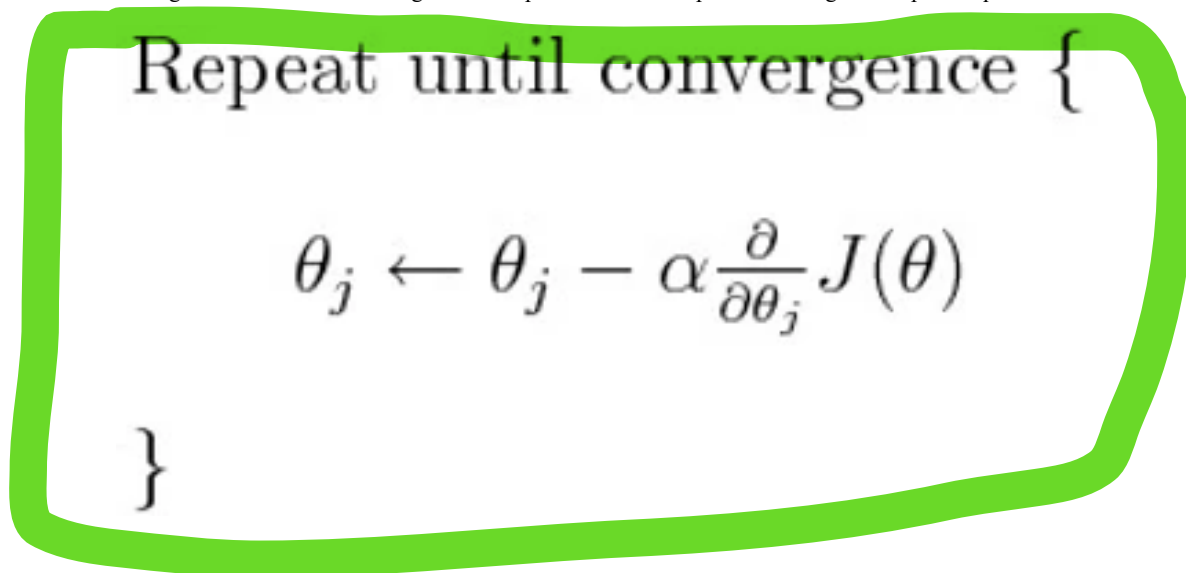
## Gradient Descent

Gradient descent is an optimization algorithm that's used when training a machine learning model. It's based on a convex function and tweaks its parameters iteratively to minimize a given function to its local minimum.

**WHAT IS GRADIENT DESCENT? Gradient Descent is an optimization algorithm for finding a local minimum of a differentiable function. Gradient descent is simply used to find the values of a function's parameters (coefficients) that minimize a cost function as far as possible.**

You start by defining the initial parameter's values and from there gradient descent uses calculus to iteratively adjust the values so they minimize the given cost-function.

The weight is initialized using some initialization strategies and is updated with each epoch according to the update equation.

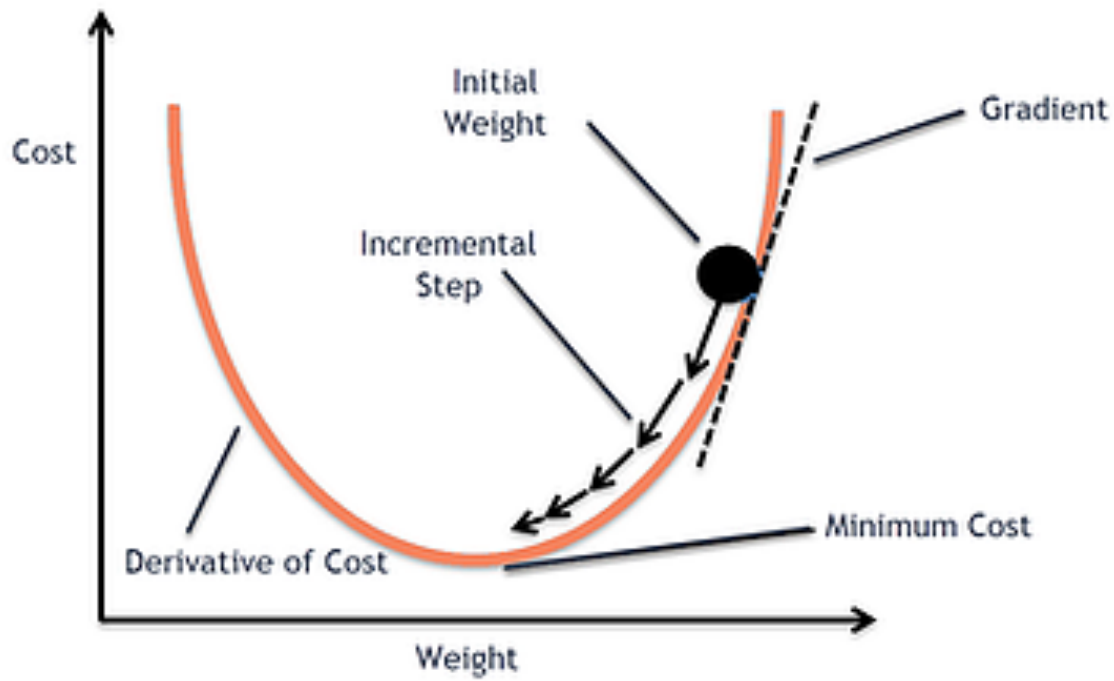


Repeat until convergence {

$$\theta_j \leftarrow \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

}

The above equation computes the gradient of the cost function  $J(\theta)$  w.r.t. to the parameters/weights  $\theta$  for the entire training dataset:

