

What are Activation functions and what are it uses in a Neural Network Model?

Activation functions are really important for a Artificial Neural Network to learn and make sense of something really complicated and Non-linear complex functional mappings between the inputs and response variable. *They introduce non-linear properties to our Network. Their main purpose is to convert a input signal of a node in a A-NN to an output signal.* That output signal now is used as a input in the next layer in the stack.

Specifically in A-NN we do the sum of products of inputs(\mathbf{X}) and their corresponding Weights(\mathbf{W}) and apply a Activation function $\mathbf{f}(\mathbf{x})$ to it to get the output of that layer and feed it as an input to the next layer.

The question arises that why can't we do it without activating the input signal?

If we do not apply a Activation function then the output signal would simply be a simple **linear function**. A *linear function* is just a polynomial of **one degree**. Now, a linear equation is easy to solve but they are limited in their complexity and have less power to learn complex functional mappings from data. A Neural Network without Activation function would simply be a **Linear regression Model**, which has limited power and does not performs good most of the times. We want our Neural Network to not just learn and compute a linear function but something more complicated than that. *Also without activation function our Neural network would not be able to learn and model other complicated kinds of data such as images, videos , audio , speech etc. That is why we use Artificial Neural network techniques such as Deep learning to make sense of something complicated ,high dimensional,non-linear -big datasets,*

where the model has lots and lots of hidden layers in between and has a very complicated architecture which helps us to make sense and extract knowledge from such complicated big datasets.

So why do we need Non-Linearities?

Non-linear functions are those which have degree more than one and they have a curvature when we plot a Non-Linear function. Now we need a Neural Network Model to learn and represent almost anything and any arbitrary complex function which maps inputs to outputs. Neural-Networks are considered **Universal Function**

Approximators. *It means that they can compute and learn any function at all.*

Almost any process we can think of can be represented as a functional computation in Neural Networks.

Hence it all comes down to this, we need to apply a Activation function $f(x)$ so as to make the network more powerful and add ability to it to learn something complex and complicated from data and represent non-linear complex arbitrary functional mappings between inputs and outputs. Hence using a non linear Activation we are able to generate non-linear mappings from inputs to outputs.

Also another important feature of a Activation function is that it should be differentiable. We need it to be this way so as to perform backpropagation optimization strategy while propagating backwards in the network to compute gradients of Error(loss) with respect to Weights and then accordingly optimize weights using Gradient descend or any other Optimization technique to reduce Error.

Just always remember to do :

“Input times weights , add Bias and Activate”

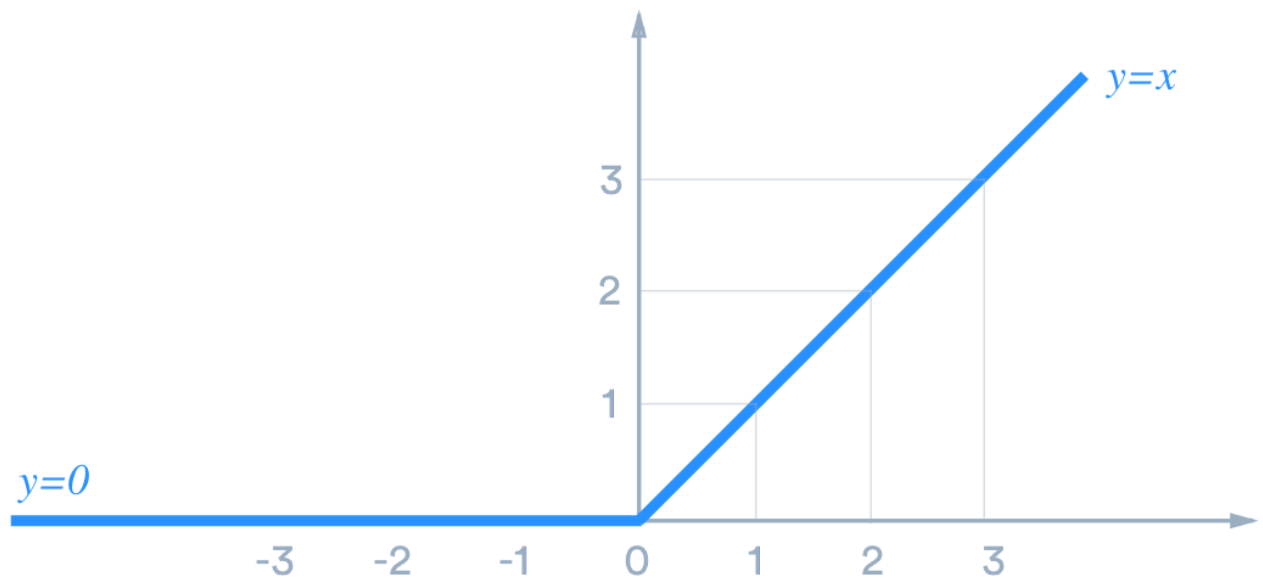
Most popular types of Activation functions -

1. *Sigmoid or Logistic*
2. *Tanh — Hyperbolic tangent*
3. **ReLU -Rectified linear units**
4. Softmax

Understanding ReLu ..

ReLU stands for rectified linear unit, and is a type of activation function.

Mathematically, it is defined as $y = \max(0, x)$. Visually, it looks like the following:



ReLU is the most commonly used activation function in neural networks, especially in CNNs. If you are unsure what activation function to use in your network, ReLU is usually a good first choice.

How does ReLU compare

ReLU is linear (identity) for all positive values, and zero for all negative values.

This means that:

- It's cheap to compute as there is no complicated math. The model can therefore take less time to train or run.
- It converges faster. Linearity means that the slope doesn't plateau, or "saturate," when x gets large. It doesn't have the vanishing gradient problem suffered by other activation functions like sigmoid or tanh.

- It's sparsely activated. Since ReLU is zero for all negative inputs, it's likely for any given unit to not activate at all. This is often desirable (see below).

Sparsity

Note: We are discussing model sparsity here. Data sparsity (missing information) is different and usually bad.

Why is sparsity good? It makes intuitive sense if we think about the biological neural network, which artificial ones try to imitate. While we have billions of neurons in our bodies, not all of them fire all the time for everything we do. Instead, they have different roles and are activated by different signals.

Sparsity results in concise models that often have better predictive power and less overfitting/noise. In a sparse network, it's more likely that neurons are actually processing meaningful aspects of the problem. For example, in a model detecting cats in images, there may be a neuron that can identify ears, which obviously shouldn't be activated if the image is about a building.

Finally, a sparse network is faster than a dense network, as there are fewer things to compute.

Dying ReLU

The downside for being zero for all negative values is a problem called “dying ReLU.”

A ReLU neuron is “dead” if it's stuck in the negative side and always outputs 0. Because the slope of ReLU in the negative range is also 0, once a neuron gets

negative, it's unlikely for it to recover. Such neurons are not playing any role in discriminating the input and is essentially useless. Over the time you may end up with a large part of your network doing nothing.

You may be confused as of how this zero-slope section works in the first place. Remember that a single step (in SGD, for example) involves multiple data points. As long as not all of them are negative, we can still get a slope out of ReLU. The dying problem is likely to occur when learning rate is too high or there is a large negative bias.

