Improving a Neural

Improving a NN

1. Fine tuning Hyper parameters
2. By solving problems
   - Vanishing / Exploding Gradient
   - Not enough data
   - Slow training
   - Overfitting

---

1. Fine tuning Hyper parameters.
   a. No. of Hidden Layers
   b. No. of neurons per layer
   c. learning rate
   d. Optimizer
   e. Batch size
   f. Activation function
   g. Epochs

a. No. of Hidden Layers.

generally, it's better to stop adding more hidden layers when your model does overfitting.

It's also preffered to have more hidden layer than having alot of neurons with just one hidden layer.

∴ a NN with 3 hidden layer (30 neuron each) is better than having a NN with 1 hidden layer having 500 neurons.

This is because the hidden layers (3) will help uncover pattern more effectively. 1st layer detects edges, 2nd the shape, 3rd the detail.

b. No. of Neuron / layer

The simple layer is to have sufficient neurons. We can start with having more neurons, and can remove them based on overfitting or vice versa.

e. Batch size.

There are two opinions on this:
either use (8 - 32) small batch size
or use (≤192 or random) large batch size.

- Having small batch size will give you accurate results but will consume alot of time
- Having a large batch size will give you faster result but has low accuracy.
- Therefore, a better way is to use a technique where learning rate changes w.r.t the epochs. → learning rate scheduler

g. Epochs

Any value is fine.

- Accurate way is to use early stopping method

The main problem is Overfitting.

Possible Solutions:

i) Add more data

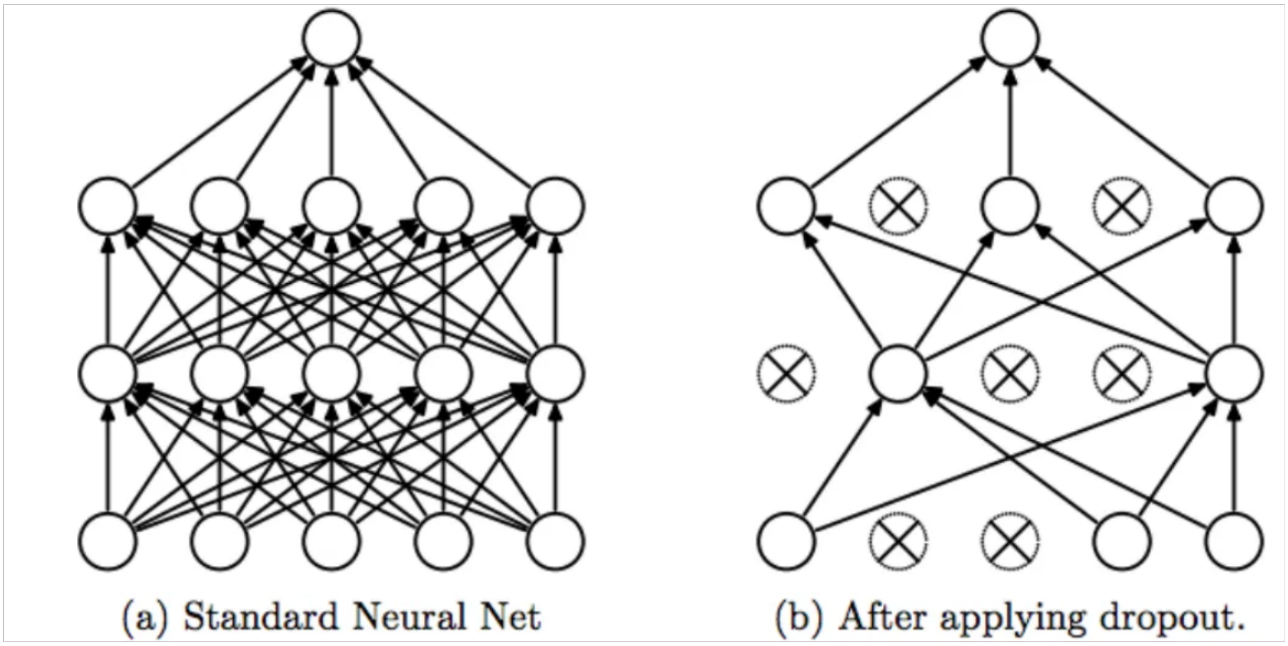
ii) Reduce Complexity of model

iii) Early Stopping

iv) L1 and L2 Regularization

v) Dropout

Dropout:



(a) Standard Neural Net

(b) After applying dropout.

Dropout is a technique that randomly disables (or drop) a fraction of neurons during each training iteration. This prevents the network from becoming too dependent on certain nodes and encourages it to learn more generalized features, which helps it perform better on new data.

1. **During Training:** At each training iteration, dropout randomly disables a fraction (e.g., 50%) of neurons in the network. This effectively creates a new, smaller neural network with fewer neurons for that iteration. As a result, the model learns to work without depending on any one specific neuron, which reduces overfitting.

2. **During Testing:** During testing or inference (when the model is predicting on new data), all neurons are active. However, the weights of the neurons are scaled down by the dropout rate to account for the different training structure. This scaling is typically by a factor of (1 — dropout rate) to ensure the same output range as during training.

↓

While doing prediction, we cannot simply give the same final weight to the variables, this is because, while training the neuron / variable was not present all the time. Therefore, if for ex. the dropout $p = 0.25$, then it means that neuron was not present 25% of the time. So, we use: $w(1-p)$ → probability of being present

Regularization

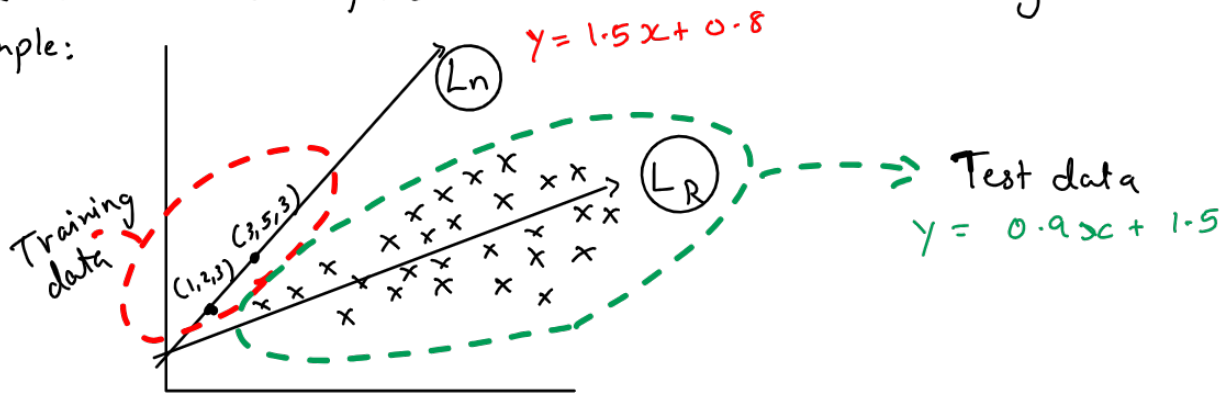
Ridge regularization, also known as L2 regularization, is a technique used in linear regression and other machine learning models to prevent overfitting by adding a penalty to the loss function based on the sum of squares of the model's coefficient.

