Week-3

Variants of Gradient Descent

What is the need for advanced optimization?

- Slow conversion rate
- Gradient gets stuck in the local optima

Gradient descent optimization algorithms

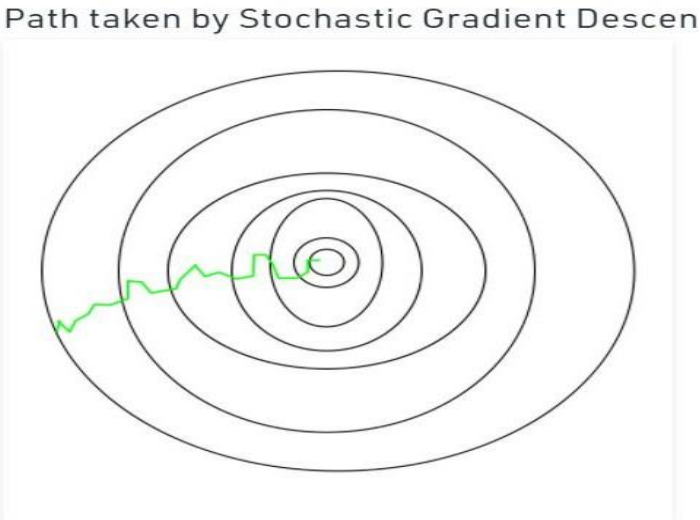
- Gradient descent
- Stochastic Gradient Descent
- Mini Batch Gradient descent
- Gradient descent Momentum
- RMS prop
- Adam

In this course, we will be discussing RMSprop and Adam

Stochastic Gradient Descent(SGD)

- Stochastic Gradient descent is an optimization algorithm used in Machine Learning.
- During the training period to find the derivative loss function random data point is selected instead of whole data for each iteration.

  For Example:- A dataset consists of 1000 data points and to calculate the derivative loss function it will be considering only one data point at a time.
- In SGD convergence to global minima happens very slowly as it will take a single record in each iteration during forward and backward propagation.

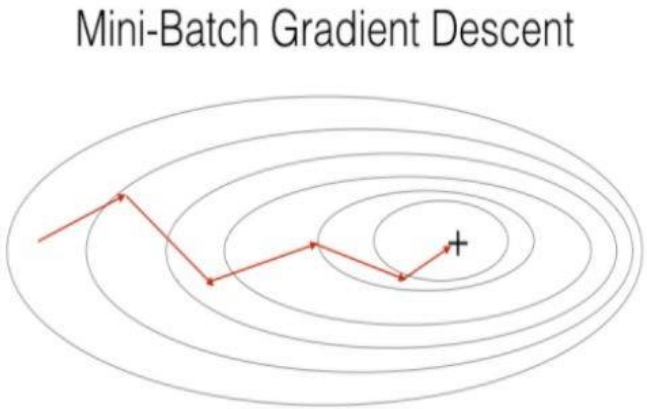


Mini Batch Gradient Descent

- Mini Batch Gradient descent is a variant of gradient descent. During the training process here a batch of K data points are taken to calculate the derivative loss function.

  For Example:- A dataset consists of 1000 data points to calculate the derivative loss function. It will be considering K=100 data point at a time. For each iteration, it will take 100 data points and calculate the loss function.

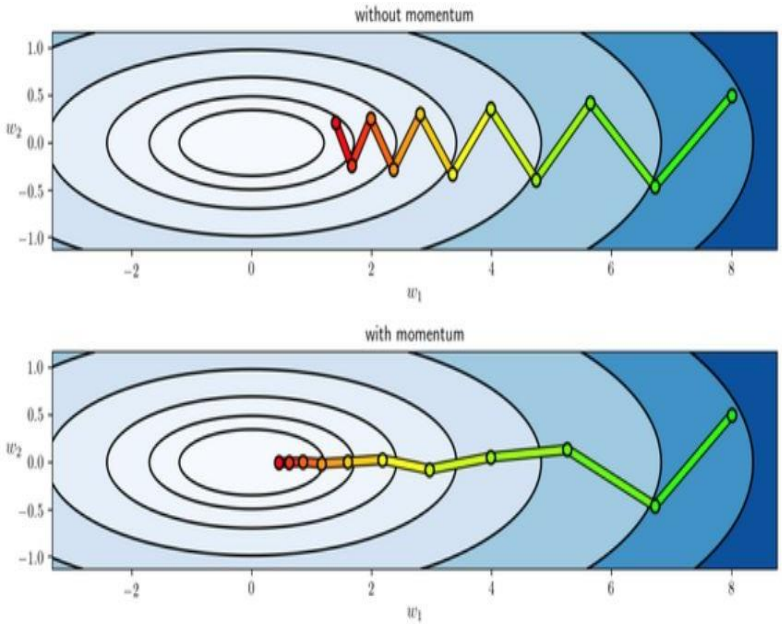
- Compare to SGD, Mini Batch Gradient descent will converge fastly to global minima. Due to Batch-wise convergence, it will produce noise in its path to global minima.



Gradient Descent with Momentum

- To overcome the noisy data produced by the Mini batch Gradient descent there is another variant of Gradient Descent called Gradient descent with momentum which will smoothen the noisy data.

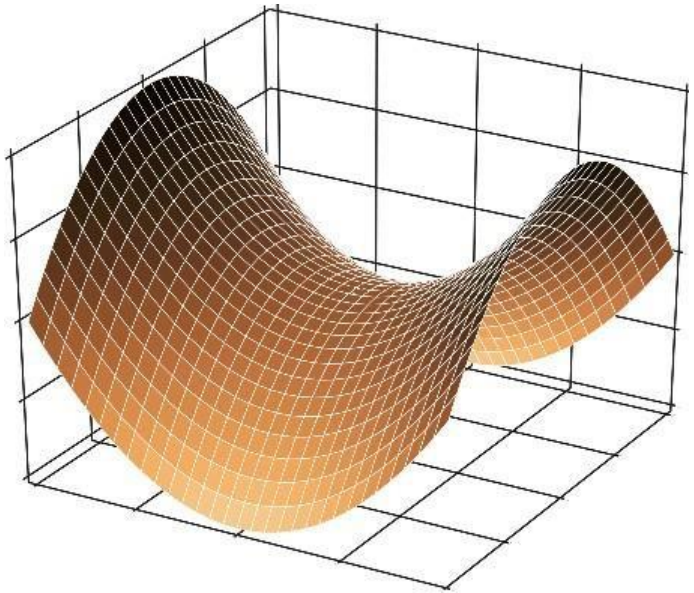
- Gradient Descent with Momentum uses exponentially weighted averages of gradients over the previous iteration to stabilize the convergence.



