58.1 Deep Multi-layer perceptrons: 1980s to 2010s

Neural Network problems before 2010

1. Vanishing gradients
2. Too little data
3. Too little Compute

Reasons for deep learning popularity after 2010

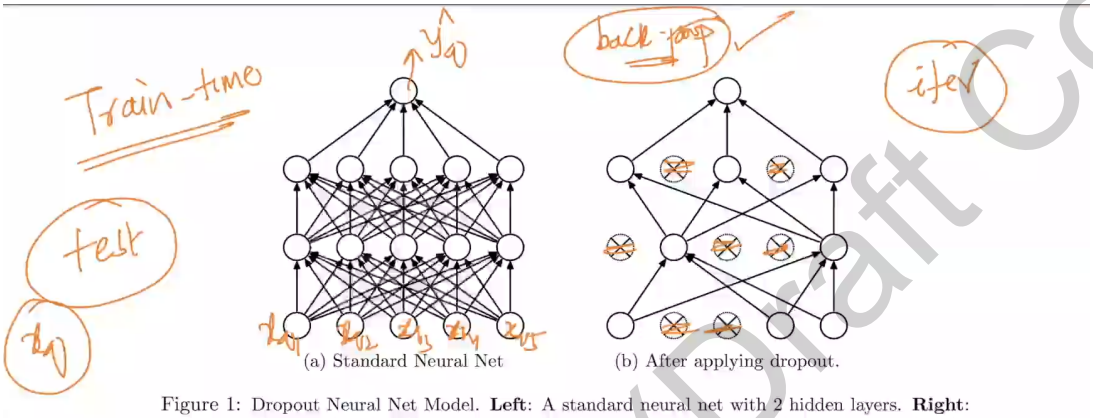
1. Increase in Data Consumption
2. Compute accessibility (gpu)
3. Good Algorithms

58.2 Dropout layers & Regularization

Similarity between random forest and dropout

Randomization for Regularization forms the basis for dropout just like in Random Forest. It means we choose the data to feed to the model to some degree with randomness, this induces regularization.

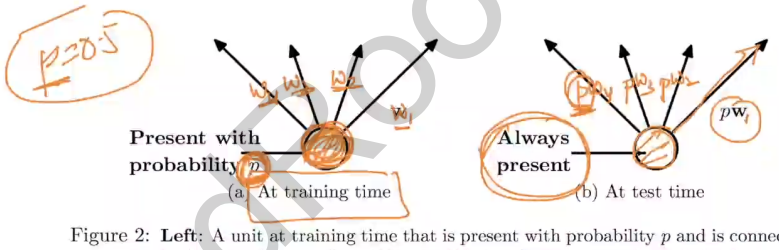
Working At Train Time



(a) Standard Neural Net (b) After applying dropout.

Figure 1: Dropout Neural Net Model. **Left**: A standard neural net with 2 hidden layers. **Right**: An example of a thinned net produced by applying dropout to the network on the left. Crossed units have been dropped.

Working of Dropout At Test Time



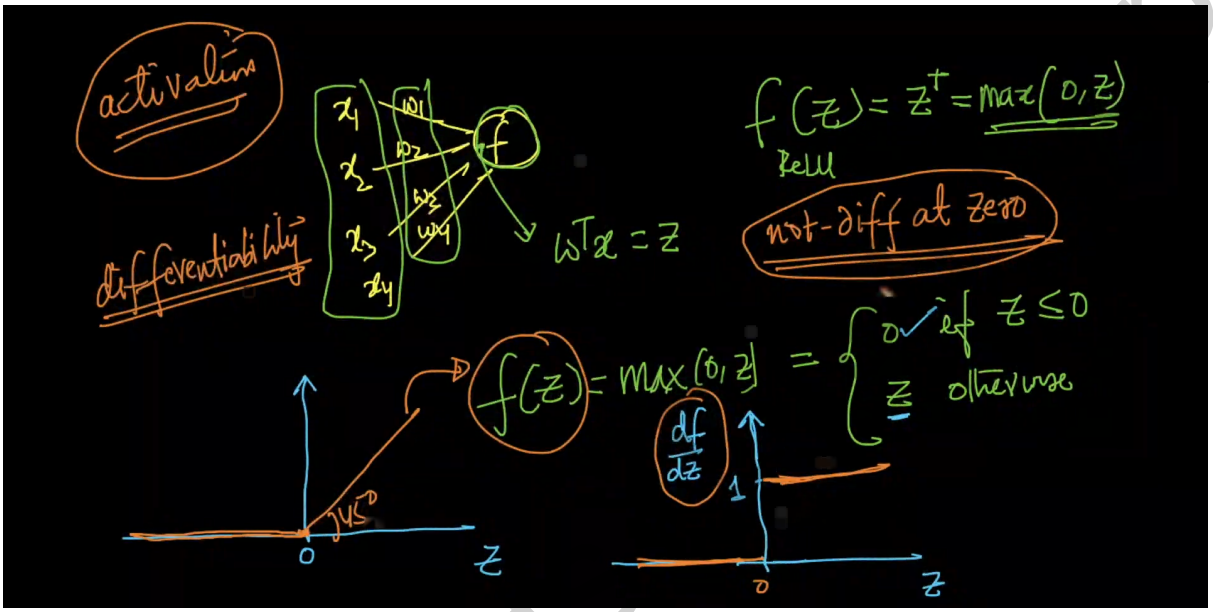
(a) At training time (b) At test time

Figure 2: **Left**: A unit at training time that is present with probability p and is connected to units in the next layer with weights $\mathbf{w}$. **Right**: At test time, the unit is always present and the weights are multiplied by p. The output at test time is same as the expected output at training time.

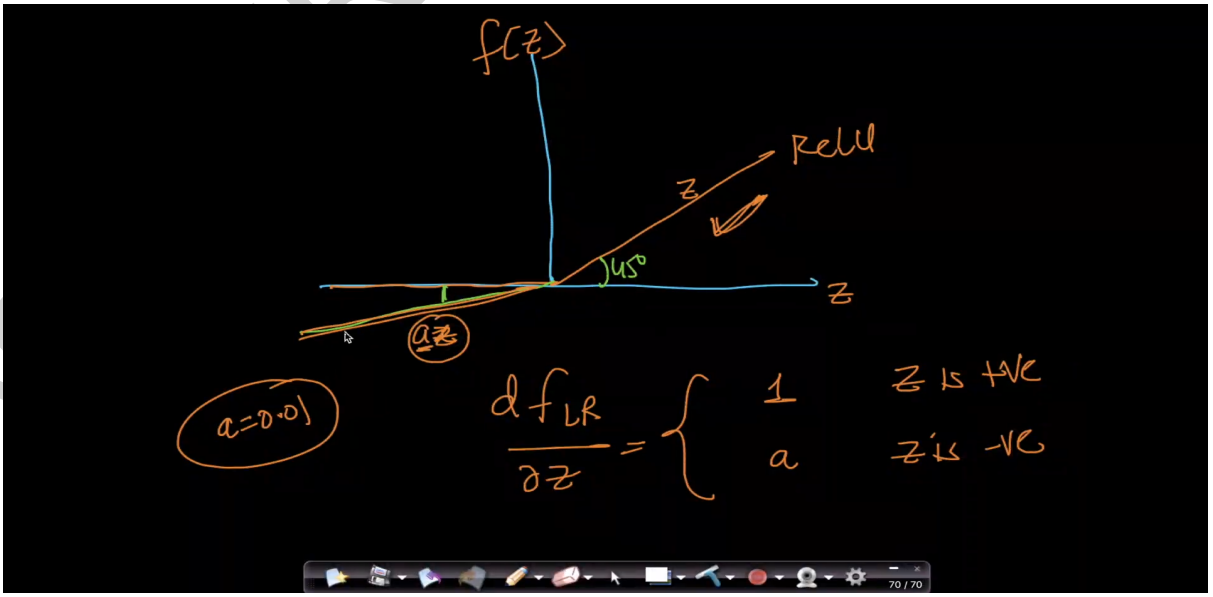
'p' in the above diagram is a hyperparameter in the Dropout layer. Dropout is very popularly used to regularize neural networks.

58.3 Rectified Linear Units

ReLU activations are used to combat vanishing gradients which were very prevalent in classical neural nets which used sigmoid or tanh activations.
Computation of gradient of ReLU is very efficient and it speeds up the convergence of the network compared to sigmoid or tanh.



ReLU is not differentiable at 0.
Derivative of ReLU is 0 or 1, this prevents explosion or vanishing gradients, but 0 derivative causes a problem with dead activations.
Leaky ReLU aims to solve this problem of ReLU.



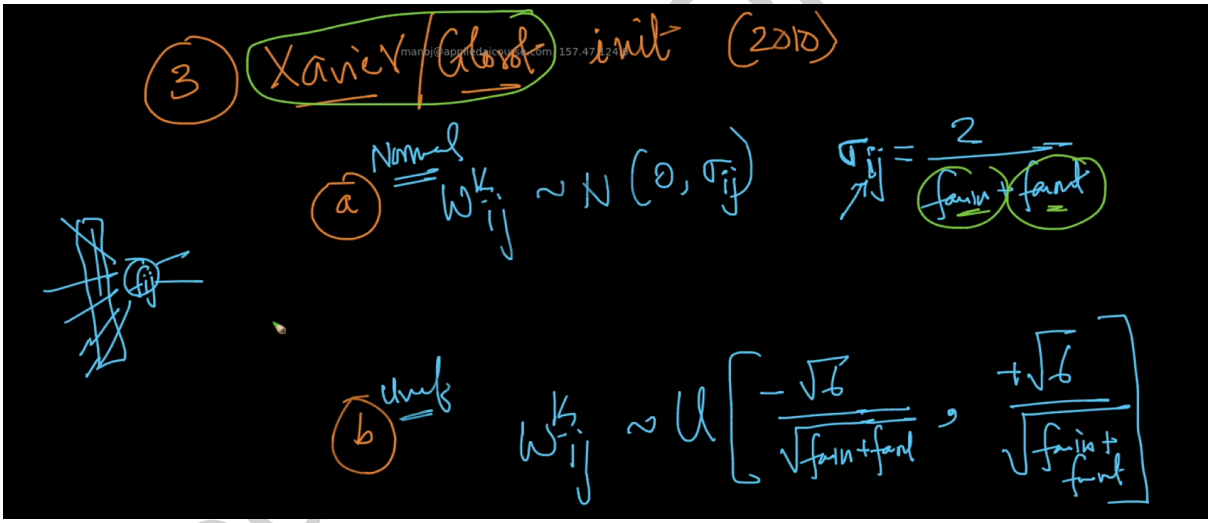
58.4 Weight Initialization.