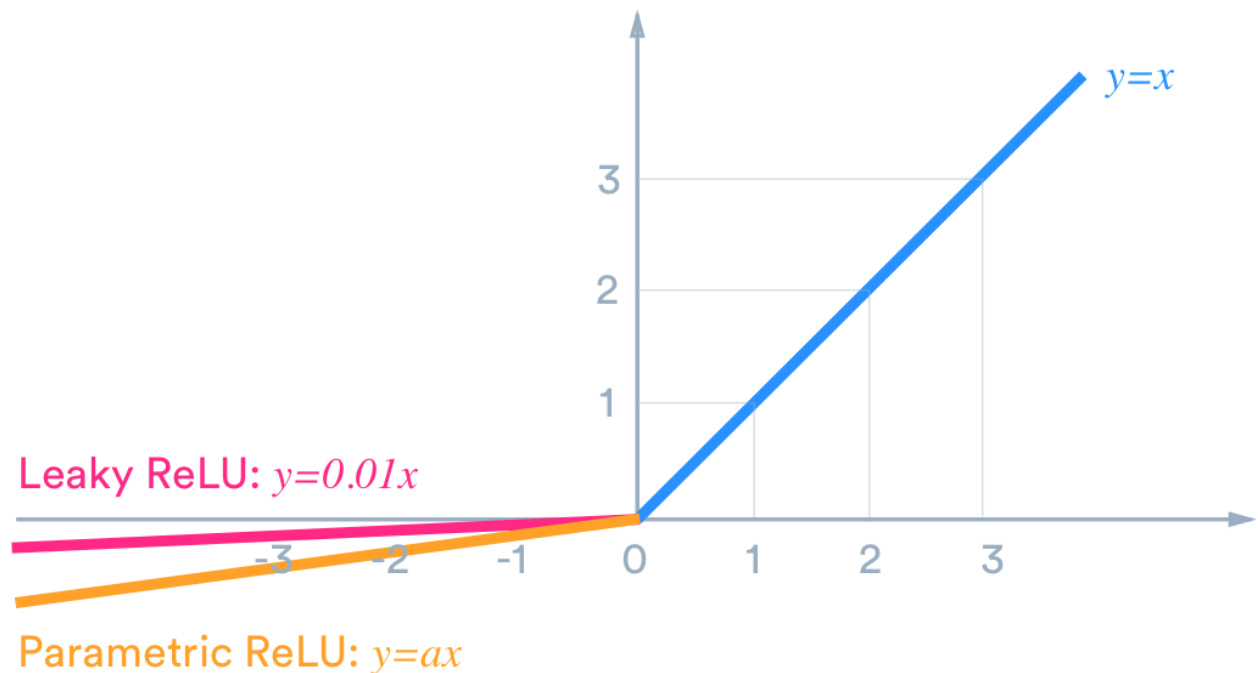
Lower learning rates often mitigates the problem. If not, leaky ReLU and ELU are also good alternatives to try. They have a slight slope in the negative range, thereby preventing the issue.

Variants

Leaky ReLU & Parametric ReLU (PReLU)

Leaky ReLU has a small slope for negative values, instead of altogether zero. For example, leaky ReLU may have $y = 0.01x$ when $x < 0$.

Parametric ReLU (PReLU) is a type of leaky ReLU that, instead of having a predetermined slope like 0.01, makes it a parameter for the neural network to figure out itself: $y = ax$ when $x < 0$.



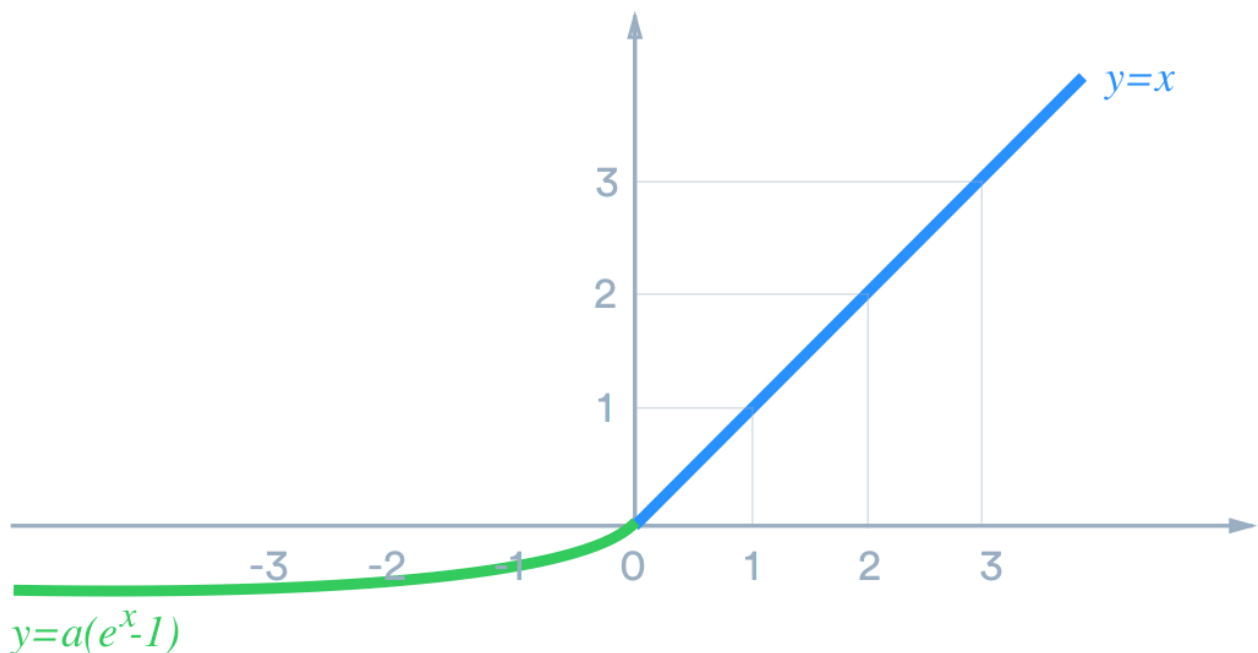
Leaky ReLU has two benefits:

- It fixes the “dying ReLU” problem, as it doesn’t have zero-slope parts.
- It speeds up training. There is evidence that having the “mean activation” be close to 0 makes training faster. (It helps keep off-diagonal entries of the Fisher information matrix small, but you can safely ignore this.) Unlike ReLU, leaky ReLU is more “balanced,” and may therefore learn faster.

Be aware that the result is not always consistent. Leaky ReLU isn’t always superior to plain ReLU, and should be considered only as an alternative.

Exponential Linear (ELU, SELU)

Similar to leaky ReLU, ELU has a small slope for negative values. Instead of a straight line, it uses a log curve like the following:



It is designed to combine the good parts of ReLU and leaky ReLU — while it doesn't have the dying ReLU problem, it saturates for large negative values, allowing them to be essentially inactive.

ELU was first proposed in [this paper](#). It is sometimes called Scaled ELU (SELU) due to the constant factor a .

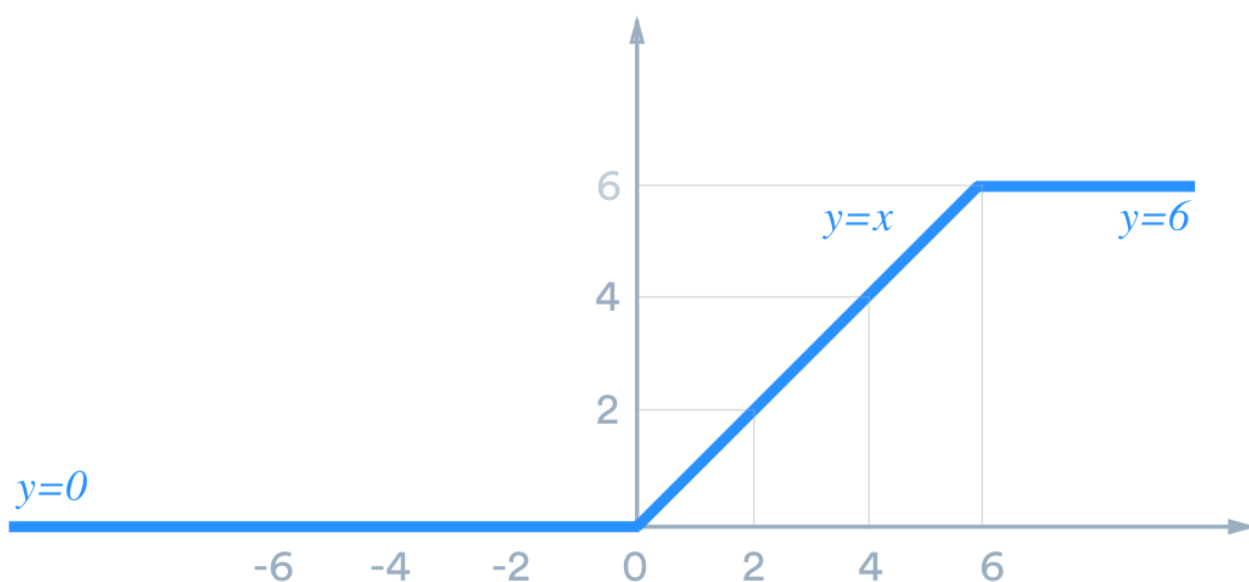
Concatenated ReLU (CReLU)

Concatenated ReLU has *two* outputs, one ReLU and one negative ReLU, concatenated together. In other words, for positive x it produces $[x, 0]$, and for negative x it produces $[0, x]$. Because it has two outputs, CReLU doubles the output dimension.

CReLU was first proposed in [this paper](#).

ReLU-6

You may run into ReLU-6 in some libraries, which is ReLU capped at 6. In other words, it looks like the following:



It was first used in [this paper](#) for CIFAR-10, and 6 is an arbitrary choice that worked well. According to the authors, the upper bound encouraged their model to learn sparse features earlier.

How do I use it?