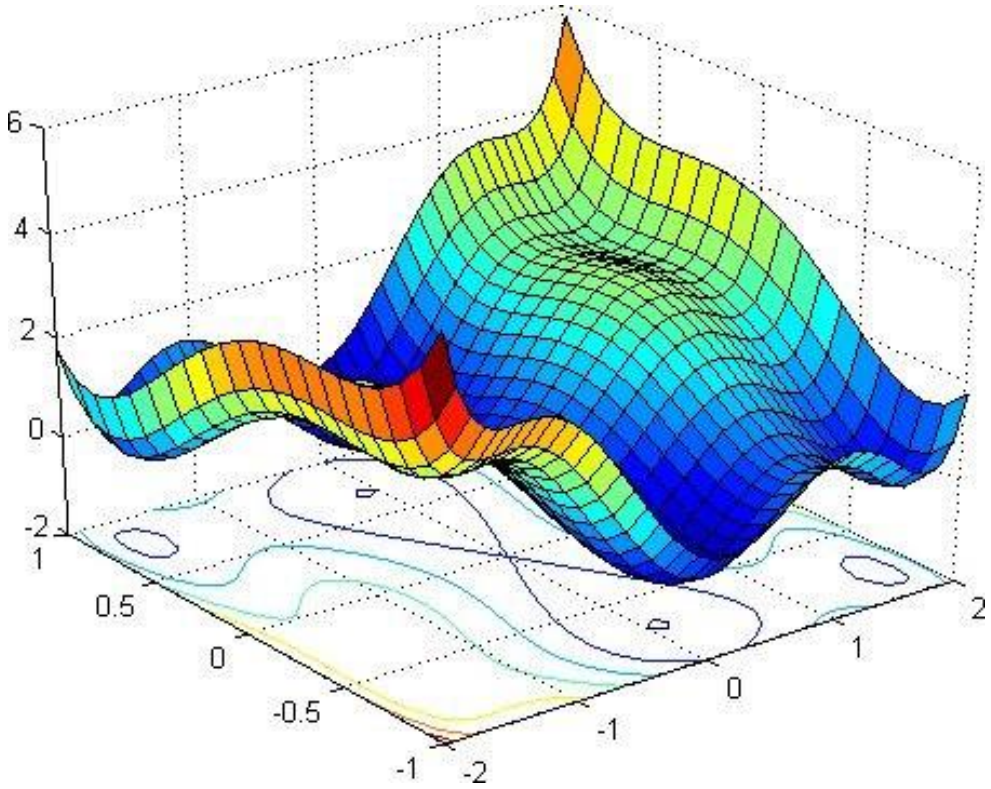
The Problem of local optimum and saddle point

Saddle points, Hessian and long local furrows

- In multiple dimensions:
  - Some weights may have reached a local minima while others have not
  - Some weights may have almost zero gradient



Complicated loss functions



Saddle points, Hessian and long local furrows

- This all is characterized by a second derivative matrix called the Hessian, which is difficult to compute and invert
- Different techniques try to approximate Hessian computation and inversion
- Usually, they treat each weight independently and set learning rates for each differently
- Examples: RMSprop, ADAM, Nesterov momentum

RMSProp

RMSprop

- RMSprop optimization is termed as Root Mean Squared propagation optimization technique.
- It is a gradient based optimization technique used in training a neural network.
- Gradients of very complex functions like neural networks have a tendency to either vanish or explode as the data propagates through the function.
- RMSprop deals with this issue by using a moving average of squared gradients to normalize the gradient.
- This balances the step size, decreasing the step for large gradients to avoid exploding and increasing the step for small gradients to avoid vanishing.
- RMSprop is preferred over gradient descent because it has a very good convergence speed towards the minima of the cost function.

How it works

- Here Vdw and Vdb are the exponential averages of squares of gradients.
- β is a hyperparameter and W is the weight term
- dw2 and db2 are the second-order derivative of the cost function with respect to the weight and bias respectively.
- α is the learning rate. In RMSprop learning rate is not a hyperparameter. It is managed by the algorithm itself, instead of being tuned.
- ε epsilon is the additive term. It just ensures that the denominator does not get equal to zero when Vdb or Vdw becomes equal to zero. Ideally, its value is 10-8.

$$v_{dw} = \beta \cdot v_{dw} + (1 - \beta) \cdot dw^2$$

$$v_{db} = \beta \cdot v_{dw} + (1 - \beta) \cdot db^2$$

$$W = W - \alpha \cdot \frac{dw}{\sqrt{v_{dw}} + \epsilon}$$

$$b = b - \alpha \cdot \frac{db}{\sqrt{v_{db}} + \epsilon}$$

- Ideally, dw is small and db is high. Hence step size of w increases(divided by a smaller number) and that of b decreases(divided by a larger number). Due to this the convergence in the direction of weight increases in RMSprop.

Estimating moments of the gradient

- In stochastic gradient descent with momentum, we compute the first moment of the gradient (g_t) given as m_t in the below diagram.
- In RMSprop we compute the second moment of the derivative. It is given as V_t in the diagram.
- β_1 and β_2 are tunable hyperparameters of the algorithm.

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

Here m_t is the first moment(mean) of the gradient.
While V_t is the second moment of the gradient.

Why RMSprop enhances convergence speed.

- The generic problem in traditional optimization algorithms like gradient descent is the speed of convergence. The gradient in the direction of weight is low and in the direction of bias is high.
- Let us understand this in the dimensions of weights and bias of gradient descent. In an ideal scenario, we need a high convergence speed in the direction of weight(or horizontal direction) and a slow speed in the direction of bias(vertical direction)In gradient descent.
- In GD we have uncontrolled steps in both directions. This hinders the freedom of choosing a bigger learning rate(Due to the risk of overshooting minima) while attempting to speed up the convergence.
- While using RMSprop we dampen the movement in the direction of bias and parallelly speed up the same in the direction of weight. This helps to solve the problem in two ways -
    - It speeds up the convergence in the direction of weight.
    - It slows down the convergence speed in the direction of bias.

Adam

Adam

- Adam can be defined as the merger of RMS prop and SGD with momentum.
- Like RMSprop, it utilizes the squared gradients to scale the learning rate, and similarly, like SGD with momentum, it uses the moving average of the gradient as an alternative to the gradient itself.
- Adam utilizes the estimations of the first and second moments of the gradient to adapt the learning rate for each weight in the neural network.
- The first moment is the mean and the second moment is not centered variance, ie. During the calculation of variance, we don’t subtract the mean.

Adam(Continued)

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

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

m_t and v_t are estimates of first and second moment respectively

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

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

$\hat{m}_t$ and $\hat{v}_t$ are bias corrected estimates of first and second moment respectively

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

ADAM optimization technique

- Adam optimizer is termed as Adaptive moment estimation optimizer.
- Algorithms like Stochastic gradient descent with momentum, RMSprop, and Adam's optimizer are similar in performance but Adam has a slight edge over the others due to the bias correction performed in Adam.
- In reality, Adam optimizer is a combination of Stochastic gradient descent with momentum and the RMSprop. In Adam, the learning rate becomes a tunable hyperparameter while in RMSprop it is an adaptive parameter.
- Adam optimizer can be explained in below three steps:-
  - Estimating moments of the gradient
  - Bias correction step
  - Updating the weights

Bias correction step

- In Adam's optimization, the most crucial step is the bias correction part. In RMSprop we initiate the momentum terms(mt and vt)with 0 which is a bias. In Adam optimization, the bias correction step makes it independent of the initial value.

- The below steps are called bias correction steps.

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

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

- The ideal value of $\beta 1$ is 0.9 while that of $\beta 2$ is 0.999.

Updating the weights.

- Now finally the weights are updated as follows -

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

In practice:

- Begin with SGD with vanilla momentum
- Using Adam is a good choice in many cases

What is Weight Initialization

Weight Initialization

What is Weight Initialization?