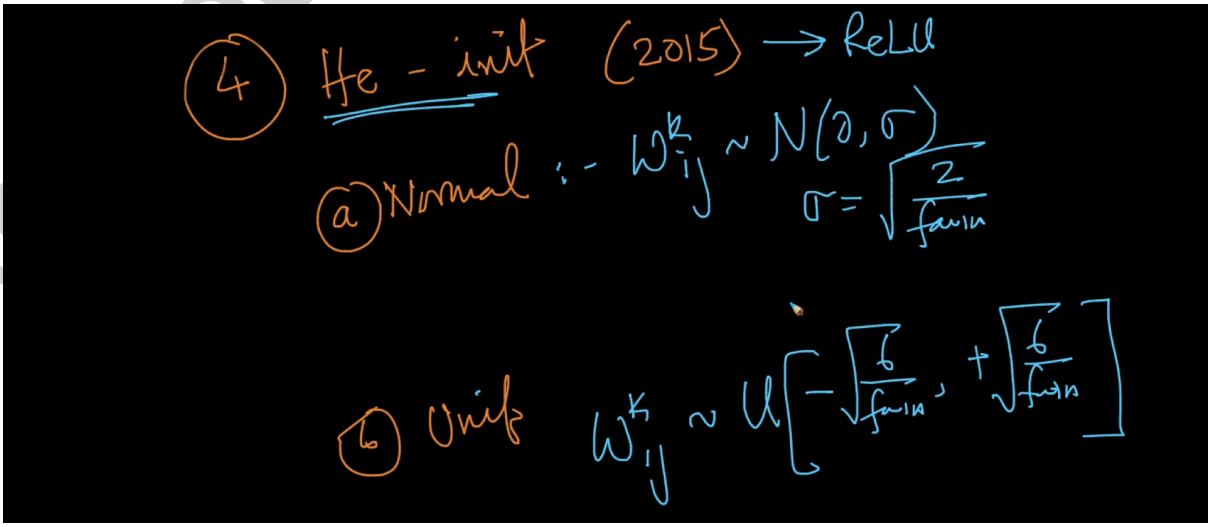
Initializing every weight to 0 means that every weight in the model learns the same thing which is ineffective. So weights are to be asymmetric so that each weight contributes to the training of the model in different ways. This is similar to using different base models in an ensemble,to improve the ensemble performance.

Even initializing to large -ve numbers will pose problems when using relu activations as this increases the dead activations in the model.

Ideas on initializing weights:

- Weights should be small and not all zeros with good variance among them.
- He initialization which works well for ReLU's
- Glorot initialization works well for sigmoid activations.



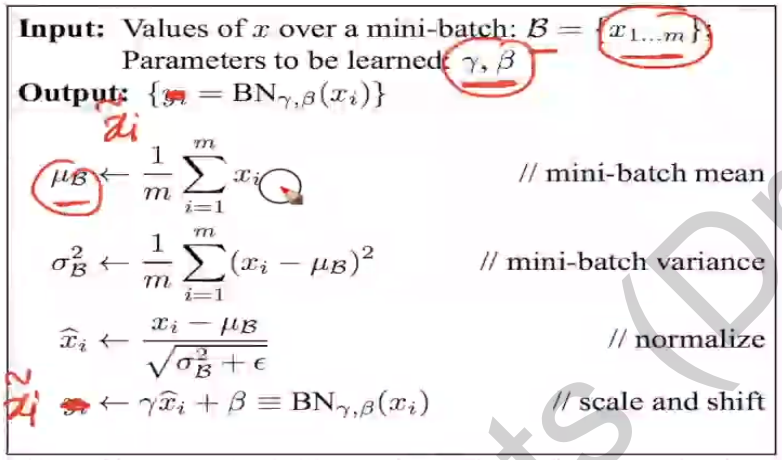


58.5 Batch Normalization

Need for Batch normalization:

A small change in the activations of early layers will be responsible for large changes in activations of later layers. Now this means the neurons at later layers will be experiencing large changes in distribution from batch to batch. This problem is called Internal Covariance Shift.

Batch Norm solves this problem by normalizing on every batch. So that distribution changes are less for later layers from batch to batch.



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

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

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

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

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

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

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

After Batch norm normalizes a batch, it scales it by gamma, and shifts it by beta.This can be thought of as a layer deciding by itself on what should be the mean and standard distribution of incoming data. Gamma and beta are parameters of a batch norm layer which are learnt through backprop.

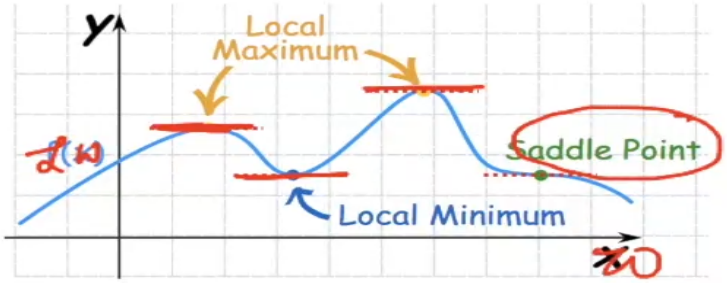
Advantages using Batch Norm Layer:

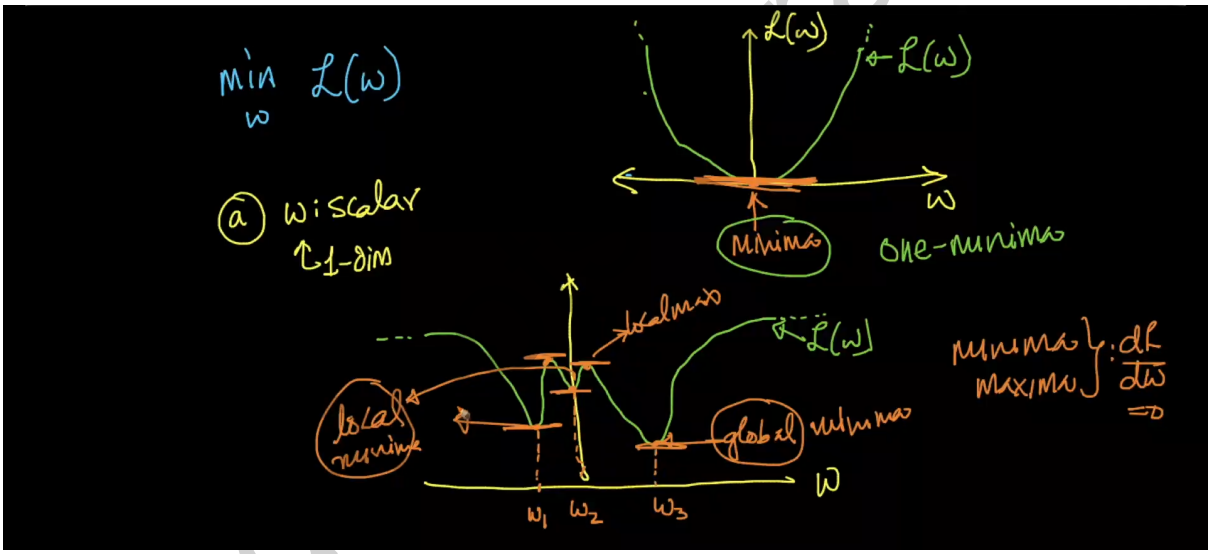
- Faster Convergence can be achieved by batch norm.
- Avoids internal covariate shift.
- Batch Norm also acts as a weak regularizer. They can work alongside dropout layers.

58.6 Optimizers: Hill-descent analogy in 2D

Minima, Maxima and Saddle points:

When gradient is 0 at some point in the curve then it could mean a minima or maxima or saddle point.
Plain SGD can get stuck at saddle points, so there is a need for advanced techniques to combat this problem.



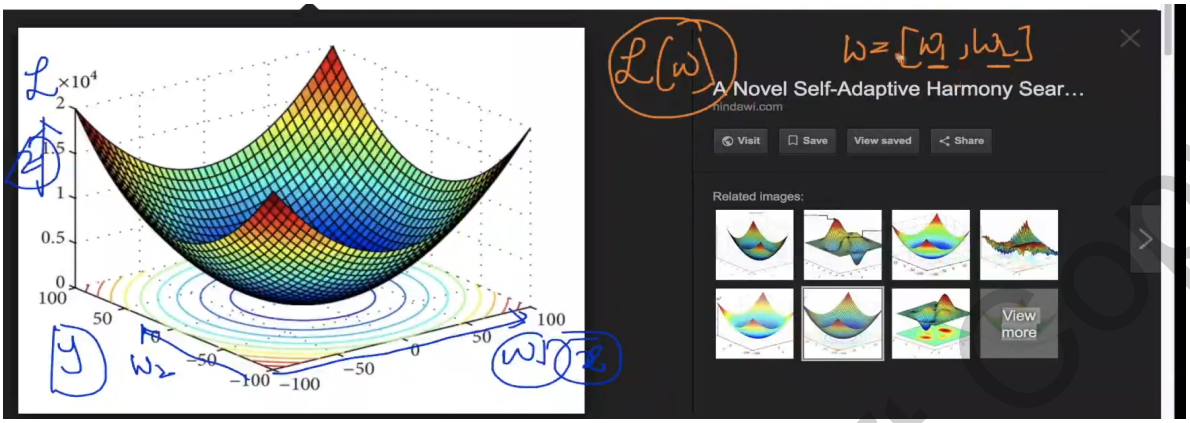


Convex and Non Convex Functions:

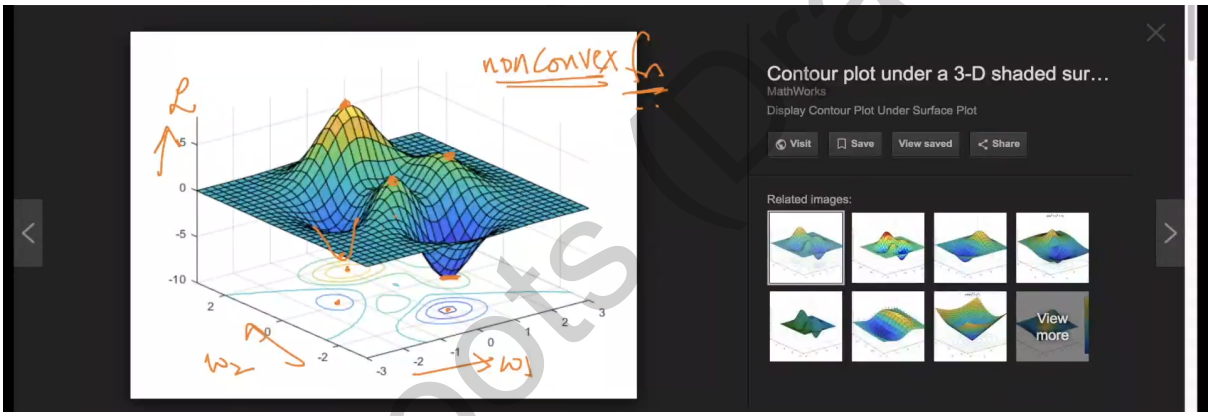
- Convex Functions have a single maxima or minima.
- Non convex functions have multiple local minima or maxima.
- Logistic Regression, Linear Regression, SVM have Convex loss functions.
- Loss functions in deep learning are non-convex. So different initializations will lead to convergence at different minima.