ReLU and its variants come standard with most frameworks.

TensorFlow

TensorFlow provides ReLU and its variants through the `tf.nn` module. For example, the following creates a convolution layer (for CNN) with `tf.nn.relu`:

```
import tensorflow as tf

conv_layer = tf.layers.conv2d(

    inputs=input_layer,

    filters=32,

    kernel_size=[5, 5],

    padding='same',

    activation=tf.nn.relu,

)
```

Keras

Keras provides ReLU and its variants through the `keras.layers.Activation` module. The following adds a ReLU layer to the model:

```
from keras.layers import Activation, Dense
```

```
model.add(Dense(64, activation='relu'))
```

PyTorch

PyTorch provides ReLU and its variants through the `torch.nn` module. The following adds 2 CNN layers with ReLU:

```
from torch.nn import RNN
```

```
model = nn.Sequential(
```

```
    nn.Conv2d(1, 20, 5),
```

```
    nn.ReLU(),
```

```
    nn.Conv2d(20, 64, 5),
```

```
    nn.ReLU()
```

```
)
```

It can also be used directly:

```
import torch
```

```
from torch import autograd, nn
```

```
relu = nn.ReLU()
```

```
var = autograd.Variable(torch.randn(2))
```

```
relu(var)
```

Certain PyTorch layer classes take `relu` as a value to their `nonlinearity` argument. For example, the following creates an RNN layer with ReLU:

```
rnn = nn.RNN(10, 20, 2, nonlinearity='relu')
```

Softmax Function

Softmax turns arbitrary real values into probabilities, which are often useful in Machine Learning. The math behind it is pretty simple: given some numbers,

1. Raise [e](#) (the mathematical constant) to the power of each of those numbers.
2. Sum up all the exponentials (powers of
3. e
4. e). This result is the *denominator*.
5. Use each number's exponential as its *numerator*.
6. $\text{Probability} = \frac{\text{Numerator}}{\text{Denominator}}$
7. Probability =

8. Denominator

9. Numerator

10.

11..

Written more fancily, Softmax performs the following transform on

n

n numbers

$x_1 \dots x_n$

x

1

$\dots x$

n

:

$$s(\mathbf{x}_i) = \frac{e^{\mathbf{x}_i}}{\sum_{j=1}^n e^{\mathbf{x}_j}}$$

s(*x*

i

)=

Σ

j=1

n

e

x

j

e

x

i

The outputs of the Softmax transform are always in the range

$[0, 1]$

$[0,1]$ and add up to 1. Hence, they form a probability distribution.

A Simple Example

Say we have the numbers -1, 0, 3, and 5. First, we calculate the denominator:

$$\begin{aligned} \text{Denominator} &= e^{-1} + e^0 + e^3 + e^5 \\ &= \boxed{169.87} \end{aligned}$$

Denominator

$=e$

-1

$+e$

0

$+e$

3

$+e$

5

$=$

169.87

Then, we can calculate the numerators and probabilities:

x	Numerator (e^x)	Probability ($\frac{e^x}{169.87}$)
-1	0.368	0.002
0	1	0.006
3	20.09	0.118
5	148.41	0.874

The bigger the

x

x , the higher its probability. Also, notice that the probabilities all add up to 1, as mentioned before.

Implementing Softmax in Python

Using [numpy](#) makes this super easy:

```
import numpy as np
```

```
def softmax(xs):
```

```
    return np.exp(xs) / sum(np.exp(xs))
```

```
xs = np.array([-1, 0, 3, 5])
```

```
print(softmax(xs)) # [0.0021657, 0.00588697, 0.11824302,  
0.87370431]
```

[np.exp\(\)](#) raises e to the power of each element in the input array.

Why is Softmax useful?

Imagine building a [Neural Network](#) to answer the question: *Is this picture of a dog or a cat?*

A common design for this neural network would have it output 2 real numbers, one representing *dog* and the other *cat*, and apply Softmax on these values. For example, let's say the network outputs

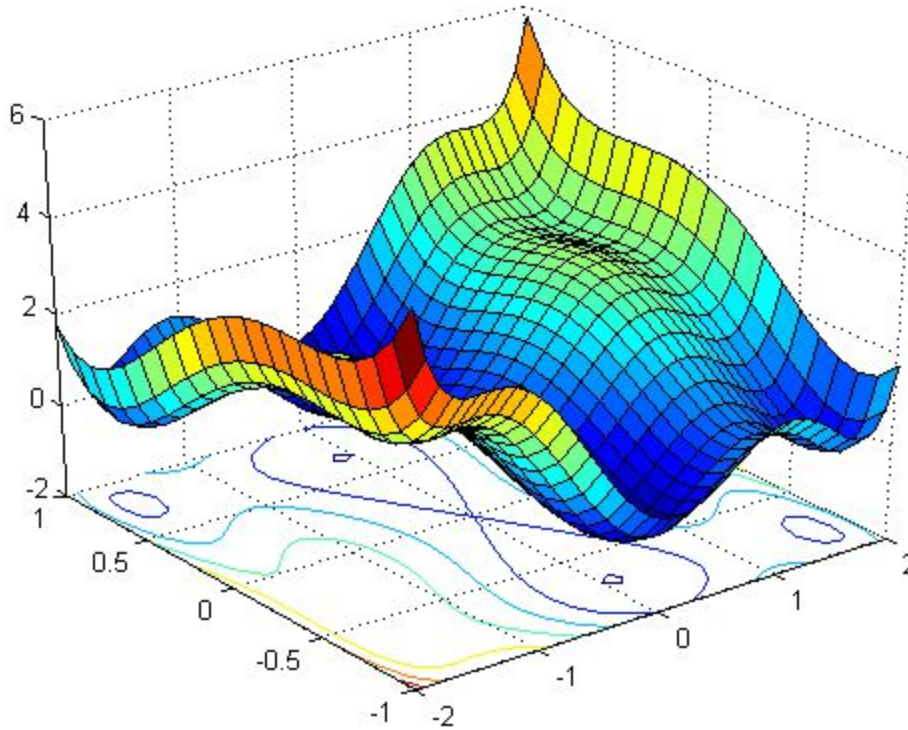
$[-1, 2]$

$[-1, 2]$:

Animal	x	e^x	Probability
	x	e	
	x		
Dog	-1	0.368	0.047
Cat	2	7.39	0.953

This means our network is 95.3% confident that the picture is of a cat. Softmax lets us answer classification questions with probabilities, which are more useful than simpler answers (e.g. binary yes/no).

Introduction to Loss Functions



The loss function is the bread and butter of modern [machine learning](#); it takes your algorithm from theoretical to practical and transforms neural networks from glorified matrix multiplication into [deep learning](#).

This post will explain the role of loss functions and how they work, while surveying a few of the most popular from the past decade.

What's a Loss Function?