Here, to reduce the overall loss function, we add one more regularization term along with the loss in the loss function

$$\text{Loss} = \sum_{i=1}^{n} (Y_i - \hat{Y}_i)^2 + \underbrace{\lambda(m^2)}_{\text{extra term (This is a hyperparameter)}}$$

The overall loss function would be decreased by this above eq.

example:



Suppose Ln is the predicted line based on the training data and L_R is the actual line where it is supposed to be.

We can clearly see that there is overfitting, as the model is not performing well on a new data.

Calculating the Loss :

Loss L_N

$\lambda = 1$

$0 + (1.5)^2$

$= \boxed{2.25}$

Loss L_R

$\lambda = 1$

$(2.3 - 0.9 - 1.5)^2$

$+ (5.3 - 2.7 - 1.5)^2$

$+ (0.9)^2$

$= (0.1)^2 + (1.1)^2$

$+ (0.9)^2$

$= \boxed{2.03}$

$\therefore$ The loss function has been reduced.

$y = mx + b$

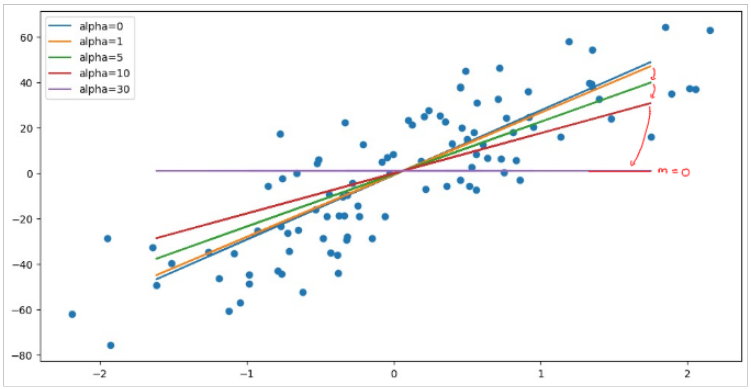
↓

The m is the most important value since, the more m value, the more loss

The main difference between Ridge and Lasso is that in Ridge, even if you max the alpha, coefficient never goes 0. Therefore, the value of coefficients will always be something (>0). But in lasso, the value of when increased, after certain threshold, the value of coefficients(m) will be zero. So it means that the slope has no effect on the overall model.

$$y = \underbrace{mx}_{m=0} + b$$

This will indirectly work as feature selection.



We can notice, on increasing the value of alpha, the slope decreases and after certain alpha, the slope is flat, resulting in underfitting.

L2 regularization works very well in deep learning

Activation Functions

In ANN, each neuron forms a weighted sum of its inputs and passes the resulting scalar value through a function referred to as an activation function or transfer function. If a neuron has n inputs then the output or activation of a neuron is:

$$a = g(w_1x_1 + w_2x_2 + w_3x_3 + \cdots w_nx_n + b)$$

The function g is reffered to as activation func

The simple reason to use a activation function is that it can capture non-linear data. If we dont use activation function, the ANN will behave like a linear model.

The most ideal activation function would be to always have

1. A non linear Activation function
2. Should always be differentiable
3. Computationally inexpensive (derivative calculation should be fast)
4. Zero Centered or Normalized (ex. tanh)

⑤ Non-Saturating

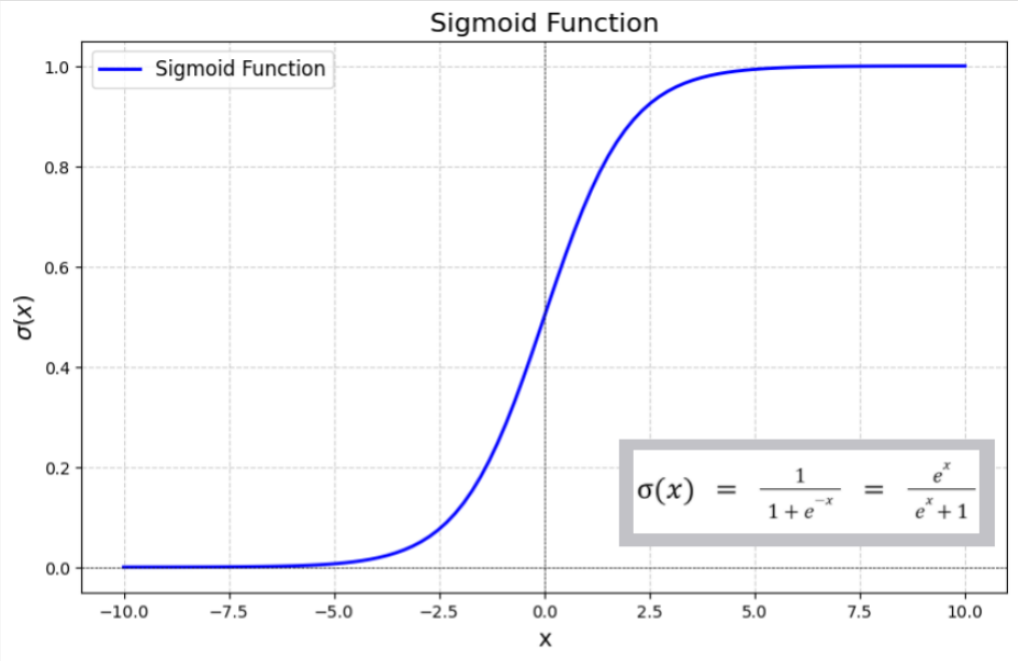
Means it should not squeeze its input in a limited space (ie. 0-1)

Relu is a non-saturating function

Max (0, x)

functions Discussion:

1] Sigmoid AF



Advantages

1) [0,1] → probability → output layer → Binary Classificat

2] Non-Linear function

3] Differentiable function

Disadvantages

1) It is a saturating function

saturating function: A function that squeezes its input space. (like 0-1)

This causes vanishing gradient problem.

In backprop algo, when the gradient space is too small (0.0000 ...) then when we derivate the function, there will be no change so no training.