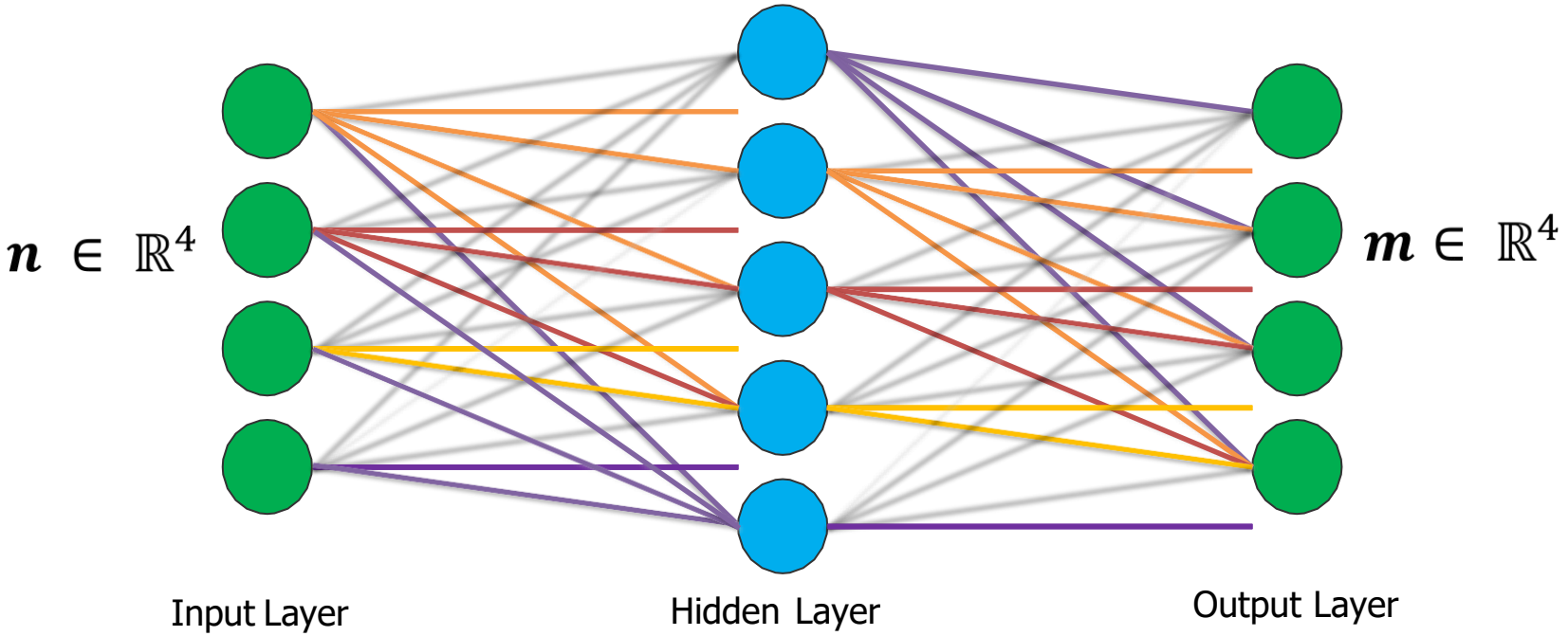
- Weight Initialization is a procedure to assign weights to a neural network with some small random values during the training process in a neural network model.

Why do we need Weight Initialization?

- The purpose of using the weight initialization technique is to prevent the neural network from exploding or vanishing gradient during the forward and backward propagation. If either occurs, the neural network will take a longer time to converge.

What happens when W=0 initialization is used?

Total Parameters = 29



Different Cases

1. **Initialize all weights with 0** - Your network will not learn as all the weights are the same.

2. **Initialize with random numbers** - Works okay for small networks (similar to our two-layer MNIST classifier), but it may lead to distributions of the activations that are not homogeneous across the layers of the network.

1. Initializing all weights to 0

- This makes your model equivalent to a linear model.
- When you set all weights to 0, the derivative with respect to loss function is the same for every w in every layer, thus, all the weights have the same values in the subsequent iteration.
- This makes the hidden units symmetric and continues for all the n iterations you run.
- Thus setting weights to zero makes your network no better than a linear model.

Food for thought: What if we were to initialize all weights to a random constant, ex 10?

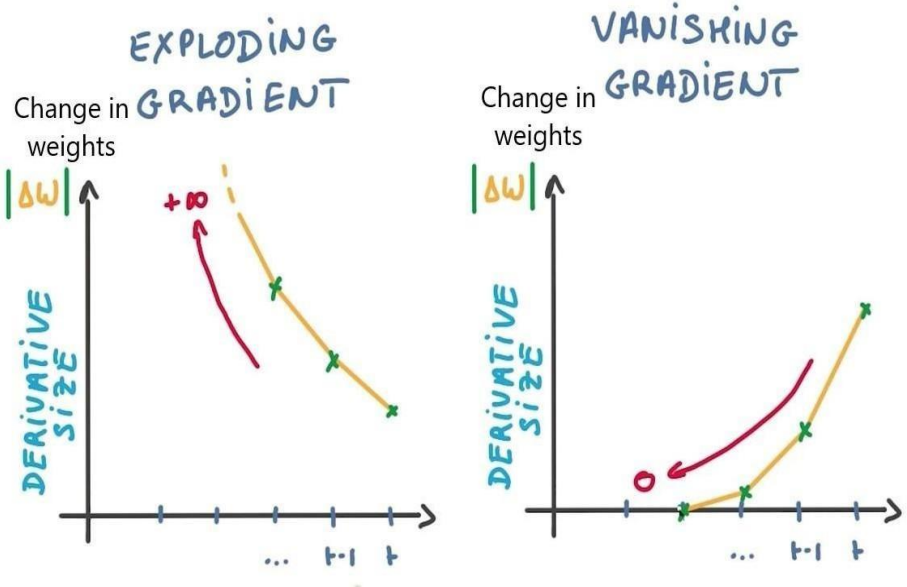
2. Initializing weights randomly

Initializing weights randomly, following standard normal distribution (np.random.randn(n, n-1) in Python) while working with a (deep) neural network can potentially lead to 2 issues — vanishing gradients or exploding gradients.

Vanishing Gradient Descent

Vanishing gradients – In the case of deep networks, for any activation function, *abs(dW)* will get smaller and smaller as we go backward with every layer during backpropagation. The earlier layers are the slowest to train in such a case.

The weight update is minor and results in slower convergence. This makes the optimization of the loss function slow. In the worst case, this may completely stop the neural network from training further.