58.7 Optimizers: Hill-descent in 3D

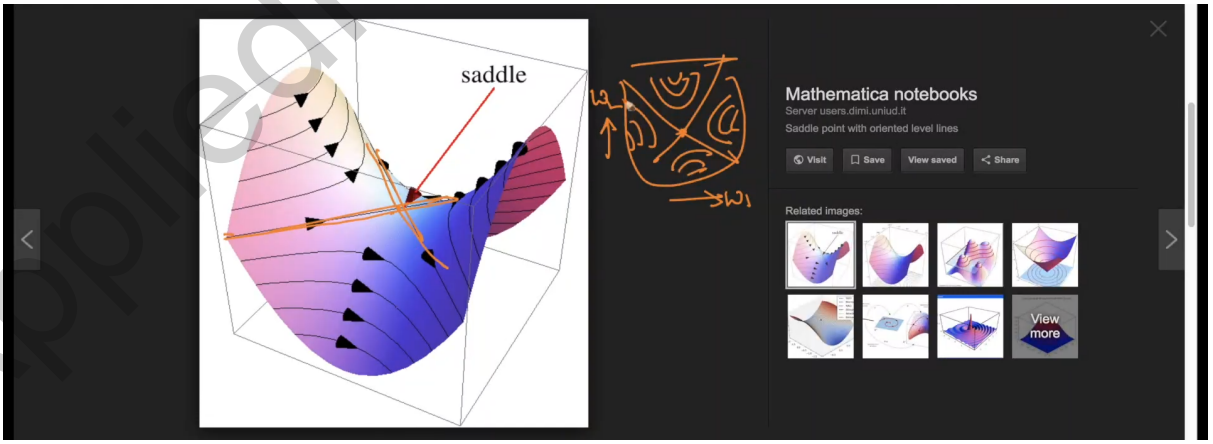
Convex Loss function Example



Non Convex function 3d example

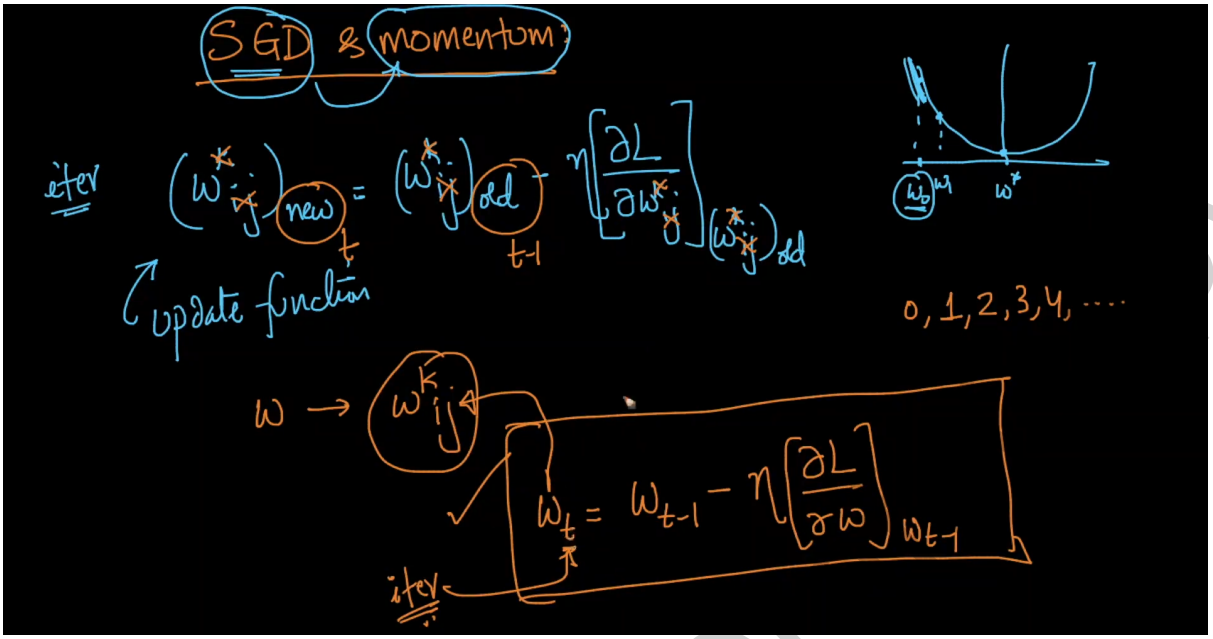


Saddle point contour plot



$y = x^3$ has a saddle point at $x = 0$

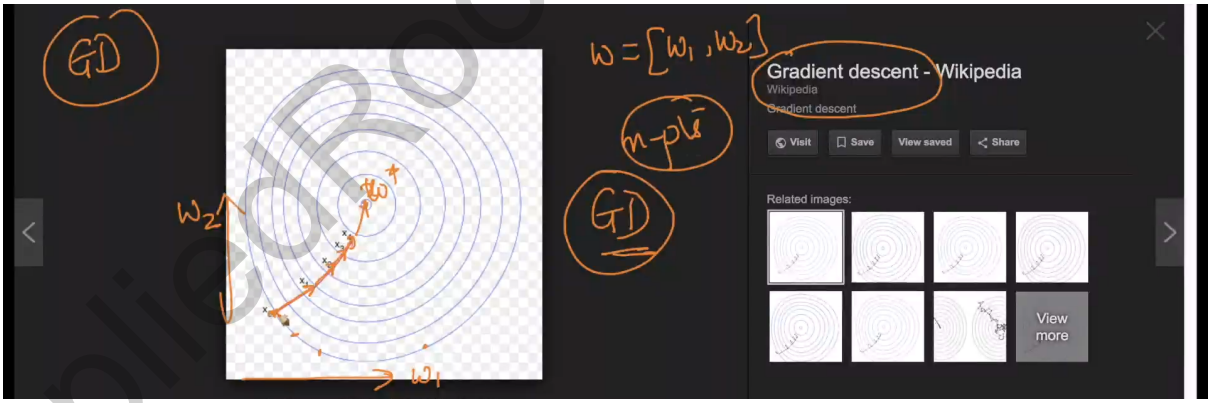
58.8 SGD Recap



Gradient Update done using:-

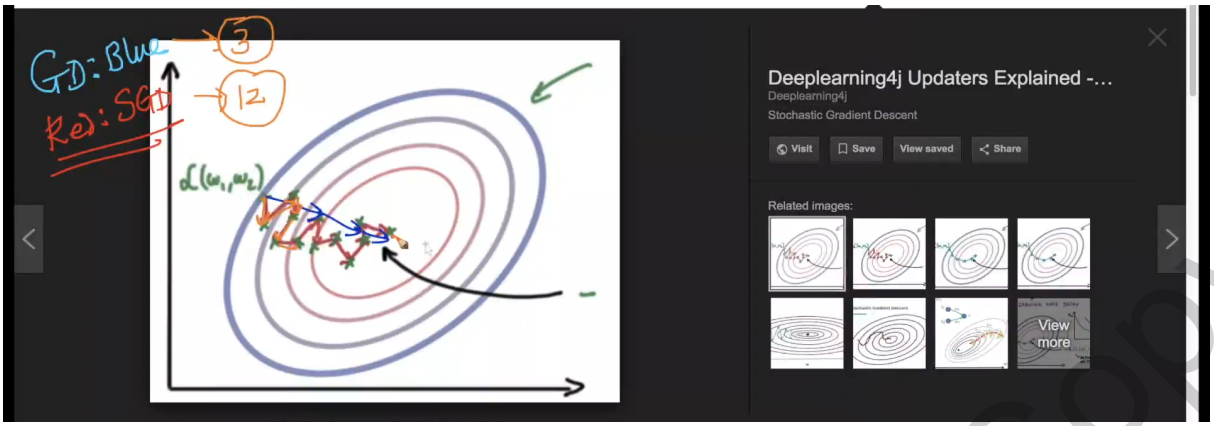
- 1 point is called Stochastic gradient descent
- All the n points in the dataset is Gradient descent.
- A Subset of data is mini batch gradient descent(very common).

GD convergence example on contour plot



Update from mini batch gradient descent takes less time than GD, and the updates from mini batch are very noisy compared to GD. Due to this convergence in mini batchGD requires more updates compared to GD.