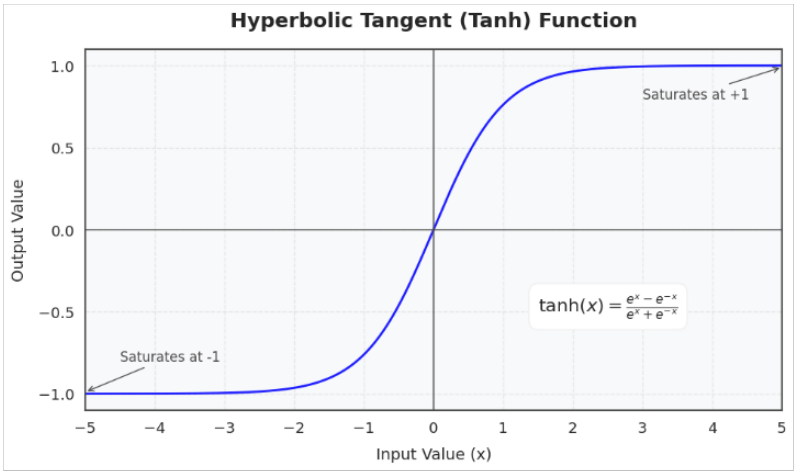
This is the biggest reason that sigmoid AF is almost never used in hidden layers.

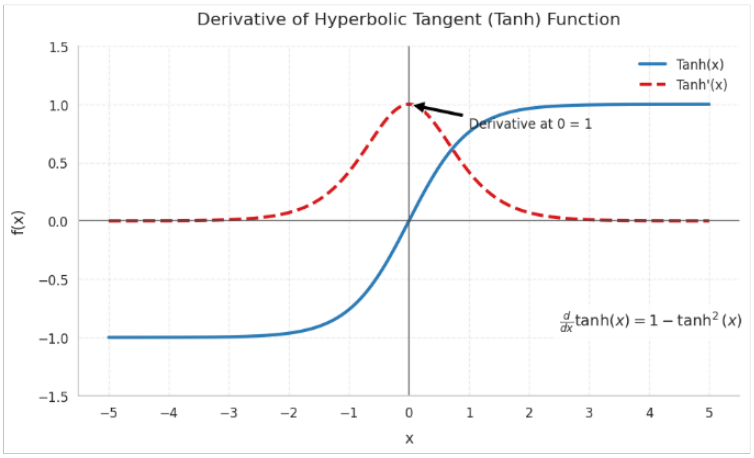
2) Non zero centered AF : Slow training

derivatives are only +ve or only -ve due to which its slow

3) Computationally expensive : hard to calculate derivative due to the presence of exponents.

2] Tanh AF





Advantages

1] Non-linear AF

2] Differentiable

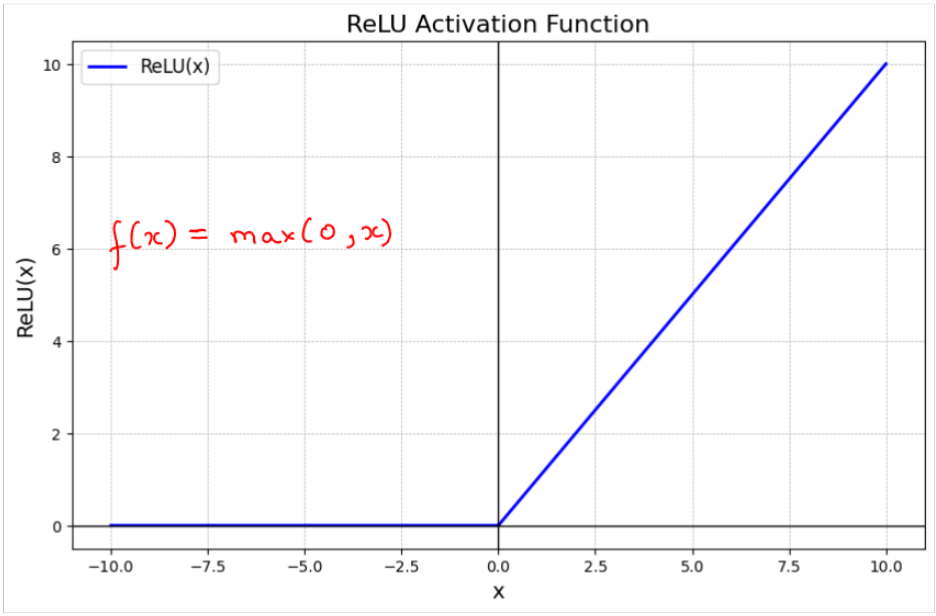
3] Zero-centered : gradient will be both +ve & -ve that leads to faster training.

Disadvantages

1] Saturating function → Vanishing Gradient problem

2] Computationally expensive → slow.

3] ReLu AF



Advantages

1) Non Linear AF : Because it's max(0, x)

2] Not saturated in the +ve region: no limit

3] Computationally inexpensive

4] Convergence is faster than sigmoid and tanh.

Disadvantages

1) Not completely differentiable at zero

2) Not zero centered → To solve this, we use Batch normalization

Problems in RELU and its variants.

Relu : Rectified Linear Unit.

The "dying Relu" problem (also called as dead relu) is when ReLU neurons end up outputting 0 for essentially all inputs and stop updating during training. Because their output is always 0, their gradient is also 0, so those neurons never change and effectively drop out of the model's capacity to learn.

Why it happens?

1) ReLU is $\max(0, x)$, so any sufficiently -ve pre-activation (x) gives output 0 and gradient 0.

2) High learning rates can push weights so far that a neuron's pre-activation becomes -ve for all training inputs, so it stays at 0 forever.

3) Large -ve biases

4) As networks get deeper, the probability that many ReLU units fall into this inactive regime increases; theory shows deep ReLU nets "die" in probability as depth goes to infinity.

Solutions

1). Set low learning rates.

2) Set bias to be +ve value : 0.01 is a proven value to avoid dying ReLU

3) Use different variations of ReLU.

We know that the main problem with ReLU is that as soon as it becomes -ve, its derivatives become -ve and we use the derivative in update rule and the whole term becomes 0.

Different Variants :

linear Variants and Non linear variants

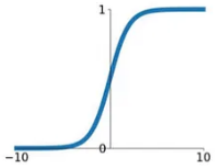
- Leaky ReLU
- Parametric ReLU

- Exponential Linear Unit (ELU)
- Scaled exponential linear unit (SeLU)