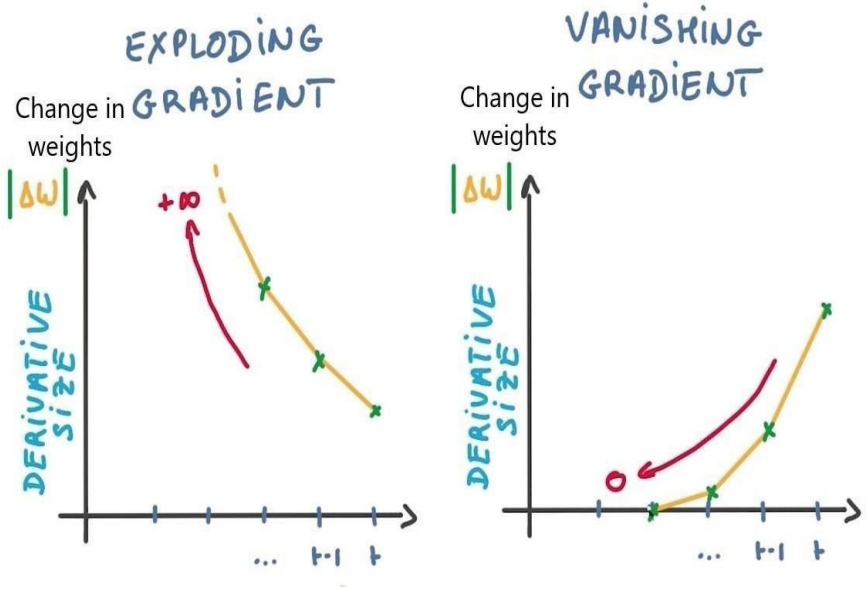


Exploding Gradient Descent

Exploding gradients – This is the exact opposite of vanishing gradients. Consider you have non-negative and large weights and small activations A (as can be the case for sigmoid(z)).

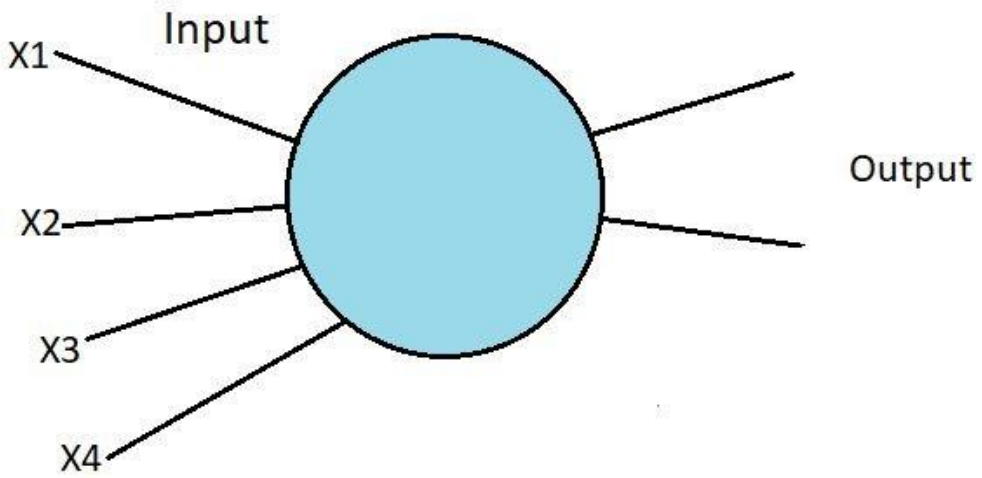
This may result in oscillating around the minima or even overshooting the optimum again and again and the model will never learn!

Another impact of exploding gradients is that huge values of the gradients may cause number overflow resulting in incorrect computations or introductions of NaN's. This might also lead to the loss of taking the value NaN.



Weight Initialization Techniques

Weight Initialization Techniques



Fan_in - No: of Inputs to the neuron
Fan_out - No: of Outputs from the neuron

1. **Random Uniform**

$$W_{ij} \sim U\left[\underbrace{\frac{-1}{\sqrt{fan\_in}}}_{a}, \underbrace{\frac{1}{\sqrt{fan\_in}}}_{b}\right]$$

U- Uniform Distribution
N- Normal Distribution
a- Upper Limit
b- Lower Limit
W_{ij}- Weight

2. Xavier Initialization - we want to initialize the weights with random values that are not “too small” and not “too large.”

Xavier Normal

$$W_{ij} \sim N(0, \sigma)$$

$$\sigma = \sqrt{\frac{2}{fan\_in + fan\_out}}$$

Xavier Uniform

$$W_{ij} \sim U\left[\underbrace{-\sqrt{\frac{6}{fan\_it + fan\_out}}}_{a}, \underbrace{\sqrt{\frac{6}{fan\_in + fan\_out}}}_{b}\right]$$

3. HE Initialization

HE- Normal

$$W_{ij} \sim N(o, \sigma)$$

$$\sigma = \sqrt{\frac{2}{fan\_in}}$$

HE - Uniform

$$W_{ij} \sim U\left[\underbrace{-\sqrt{\frac{6}{fan\_in}}}_{a}, \underbrace{\sqrt{\frac{6}{fan\_in}}}_{b}\right]$$

Best Practices

Which is Good?

Weight initialization techniques for different activation functions:

Activation Function	Weight Initialization Technique
Relu, Leaky Relu	HE Initialization
Sigmoid, TanH	Xavier Initialization

Proper Initialization is an active area of research

Understanding the difficulty of training deep feedforward neural networks
by Glorot and Bengio, 2010

Exact solutions to the nonlinear dynamics of learning in deep linear neural networks
by Saxe et al, 2013

Delving deep into rectifiers: Surpassing human-levll performance on ImageNet classification
by He et al., 2015

Data-dependent Initializations of Convolutional Neural Networks
by Krähenbühl et al., 2015

All you need is a good init
by Mishkin and Matas, 2015

Centered Weight Normalization in Accelerating Training of Deep Neural Networks
by Huang et al., 2017

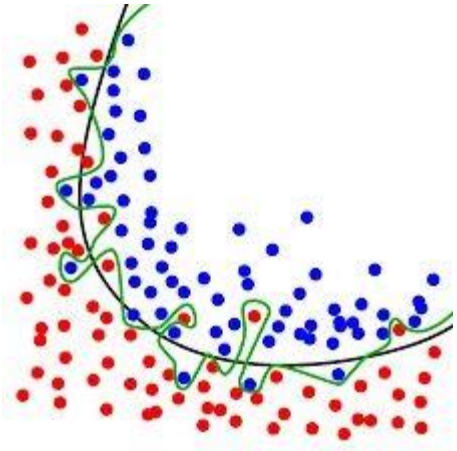
Adjusting for Dropout Variance in Batch Normalization and Weight Initialization: by Hendrycks et al., 2017

Overfitting in Neural Network

Overfitting in ANN

Neural networks are prone to overfitting because of the larger number of parameters.

ANN is able to model higher order and complex functions which makes it more prone to overfitting .