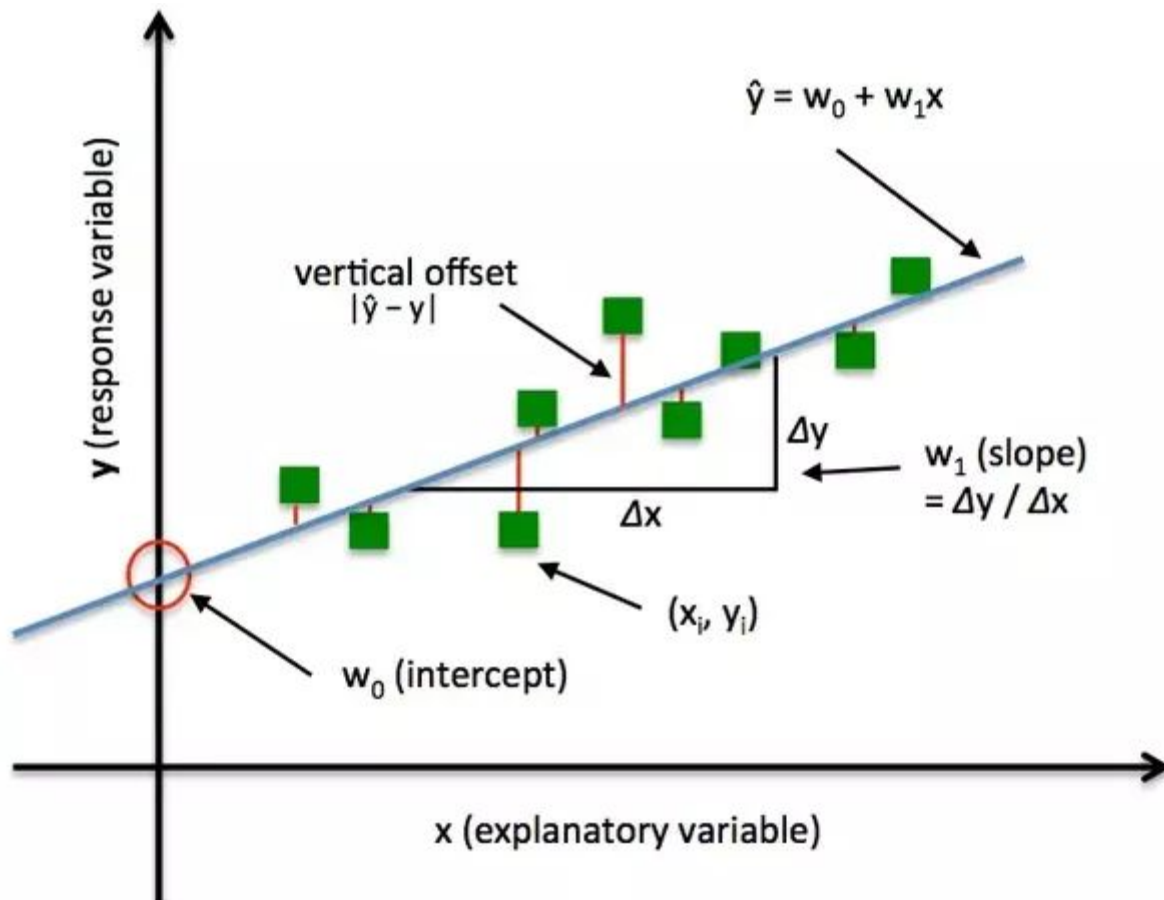
At its core, a loss function is incredibly simple: it's a method of evaluating how well your algorithm models your dataset. If your predictions are totally off, your loss function will output a higher number. If they're pretty good, it'll output a lower number. As you change pieces of your algorithm to try and improve your model, your loss function will tell you if you're getting anywhere.

In fact, we can design our own (very) basic loss function to further explain how it works. For each prediction that we make, our loss function will simply measure the absolute difference between our prediction and the actual value. In mathematical notation, it might look something like $abs(y_predicted - y)$. Here's what some situations might look like if we were trying to predict how expensive the rent is in some NYC apartments:

Our Predictions	Actual Values	Our Total Loss
Harlem: \$1,000 SoHo: \$2,000 West Village: \$3,000	Harlem: \$1,000 SoHo: \$2,000 West Village: \$3,000	0 (we got them all right!)
Harlem: \$500 SoHo: \$2,000 West Village: \$3,000		500 (we were off by \$500 in Harlem)
Harlem: \$500 SoHo: \$1,500 West Village: \$4,000		2000 (we were off by \$500 in Harlem, \$500 in SoHo, and \$1,000 in the West Village)

Notice how in the loss function we defined, it doesn't matter if our predictions were too high or too low. All that matters is how incorrect we were, directionally agnostic. This is *not* a feature of all loss functions: in fact, your loss function will vary significantly based on the domain and unique context of the problem that you're applying machine learning

to. In your project, it may be much worse to guess too high than to guess too low, and the loss function you select must reflect that.



Different Types and Flavors of Loss Functions

A lot of the loss functions that you see implemented in machine learning can get complex and confusing. Consider [this paper from late 2017](#), entitled *A Semantic Loss Function for Deep Learning with Symbolic Knowledge*. There's more in that title that I don't understand than I do. But if you remember the end goal of all loss

functions—measuring how well your algorithm is doing on your dataset—you can keep that complexity in check.

We'll run through a few of the most popular loss functions currently being used, from simple to more complex.

Mean Squared Error

Mean Squared Error (MSE) is the workhorse of basic loss functions: it's easy to understand and implement and generally works pretty well. To calculate MSE, you take the difference between your predictions and the ground truth, square it, and average it out across the whole dataset.

Implemented in code, MSE might look something like:

```
def MSE(y_predicted, y):  
  
    squared_error = (y_predicted - y) ** 2  
  
    sum_squared_error = np.sum(squared_error)  
  
    mse = sum_squared_error / y.size  
  
    return(mse)
```

Likelihood Loss

The [likelihood function](#) is also relatively simple, and is commonly used in classification problems. The function takes the predicted probability for each input example and multiplies them. And although the output isn't exactly human interpretable, it's useful for comparing models.

For example, consider a model that outputs probabilities of [0.4, 0.6, 0.9, 0.1] for the ground truth labels of [0, 1, 1, 0]. The likelihood loss would be computed as $(0.6) * (0.6) * (0.9) * (0.9) = 0.2916$. Since the model outputs probabilities for TRUE (or 1) only, when the ground truth label is 0 we take $(1-p)$ as the probability. In other words, we multiply the model's outputted probabilities together for the actual outcomes.

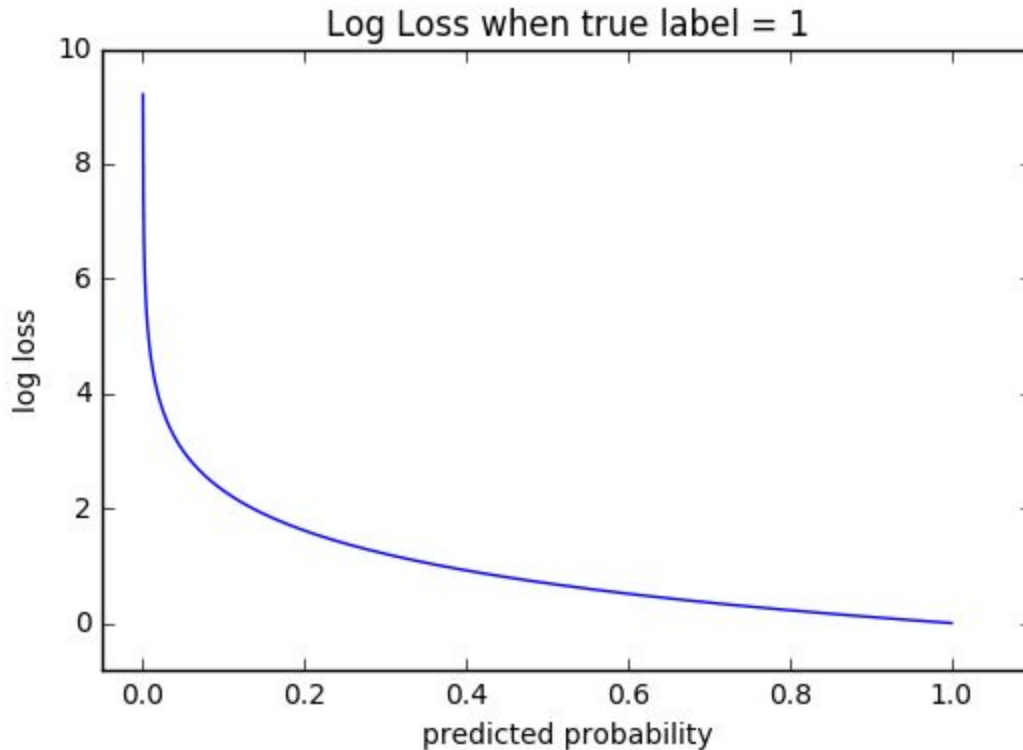
Log Loss (Cross Entropy Loss)

[Log Loss](#) is a loss function also used frequently in classification problems, and is one of the most popular measures for [Kaggle](#) competitions. It's just a straightforward modification of the likelihood function with logarithms.

$$-(y \log(p) + (1 - y) \log(1 - p))$$

This is actually exactly the same formula as the regular likelihood function, but with logarithms added in. You can see that when the actual class is 1, the second half of the function disappears, and when the actual class is 0, the first half drops. That way, we just end up multiplying the log of the actual predicted probability for the ground truth class.

The cool thing about the log loss loss function is that it has a kick: it penalizes heavily for being *very confident* and *very wrong*. Predicting high probabilities for the wrong class makes the function go crazy. The graph below is for when the true label = 1, and you can see that it skyrockets as the predicted probability for label = 0 approaches 1.



Source: [Fast.ai](#)

Loss Functions and Optimizers

Loss functions provide more than just a static representation of how your model is performing—they're how your algorithms fit data in the first place. Most machine learning

algorithms use some sort of loss function in the process of optimization, or finding the best parameters (weights) for your data.

For a simple example, consider [linear regression](#). In traditional “least squares” regression, the line of best fit is determined through none other than MSE (hence the least squares moniker)! For each set of weights that the model tries, the MSE is calculated across all input examples. The model then optimizes the MSE functions—or in other words, makes it the lowest possible—through the use of an optimizer algorithm like [Gradient Descent](#).