Activation Functions

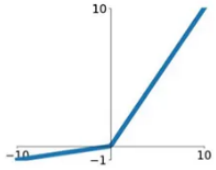
Sigmoid

$\sigma(x) = \frac{1}{1+e^{-x}}$



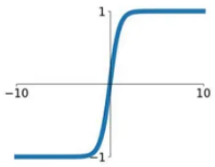
Leaky ReLU

$\max(0.1x, x)$



tanh

$\tanh(x)$

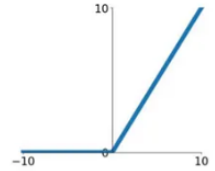


Maxout

$\max(w_1^T x + b_1, w_2^T x + b_2)$

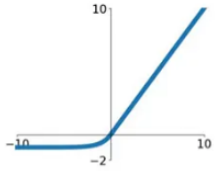
ReLU

$\max(0, x)$

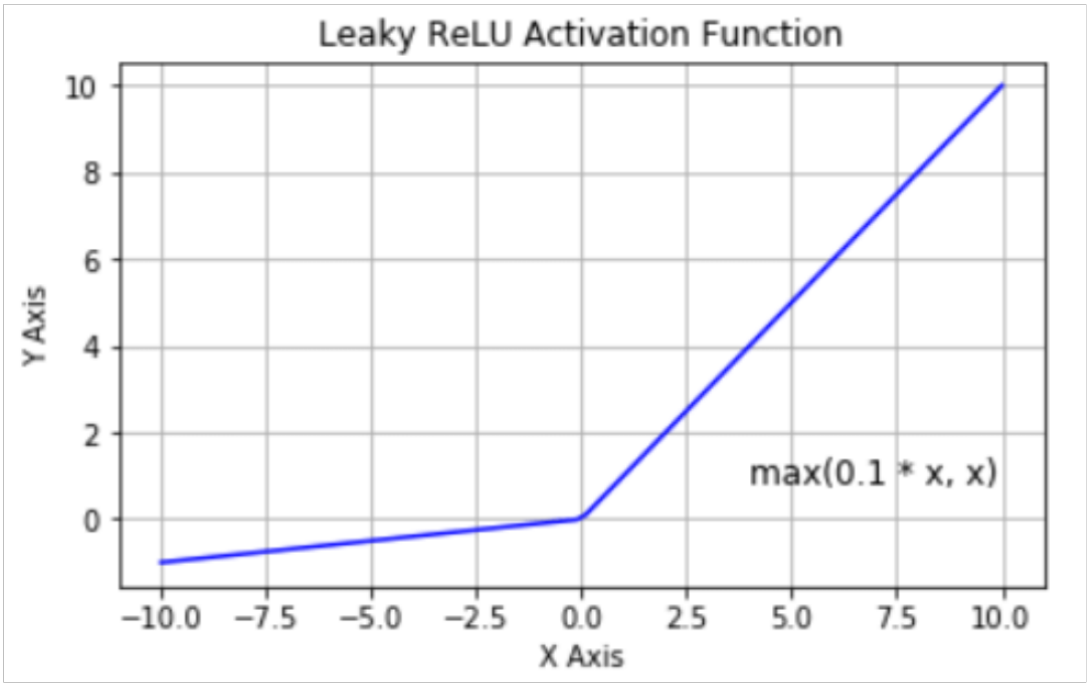


ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Leaky ReLU

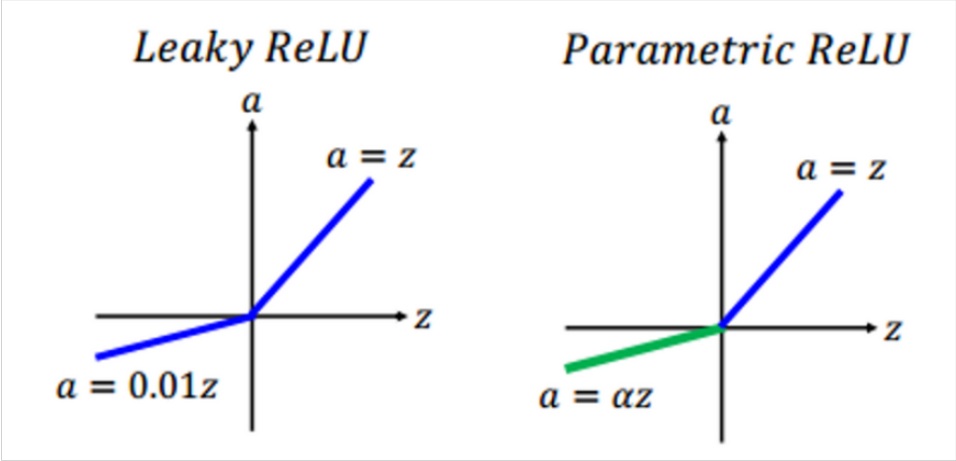


This is a modified version of ReLU where it keeps a small, non-zero slope for -ve inputs instead of outputting exactly 0.

Advantages

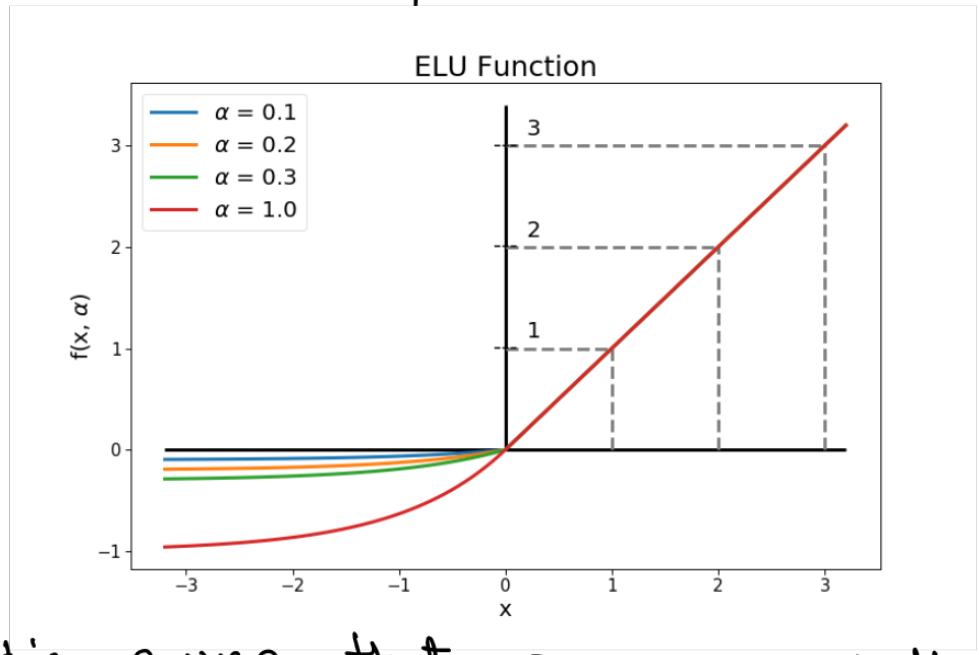
1) Non-saturated → Unbounded (no limits)
2) Easily Computed
3) No dying ReLU
4) Close to 0 centered.

2) Parametric ReLU



Everything is exactly same as Leaky ReLU except that here instead of using 0.01 (from Leaky ReLU) we can change the value.

3) ELU : Exponential Linear unit.



$$ELU$$

$$\begin{cases} x & for\ x \geqslant 0 \\ \alpha(e^x - 1) & for\ x < 0 \end{cases}$$

It's proven that ELU gives better results than ReLU because it has no boundary on -ve side aswell.