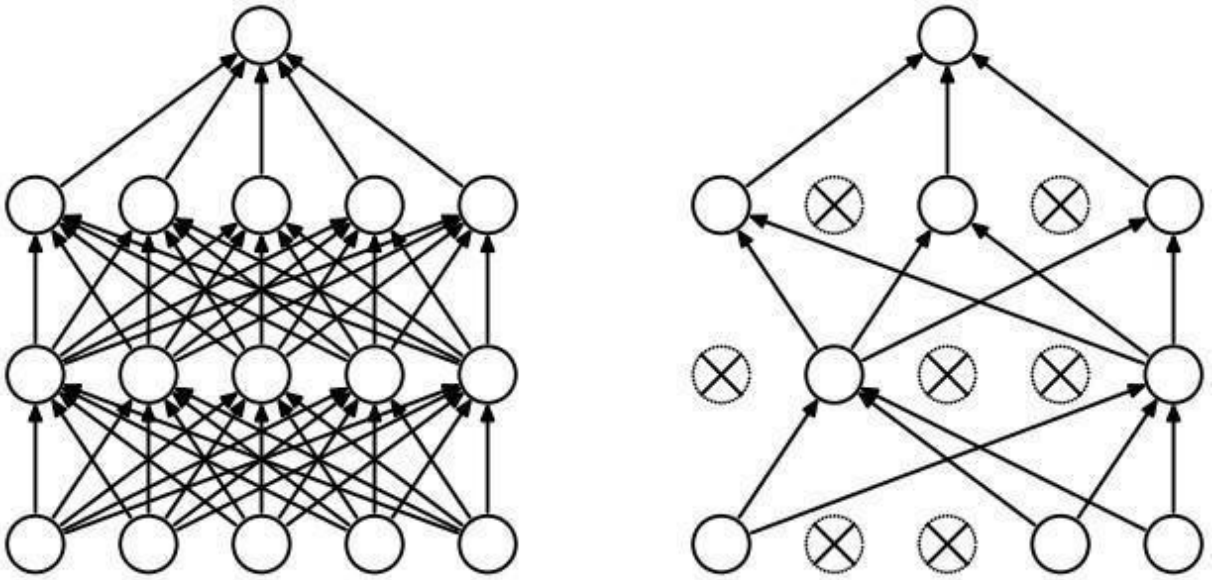
https://en.wikipedia.org/wiki/Overfitting

Regularization

Solution to overfitting: Regularization

Regularization: Dropout

"During training, randomly set some neurons to zero in the forward pass"



(a) Standard Neural Net

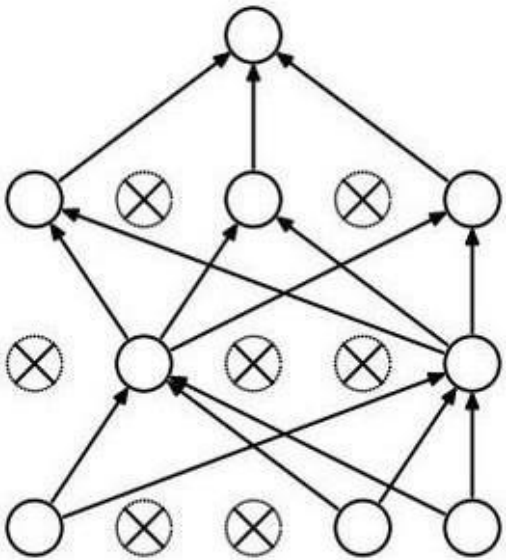
(b) After applying dropout.

Why do we need Dropout?

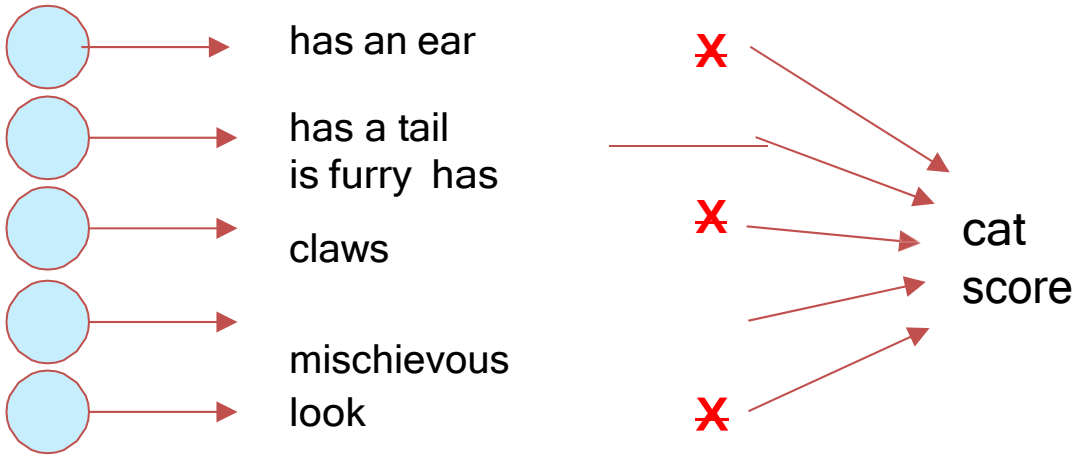
The answer to this question is “to prevent over-fitting”.

A fully connected layer occupies most of the parameters, and hence, neurons develop co-dependency amongst each other during training which curbs the individual power of each neuron, leading to over-fitting of the training data.

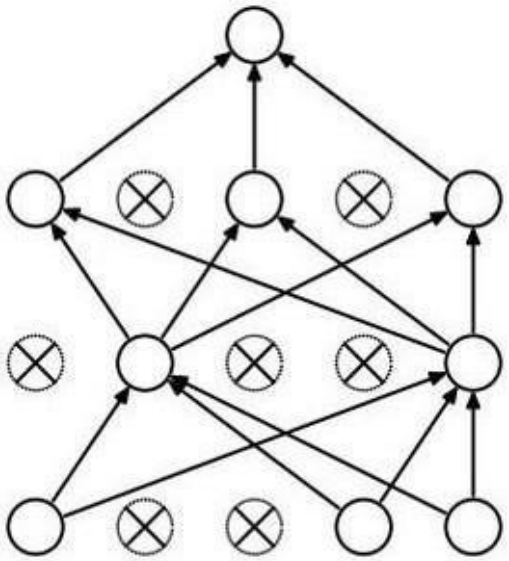
How could this possibly be a good idea?



Forces the network to have a redundant representation.



How could this possibly be a good idea?



Another interpretation:

A dropout is an approach to regularization in neural networks which helps to reduce interdependent learning amongst the neurons.

At test time...

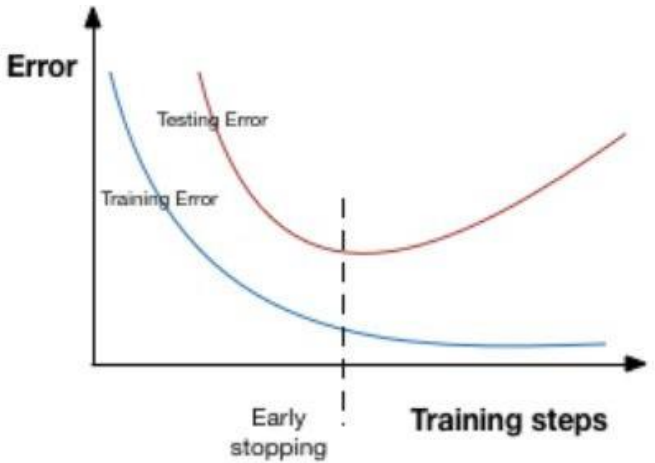
At test time all neurons are always ON

We must scale the activations so that for each neuron:

output at test time = expected output at training time

Regularization: Early Stopping

- Early stopping is a technique similar to cross-validation where a part of training data is kept as the validation data. When the performance of the validation data starts worsening, the model will immediately stop the training.
- We usually reduce overfitting by observing the training/validation accuracy gap during training and then stop at the right point.
- In the given fig, the model will stop training at the dotted line since after that our model will start overfitting on the training data.