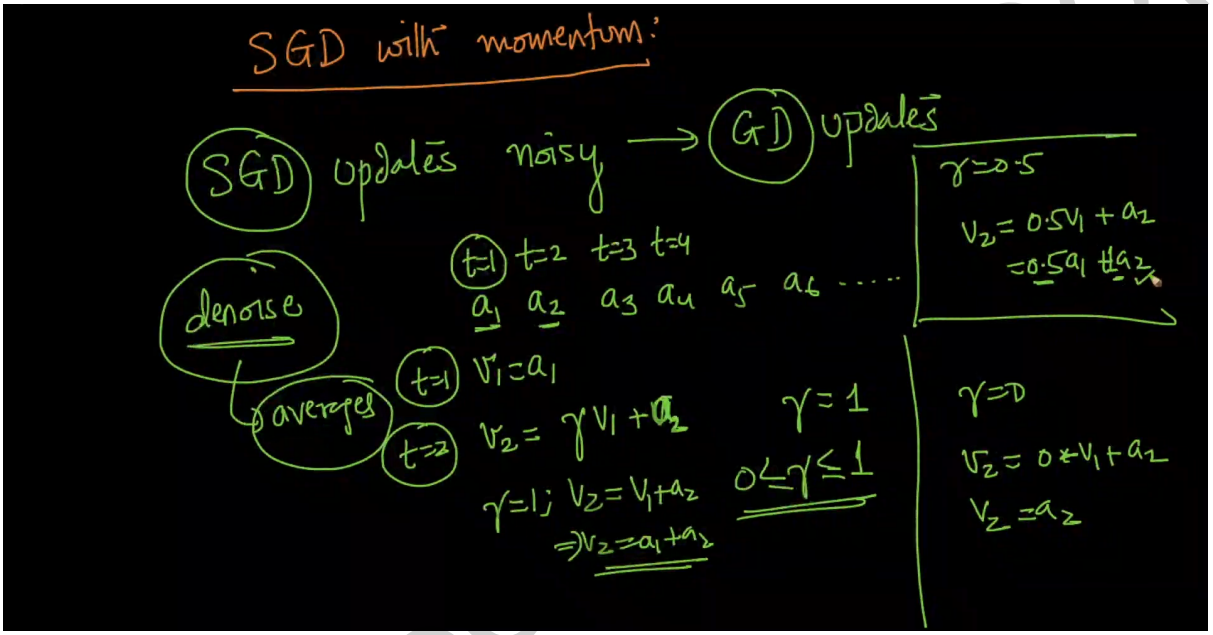
SGD vs GD updates comparison



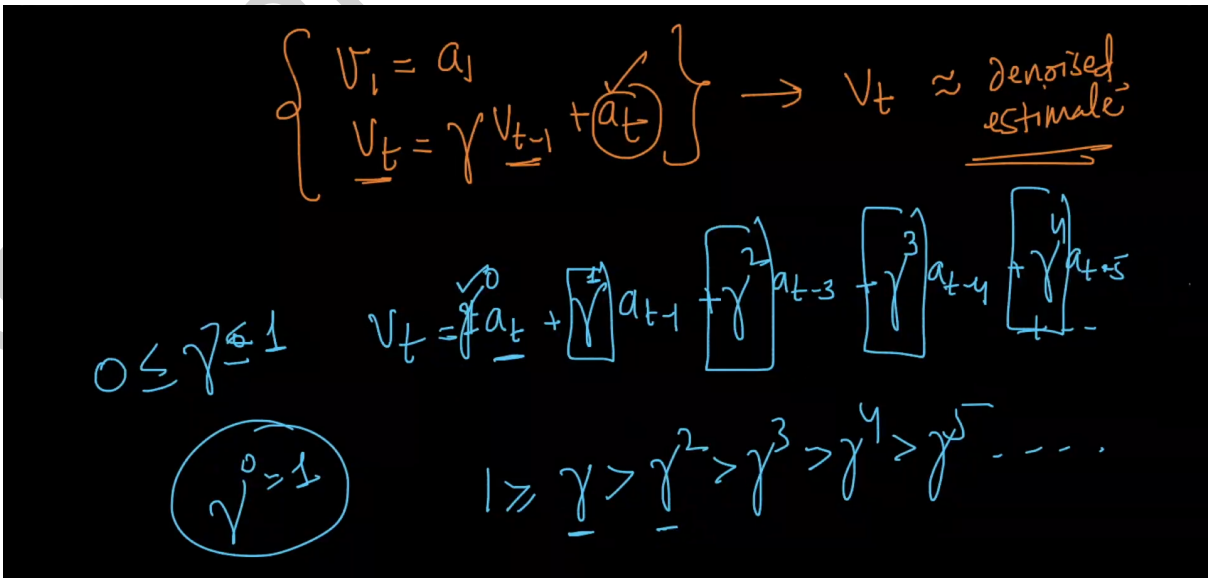
58.9 Batch SGD with momentum

Denoising the gradient updates

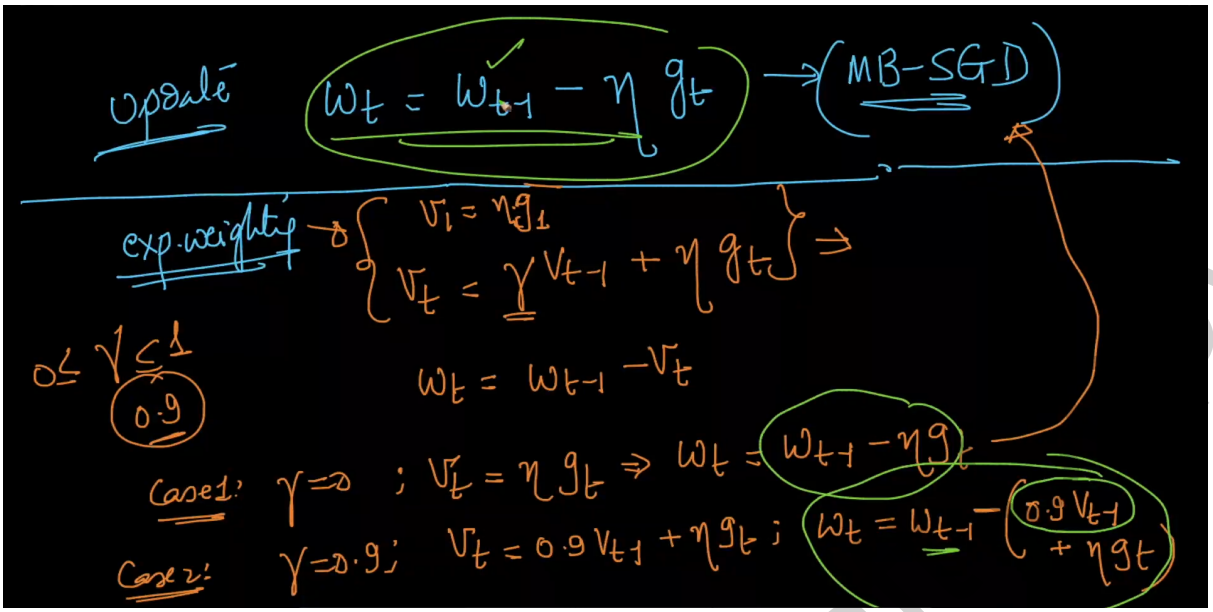
So to denoise the updates we can take the idea of weighted averaging. This is done in SGD with momentum with a weight gamma, here gamma will be a hyperparameter which can be between 0 and 1.In particular we use exponential weighted average here.



Momentum is generally expressed as a recurrence relation, which breaks down into the following equation.



Modified gradient update equation after involving momentum in it.



Momentum speeds up the convergence dramatically. It also sort of sees where actually we are heading through the noise and speeds up in that direction(towards a minima).

Notice the longer steps in image 3 compared to image 2

Image 2: SGD without momentum

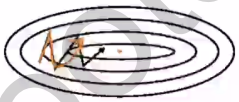
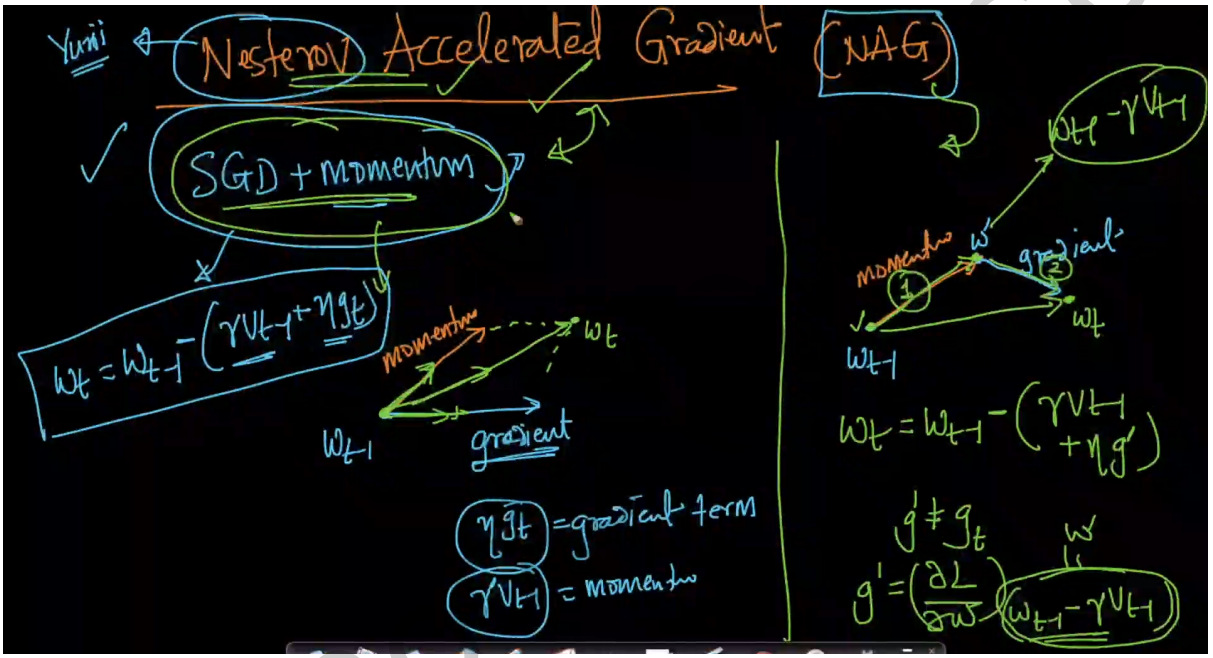


Image 3: SGD with momentum

58.10 Nesterov Accelerated Gradient

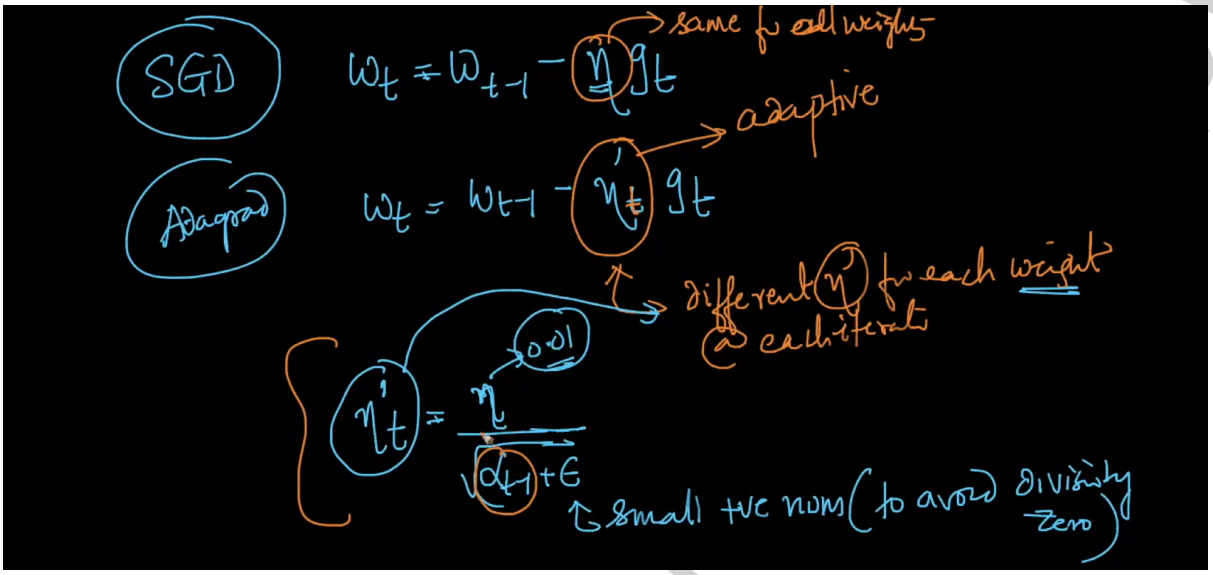
This works very similarly to SGD+momentum. Except here the gradient is computed after the momentum has been included in the gradient update unlike SGD+momentum where gradient is computed alongside momentum.

Here we can notice that the gradient of SGD+momentum is done at w_{t-1} and gradient is computed for NAG at w^{l}.

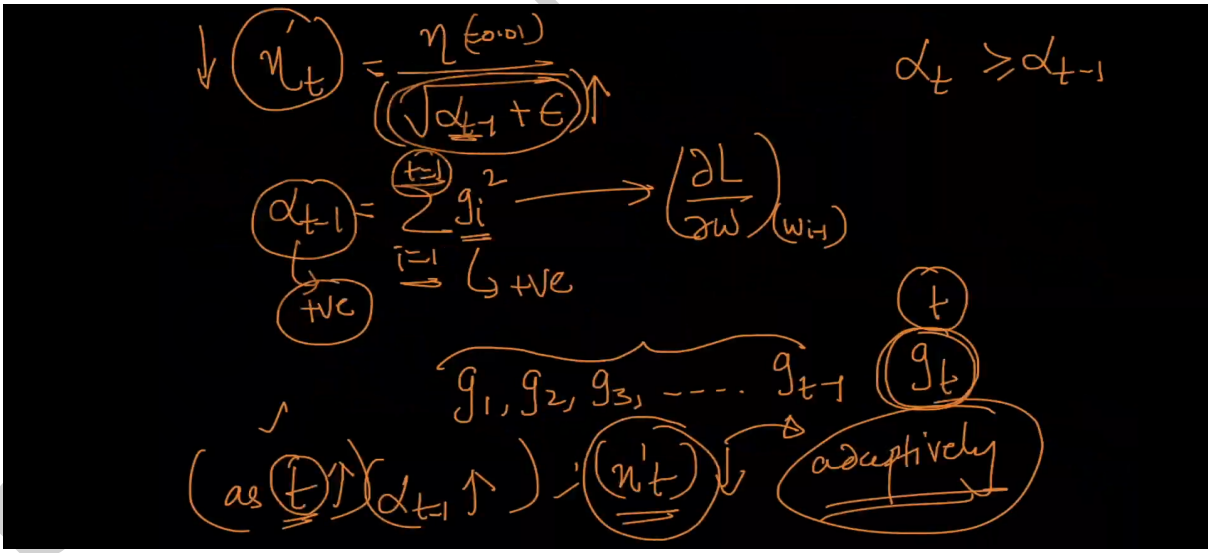


58.11 Optimizers:Adagrad

Different Learning Rate for each parameter is the idea of Adagrad.This is important in dealing with sparse features as they are to be treated differently than dense features.