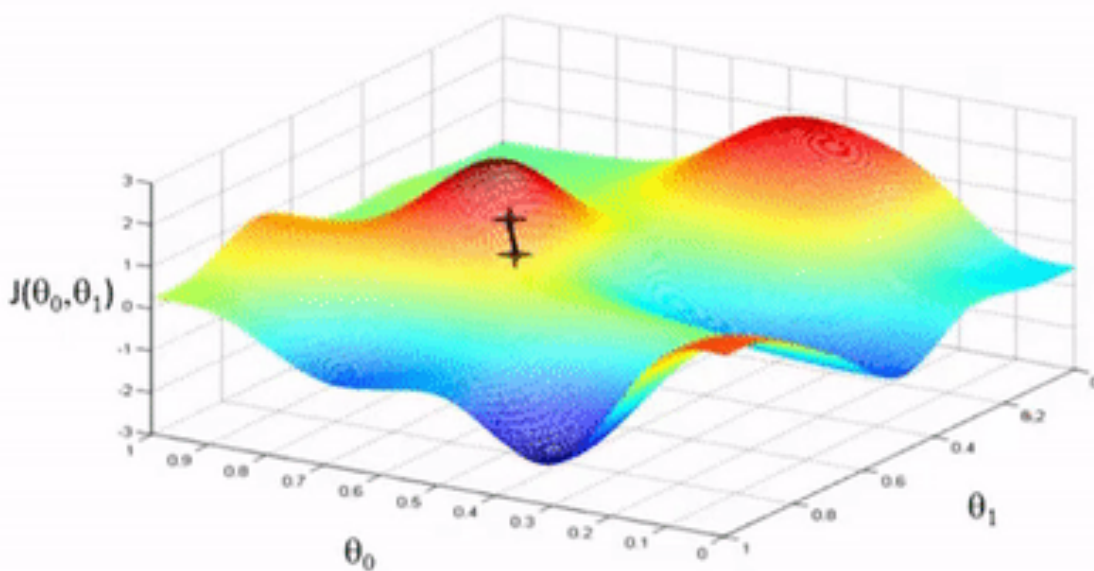


[Image source](#)

Our aim is to get to the bottom of our graph(Cost vs weights), or to a point where we can no longer move downhill—a local minimum.

Okay now, what is Gradient?

**"A gradient measures how much the output of a function changes if you change the inputs a little bit."**—Lex Fridman (MIT)



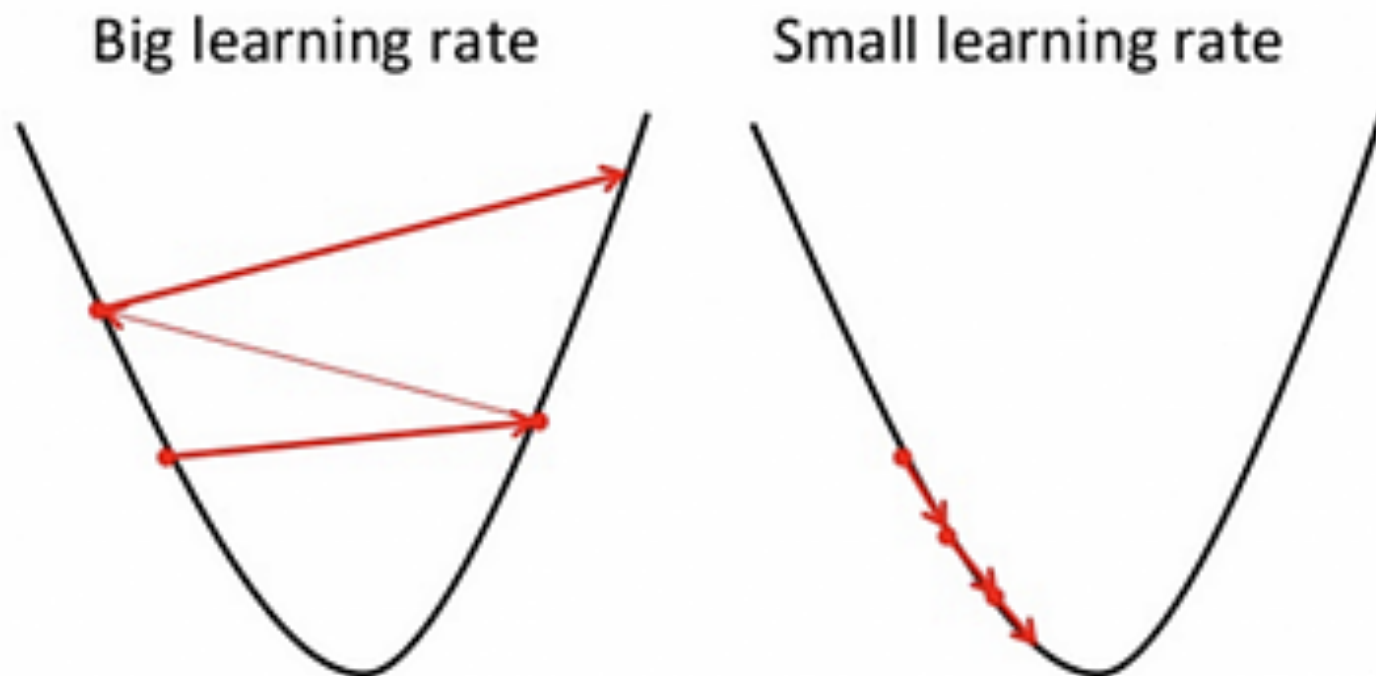
Andrew Ng

[Image source](#)

## Importance of Learning rate

How big the steps are gradient descent takes into the direction of the local minimum are determined by the learning rate, which figures out how fast or slows we will move towards the optimal weights.