L2 Regularization

- Due to the addition of the regularization term, the weights decrease because it assumes that a neural network with a smaller weight leads to a simpler model. Therefore, it will also reduce the overfitting problem to quite an extent.

- Weight updation equation with regularization term is given in fig (b).

- If the learning rate is high difference in (1-nl) will be low and weights wi go down.

- If the learning rate is low the difference in (1-nl) will be high and weights will be high.

$$W_{i,j} - n\frac{\Delta(L)}{\Delta W_{ij}}$$

Fig (a)

$$W_{ij}^{*}\ (1\text{-}nl) - n\left[\frac{\Delta L}{\Delta W_{ij}}\right]$$

Fig (b)

n - Learning Rate
l - Regularization Parameter
(1-nl) - Weight Decay

Noise in Neural Networks

- Training a neural network with smaller datasets may be prone to overfit the model so adding some noise during the training process can make the model more robust and reduce generalization error.

- Noise can be added in various formats. It is mainly added to inputs, weights, activation functions, and even gradients.

- The commonly used noise for training is Gaussian noise and it is mainly added to the input variables. Gaussian noise is simply termed to be a noise with a mean of zero and standard deviation of one and adding this noise to the inputs is termed as jitter.

- Here the amount of noise to be added is a hyperparameter.

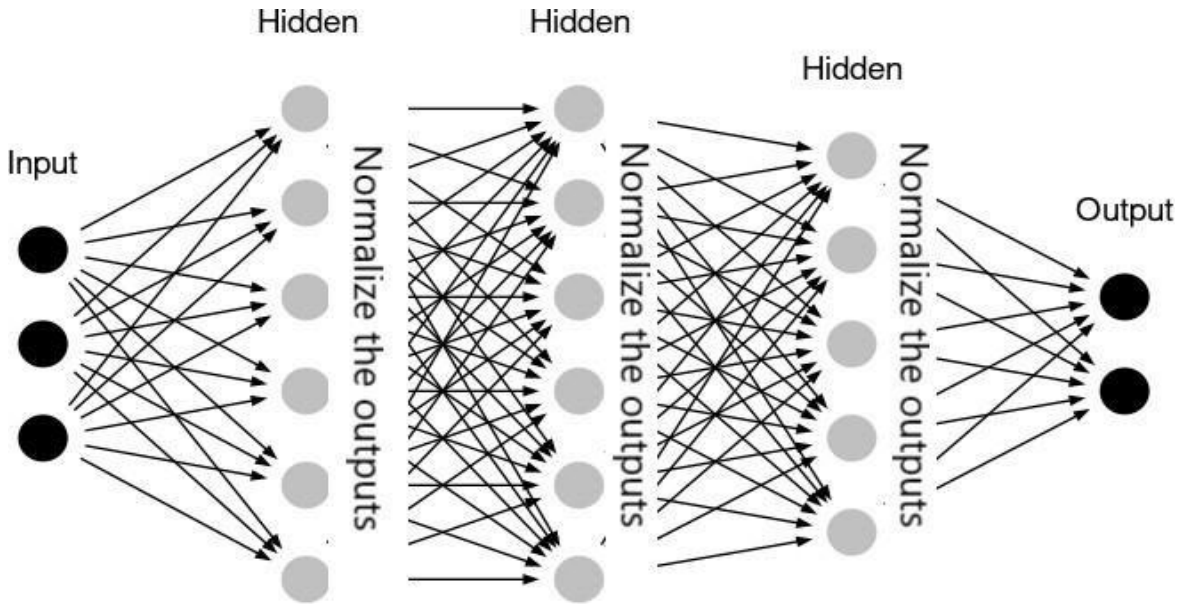
Introduction to Batch Normalization

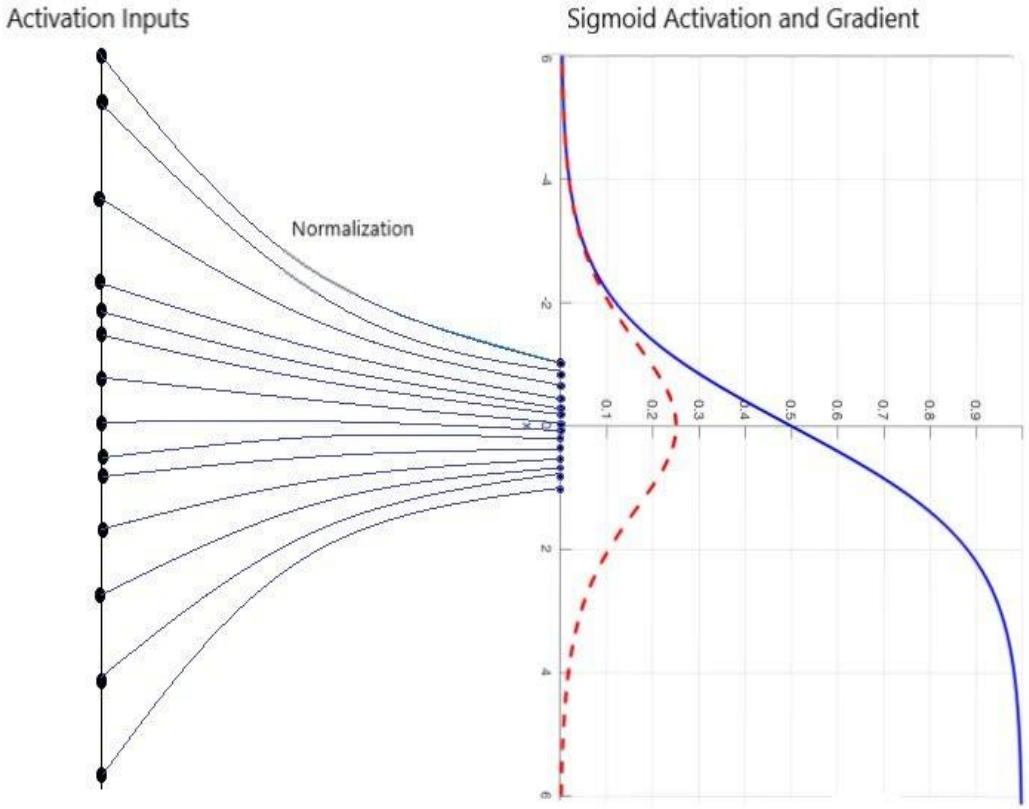
Batch Normalization

Batch normalization is a technique for improving the performance and stability of neural networks.

The idea is to normalize the inputs of each layer in such a way that they have a mean output activation of zero and a standard deviation of one.

This is analogous to how the inputs to networks are standardized.





Batch Normalization

Due to these normalization “layers” between the fully connected layers, the range of input distribution of each layer stays the same, no matter the changes in the previous layer.

Given x inputs from the k-th neuron. Consider a batch of activations at some layer. For making each feature dimension unit gaussian,

Use:

$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

Scale and Shift

There are usually two stages in which Batch Normalization is applied:

1. Before the activation function (non-linearity)
2. After non-linearity

For sigmoid and tanh activation, the normalized region is more linear than nonlinear.

A few issues:

For sigmoid and tanh activation, the normalized region is more linear than nonlinear.

For ReLU activation, half of the inputs are zeroed out.

So, some transformation has to be done to move the distribution away from 0.