Learning rate is framed using previous gradients as follows



Advantages

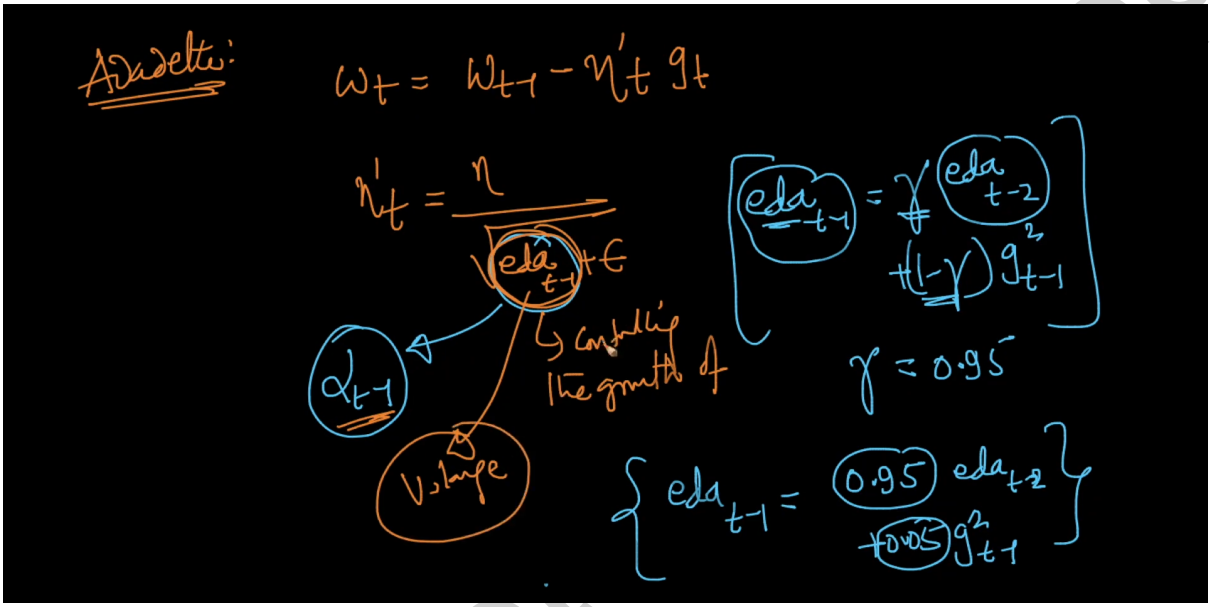
- No need for manually tuning the learning rate.
- Sparse features are taken care of.

Only drawback here is as iterations increase $alpha_t$ grows large, then eta_t becomes very small leading to slower convergence.

58.12 Optimizers:Adadelta and RMSProp

Adagrad has a problem of $alpha_t$ becoming very large, Adadelta uses exponential decaying average(eda) to combat this.

Adadelta proposes to use eda of squared gradients instead of sum of squared gradients like in Adagrad.

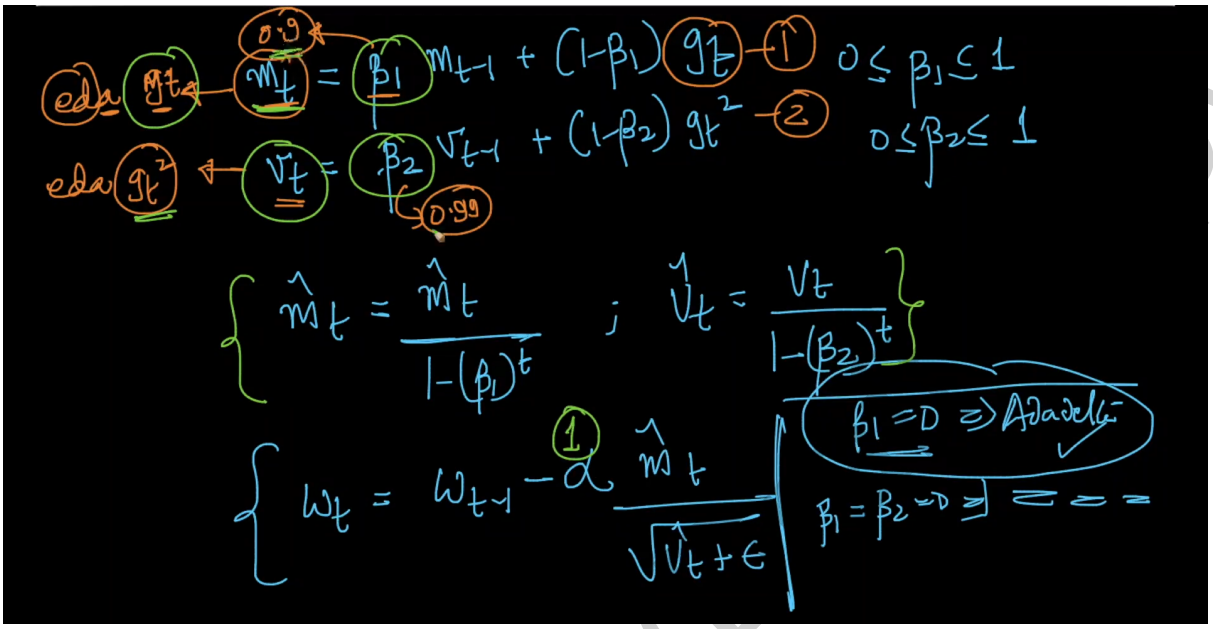


Here note that weights in the eda formula are gamma and 1-gamma which sum up to 1, so we are essentially taking a weighted average. Whereas in the momentum lecture the weights were gamma for momentum and 1 for current gradient so it was a weighted sum.

Working of RMSProp is very similar to Adadelta.

58.13 Adam

Considers the exponential decaying average of both gradients and squared gradients.



For beta$_1$= 0, Adam works like Adadelta.

Mean and Variance are known as first order moment and second order moment respectively.

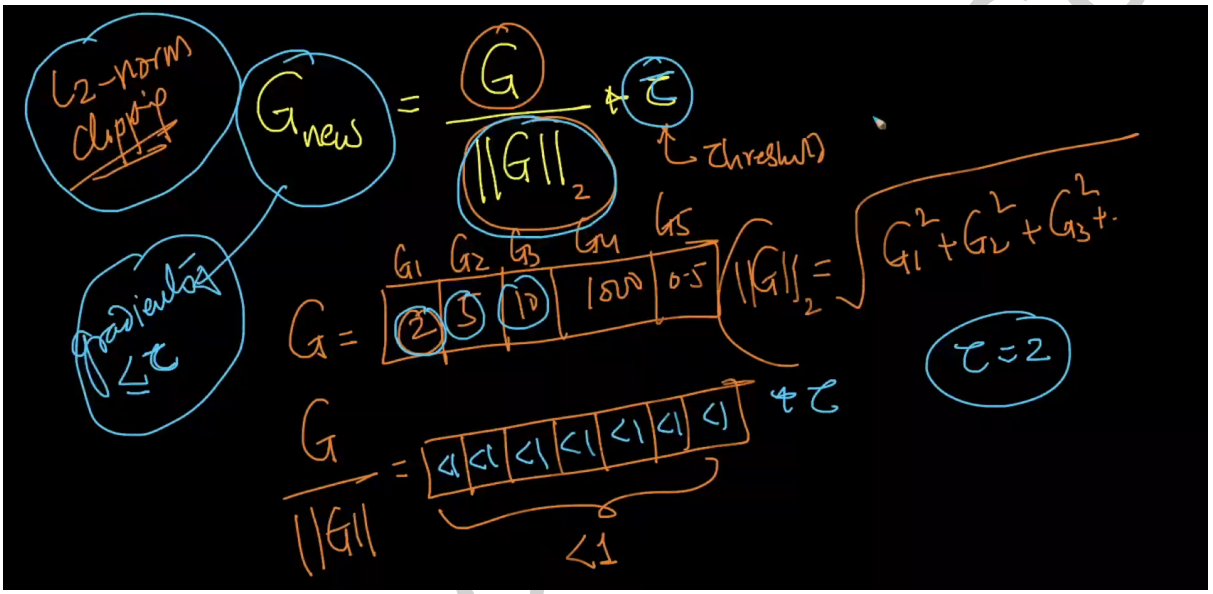
58.14 Which algorithm to choose when?

- Adam - very well used, works well on almost all cases, fairly well.
- Plain SGD- gets stuck at saddle points.
- Momentum and NAG- works well in most cases but slow.
- Adagrad- works well on sparse data.

58.15 Gradient Checking and clipping

It is a good practice to monitor gradient updates. Gradients play a very important role in the optimization process.
Monitoring should be done for each layer and every epoch.
They help in noticing vanishing gradients and exploding gradients.
Clipping of gradients helps in solving exploding gradients problem.

L2 norm clipping is done as follows:

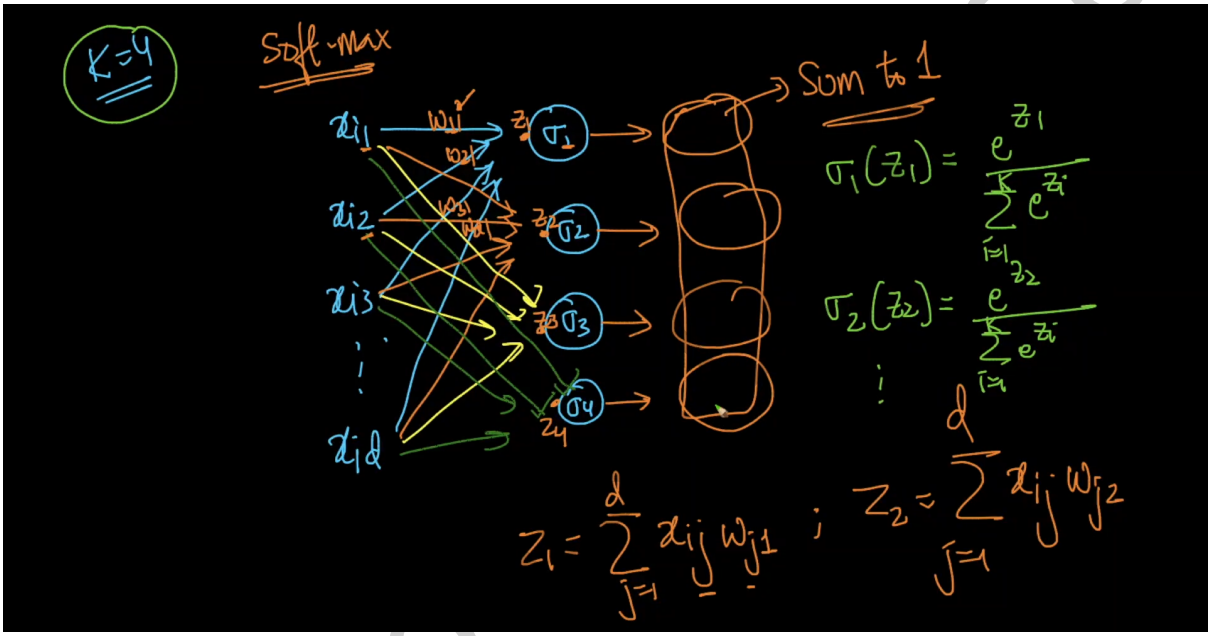


Here we are restricting the L2 norm of the gradients to a threshold(tau).