Just like there are different flavors of loss functions for unique problems, there is no shortage of different optimizers as well. That’s beyond the scope of this post, but in essence, the loss function and optimizer work in tandem to fit the algorithm to your data in the best way possible.

What are Optimization Algorithms ?

Optimization algorithms helps us to ***minimize (or maximize)*** an **Objective** function (*another name for **Error** function*) $E(\mathbf{x})$ which is simply a mathematical function dependent on the Model’s internal **learnable parameters** which are used in computing the target values(\mathbf{Y}) from the set of *predictors*(\mathbf{X}) used in the model. For example — we call the **Weights**(\mathbf{W}) and the **Bias**(\mathbf{b}) values of the neural network as its internal learnable *parameters* which are used in computing the output values and are learned and updated in the direction of optimal solution i.e

minimizing the **Loss** by the network's training process and also play a major role in the **training** process of the Neural Network Model .

The internal parameters of a Model play a very important role in efficiently and effectively training a Model and produce accurate results. This is why we use various Optimization strategies and algorithms to update and calculate appropriate and optimum values of such model's parameters which influence our Model's learning process and the output of a Model.

Types of optimization algorithms ?

Optimization Algorithm falls in 2 major categories -

1. **First Order Optimization Algorithms** — These algorithms minimize or maximize a Loss function $E(\mathbf{x})$ using its **Gradient** values with respect to the parameters. Most widely used First order optimization algorithm is **Gradient Descent**. The First order

derivative tells us whether the function is decreasing or increasing at a particular point. First order Derivative basically give us a **line** which is ***Tangential** to a point on its Error Surface.*

What is a Gradient of a function?

A **Gradient** is simply a vector which is a multi-variable generalization of a **derivative**(dy/dx) which is the instantaneous rate of change of **y with respect to x**. The difference is that to calculate a derivative of a function which is dependent on more than one variable or multiple variables, a **Gradient takes its place. And a gradient is calculated using Partial Derivatives** . Also another major difference between the **Gradient** and a **derivative** is that a **Gradient** of a function produces a **Vector Field**.

A **Gradient** is represented by a **Jacobian** Matrix — which is simply a Matrix consisting of **first order partial Derivatives(Gradients)**.

Hence summing up, a derivative is simply defined for a function dependent on single variables, whereas a Gradient is defined for function dependent on multiple variables. Now let's not get more into Calculus and Physics.

2. Second Order Optimization Algorithms — Second-order methods use the **second order derivative** which is also called **Hessian** to minimize or maximize the **Loss** function. The **Hessian** is a Matrix of ***Second Order Partial Derivatives***. ***Since the second derivative is costly to compute, the second order is not used much***. The second order derivative tells us whether the ***first derivative*** is increasing or decreasing which hints at the function's curvature. Second Order Derivative provide us with a **quadratic** surface which touches the curvature of the **Error Surface**.

Some Advantages of Second Order Optimization over First Order —

Although the Second Order Derivative may be a bit costly to find and calculate, but the advantage of a **Second order**

Optimization Technique is that it does not neglect or ignore the **curvature of Surface**. Secondly, in terms of Step-wise Performance they are better.

You can search more on second order Optimization Algorithms here-<https://web.stanford.edu/class/msande311/lecture13.pdf>

So which Order Optimization Strategy to use ?

1. Now **The First Order Optimization** techniques are easy to compute and less time consuming , converging pretty fast on large data sets.
2. **Second Order Techniques** are faster only when the **Second Order Derivative** is known otherwise, these methods are always slower and costly to compute in terms of both time and memory.

*Although ,sometimes **Newton's Second Order Optimization** technique can sometimes Outperform **First Order Gradient Descent** Techniques because Second Order Techniques will not get stuck around paths of slow convergence around **saddle points** whereas **Gradient Descent** sometimes gets stuck and does not converges.*

Best way to know which one converges fast is to try it out yourself.

Now what are the different types of Optimization Algorithms used in Neural Networks ?

Gradient Descent

Gradient Descent is the most important technique and the foundation of how we train and optimize ***Intelligent Systems***.

What is does is —

“Oh Gradient Descent — Find the Minima ,
control the variance and then update the
Model’s parameters and finally lead us to
Convergence”

$\theta = \theta - \eta \cdot \nabla J(\theta)$ — is the formula of the parameter updates, where
‘ η ’ is the learning rate , ‘ $\nabla J(\theta)$ ’ is the **Gradient** of **Loss**
function- $J(\theta)$ w.r.t *parameters-‘ θ ’*.

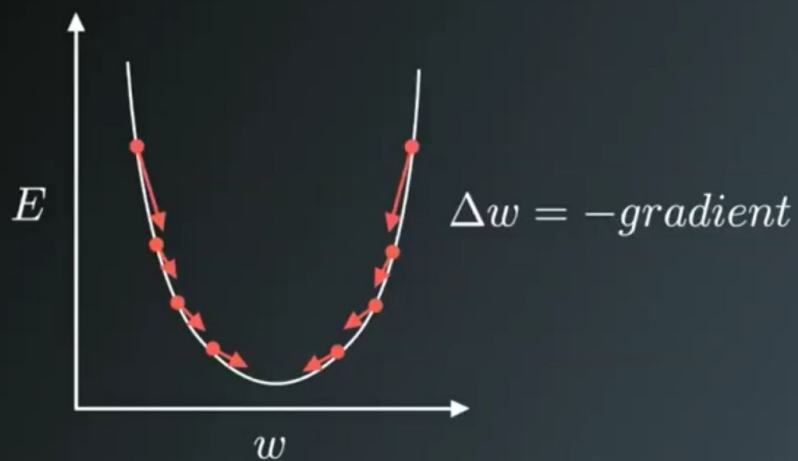
It is the most popular Optimization algorithms used in optimizing
a Neural Network. Now gradient descent is majorly used to do
Weights updates in a Neural Network Model , i.e update and
tune the Model’s parameters in a direction so that we can
minimize the **Loss function**. Now we all know a Neural Network
trains via a famous technique called **Backpropagation** , in
which we first propagate forward calculating the dot product of
Inputs signals and their corresponding Weights and then apply a

activation function to those sum of products, which transforms the input signal to an output signal and also is important to model complex Non-linear functions and introduces **Non-linearities** to the Model which enables the Model to learn almost any *arbitrary functional mappings*.

After this we propagate **backwards** in the Network carrying **Error** terms and updating **Weights** values using *Gradient Descent*, in which we calculate the gradient of **Error(E) function** with respect to the **Weights (W)** or the parameters , and update the parameters (here **Weights**) in the opposite direction of the Gradient of the Loss function w.r.t to the Model's parameters.



$$E = \frac{1}{2}(y - f(\sum w_i x_i))^2$$



Weight updates in the opposite direction of the Gradient

*The image on above shows the process of Weight updates in the opposite direction of the Gradient Vector of Error w.r.t to the Weights of the Network. The **U-Shaped** curve is the Gradient(slope). As one can notice if the Weight(**W**) values are too small or too large then we have large Errors , so want to update and optimize the weights such that it is neither too small nor too large , so we descent downwards opposite to the Gradients until we find a **local minima**.*

Variants of Gradient Descent-

The traditional *Batch Gradient Descent* will calculate the gradient of the whole Data set but *will perform only **one update** , hence it can be very slow and hard to control for datasets which are very very large and don't fit in the Memory.*

How big or small of an update to do is determined by the Learning Rate - η , and it is guaranteed to converge to the **global minimum** for convex error surfaces and to a **local minimum** for non-convex surfaces. *Another thing while using Standard*

batch Gradient descent is that it computes redundant updates for large data sets.

The above problems of Standard Gradient Descent are rectified in Stochastic Gradient Descent.