A **scaling factor γ** and **shifting factor β** are used to do this.

$$y^{(k)} = \gamma^{(k)} \widehat{x}^{(k)} + \beta^{(k)}$$

Final Definition of Batch Normalization

Normalize:

$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

And then allow the network to squash the range if it wants to:

$$y^{(k)} = \gamma^{(k)}\widehat{x}^{(k)} + \beta^{(k)}$$

Where

$\gamma^{(k)}$ and $\beta^{(k)}$ are learnable parameters

$$\gamma^{(k)} = \sqrt{\mathrm{Var}[x^{(k)}]}$$

$$\beta^{(k)} = \mathrm{E}[x^{(k)}]$$

Batch Normalization - Mini Batches

- Improves gradient flow through the network
- Allows higher learning rates
- Reduces the strong dependence on initialization
- Acts as a form of regularization in a funny way, and slightly reduces the need for dropout, maybe

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

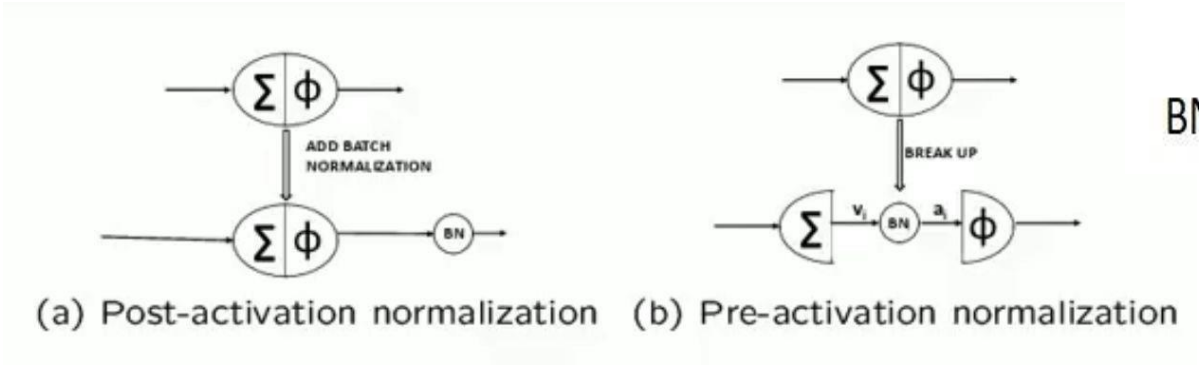
Benefits of Batch Normalization

1. Network trains faster
2. Provides some regularization

Fitting the batch norm in the deep layers

- Batch Normalization is used on the input side just on the inputs of the layer before or after the activation function in the previous layer.

- It may be appropriate to use the Batch Normalization technique before the activation function for activations that may result in non-normal distributions like the ReLu activation function.



Φ - Activation Function

∑ - Summation of Inputs and Weights

BN - Batch Normalization

Batch Normalization at Test time

Batch Normalization during Test time

- Usually, when we use Batch Normalization, it is required to analyze its behavior during test time. As the network gets trained, we try to normalize using population rather than using mini-batch statistics.

- Alternatively, we use the exponential moving average to compute the mean and variance for test time.

When Batch-norm does not work

- A feature of BatchNorm in training mode changes the absolute scale of the features according to the batch statistics, which is a random variable, while the relative distances between features are preserved.

- This is completely fine for e.g. classification and segmentation tasks, where the semantics of the image are invariant to arbitrary scaling and shifting of the channel values.

- The Batch norm should not be used mostly with dropouts as dropouts shift the variance keeping the average constant.

When Batch-Norm does not work

- For example, that you run a batch of images through BatchNorm. The absolute color information is lost, but the dogs will still look like dogs and cats will still look like cats.

- The information in the image data is preserved and the images can still be classified or segmented.

- In other words, in a dog/cat classification problem, as long as the features from an image of a dog remain more “dog-like” than “cat-like” after batch normalization, we do not worry too much about the absolute level of dog-likeness.

Guidelines for Dropout and Batch-Normalization

Guidelines for Batch-norm

- Batch normalization may be used on the inputs to the layer before or after the activation function in the previous layer.
- It may be more appropriate after the activation function if for s-shaped functions like the hyperbolic tangent and logistic function.
- It may be appropriate before the activation function for activations that may result in non-Gaussian distributions like the rectified linear activation function, the modern default for most network types.
- Using batch norm depends on the problem, it should be used when the absolute scale of the features do not matter

Guidelines for Batch-norm

- The Batch norm should be used in the deeper network
- Batch Normalization (BN) often leads to a worse performance when It is combined with dropout in both theoretical and statistical aspects.
- Theoretically, Dropout shifts the variance of a specific neural unit when we transfer the state of that network from train to test.
- However, BN would maintain its statistical variance, which is accumulated from the entire learning procedure, in the test phase.
- The inconsistency of that variance (we name this scheme as “variance shift”) causes the unstable numerical behavior in an inference that leads to more erroneous predictions finally when applying Dropout before BN.

Guidelines for Dropout

- The choice of which units to be dropped is random.
- The Dropout method is similar to the Random Forest technique in selecting the random features for the model building.
- What should be the Dropout ratio in a neural network is a hyperparameter. The dropout ratio (p) is between 0<=p<=1. Whenever a deep neural network is overfitting the (p) value should be high.
- A good value for the dropout ratio in the hidden layer is 0.5-0.8.
- The Dropout technique is applied to the training data, during prediction on test data all the neurons will be fully connected and one additional work is done during prediction is the dropout ratio (p) with be multiplied with weights (w) in every layer.

Hyperparameter Optimization

Difference between Parameter and Hyperparameter

Parameter

- Parameters are the ones which the model will learn from the data and gives the values.

Example:-

1. In Linear regression, the weight coefficients and bias are learned from the data.

2. In the Support Vector Machine, support vectors near the margin line are parameters.

Hyperparameter

- Hyperparameters are the ones which we will be giving the model that helps in the learning process.

Example:-

1. In Decision Tree and Random Forest
   Max_Depth, Min_sample, Min_sample_leaf, max_leaf_node, and criterion are the hyperparameters given to DT and RF.

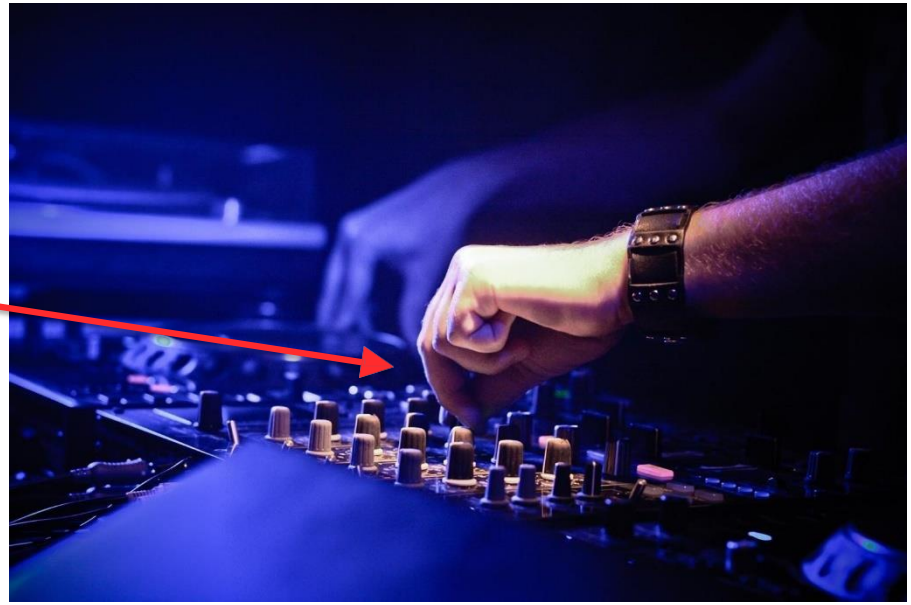
Hyperparameters in Neural Networks

- Activation Function
- Number of Neurons in each layer
- Number of Hidden Layers
- Optimizer
- Loss/ Cost Function e.t.c