Advantages

1) Zero centered : faster convergence.

2) Better generalized Results.

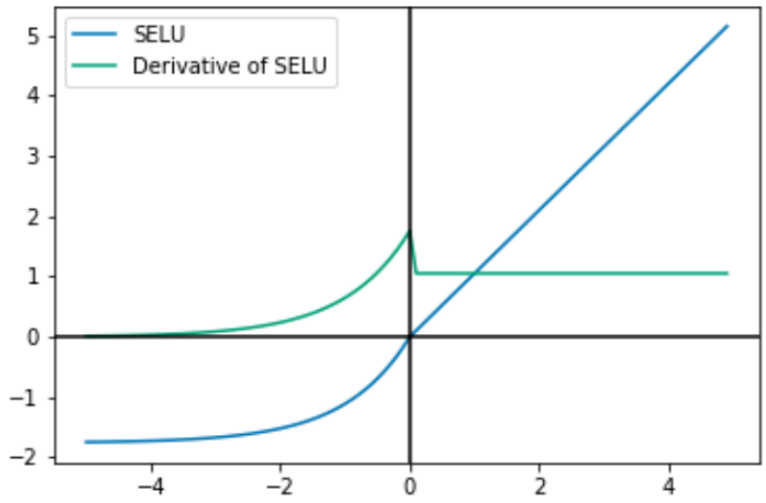
3) Always Continous as well as differentiable

4) Not a Dying ReLU problem

Disadvantages

1) Computationally expensive because of exponent calculation but since, its zero centered, convergence reaches faster.

4) SeLU - Scaled Exponential Linear Unit



$$\text{selu}(x) = \lambda \begin{cases} x & \text{if } x > 0 \\ \alpha e^x - \alpha & \text{if } x \leqslant 0 \end{cases}.$$

This just have an extra λ that helps the network self-normalize.

Values of α and λ are hard coded:

```
alpha = 1.6732632423543772848170429916717
scale = 1.0507009873554804934193349852946
```

This is new recent publish, so there is less adoption yet.

Advantage:

It is self normalize : the output from this activation func is that it is normalized (m=0, σ=1) therefore, it reaches convergence faster.

Weight Initialization Techniques

Weight initialization is one of the most important things to keep in mind while training a model.

What not to do?

Case 1: Zero Initialization

lets take 3 different AF:

ReLU
Tanh
Sigmoid.

} We are taking one AF at a time and applying it to 1st hidden layer

→ for ReLU, when we initialize the weights as 0, during forward propagation, because the AF is $\max(0, x)$, the output will be 0 due to which during back propagation, the weights will stay the same.

$$w_{new} = w_{old} - \eta \frac{\partial L}{\partial w_{old}} \quad (0)$$

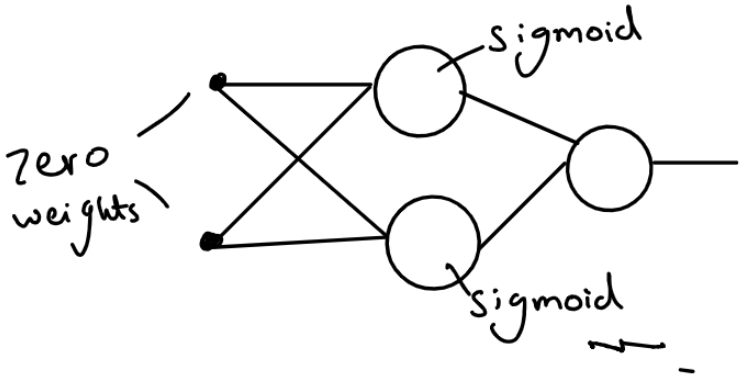
$w_{new} = w_{old}$ → equal and 0

Because of this, the model will stop training.

Same case goes for tanh function

Sigmoid AF:

At value 0, the value of AF will be 0.5



When we back propegate, there are alot of places where the same derivatives are used which will result in having the same output. So both weights will increase together.

So, no matter the no. of neurons in 1st hidden layer, if the AF is sigmoid and weights are 0, then the entire hidden layer will behave as a single perceptron. Therefore it is simply linear model

This happens because $w=0$ then every neuron outputs 0.5 and hence all have same weights.

Case 2: Non zero Constant Value.

If we use a same value (lets say 0.7), this will behave exactly same as what happens with sigmoid and zero weight combination. So, it will act as a linear model.

Case 3: Random Initialization

When we use very very small random (0.0001 etc) value initialization, tanh and sigmoid will tend to have vanishing gradient problem and ReLU will have only one issue which will be : extremely slow convergence.

When we use big numbers (0-1)
In case of tanh / Sigmoid, :

- saturation happens : - Slow convergence
  - Vanishing Gradient.

for ReLU, gradient's are unstable

In summary, we cannot do these :

→ zero initialization
→ non zero constant values
→ extremely small values (0.0001)
→ very large values (0-1)

What techniques are used for weight initialization? (Solutions)

i) Xavier / Glorant Initialization
ii) He Initialization

} i) normal type
ii) uniform type

Both of these techniques set initial weights to maintain consistent variance of activations and gradients across layers, preventing vanishing / exploding gradients.

1) Xavier initialization

Used with tanh / sigmoid AF.

Xavier initialization is designed to keep the variance of activations and gradients roughly the same across all layers during forward and backward propagation. This prevents vanishing or exploding gradients, especially in deep networks.

If weights are too large, activations explode, too small, activations vanish. Therefore, Xavier finds a "sweet spot" that maintains the flow in the network.

$$W \sim N\left(0, \sqrt{\frac{2}{n_{in} + n_{out}}}\right) \rightarrow \text{Xavier normal}$$

$$\text{Mean} = 0, \quad \text{Variance} = \frac{2}{n_{in} + n_{out}}$$

n_{in} = no. of input units

n_{out} = no. of output units

Xavier Uniform :

$$W \sim N\left(-\sqrt{\frac{6}{n_{in} + n_{out}}}, \sqrt{\frac{6}{n_{in} + n_{out}}}\right)$$

Difference: Uniform tends to have bounded values while normal can ocassionally produce larger outliers.

The simple intuition is that if we have :

Very large weights: Actual values gets distorted
- Network cant learn

Very small weights : Signal gets lost in noise
- Network cant learn

(Signal means actual values)

Using these techniques: Signal passes through cleanly by using $1/\sqrt{n}$ scaling.
- Network learns

Step by Step explaination

1. Random weights are assigned (from normal or uniform distribution)

This distribution depends on what type of intitialization for Xavier is used. (uniform or normal)

2. Multiplies them by $1/\sqrt{n}$ to shrink to correct range

3. Done.

Here, $1/\sqrt{n}$ is just for understanding purpose. actual formulas are used here.