1. Stochastic gradient descent

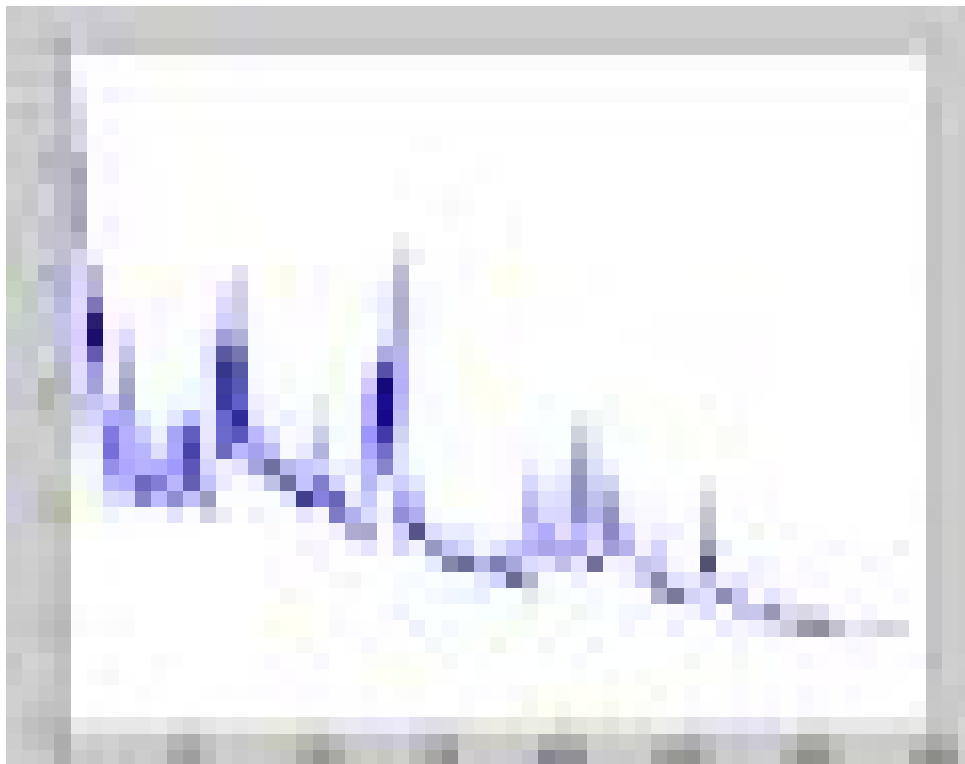
Stochastic Gradient Descent(SGD) on the other hand performs a parameter update for **each training example** .It is usually much faster technique.It performs one update at a time.

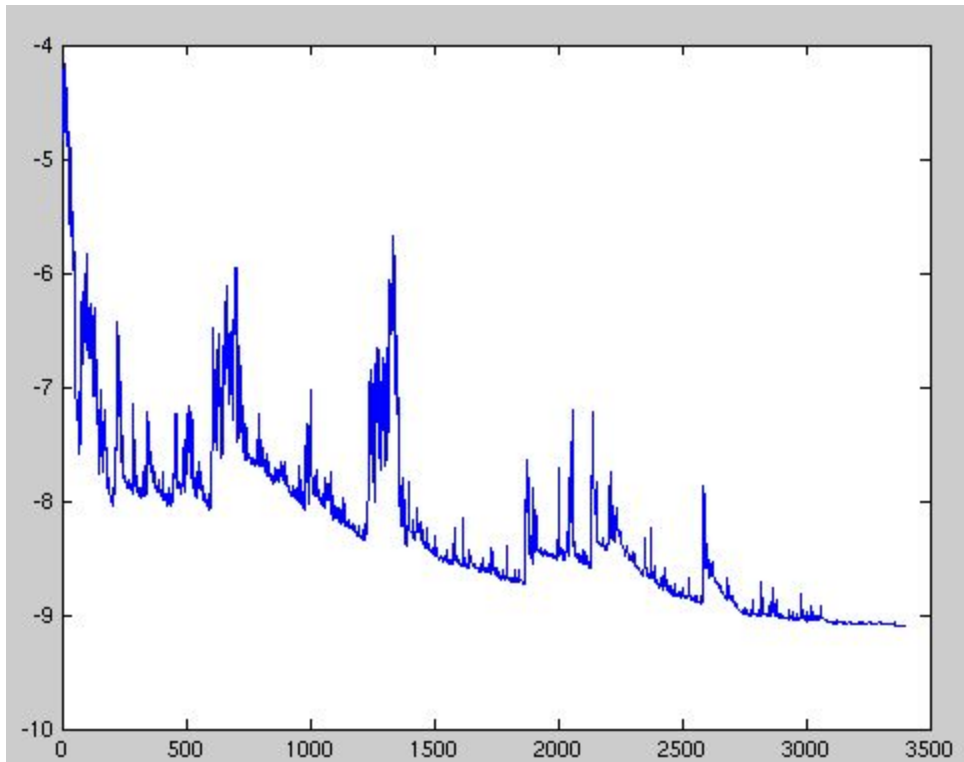
$\theta = \theta - \eta \cdot \nabla J(\theta; x(i); y(i))$, where $\{x(i), y(i)\}$ are the training examples.

*Now due to these **frequent updates** ,parameters updates have **high variance** and causes the **Loss function to fluctuate to different intensities**. This is actually a good thing because it helps us **discover new and possibly better local minima** , whereas Standard Gradient Descent will only converge to the minimum of the basin as mentioned above.*

But the problem with SGD is that due to the frequent updates and fluctuations it ultimately complicates the convergence to the exact minimum and will keep overshooting due to the frequent fluctuations .

Although, it has been shown that as we slowly decrease the learning rate- η , SGD shows the same convergence pattern as Standard gradient descent.





High Variance parameter updates for each training example cause the Loss function to fluctuate heavily due to which we might not get the minimum value of parameter which gives us least Loss value.

The problems of high variance parameter updates and unstable convergence can be rectified in another variant called *Mini-Batch Gradient Descent*.

2. Mini Batch Gradient Descent

An improvement to avoid all the problems and demerits of SGD and standard Gradient Descent would be to use **Mini Batch Gradient Descent** as it takes the best of both techniques and performs an update for every batch with n training examples in each batch.

The advantages of using Mini Batch Gradient Descent are —

1. It Reduces the variance in the parameter updates , which can ultimately lead us to a much better and stable convergence.
2. Can make use of highly optimized matrix optimizations common to state-of-the-art deep learning libraries that make computing the gradient w.r.t. a mini-batch very efficient.
3. Commonly Mini-batch sizes Range from 50 to 256, but can vary as per the application and problem being solved.
4. Mini-batch gradient descent is typically the algorithm of choice when training a neural network nowadays

P.S —Actually the term SGD is used also when mini-batch gradient descent is used .

Challenges faced while using Gradient Descent and its variants —

1. Choosing a proper learning rate can be difficult. A learning rate that is too small leads to painfully *slow convergence* i.e will result in **small** baby steps towards finding optimal parameter values which minimize loss and finding that valley which directly affects the overall training time which gets too large. While a learning rate that is too **large** can hinder convergence and cause the loss function to fluctuate around the minimum or even to diverge.
2. Additionally, the same learning rate applies to all parameter updates. If our data is sparse and our features have very different frequencies, we might not want to update all of them to the same extent, but perform a larger update for rarely occurring features.

3. Another key challenge of minimizing highly non-convex error functions common for neural networks is avoiding getting trapped in their numerous ***sub-optimal local minima***. Actually, Difficulty arises in fact not from local minima but from ***saddle points***, i.e. *points where one dimension slopes up and another slopes down*. These saddle points are usually surrounded by a plateau of the same error, which makes it notoriously hard for SGD to escape, as the gradient is close to zero in all dimensions.

Optimizing the Gradient Descent

Now we will discuss about the various algorithms which are used to further optimize Gradient Descent.

Momentum

The high variance oscillations in SGD makes it hard to reach convergence , so a technique called ***Momentum*** was invented which ***accelerates SGD*** by navigating along the relevant direction and softens the oscillations in irrelevant directions.In

other words all it does is adds a fraction ' γ ' of the update vector of the past step to the current update vector.

$$\mathbf{V}(t) = \gamma \mathbf{V}(t-1) + \eta \nabla J(\theta).$$

and finally we update parameters by $\theta = \theta - \mathbf{V}(t)$.

The momentum term γ is usually set to 0.9 or a similar value.

*Here the **momentum** is same as the momentum in classical physics , as we throw a ball down a hill it gathers momentum and its velocity keeps on increasing.*

The same thing happens with our parameter updates —

1. It leads to faster and stable convergence.
2. Reduced Oscillations

The **momentum** term γ increases for dimensions whose gradients point in the same directions and *reduces updates* for dimensions whose gradients change directions. *This means it*

does parameter updates only for relevant examples. This reduces the unnecessary parameter updates which leads to faster and stable convergence and reduced oscillations.

Nesterov accelerated gradient

A researcher named Yurii Nesterov saw a problem with Momentum —

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.

What actually happens is that as we reach the minima i.e the lowest point on the curve ,the **momentum** is pretty high and it doesn't know to ***slow*** down at that point due to the high momentum *which could cause it to miss the minima entirely and continue moving up. This problem was noticed by Yurii Nesterov.*

He published a research paper in 1983 which solved this problem of momentum and we now call this strategy ***Nesterov Accelerated Gradient***.