58.16 Softmax and Cross-entropy for multi-class classification

Softmax is used for multi-class classification typically along with Cross-entropy loss.
Softmax is a generalization of Logistic Regression to multi-class setting.
Sum of all the outputs of a Softmax = 1.

Formulation of Softmax



Multi class log loss is also called Cross-entropy.

Cross-entropy for N points in M class classification

$$-\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M} y_{ij}\log p_{ij}$$

Here y_{ij} is 1 if point i belongs to class j else it is 0.
p_{ij} is the probability that point i belongs to class j given point i
{p(yi belongs to class j | i)}

58.17 How to train a Deep MLP

Guidelines:

1. Preprocessing of Data.
2. Weight initialization for sigmoid use Xavier/Glorot, for ReLU activations use He initialization.
3. ReLU activation works for most cases fairly well.
4. BatchNorm helps very well to train deep neural nets.
5. Use dropout when in need for regularization.
6. Adam optimizer gives good results.
7. In Deep Learning lot of hyperparameters are present like no of layers, no of neurons per layer, learning rate, dropout rate, etc,
8. Loss functions for 2 class classification use binary log loss, for multiclass classification use multi class log loss, for regression use MSE.
9. Monitor the gradients, use clipping of gradients if needed.
10. Plot the graph for epochs vs loss and epochs vs metric.
11. Avoid overfitting, it is very easy to overfit in deep learning.

58.18 - Auto Encoders

In a nutshell, an auto-encoder is a neural network that performs dimensionality reduction. It is a neural network-based dimensionality reduction technique and is much better when compared to PCA and T-SNE.

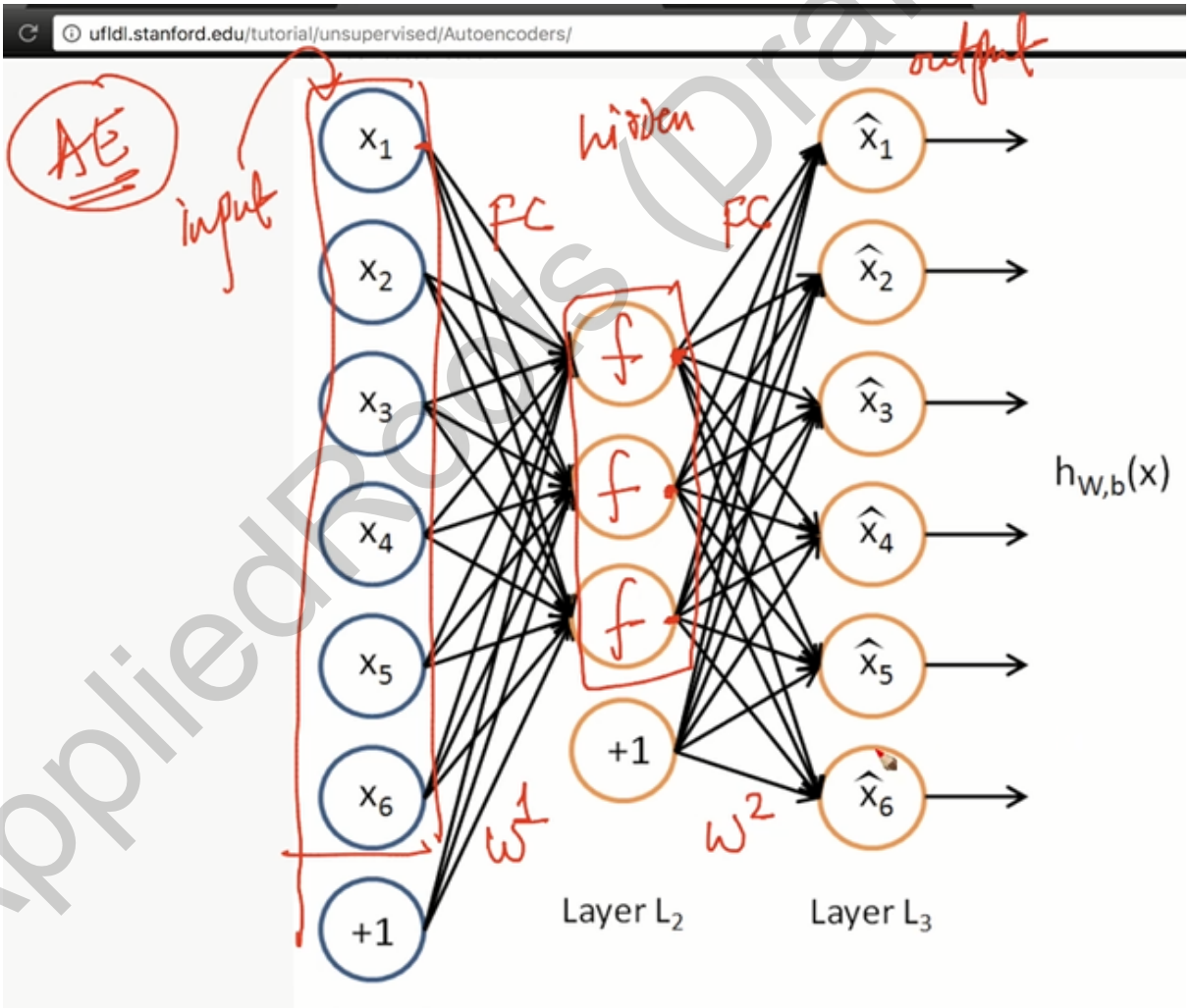
Auto Encoder with a single hidden layer

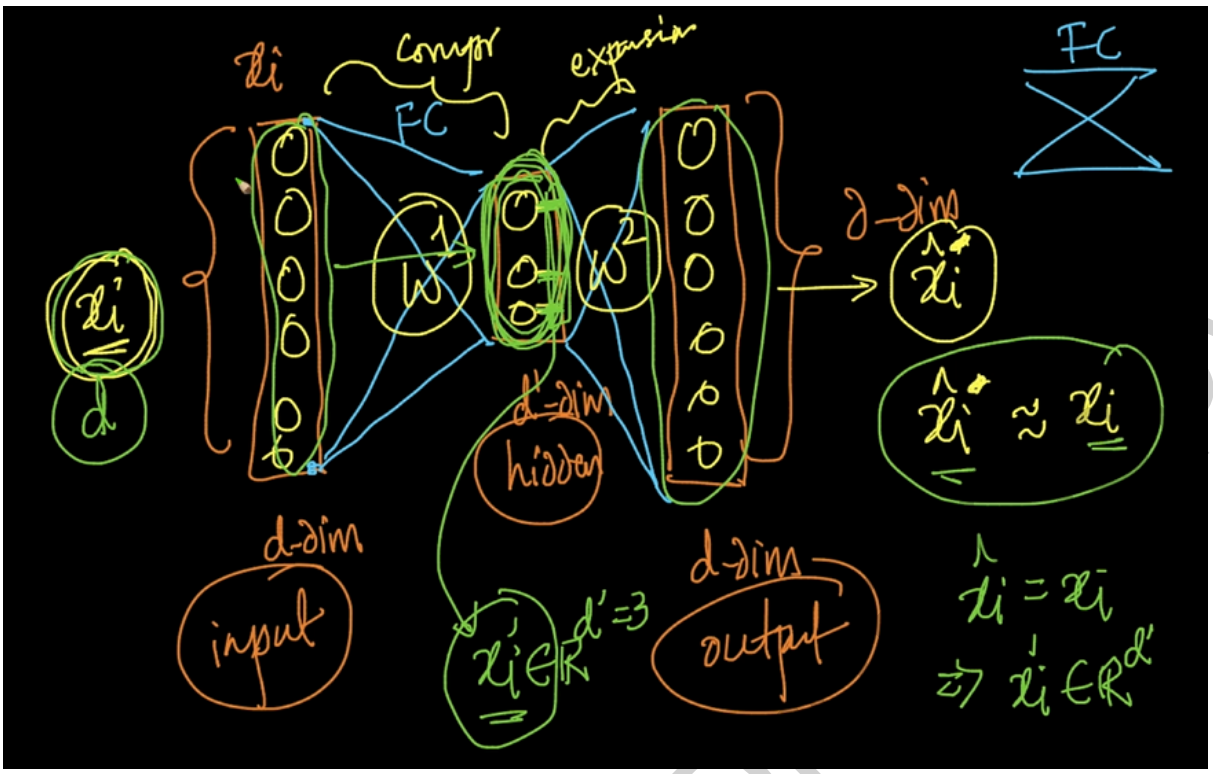
Let us assume the given input dataset 'D' is of 'd' dimensions

$D = \{x_i\}_{i=1}^{n}$ $(x_i \in R^d)$

Our job is to construct a new dataset $D^l = \{x_i^l\}_{i=1}^{n}$ $(x_i^l \in R^{d'})$ (where $d^l < d$)

For example, if our dataset consists of 6 dimensions and we have to convert it to 3 dimensions. (ie., $x_i \in R^6$, $x_i^l \in R^3$, $d = 6$, $d^l = 3$)



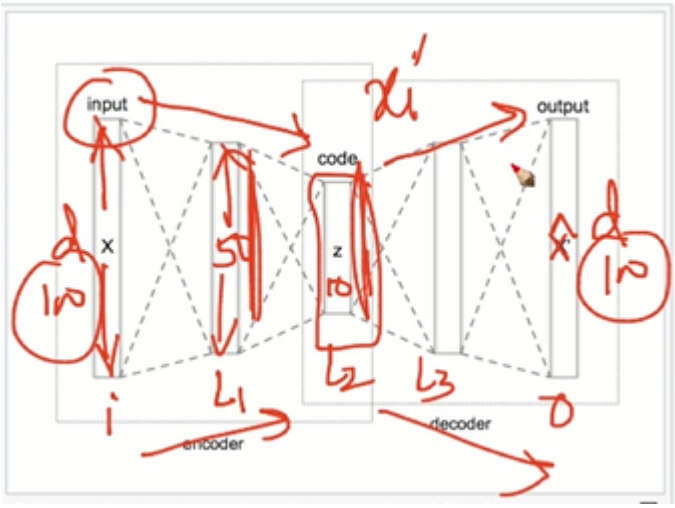


Here if $\mathbf{x_i \hat{} \approx x_i}$, then we can say $xi^{l} \in R^{d'}$. We can encode all the information by recreating all the 'd' dimensions from the 'd^{l}' dimensions. The above neural network is an autoencoder with one hidden layer. At the end, we need $\mathbf{x_i \hat{} \approx x_i}$. So for this problem, the optimization function that needs to be minimized is $\mathbf{L(xi, xi\hat{}) = ||(x_i - x_i\hat{})||_2}$

If $\mathbf{x_i \approx x_i \hat{},}$ then L=0. We can use the available optimization techniques like Adam, Adadelta, etc.

Auto-encoders with multiple hidden layers

The architecture of multi-layered auto-encoders is given below. This kind of auto-encoder is also known as **Deep Auto-encoder**. We can apply dropouts here.