Formula	What it means
1/√n	Simple intuition: scale down by number of inputs
2/(n_in + n_out)	Xavier's actual formula: balances forward & backward, accounts for tanh/sigmoid
2/n_in	He's actual formula: focuses on forward pass, accounts for ReLU

2) He Initialization

Every thing is same as Xavier/glorant initialization except the formola and where it's used with:

He is used with ReLU

Normal → $W \sim N\left(0, \sqrt{\frac{2}{n_{in}}}\right)$

Uniform → $W \sim U\left(-\sqrt{\frac{6}{n_{in}}}, \sqrt{\frac{6}{n_{in}}}\right)$

n_in = no. of input connections to a neuron (incoming)

n_out = no. of output connections to a neuron (outgoing)

The reason we han n_in and n_out is because n_in is used during forward propagation while during back propagation, gradient flows backward, so we have n_out.

Xavier is designed to handle both in and out because tanh/sigmoid have complex gradients.

while He is designed to only have in (pass) because He it uses ReLU's gradients which are simple and forward pass is more critical.

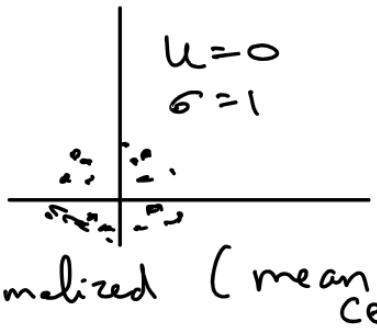
Batch Normalization

Batch Normalization (BN) is an algorithmic method which makes the training of Deep neural networks (DNN) faster and more stable.

It consists of normalizing activation vectors from hidden layers using the mean and variance of the current batch. This normalization step is applied right before (or right after) the non linear function.

Why use Batch normalization?

1) When your data is unnormalised, there are higher chances that while finding minima during gradient descent, it can shoot out. And Because of this shootout prevention, we have to use small learning rate which eventually makes the overall training slow.



2) Reduces Internal Covariate shift

Covariate Shift : It occurs when the distribution of input features (x) changes between training and test data, but the relationship between inputs and outputs stays the same

Example : You train a model to recognize cats using indoor photos but test it on outdoor photos. The lighting, backgrounds and colors are different (input distribution changed) but a cat is still a cat (underlying relationship is the same.)

This makes the model perform poorly because it has not seen data from the new distribution during training.

Internal Covariate Shift

It is the same concept but happening inside the network between layers during trainings.

As the network trains and weights update, the distribution of inputs to each layer keeps changing. For example :

- Layer 2 recieves input from Layer 1
- As Layer 1's weights update, the distribution of its outputs changes
- This means Layer 2 is constantly trying to learn from a "moving target"

Why is this a problem?

- Each layer must continuously adapt to new input distributions
- Slows down training
- Forces you to use smaller learning rate
- Makes the network harder to train.

How batch normalization helps : By normalizing inputs at each layer, it reduces how much distributions shift around (makes sure that the data stays normalized : $\mu = 0$, $\sigma = 1$), giving each layer a more stable foundation to learn from. This makes the layer focus on learning the actual patterns rather than constantly adapting to distribution changes.

How BN Works? (during training)

We use mini-Batch Gradient descent.

① firstly, a batch of data points is forward propagated, and it's mean and variance is computed for that Batch data points.

$$u = \frac{1}{m}\sum_{i=1}^{m} z_{11}^{i} \quad , \quad \sigma = \sqrt{\frac{1}{m}\sum_{i=1}^{m}(z_{11}^{i} - u_B)}$$

This calculation of u and σ is always done for each neuron seperately.

② Once that's done, we normalize

$$z^{i} = \frac{z^{i} - u_B}{\sigma_B + (\epsilon)}$$

→ The epsilon is a small constant added for numerical stability to prevent zero.

③ Scale and Shift

$$y_i = \gamma \hat{x_i} + \beta$$

Here γ (gamma) and β (Beta) are learnable parameters (internally managed by libraries) that allow the network to undo the normalization if needed. This is crucial because sometimes the optimal representation is not always zero mean and unit-variance (normalized: $u=0$, $\sigma=1$).
Sometimes, the model does not want us to normalize each layer's output which is why, if the model want to have another distribution, it can scale it using these learnable parameters)

During Test :

Since while training, we were using batches and normalizing the neuron's output using the batch's mean and variance, but while testing, there is only one data point at a time, so we cant calculate u and σ?

This is where the concept of Exponentially weighted moving average comes. (EWMA)

During training, we forward propagate batches. The model's internal system maintains the EWMA using the means of each batches $u_{11}, u_{22} \dots u_{nn}$.

Same goes for σ (variance). each batch has its different σ and u, and by this, EWMA is maintained.

Now, when its time for training, we send out one data point at a time, the last value for the EWMA mean and EWMA variance is used this time as μ and σ for that data point

Therefore, while training a model, the model maintains a total of 4 parameters values. $\gamma, \beta, \mu, \sigma$

γ, β: learnable parameters

μ, σ: non learnable parameters

Advantages of BN

1. Training is more stable which enables us to select a better range of hyperparameters
2. faster training time ⟶ higher learning rate
3. Acts as a Regularizer ⟶ (not that strong, but a side +ve impact)
4. Reduces the impact of Batch Normalization.
5. Faster Convergence

Disadvantages

1. Batch size dependency : Performance degrades with small batches
2. Not suitable for RNN's
3. Adds Complexity : Extra hyperparameters and computational cost per layer
4. Memory Overhead : Requires additional parameters to maintain such as $\gamma, \beta, \mu, \sigma$, EWMA
5. Training/inference discrepancy : Uses batch statistics during training but running averages during inference, causing potential behaviour changes.

Optimizers

Optimizers are algorithm that adjust the parameters (weights and biases) of NN during training to minimize the loss function.

The most common is Gradient Descent.

GD has three types:

Stochastic GD
Batch GD
Mini-Batch GD

There are other optimizers as well that are in use nowadays due to the following problems with GD.

1) We have to find the most optimal learning rate
   - To solve this, we have learning rate scheduler.
2) The other problem because of constant η is that no matter the dimensions of data, the η will be same for all directions
3) local minima problem.
4) Saddle point problem.

In order to solve all of these issues, we have new optimizers.