For gradient descent to reach the local minimum we must set the learning rate to an appropriate value, which is neither too low nor too high. This is important because if the steps it takes are too big, it may not reach the local minimum because it bounces back and forth between the convex function of gradient descent (see left image below). If we set the learning rate to a very small value, gradient descent will eventually reach the local minimum but that may take a while (see the right image).



[Image Source](#)

So, the learning rate should never be too high or too low for this reason. You can check if you're learning rate is doing well by plotting it on a graph.

In code, 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
```

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`.

### Advantages:

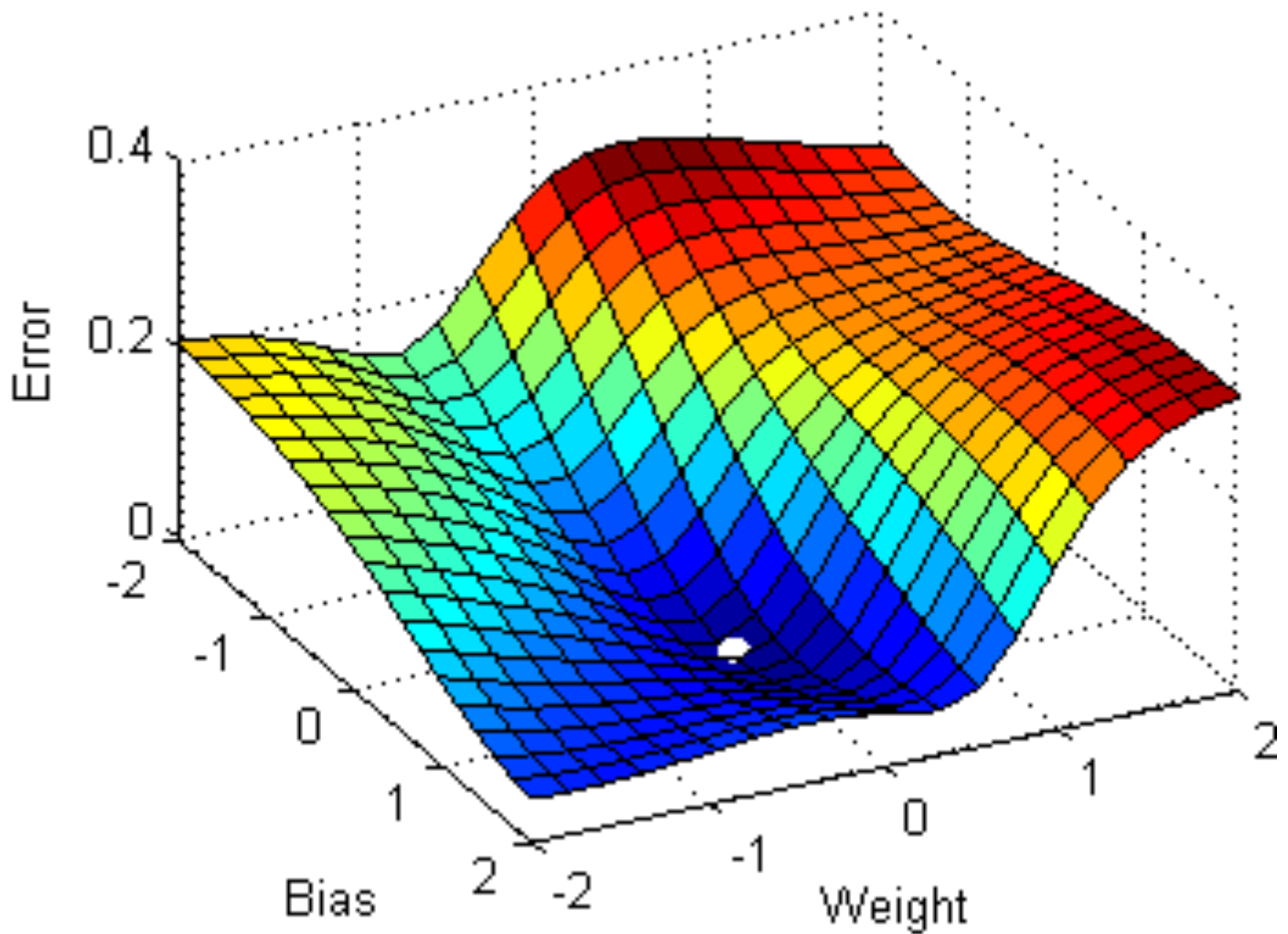
1. Easy computation.
2. Easy to implement.
3. Easy to understand.

### Disadvantages:

1. May trap at local minima.
2. Weights are changed after calculating the gradient on the whole dataset. So, if the dataset is too large then this may take years to converge to the minima.
3. Requires large memory to calculate the gradient on the whole dataset.

## Stochastic Gradient Descent (SGD)

SGD algorithm is an extension of the Gradient Descent and it overcomes some of the disadvantages of the GD algorithm. Gradient Descent has a disadvantage that it requires a lot of memory to load the entire dataset of  $n$ -points at a time to compute the derivative of the loss function. **In the SGD algorithm derivative is computed taking one point at a time.**



Stochastic Gradient Descent [Image Source](#)

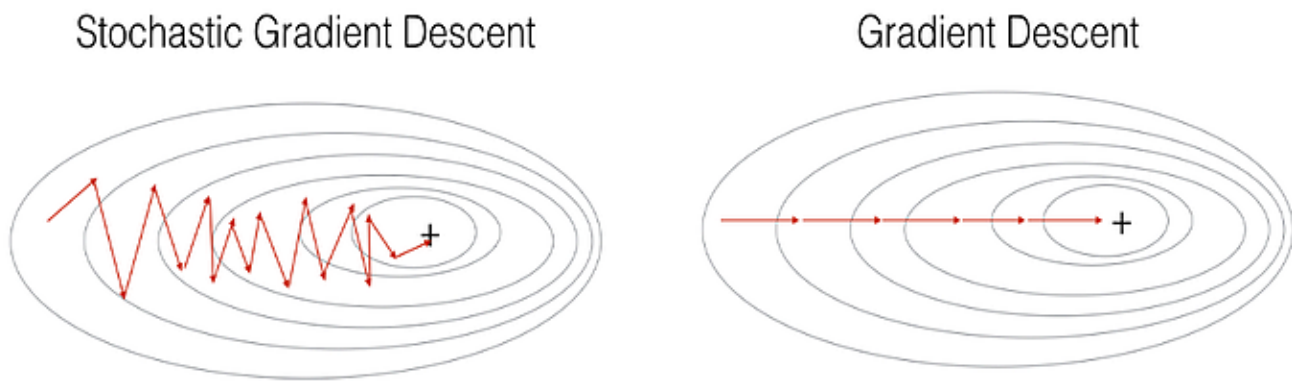
SGD performs a parameter update for *each* training example  $\mathbf{x}(\mathbf{i})$  and label  $\mathbf{y}(\mathbf{i})$ :

$$\theta = \theta - \alpha \cdot \partial (J(\theta; \mathbf{x}(\mathbf{i}), \mathbf{y}(\mathbf{i}))) / \partial \theta$$

where  $\{\mathbf{x}(\mathbf{i}), \mathbf{y}(\mathbf{i})\}$  are the training examples.

To make the training even faster we take a Gradient Descent step for each training example. Let's see what the implications would be in the image below.





**Figure 1 : SGD vs GD**

"+" denotes a minimum of the cost. SGD leads to many oscillations to reach convergence. But each step is a lot faster to compute for SGD than for GD, as it uses only one training example (vs. the whole batch for GD).

[Image Source](#)

1. On the left, we have Stochastic Gradient Descent (where  $m=1$  per step) we take a Gradient Descent step for each example and on the right is Gradient Descent (1 step per entire training set).
2. SGD seems to be quite noisy, at the same time it is much faster but may not converge to a minimum.
3. Typically, to get the best out of both worlds we use Mini-batch gradient descent (MGD) which looks at a smaller number of training set examples at once to help (usually power of 2 -  $2^6$  etc.).
4. Mini-batch Gradient Descent is relatively more stable than Stochastic Gradient Descent (SGD) but does have oscillations as gradient steps are being taken in the direction of a sample of the training set and not the entire set as in BGD.

It is observed that in SGD the updates take more number iterations compared to gradient descent to reach minima. On the right, the Gradient Descent takes fewer steps to reach minima but the SGD algorithm is noisier and takes more iterations.

Its code fragment simply adds a loop over the training examples and evaluates the gradient w.r.t. each example.