In the method he suggested we first make a big jump based on our previous momentum then calculate the Gradient and then make a correction which results in a parameter update. Now this anticipatory update prevents us to go too fast and not miss the minima and makes it more responsive to changes.

Nesterov accelerated gradient (NAG) is a way to give our momentum term this kind of prescience. We know that we will use our momentum term $\gamma \mathbf{V}(\mathbf{t}-1)$ to move the parameters $\boldsymbol{\theta}$. Computing $\boldsymbol{\theta} - \gamma \mathbf{V}(\mathbf{t}-1)$ thus gives us an *approximation of the next position of the parameters which gives us a rough idea where our parameters are going to be*. **We can now effectively look ahead by calculating the gradient not w.r.t. to our current parameters $\boldsymbol{\theta}$ but w.r.t. the approximate future position of our parameters:**

$V(t) = \gamma V(t-1) + \eta \nabla J(\theta - \gamma V(t-1))$ and then update the parameters using $\theta = \theta - V(t)$.

One can refer more on **NAGs** here

<http://cs231n.github.io/neural-networks-3/>.

Now that we are able to adapt our updates to the slope of our error function and speed up SGD in turn, we *would also like to adapt our updates to each individual parameter to perform larger or smaller updates depending on their importance.*

Adagrad

It simply allows the learning Rate $-\eta$ to **adapt** based on the parameters. So it makes big updates for infrequent parameters and small updates for frequent parameters. For this reason, it is well-suited for dealing with sparse data.

It uses a different learning Rate for every parameter θ at a time step based on the past gradients which were computed for that parameter.

Previously, we performed an update for all parameters θ at once as every parameter $\theta(i)$ used the same learning rate η . As ***Adagrad*** uses a different learning rate for every parameter $\theta(i)$ at every time step t , we first show Adagrad's per-parameter update, which we then vectorize. For brevity, we set $\mathbf{g}(t,i)$ to be the ***gradient of the loss function*** w.r.t. to the parameter $\theta(i)$ at time step t .



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

The formula for Parameter updates

***Adagrad* modifies the general learning rate η at each time step t for every parameter $\theta(i)$ based on the past gradients that have been computed for $\theta(i)$.**

*The main benefit of **Adagrad** is that we don't need to manually tune the learning Rate. Most implementations use a default value of 0.01 and leave it at that.*

Disadvantage —

1. Its main weakness is that its learning rate- η is always Decreasing and decaying.

This happens due to the accumulation of each squared Gradients in the denominator , since every added term is positive. The accumulated sum keeps growing during training. This in turn causes the *learning rate to shrink and eventually become so small*, **that the model just stops learning entirely and stops acquiring new additional knowledge.** Because we know that as the learning rate gets smaller and smaller the ability of the Model to learn fastly decreases and

which gives very slow convergence and takes very long to train and learn i.e learning speed suffers and decreases.

This problem of **Decaying learning Rate** is Rectified in another algorithm called **AdaDelta**.

AdaDelta

It is an extension of **AdaGrad** which tends to remove the *decaying learning Rate* problem of it. Instead of accumulating all previous squared gradients, **Adadelta** limits the window of accumulated past gradients to some fixed size **w**.

Instead of inefficiently storing **w** previous squared gradients, the sum of gradients is recursively defined as a *decaying mean* of all past squared gradients. The running average **$E[g^2](t)$** at time step **t** then depends (as a fraction **γ** similarly to the Momentum term) *only on the previous average and the current gradient*.

$E[g^2](t) = \gamma \cdot E[g^2](t-1) + (1-\gamma) \cdot g^2(t)$, We set **$\gamma$** to a similar value as the momentum term, around 0.9.

$$\Delta\theta(t)=-\eta \cdot g(t,i).$$

$$\theta(t+1)=\theta(t)+\Delta\theta(t).$$



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

As the denominator is just the root mean squared (RMS) error criterion of the gradient, we can replace it with the criterion short-hand:

$$\Delta\theta_t = -\frac{\eta}{RMS[g]_t} g_t.$$

The final formula for Parameter Updates

Another thing with AdaDelta is that we don't even need to set a default learning Rate .

What Improvements we have done so far —

1. We are calculating *different learning Rates* for each parameter.
2. We are also calculating *momentum*.
3. Preventing **Vanishing(decaying) learning Rates**.

What more improvements can we do ?

Since we are calculating individual ***learning rates*** for each parameter , why not calculate individual **momentum** changes for each parameter and store them separately. This is where a new modified technique and improvement comes into play called as **Adam**.

Adam

Adam stands for **Adaptive Moment Estimation**. Adaptive Moment Estimation (Adam) is another method that computes

adaptive learning rates for each parameter. In addition to storing an exponentially decaying average of past squared gradients like **AdaDelta**, **Adam** also keeps an exponentially decaying average of past gradients $M(\mathbf{t})$, similar to momentum:

$\mathbf{M}(\mathbf{t})$ and $\mathbf{V}(\mathbf{t})$ are values of the first moment which is the **Mean** and the second moment which is the **uncentered variance** of the *gradients* respectively.

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

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

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

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

The formulas for the first Moment(mean) and the second moment (the variance) of the Gradients

Then the final formula for the Parameter update is —

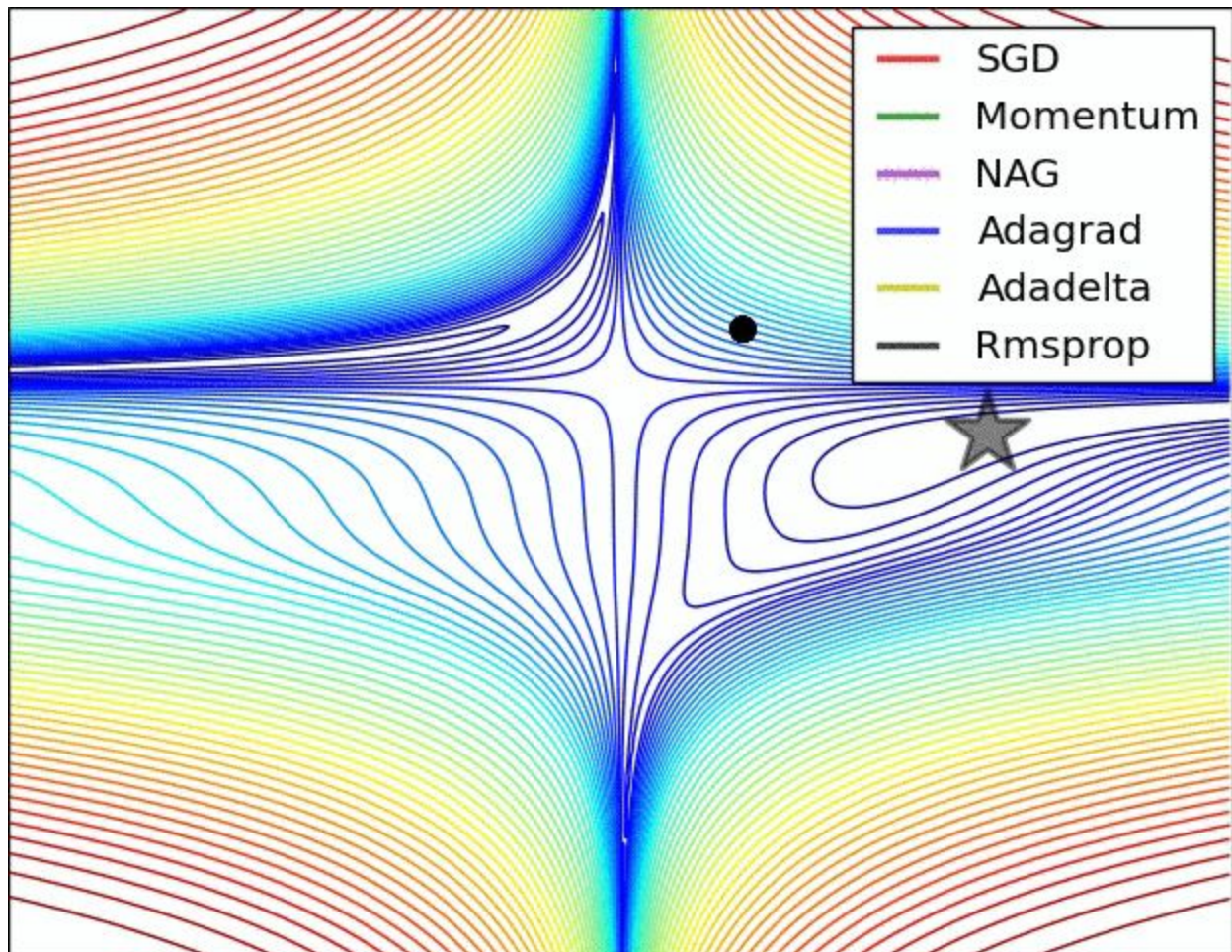
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$$

The values for β_1 is 0.9 , 0.999 for β_2 , and $(10 \times \exp(-8))$ for ϵ .