① Momentum ② AdaGrad ③ NAG ④ RMS prop ⑤ Adam

Before that, we need to learn more about Exponential weighted moving average (EWMA)

Exponential Weighted Moving Average (EWMA)

EMWA is a smoothing technique that gives more weight to recent data points while still considering older ones, with weights decreasing exponentially as you go back in time.

$$V_t = \beta V_{t-1} + (1-\beta) y_t$$ — also called θ_T

V_t is the EWMA at time t

β is decay parameter ($0.9 - 0.999$)

y_t is actual value at time t

V_{t-1} is the previous EWMA

Mathematical Intuition:

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

$V_0 = 0$,

$\rightarrow V_1 = (1-\beta)\theta_1$

$\rightarrow V_2 = \beta V_1 + (1-\beta)\theta_2$

$= \beta(1-\beta)\theta_1 + (1-\beta)\theta_2$ (Replacing value of V_1 from upper equation)

$\rightarrow V_3 = \beta V_2 + (1-\beta)\theta_3$

$= \beta((1-\beta)\theta_1 + (1-\beta)\theta_2) + (1-\beta)\theta_3$

$= \beta(1-\beta\theta_1) + \beta(1-\beta)\theta_2 + (1-\beta)\theta_3$

$\rightarrow V_4 = \beta V_3 + (1-\beta)\theta_4$

$= \beta^3(1-\beta)\theta_1 + \beta^2(1-\beta)\theta_2 + \beta(1-\beta)\theta_3 + (1-\beta)\theta_4$

$= (1-\beta)[\beta^3\theta_1 + \beta^2\theta_2 + \beta\theta_3 + \theta_4]$

Notice here, that θ_1 value is multiplied by β^3 while θ_2 with β^2 and so on. and the optimal value in DL for β is in 0.9 range. So, $(0.9)^3 < (0.9)^2$

Therefore, the older points gets less importance.

① SGD in Momentum

SGD in momentum is a optimization technique that accelerates gradient descent by accumulating velocity in directions of persistent gradients, helping escape local minima and smooth out oscillations.

So, the main intuition is that when several consecutive gradients point in roughly the same direction, the momentum accumulates and says "If we have been consistently going right for a while now so lets build up speed in that direction.

The key insight: Instead of taking a step based purely on the current gradient, you are taking a step based on a weighted average of recent gradients.
So if the last 10 gradients all pointed right, your velocity vector builds up strongly to right, and you take a much larger effective step.

Vanilla SGD: $\boxed{w = w - \eta\, dw}$ (dw: The raw gradient)

SGD Momentum: $V_t = \beta(V_{t-1}) + (1-\beta)\Delta J(\theta_t)$

$\therefore \boxed{w = w - \eta V_t}$

V_t is literally a weighted sum of all past gradients with exponentially decaying weights

Summary

The problem: Vanilla SGD is slow and zigzags especially in ravines where gradients oscillate.

The Solution: Build up "velocity" by remembering recent gradients - go faster in consistent directions, smooth out oscillations.

The process: i) See which direction the loss function says to go (Compute gradient)
Here we actually compute gradient first and then based on that update velocity!
ii) Mix that direction with your accumulated momentum
iii) Move in momentum direction (update parameter using velocity not raw gradient)

Benefit: faster convergence, smoother trainings, handles noisy gradients better.
The only problem with SGD Momentum is "Overshooting at Minima"

Nesterov Accelerated Gradient (NAG)

NAG is a smarter version of momentum that "Looks Ahead" before computing the gradient. It's like anticipating where you are going instead of blindly following momentum.

Simplified intuition

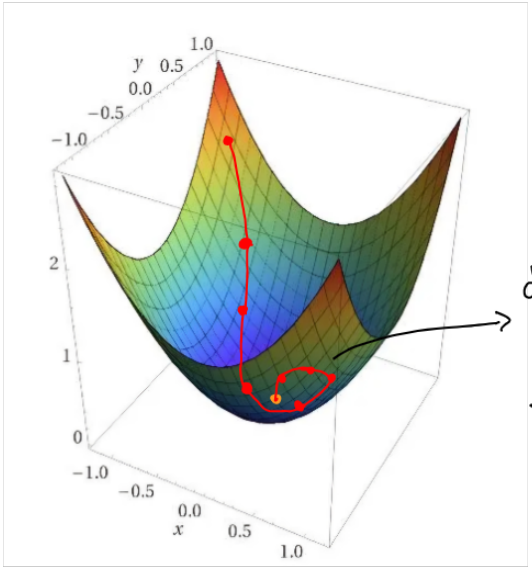
In SGD momentum, the main problem is that it goes a bit ahead of minima and again comes back and again to end up at the exact minima

This happens because total calculation has two components:

momentum + gradient (raw)

↓

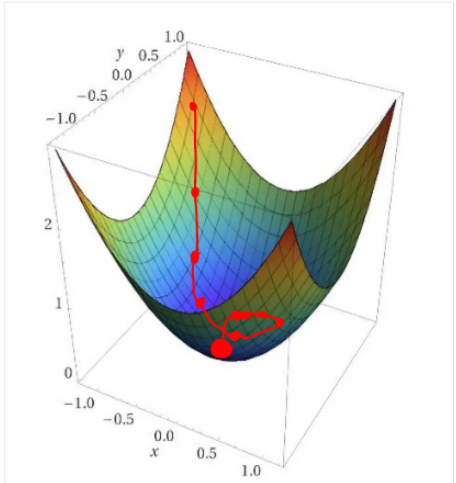
SGD momentum combines both of these to take next jump

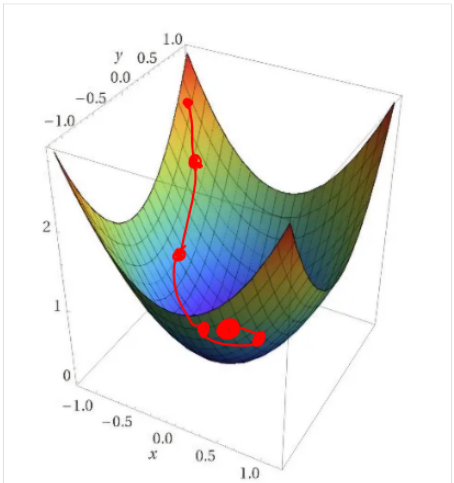


does not directly go to the minima, instead takes a little longer route.

What NAG does is that it only calculates the momentum and jumps and after reaching there it calculates the gradient. So, it will have like a dampning effect on the overall gradient.

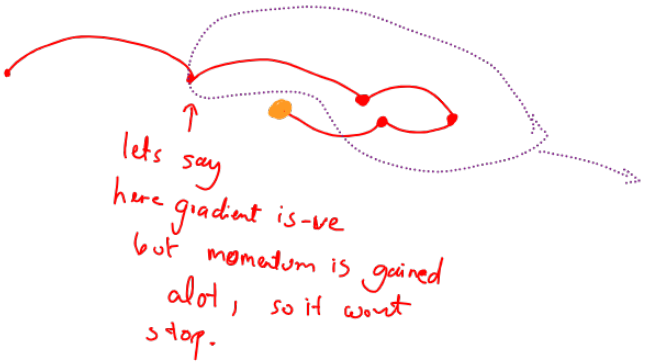
So, Since NAG calculates gradient after landing, what happens is that that gradient might change its direction backwards which will give more influence.





So, if we consider SGD Momentum, here's what exactly happens.