There are certain extensions of auto-encoders.

a) Denoising auto-encoder

Let the given dataset be $D = \{x_1, x_2, x_3, \ldots, x_n\} \in R^d$ be the actual data. Let the data we get be $D^{\sim} = \{x_1^{\sim}, x_2^{\sim}, x_3^{\sim}, \ldots, x_n^{\sim}\}$ which is noisy and corrupted data (ie., the actual data with noise added).

If we now apply an auto-encoder on top of this noisy data,

$x_i^{\sim} \rightarrow$ Auto-Encoder $\rightarrow x_i^{l} \in R^{d'}$ $(d^{l}<d)$

This result 'x_i^{l}' is more robust to noise. The auto-encoder here, while reducing the dimensionality, also reduces the noise and the corrupted data.

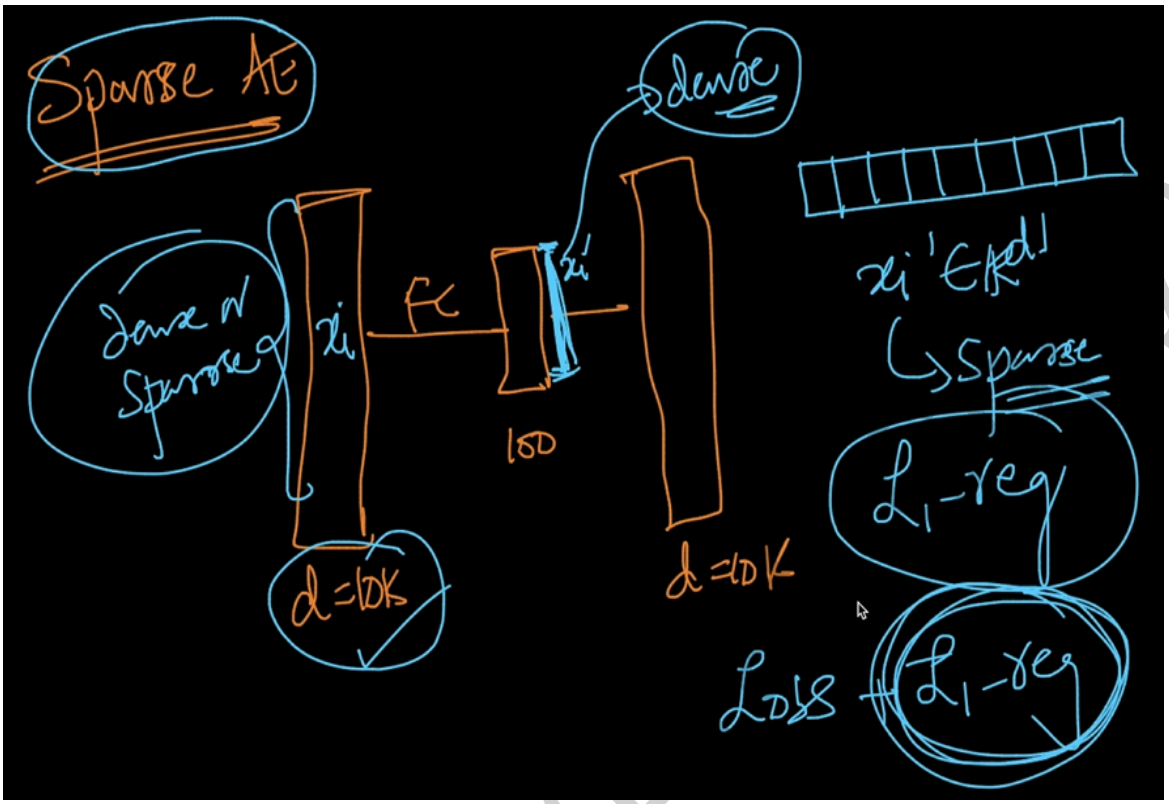
In general, whenever the developers want to denoise the data, they take the noise-free actual data 'x_i' and add some noise to it. Let the data with noise be denoted as '$x_i^{\sim}$'. ($x_i^{\sim} = x + N(0,\sigma)$)

where $N(0,\sigma) \rightarrow$ noise

On top of this data, they apply an auto-encoder to reduce the dimensionality and make the data robust to the noise.

If '$D^{\sim}$'(ie., the actual data with noise added) is directly given, then we can directly apply an auto-encoder on top of it. Otherwise, we have to add the Gaussian noise to the actual data, and then apply an auto-encoder. The output of the auto-encoder is the robust noise-free representation of the data.

b) Sparse auto-encoder



Let us assume our input data is high dimensional data like the output obtained from BOW/TF-IDF vectorizations. Suppose we need a fewer dimensional representation of the data.

Irrespective of whether our input data is sparse or dense if we want the output of the hidden layer(ie., the d^l dimensional vector) to be sparse, then we could get $\mathbf{x}_i^l$ by adding an auto-encoder with L_1 regularization, which adds sparsity. So to the loss function, we have to add L_1 regularization.

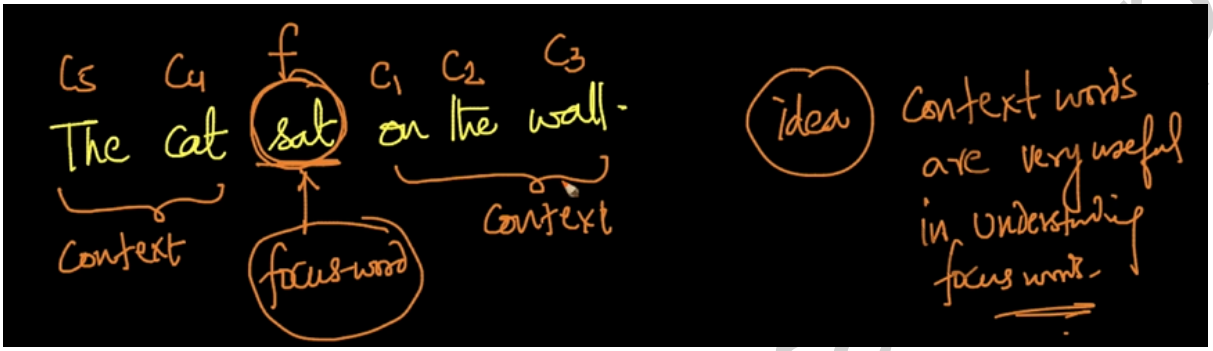
Along with the dimensionality reduction, auto-encoders can also give the best feature representations of the data in lower dimensions when compared to PCA and T-SNE.

58.19 - Word2Vec: CBOW

We have seen in the Word2Vec model, a vector is created for each word, and semantically similar words have similar vectors. There are two types of algorithms for Word2Vec. They are

a) CBOW (Continuous Bag of Words)

b) Skip-gram



If we want to make an analysis(or put focus) on a word and build a vector for it, then that word is called the **Focus word**, and the rest of the words are called the **Context words**.

In the above example, if we want to make an analysis on the word “sat”, then it becomes the focus word and the rest of the words become the context words.

CBOW - Procedure

Step 1

Let us assume we have a dictionary/vocabulary of words. Let it be declared as **‘V’**. Let the length of the vocabulary be denoted as **‘v’**.

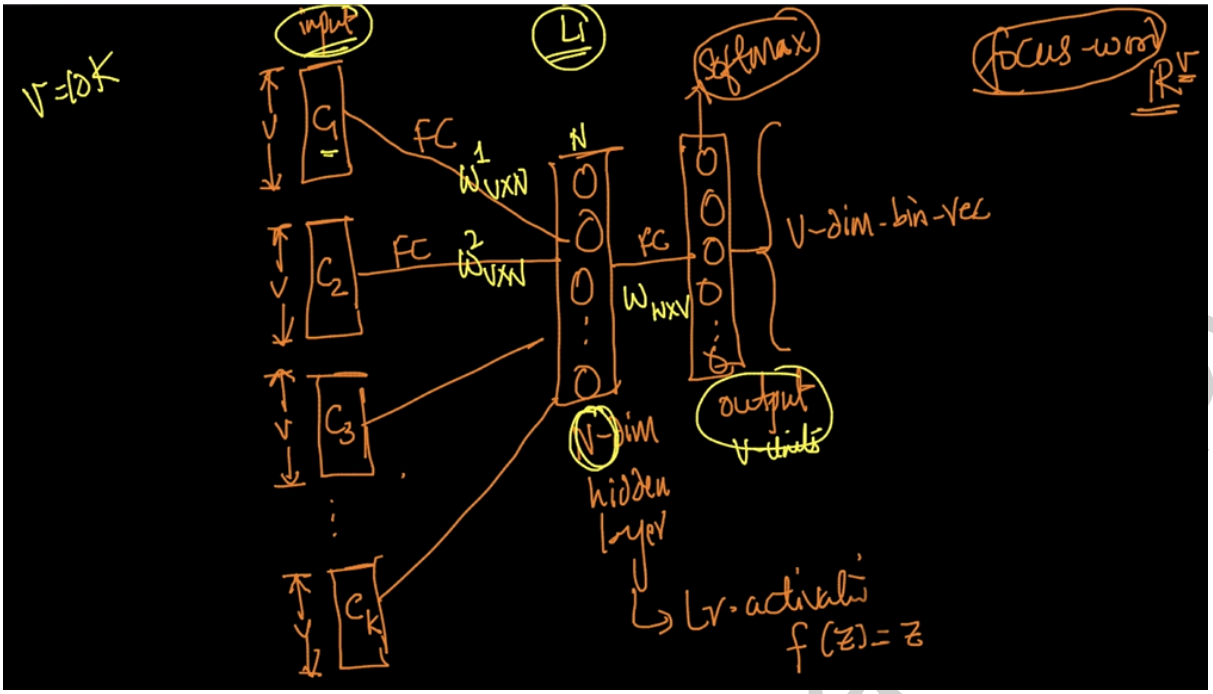
Step 2

Convert each word into a vector format using the one-hot encoding (ie., converting each word ‘w_i’ into ‘v’ dimensional binary vector)

$w_i \in R^v$ (binary vector of ‘v’ dimensions)

Step 3

Create a v-dimensional vector for all the context words present in the given text.



We get the final output in the form of a matrix 'W_{NxV}', which is of the order 'NxV'. In this matrix, the i^{th} column represents the word vector of the word 'w_i'.