```
for i in range(nb_epochs):
    np.random.shuffle(data)
    for example in data:
        params_grad = evaluate_gradient(loss_function, example, params)
        params = params - learning_rate * params_grad
```

#### Advantage:

Memory requirement is less compared to the GD algorithm as the derivative is computed taking only 1 point at once.

#### Disadvantages:

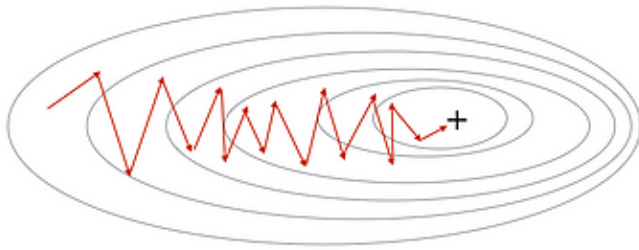
1. The time required to complete 1 epoch is large compared to the GD algorithm.
2. Takes a long time to converge.
3. May stuck at local minima.

#### Mini Batch Stochastic Gradient Descent (MB-SGD)

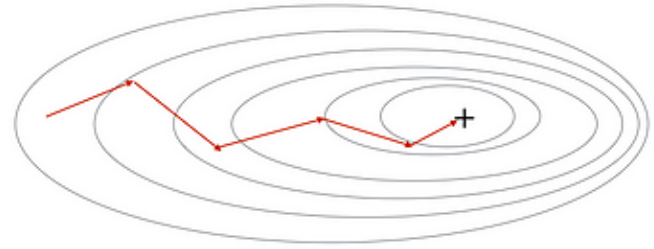
MB-SGD algorithm is an extension of the SGD algorithm and it overcomes the problem of large time complexity in the case of the SGD algorithm. MB-SGD algorithm takes a batch of points or subset of points from the dataset to compute derivative.

It is observed that the derivative of the loss function for MB-SGD is almost the same as a derivative of the loss function for GD after some number of iterations. But the number of iterations to achieve minima is large for MB-SGD compared to GD and the cost of computation is also large.

## Stochastic Gradient Descent



## Mini-Batch Gradient Descent



[Image Source](#)

The update of weight is dependent on the derivative of loss for a batch of points. The updates in the case of MB-SGD are much noisier because the derivative is not always towards minima.

MB-SGD divides the dataset into various batches and after every batch, the parameters are updated.

$$\theta = \theta - \alpha \cdot \partial(\mathcal{J}(\theta; \mathbf{B}(i))) / \partial \theta$$

where  $\{\mathbf{B}(i)\}$  are the batches of training examples.

In code, instead of iterating over examples, we now iterate over mini-batches of size 50:

```
for i in range(nb_epochs):
    np.random.shuffle(data)
    for batch in get_batches(data, batch_size=50):
        params_grad = evaluate_gradient(loss_function, batch, params)
        params = params - learning_rate * params_grad
```

### Advantages:

Less time complexity to converge compared to standard SGD algorithm.

### Disadvantages:

1. The update of MB-SGD is much noisier compared to the update of the GD algorithm.
2. Take a longer time to converge than the GD algorithm.
3. May get stuck at local minima.

### SGD with momentum

A major disadvantage of the MB-SGD algorithm is that updates of weight are very noisy. SGD with momentum overcomes this disadvantage by denoising the gradients. Updates of weight are dependent on noisy derivative and if we somehow denoise the derivatives then converging time will decrease.

The idea is to denoise derivative using exponential weighting average that is to give more weightage to recent updates compared to the previous update.

It accelerates the convergence towards the relevant direction and reduces the fluctuation to the irrelevant direction. One more hyperparameter is used in this method known as momentum symbolized by ' $\gamma$ '.

$$\mathbf{v}(t) = \gamma \cdot \mathbf{v}(t-1) + \alpha \cdot \partial(\mathcal{J}(\theta)) / \partial \theta$$

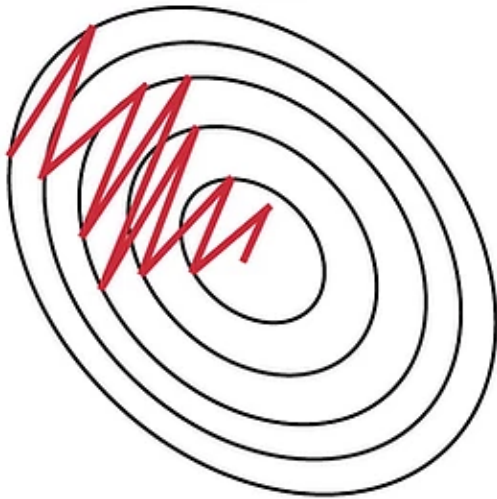
Now, the weights are updated by  $\theta = \theta - \mathbf{v}(t)$ .



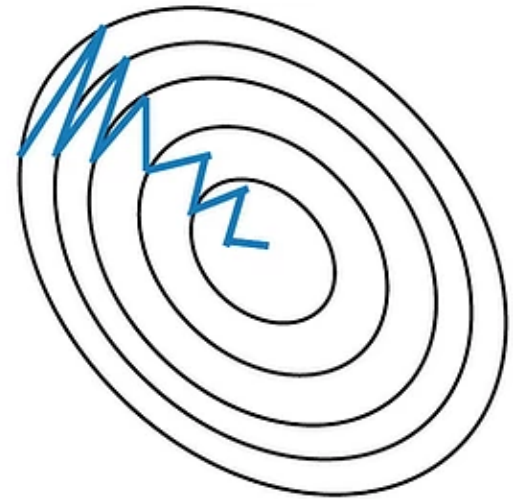
The momentum term  $\gamma$  is usually set to 0.9 or a similar value.

Momentum at time 't' is computed using all previous updates giving more weightage to recent updates compared to the previous update. This leads to speed up the convergence.

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.  $\gamma < 1$ ). 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.



Stochastic Gradient  
Descent **without**  
Momentum



Stochastic Gradient  
Descent **with**  
Momentum

[Image Source](#)

The diagram above concludes SGD with momentum denoises the gradients and converges faster as compared to SGD.

#### Advantages:

1. Has all advantages of the SGD algorithm.
2. Converges faster than the GD algorithm.

#### Disadvantages:

We need to compute one more variable for each update.

#### Nesterov Accelerated Gradient (NAG)

*Just one Peak*

The idea of the NAG algorithm is very similar to SGD with momentum with a slight variant. In the case of SGD with a momentum algorithm, the momentum and gradient are computed on the previous updated weight.

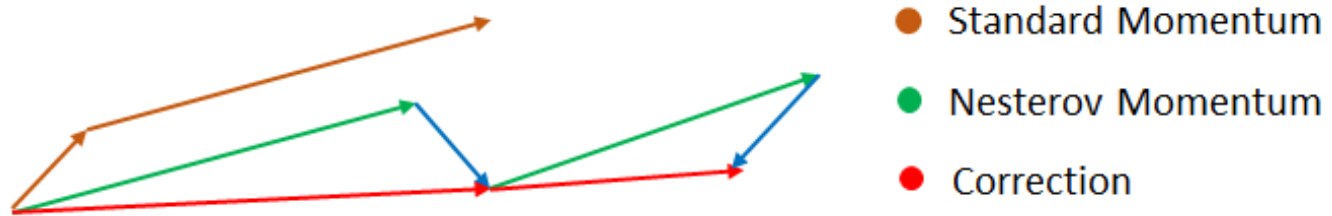
Momentum may be a good method but if the momentum is too high the algorithm may miss the local minima and may continue to rise up. So, to resolve this issue the NAG algorithm was developed. It is a look ahead method. We know we'll be using  $\gamma \cdot \nabla V(\mathbf{t}-1)$  for modifying the weights so,  $\mathbf{t} - \gamma \nabla V(\mathbf{t}-1)$  approximately tells us the future location. Now, we'll calculate the cost based on this future parameter rather than the current one.

$$\mathbf{v}(\mathbf{t}) = \gamma \cdot \mathbf{v}(\mathbf{t}-1) + \alpha \cdot \partial (J(\mathbf{t} - \gamma \mathbf{v}(\mathbf{t}-1))) / \partial \theta$$

and then update the parameters using  $\theta = \theta - \mathbf{v}(\mathbf{t})$

Again, we set the momentum term  $\gamma$  to a value of around 0.9. While Momentum first computes the current gradient (small brown vector in Image 4) and

then takes a big jump in the direction of the updated accumulated gradient (big brown vector), NAG first makes a big jump in the direction of the previously accumulated gradient (green vector), measures the gradient and then makes a correction (red vector), which results in the complete NAG update (red vector). This anticipatory update prevents us from going too fast and results in increased responsiveness, which has significantly increased the performance of RNNs on a number of tasks.