The velocity is calculated by adding up the momentum and the gradient together.



If we do the same for NAG, the overall distance will be less because gradient is calculated later on which damps the impact a little.

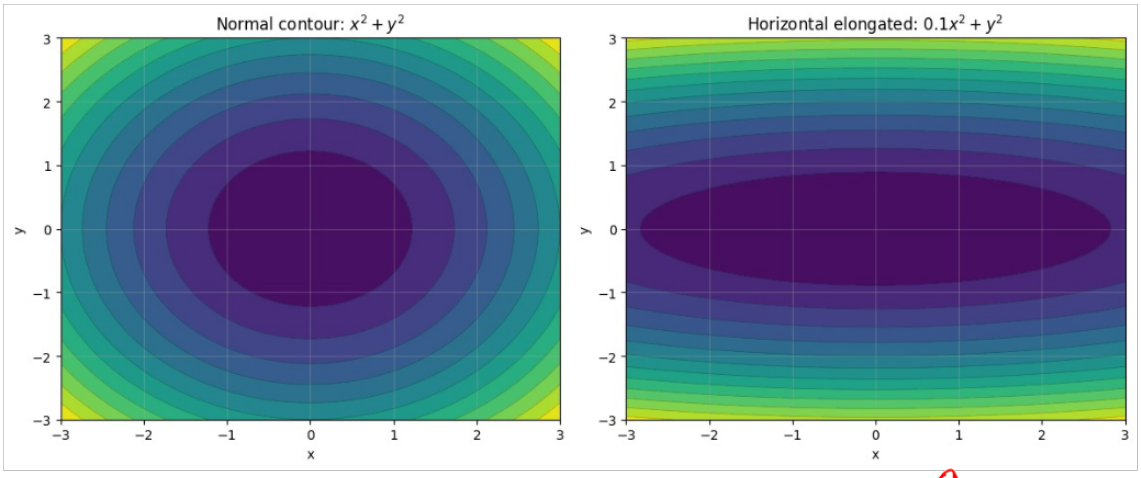
Ada Grad – Adaptive Gradient

Adagrad is an optimization algorithm that adapts the learning rate for each parameter individually based on the historical magnitudes of gradients for that parameter. Parameters with large cumilative gradients get smaller learning rates, while parameters with small cumilative gradients get larger learning rate.

The core problem it solves: There are times when your dataset has different scales for each parameters. In SGD, all parameters share the same learning rates which is inefficient when:

i) features have different scales
ii) Some features are sparse (rarely)
iii) Some parameters need frequent large updates others need infrequent small updates.

Solution: Give each parameter its own adaptive learning rate that automatically adjusts based on gradient history.



This is where you use adagrad.

Initialize:

$$G_0 = 0 \text{ (for each parameter)}$$

For each iteration t:

1. **Compute gradient:**

$$\nabla J(\theta_t)$$

2. **Accumulate squared gradients:**

$$G_t = G_{t-1} + (\nabla J(\theta_t))^2$$

(element-wise squaring and addition)

3. **Update parameters with adaptive learning rate:**

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{G_t + \epsilon}} \odot \nabla J(\theta_t)$$

Where:

- α = base learning rate (e.g., 0.01)
- G_t = sum of squared gradients up to time t
- ϵ = small constant (e.g., 10^{-8}) to avoid division by zero
- $\odot$ = element-wise multiplication

When to use?

- Sparse data (when rare features needs attention, but is filled mostly with 0 (we needs))
- Diff feature scales
- Convex problems

When not to use? Biggest disadvantage

- Not to use in deep neural networks (learning rate dies -> gets stuck)
- Non-convex optimization
- long training

RMS Prop

RMS prop : Root Mean Squared propagation

→ This is simply an improvement on Adagrad.

RMS prop is an adaptive learning rate optimizer that fixes Adagrad's dying learning rate problem by using an exponentially weighted moving average (EWMA) of squared gradients instead of accumulating all past gradients. This allows recent gradients to have more importance while old gradients are gradually forgotten, enabling the learning rate to stabilize rather than decay to zero.

Adagrad's Problem : (no forgetting) $v_t = v_{t-1} (\Delta J_i)^2 \longrightarrow \infty$

- Accumulates all past gradients equally.
- V only grows, never shrinks.
- Old gradients from iteration 1 still affect 10000 iteration

RMSprop (with forgetting) :

$$v_t = \beta v_{t-1} + (1-\beta)(\nabla J)^2$$

- Uses EWMA
- Recent gradients weighted more heavily
- Old gradients exponentially decay away.

(Note that there is nothing like momentum in adagrad or Rmsprop)

∴ AdaGrad : $G = G + (\nabla J)^2$ → Remembers everything forever

∴ RMSprop : $G = \beta G + (1-\beta)(\nabla J)^2$ → forgets old gradients.

Pros :
- Learning rate does not die
- Handles different feature scales
- Suitable for CNN's, RNN, and deep Networks.

Cons:
There is no major disadvantage of RMSprop at all! but still used a little less because Adam gives a little better Results.

Adam – Adaptive Moment Estimation

Adam is an optimization algorithm that combines momentum and RMSprop to get the best of both worlds. It maintains two exponentially decaying averages: the first moment (mean of gradients) for directions like momentum, and the second moment (variance of gradients) for adaptive learning rates like RMSprop. Adam also includes bias correction to fix initialization issues in early iterations.

Adam = Momentum + RMSprop + Bias Correction

- **From Momentum:** Smooth gradient direction, accelerate in consistent directions
- **From RMSprop:** Adaptive per-parameter learning rates, stable long-term training
- **Innovation:** Bias correction for proper scaling from the start

Initialize:

$m_0 = 0, v_0 = 0, t = 0$

Each iteration:

1. Compute gradient: $\nabla J(\theta_t)$

2. Update 1st moment: $m_t = \beta_1 \cdot m_{t-1} + (1-\beta_1) \cdot \nabla J(\theta_t)$

3. Update 2nd moment: $v_t = \beta_2 \cdot v_{t-1} + (1-\beta_2) \cdot (\nabla J(\theta_t))^2$

4. Bias correction: $\hat{m}_t = m_t/(1-\beta_1^t)$, $\hat{v}_t = v_t/(1-\beta_2^t)$

5. Update parameters: $\theta_{t+1} = \theta_t - \alpha \cdot \hat{m}_t/(\sqrt{\hat{v}_t} + \varepsilon)$

pros:
- Best of both world → combines momentum (direction) + RMSprop (adaptive LR)
- default hyper parameters works very well (less need for tuning)
- Bias correction → Proper scaling from 1st iteration

Cons:
- Can sometimes find very sharp minima → Overfitting
- Requires more memory
- Not always optimal

There is also a similar version of Adam called Nadam which is a combination of NAG and adam
- Slightly faster convergence, less overshooting
- Slightly more complex, minimal improvement over adam in many cases