Adam works well in practice and compares favorably to other adaptive learning-method algorithms as it converges very fast and the learning speed of the Model is quite Fast and efficient and also it rectifies every problem that is faced in other optimization techniques such as **vanishing Learning rate** , **slow convergence** or **High variance in the parameter updates** which leads to **fluctuating Loss function**

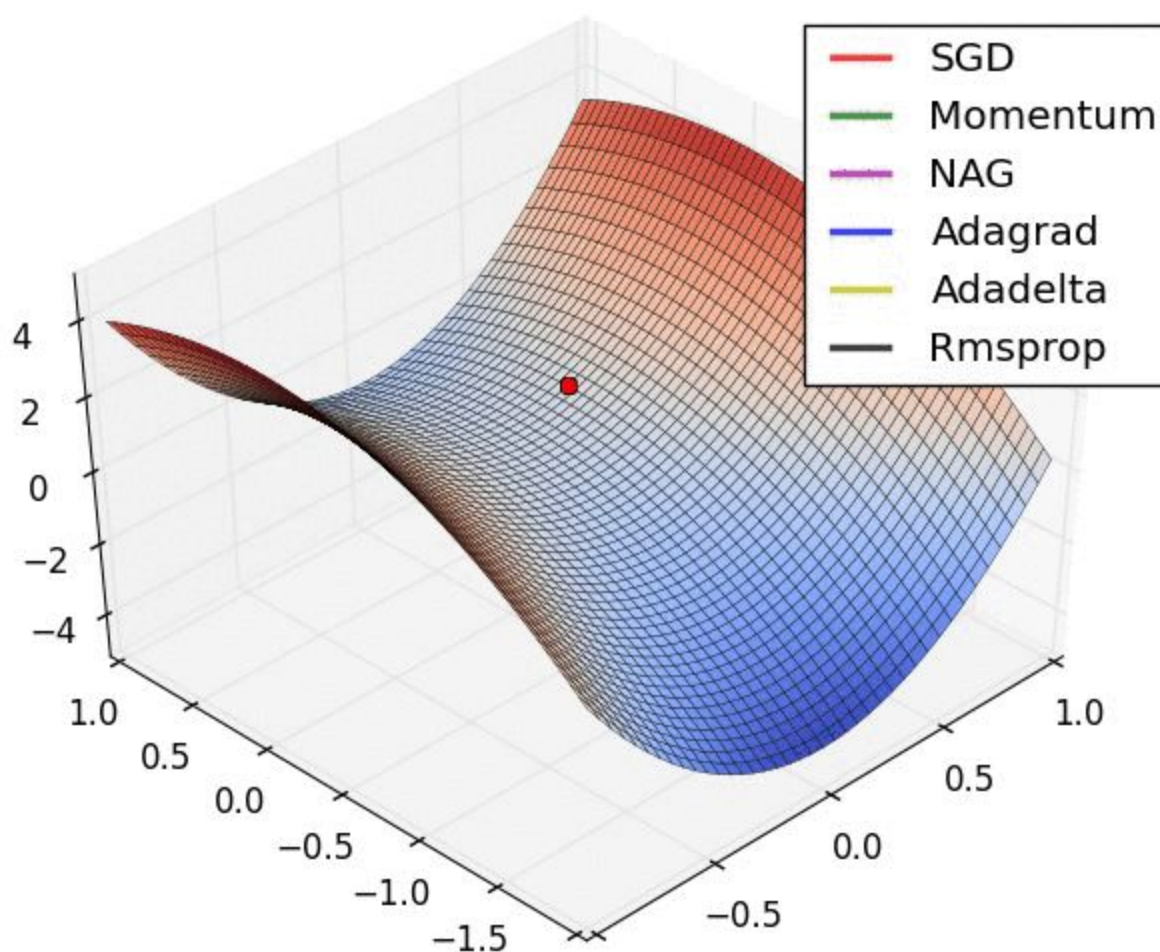
Visualization of the Optimization Algorithms





SGD optimization on loss surface contours





SGD optimization on saddle point

*From the above images one can see that The **Adaptive algorithms converge** very fast and quickly find the right direction in which parameter updates should occur. Whereas standard **SGD** , **NAG** and **momentum** techniques are very slow and could not find the right direction.*

Conclusion

Which optimizer should we use?

The question was to choose the best optimizer for our Neural Network Model in order to converge fast and to learn properly and tune the internal parameters so as to minimize the Loss function .

Adam *works well in practice and outperforms other Adaptive techniques.*

If your input data is sparse then methods such as **SGD, NAG and momentum** are inferior and perform poorly. **For sparse data sets one should use one of the *adaptive learning-rate methods*.** An additional benefit is that we won't need to adjust the learning rate but likely achieve the best results with the default value.

If one wants fast convergence and train a deep Neural Network Model or a highly complex Neural Network then **Adam or any other Adaptive learning rate techniques** should be used because they outperforms every other optimization algorithms.

Understanding Label and OneHot Encoding.

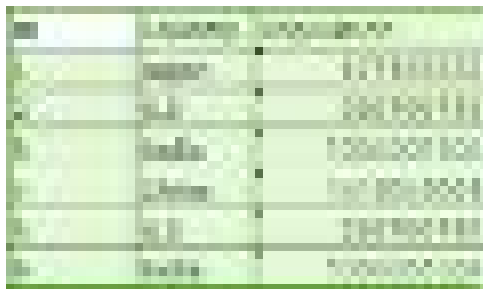
Let us understand the working of Label and One hot encoder and further, we will see how to use these encoders in python and see their impact on predictions

Label Encoder:

Label Encoding in Python can be achieved using Sklearn Library. Sklearn provides a very efficient tool for encoding the levels of categorical features into numeric values. `LabelEncoder` encode labels with a value between 0 and `n_classes-1` where `n` is the

number of distinct labels. If a label repeats it assigns the same value to as assigned earlier.

Consider below example:



ID	Country	Population
1	Japan	127185332
2	U.S	326766748
3	India	1354051854
4	China	1415045928
5	U.S	326766748
6	India	1354051854

ID	Country	Population
1	Japan	127185332
2	U.S	326766748
3	India	1354051854
4	China	1415045928
5	U.S	326766748
6	India	1354051854

If we have to pass this data to the model we need to encode the Country column to its numeric representation by using Label Encoder. After applying Label Encoder we will get a result as seen below

ID	Country	Population
1	0	127185332
2	1	326766748
3	2	1354051854
4	3	1415045928
5	1	326766748
6	2	1354051854

ID	Country	Population
1	0	127185332
2	1	326766748
3	2	1354051854
4	3	1415045928
5	1	326766748
6	2	1354051854

The categorical values have been converted into numeric values.

That's all label encoding is about. But depending on the data, label encoding introduces a new problem. For example, we have encoded a set of country names into numerical data. This is actually categorical data and there is no relation, of any kind, between the rows.

The problem here is since there are different numbers in the same column, the model will misunderstand the data to be in some kind of order, $0 < 1 < 2$.

The model may derive a correlation like as the country number increases the population increases but this clearly may not be the scenario in some other data or the prediction set. To overcome this problem, we use One Hot Encoder.

One Hot Encoder:

Now, as we already discussed, depending on the data we have, we might run into situations where, after label encoding, we might confuse our model into thinking that a column has data with some kind of order or hierarchy when we clearly don't have it. To avoid this, we 'OneHotEncode' that column.

What one hot encoding does is, it takes a column which has categorical data, which has been label encoded and then splits the column into multiple columns. The numbers are replaced by 1s and 0s, depending on which column has what value. In our example, we'll get four new columns, one for each country — Japan, U.S, India, and China.

For rows which have the first column value as Japan, the 'Japan' column will have a '1' and the other three columns will have '0's. Similarly, for rows which have the first column value as the U.S, the 'U.s' column will have a '1' and the other three columns will have '0's and so on.



ID	Country_Japan	Country_U.S	Country_India	Country_China	Population
1	1	0	0	0	127185332
2	0	1	0	0	326766748
3	0	0	1	0	1354051854
4	0	0	0	1	1415045928
5	0	1	0	0	326766748
6	0	0	1	0	1354051854

OneHot encoded country values.