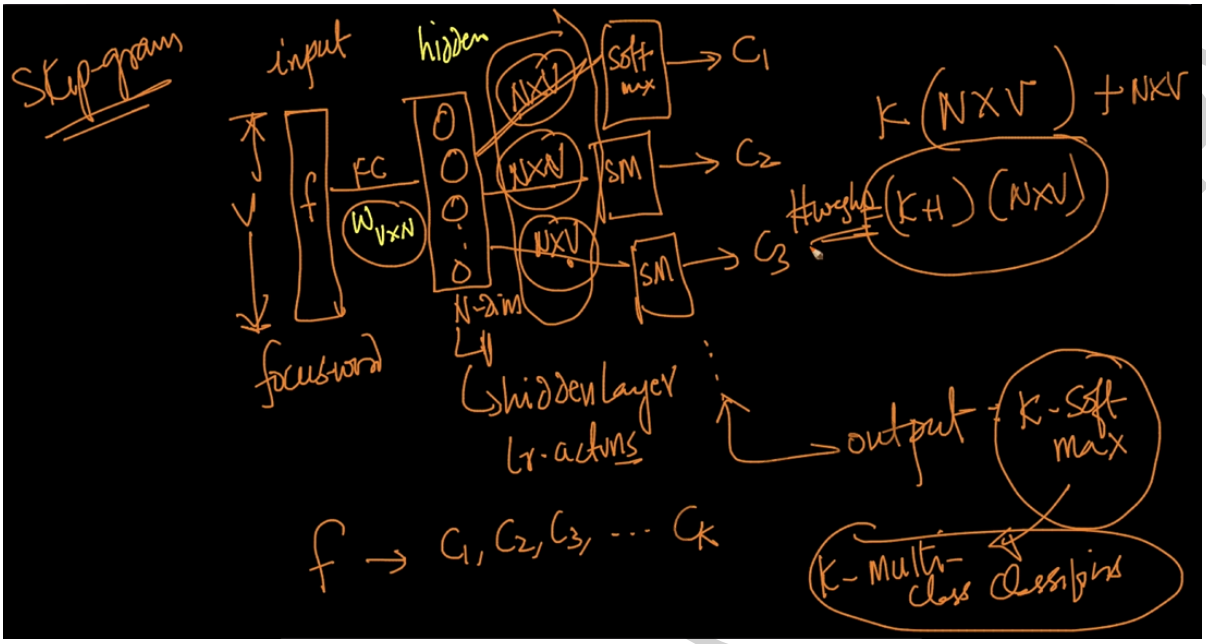
The code idea of CBOW is if the context words are given, we have to predict the focus word. The prediction of the focus word can be treated as a multi-class classification problem, as the predicted word is one among those 'v' words.

CBOW Training

a) Take all the text in the corpus.
b) Create (focus word, {context words}) combinations.
   Ex: ('sat', {'the', 'cat', 'on', 'the', 'wall'})
c) After creating the above combinations, we have to train the model using the dataset.
d) At the end of the training stage, we have all the weights computed. All the weights are summarized in 'W_{NxV}' matrix. In this matrix, we have the vectors for all the v-dimensions. The Word2Vec of each word can be obtained from this 'W_{NxV}' matrix The word vector of a word 'w_i' is nothing but the i^{th} feature in 'W_{NxV}' matrix.

58.20 - Word2Vec: Skip-grams

So far in CBOW, we have discussed how to predict the focus word, when the context words are given. But here in the skip-grams, we predict the context word, given the focus word.



Here the total number of weights = K*(N*V) + (N*V) = (K+1)*(N*V)
In the case of both CBOW and Skip-grams, the number of weights obtained at the end of the training phase is the same, and it is equal to **(K+1)*(N*V)**. The only difference is in the case of CBOW, we have only one softmax to train, whereas, in the case of Skip-gram, we have 'K' softmax classifiers to train, which is nothing but equal to training 'K' multi-class classifiers. Hence skip-gram is computationally expensive when compared to CBOW.

Advantages of CBOW

1) CBOW is faster than skip-gram because in CBOW, we have only one softmax, whereas in skip-gram, we have 'K' softmax classifiers to train.
2) CBOW works better for frequently occurring words, as the word occurs multiple times with different combinations of context words, and this particular word, when it becomes a focus word, will have many context words. Having many context words will help a lot in predicting the focus word easily at a lesser time.

Advantages of Skip-grams

1) Though skip-grams are slower, they work fairly well with a small amount of data when compared to CBOW.
2) Skip-grams work well for infrequently occurring words.

Skipgrams are much harder than CBOW, as it is difficult to predict multiple words when a single word is given when compared to predicting a single word when multiple words are given.

There is a major disadvantage with both CBOW and Skip-grams. Let 'K' be the number of context words.

For skip-grams and CBOW, as 'K' increases, the 'N' dimensional representation for a point is better. As 'N' increases, we have more dimensions, we get a better representation of the data.

In both CBOW and Skip-gram, the weights are obtained from $\mathbf{W_{NXV}}$.

The major point to be noted is that whether we train CBOW or skip-grams, the total number of weights we get is **(K+1)*(N*V)**.

For example, if N=200, K=5, V = 10000, then

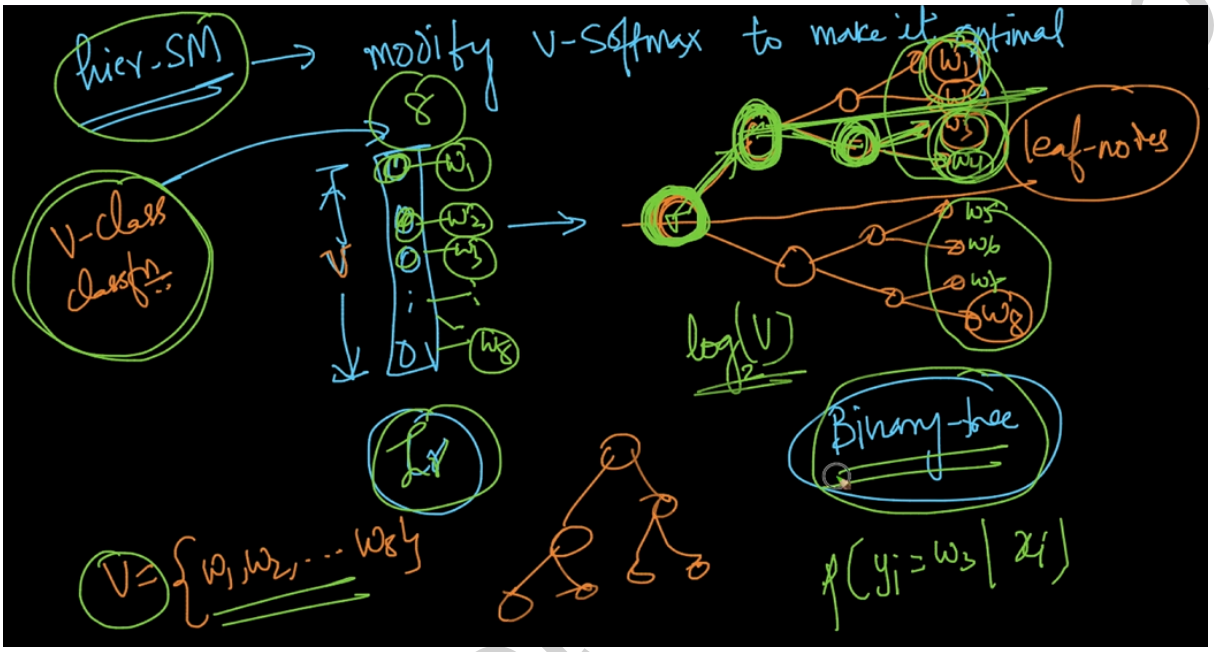
Total number of weights to be computed = **(K+1)*(N*V)** = (5+1)*(200*10000) = 12 million

Hence we need to go for algorithmic optimizations, in order to reduce the number of weights that are to be computed.

58.21 - Algorithmic Optimizations

So far we have seen both the techniques CBOW and skip-grams have millions of weights to be computed, and make the problem solving consume a huge time. In order to handle this problem, there are 2 primary algorithmic optimizations. They are

a) Hierarchical Softmax



The major computation time in both CBOW and skip-grams is taken by the softmax. The core idea of hierarchical softmax is to modify the v-softmax with v-outputs to make it optimal.

The above representation is like a sleeping tree and we are placing each word at the end of the tree leaves. Each of these leaf nodes is the word we care about in the vocabulary. When compared to the computation of $z = \Sigma w_i x_i$, computing softmax is much harder. Instead of creating a v-class softmax, it is always better to build a tree-like representation. In a v-class softmax, it computes the probability estimates for each one and then picks the one with the maximum probability.

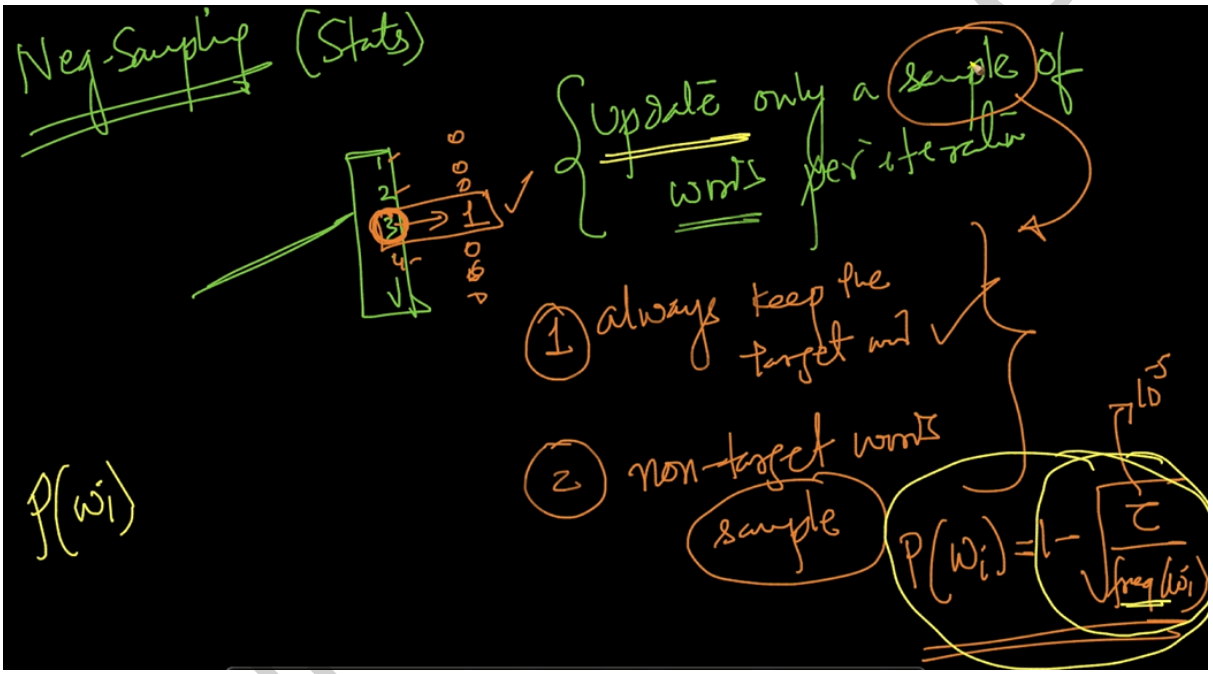
Whereas here it keeps checking the condition at a node and searches the upper and lower half trees. Once it is found in one of them, again the words are divided into 2 halves, and checking is done on both. This way, instead of checking all 'v' activation units, it just checks **log(v)** activation units. So here at the end, again the probabilities are computed. But the number of activations needed here is **log(v)**, to

compute the probabilities. This way of using a binary tree of activation functions is more efficient.

For example, if the word is 'w_4', then **condition 1** is checked first and goes into the upper tree. There **condition 2** is checked and goes into the lower tree. Then the **condition 3** is checked and then reaches 'w_4'. So initially if we have '8' words, then the number of activations used to predict the word is $\log_2(8) = 3$.

b) Negative Sampling

For any output, in the end, we have to update the weights. The core idea here is to update only a sample of words per iteration.



Step 1

The way of sampling is always to keep the target word.

Step 2

Among all the non-target words, do not update all the weights. Update only those weights picked by sampling.

Probability of sampling a word(w_i) = 1 - sqrt(τ/frequency(w_i))

The typical value of $\tau = 10^{-5}$

This probability is high for a word if it occurs more times. Hence we are updating the weights of only some non-target words to counter the imbalance between frequent and rare words. Negative Sampling is exclusively used in Skip-grams, but not in CBOW.