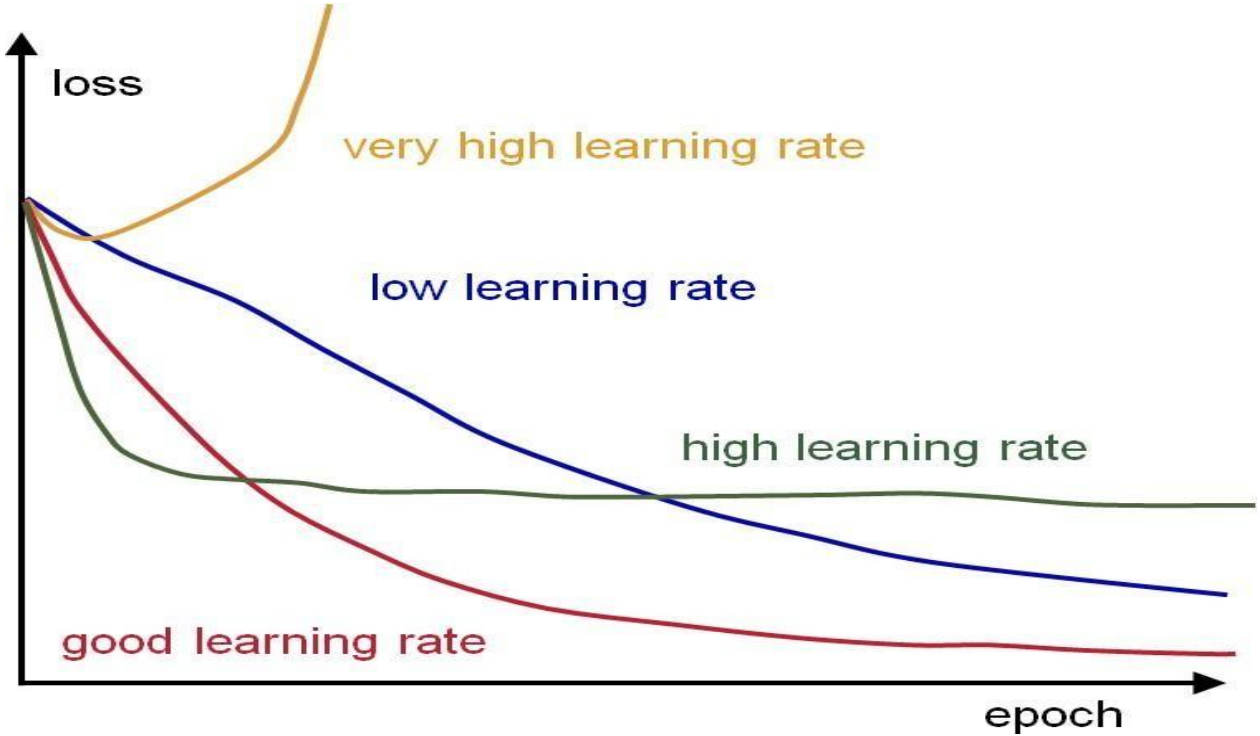
Hyperparameters to play with

- Network architecture
- Learning rate, its multiplier schedule
- Regularization (L2/Dropout strength)

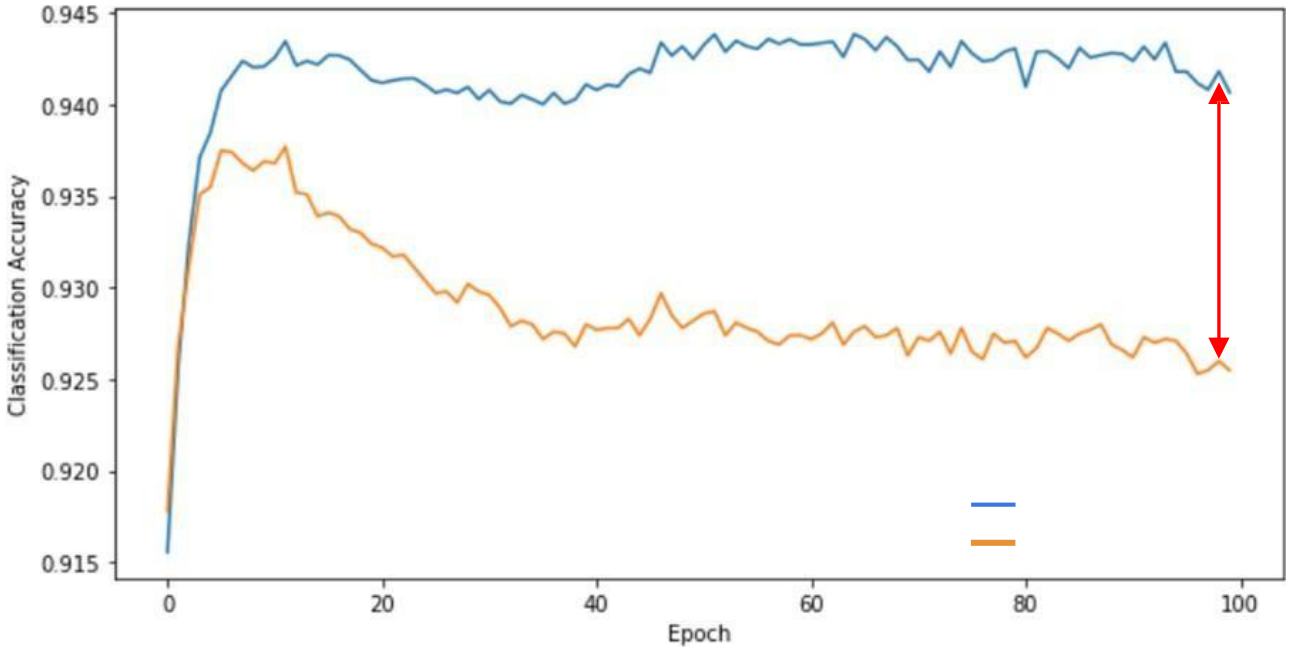
Neural networks practitioner
music = loss function



Monitor and visualize the Loss Curve



Monitor and visualize the Loss Curve



big gap = overfitting

=> increase regularization strength?

no gap - low training and validation accuracy

=> increase model capacity?

Recipe of training Neural Network

Recipe for training an ANN

1. Become one with the data

2. Set up the end-to-end training/evaluation skeleton + get dumb baselines

3. Overfit

4. Regularize

5. Tune

6. Squeeze out the juice

Please refer to this article from Andrej Karpathy