Both NAG and SGD with momentum algorithms work equally well and share the same advantages and disadvantages.

### Adaptive Gradient Descent(AdaGrad)

For all the previously discussed algorithms the learning rate remains constant. So the key idea of AdaGrad is to have an adaptive learning rate for each of the weights.

It performs smaller updates for parameters associated with frequently occurring features, and larger updates for parameters associated with infrequently occurring features.

For brevity, we use  $g_t$  to denote the gradient at time step  $t$ .  $g_{t,i}$  is then the partial derivative of the objective function w.r.t. to the parameter  $\theta_i$  at time step  $t$ ,  $\eta$  is the learning rate and  $\nabla \theta$  is the partial derivative of loss function  $J(\theta_i)$

$$g_{t,i} = \nabla_{\theta} J(\theta_{t,i}).$$

The SGD update for every parameter  $\theta_i$  at each time step  $t$  then becomes:

$$\theta_{t+1,i} = \theta_{t,i} - \eta \cdot g_{t,i}.$$

The idea behind Adagrad is to use different learning rates for each parameter based on iteration. The reason behind the need for different learning rates is that the learning rate for sparse features parameters needs to be higher compared to the dense features parameter because the frequency of occurrence of sparse features is lower.



In its update rule, Adagrad modifies the general learning rate  $\eta$  at each time step  $t$  for every parameter  $\theta_i$  based on the past gradients for  $\theta_i$ :

This algorithm performs best for sparse data because it decreases the learning rate faster for frequent parameters, and slower for parameters infrequent parameter.

Unfortunately, there is some case that the effective learning rate decreased very fast because we do accumulation of the gradients from the beginning of training

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} g_{t,i}$$

This can cause an issue that there is a point that the model will not learn again because the learning rate is almost zero.

This issue was mitigated by some algorithms that extend AdaGrad, and these algorithms will be the subject of next post.

where  $G_t$  is the sum of the squares of the past gradients w.r.t. to all parameters  $\theta$ .

The benefit of AdaGrad is that it eliminates the need to manually tune the learning rate; most leave it at a default value of 0.01.

Its main weakness is the accumulation of the squared gradients ( $G_t$ ) in the denominator. Since every added term is positive, the accumulated sum keeps growing during training, causing the learning rate to shrink and becoming infinitesimally small and further resulting in a vanishing gradient problem.

**Advantage:**

No need to update the learning rate manually as it changes adaptively with iterations.

**Disadvantage:**

As the number of iteration becomes very large learning rate decreases to a very small number which leads to slow convergence.

## AdaDelta

The problem with the previous algorithm AdaGrad was learning rate becomes very small with a large number of iterations which leads to slow convergence. To avoid this, the AdaDelta algorithm has an idea to take an exponentially decaying average.

Adadelata is a more robust extension of Adagrad that adapts learning rates based on a moving window of gradient updates, instead of accumulating all past gradients. This way, Adadelata continues learning even when many updates have been done. Compared to Adagrad, in the original version of Adadelata, you don't have to set an initial learning rate.

Instead of inefficiently storing w previous squared gradients, the sum of gradients is recursively defined as a decaying average of all past squared gradients. The running average  $E[g^2]_t$  at time step t then depends only on the previous average and current gradient:

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1 - \gamma)g_t^2$$

Usually  $\gamma$  is set to around 0.9. Rewriting SGD updates in terms of the parameter update vector:

$$\begin{aligned}\Delta\theta_t &= -\eta \cdot g_{t,i} \\ \theta_{t+1} &= \theta_t + \Delta\theta_t\end{aligned}$$

AdaDelta takes the form:

$$\Delta\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

With Adadelata, we do not even need to set a default learning rate, as it has been eliminated from the update rule.

**Adagrad**

$$\Delta\theta_t = -\frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t$$

**SGD**

$$\begin{aligned}\Delta\theta_t &= -\eta \cdot g_{t,i} \\ \theta_{t+1} &= \theta_t + \Delta\theta_t\end{aligned}$$



Replace the diagonal matrix  $G_t$  with the decaying average over past squared gradients  $E[g^2]_t$

**Adadelta**

$$\Delta\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

**RMSprop**

RMSprop in fact is identical to the first update vector of Adadelta that we derived above:

$$\begin{aligned}E[g^2]_t &= 0.9E[g^2]_{t-1} + 0.1g_t^2 \\ \theta_{t+1} &= \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t\end{aligned}$$

RMSprop as well divides the learning rate by an exponentially decaying average of squared gradients. Hinton suggests  $\gamma$  be set to 0.9, while a good default value for the learning rate  $\eta$  is 0.001.

RMSprop and Adadelta have both been developed independently around the same time stemming from the need to resolve Adagrad's radically diminishing learning rates

**Adaptive Moment Estimation (Adam)**

Adam can be looked at as a combination of RMSprop and Stochastic Gradient Descent with momentum.

Adam computes adaptive learning rates for each parameter. In addition to storing an exponentially decaying average of past squared gradients  $\mathbf{v}_t$  like Adadelta and RMSprop, Adam also keeps an exponentially decaying average of past gradients  $\mathbf{m}_t$ , similar to momentum. Whereas momentum can be seen as a ball running down a slope, Adam behaves like a heavy ball with friction, which thus prefers flat minima in the error surface.

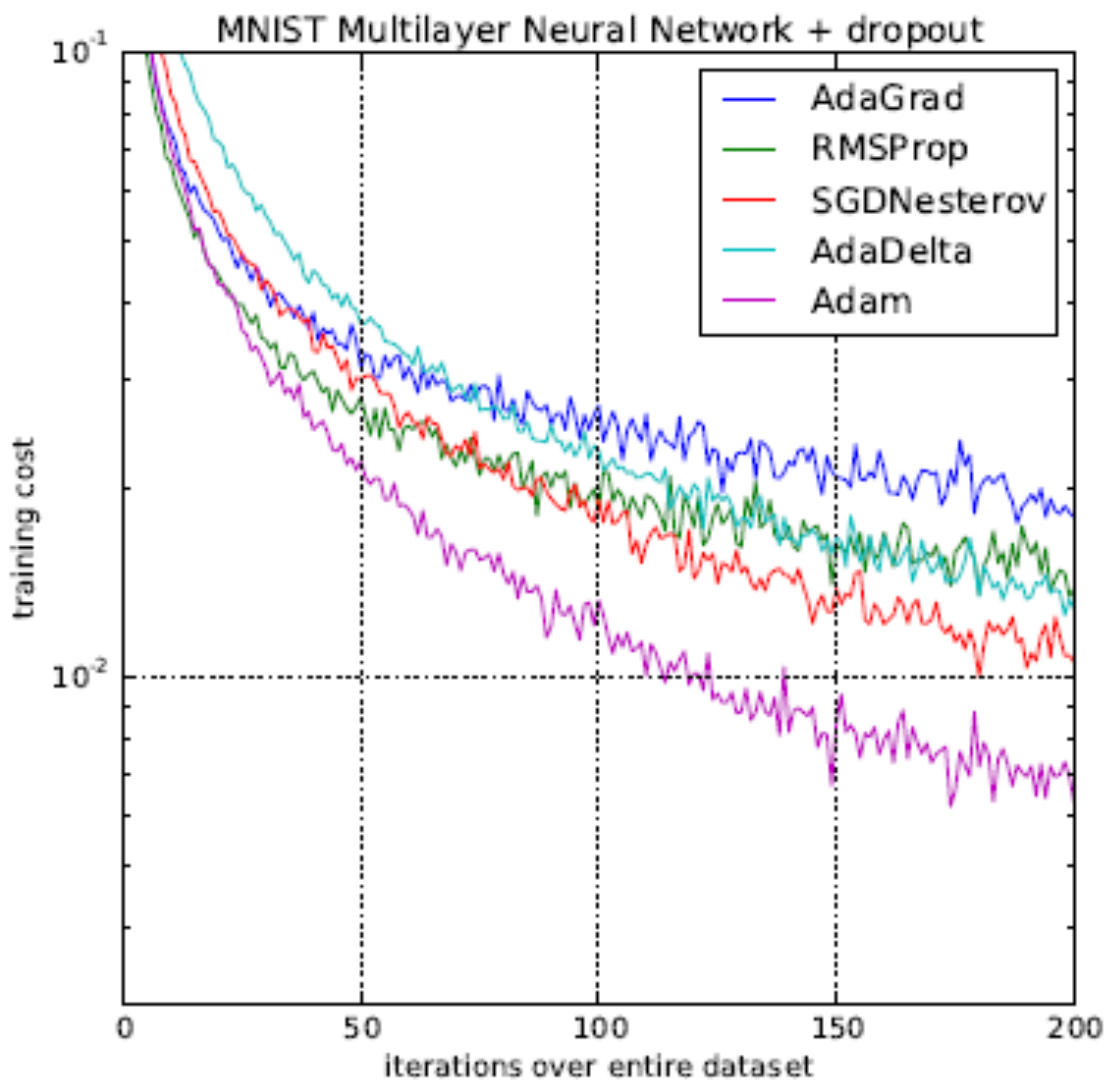
Hyper-parameters  $\beta_1, \beta_2 \in [0, 1)$  control the exponential decay rates of these moving averages. We compute the decaying averages of past and past squared gradients  $\mathbf{m}_t$  and  $\mathbf{v}_t$  respectively as follows:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

$m_t$  and  $v_t$  are estimates of the first moment (the mean) and the second moment (the uncentered variance) of the gradients respectively, hence the name of the method.

**When to choose which algorithm?**



[Image credits](#)

As you can observe, the training cost in the case of Adam is the least.

Now observe the animation at the beginning of this article and consider the following points:

1. It is observed that the SGD algorithm (red) is stuck at a saddle point. So SGD algorithm can only be used for shallow networks.
2. All the other algorithms except SGD finally converges one after the other, AdaDelta being the fastest followed by momentum algorithms.
3. AdaGrad and AdaDelta algorithm can be used for sparse data.

4. Momentum and NAG work well for most cases but is slower.
5. Animation for Adam is not available but from the plot above it is observed that it is the fastest algorithm to converge to minima.
6. Adam is considered the best algorithm amongst all the algorithms discussed above.

To understand how to implement all these optimization algorithms in neural networks using python you can have a look [here](#).

#### References:

- <https://towardsdatascience.com/overview-of-various-optimizers-in-neural-networks-17c1be2df6d5>
- <https://towardsdatascience.com/optimizers-for-training-neural-network-59450d71caf6>
- <https://towardsdatascience.com/stochastic-gradient-descent-with-momentum-a84097641a5d>
- <https://ruder.io/optimizing-gradient-descent/>

**Bio:** [Nagesh Singh Chauhan](#) is a Big data developer at CirrusLabs. He has over 4 years of working experience in various sectors like Telecom, Analytics, Sales, Data Science having specialisation in various Big data components.

[Original](#). Reposted with permission.

#### Related:

- [Hyperparameter Optimization for Machine Learning Models](#)
- [Model Evaluation Metrics in Machine Learning](#)
- [Introduction to Convolutional Neural Networks](#)

#### What do you think?

26 Responses



0 Comments   KDnuggets   Disqus' Privacy Policy

Login ▾

Recommend   Tweet   Share

Sort by Best ▾



Start the discussion...

LOG IN WITH



OR SIGN UP WITH DISQUS

Name

Be the first to comment.

Subscribe   Add Disqus to your siteAdd DisqusAdd   Do Not Sell My Data

[<= Previous post](#)

[Next post =>](#)

## Top Stories Past 30 Days

#### Most Popular

1. [Top 6 Data Science Online Courses in 2021](#)
2. [Data Scientists and ML Engineers Are Luxury Employees](#)
3. [Advice for Learning Data Science from Google's Director of Research](#)

#### Most Shared

1. [Why and how should you learn "Productive Data Science"?](#)
2. [Not Only for Deep Learning: How GPUs Accelerate Data Science & Data Analytics](#)
3. [Bootstrap a Modern Data Stack in 5 minutes with Terraform](#)



4. [GitHub Copilot Open Source Alternatives](#)
5. [Geometric foundations of Deep Learning](#)

4. [GPU-Powered Data Science \(NOT Deep Learning\) with RAPIDS](#)
5. [Become an Analytics Engineer in 90 Days](#)

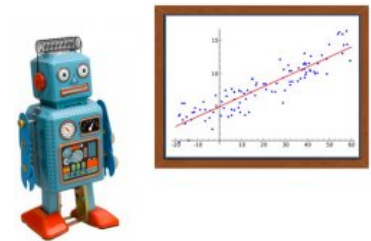
## Latest News

- [MLOPs And Machine Learning RoadMap](#)
- [3 mindset changes to become a better analyst](#)
- [How to Detect and Overcome Model Drift in MLOps](#)
- [2021 State of Production Machine Learning Survey](#)
- [The Difference Between Data Scientists and ML Engineers](#)
- [For SQL, or why I'm so over-protective of my data...](#)

## Top Stories Last Week

## Most Popular

1. [3 Reasons Why You Should Use Linear Regression Models Instead of Neural Networks](#)
2. [Most Common Data Science Interview Questions and Answers](#)
3. [How Visualization is Transforming Exploratory Data Analysis](#)
4. [GitHub Copilot Open Source Alternatives](#)
5. [How To Become A Freelance Data Scientist – 4 Practical Tips](#)



## Most Shared

1. [Bootstrap a Modern Data Stack in 5 minutes with Terraform](#)
2. [GPU-Powered Data Science \(NOT Deep Learning\) with RAPIDS](#)
3. [3 Reasons Why You Should Use Linear Regression Models Instead of Neural Networks](#)
4. [How Visualization is Transforming Exploratory Data Analysis](#)
5. [Top Stories, Jul 26 – Aug 1: GitHub Copilot Open Source Alternatives; Why and how should you learn “Productive Data Science”?](#)

## More Recent Stories

- [For SQL, or why I'm so over-protective of my data people](#)
- [DeepMind's New Super Model: Perceiver IO is a Transformer th...](#)
- [KDnuggets 21:n30, Aug 11: Most Common Data Science Intervie...](#)
- [AI in Real Life](#)
- [How My Learning Path Changed After Becoming a Data Scientist](#)
- [Practising SQL without your own database](#)
- [Visualizing Bias-Variance](#)
- [5 Tips for Writing Clean R Code](#)
- [Top Stories, Aug 2-8: 3 Reasons Why You Should Use Linear Regr...](#)
- [Including ModelOps in your AI strategy](#)
- [How to Query Your Pandas Dataframe](#)
- [Using Twitter to Understand Pizza Delivery Apprehension During...](#)
- [Bootstrap a Modern Data Stack in 5 minutes with Terraform \[Gold Blog\]](#)
- [Essential Math for Data Science: Introduction to Systems of Li...](#)
- [Be Wary of Automated Feature Selection — Chi Square Test of ...](#)
- [Most Common Data Science Interview Questions and Answers \[Silver Blog\]](#)
- [Artificial Intelligence vs Machine Learning in Cybersecurity](#)
- [How Visualization is Transforming Exploratory Data Analysis](#)
- [How To Become A Freelance Data Scientist – 4 Practical Tips](#)
- [How DeepMind Trains Agents to Play Any Game Without Intervention](#)

[KDnuggets Home](#) » [News](#) » [2020](#) » [Dec](#) » [Tutorials, Overviews](#) » Optimization Algorithms in Neural Networks ( [20:n48](#) )

© 2021 KDnuggets. | [About KDnuggets](#) | [Contact](#) | [Privacy policy](#) | [Terms of Service](#)

[Subscribe to KDnuggets News](#)

X