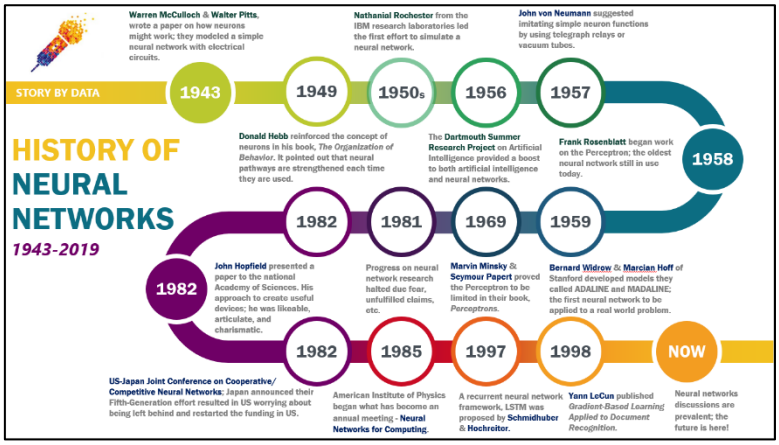
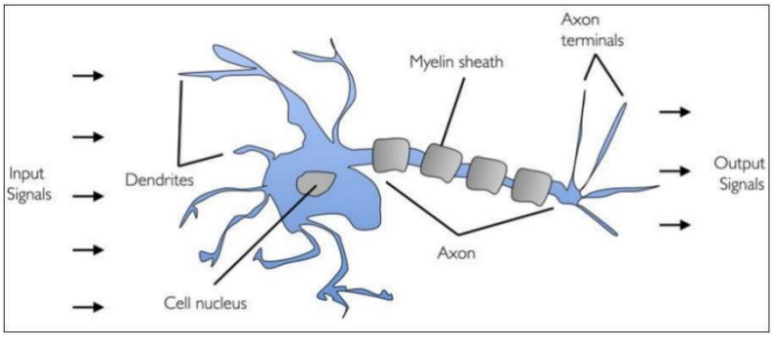
- **Neural networks and Deep Learning**




  - In 1957, Rosenblatt invented perceptron which is like logistic regression and loosely inspired from neurobiology.
  - A perceptron mimics certain parts of neurons, such as dendrites, cell bodies and axons using simplified mathematical models.
  - Working of neurons: signals can be received from dendrites, and sent down the axon once enough signals were received. This outgoing signal can then be used as another input for other neurons, repeating the process.
  - Some signals are more important than others and can trigger some neurons to fire easier. Connections can become stronger or weaker, new connections can appear while others can cease to exist.
  - We can mimic most of this process by a function that receives a list of weighted input signals and outputs signal if the sum of these weighted inputs reaches a certain threshold.
  - This simplified model does not mimic the creation or the destruction of connections (dendrites or axons) between neurons, and ignores signal timing. However, this restricted model alone is powerful enough to work with simple classification/regression tasks.
  - In 2012, For a competition at Stanford for recognizing objects out of an image (dataset consist of millions of images), Deep learning algorithms were applied and subsequently it was widely used by Google, Amazon, Facebook, etc. for applications such as voice assistant, self-driving cars, etc.
- **How Biological Neurons work?**



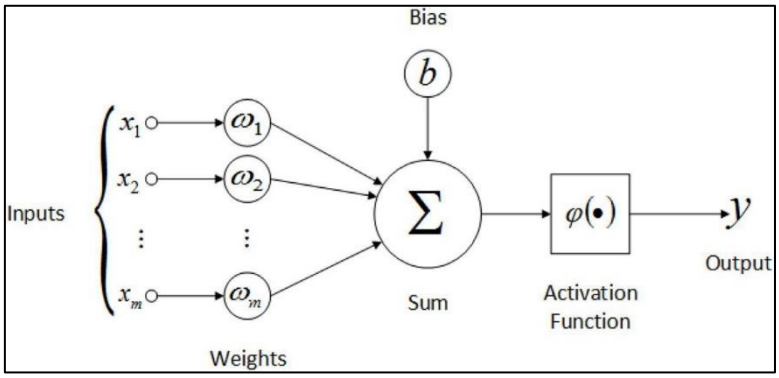

  - Dendrites receive information or signals from other neurons that get connected to it. Some dendrites are thicker which depicts the inputs which matter more while processing.
  - Information processing happens in a cell body. These take in all the information coming from the different dendrites and process that information. The processing happens only when there is enough input then only cell is activated.
  - Axon sends the output signal to another neuron for the flow of information. Here, each of the flanges connects to the dendrite or the hairs on the next one.
- **Logistic Regression and Perceptron**
  - Geometrically, Logistic regression tries to find a hyperplane which separates two classes. Here to find $\hat{y}_i$ we apply sigmoid function s.t. $\hat{y}_i = sigmoid\ (W^T x_i + b)$. we try to find $W$ & $b$.
  - Same concept is applied in perceptron. Perceptron Learning Rule states that the algorithm would automatically learn the optimal weight coefficients based on the input data. The input features are then multiplied with these weights to determine if a neuron activates or not.

$$f(x) = \sum_{j=0}^{d} w_j x_{ij} + b$$

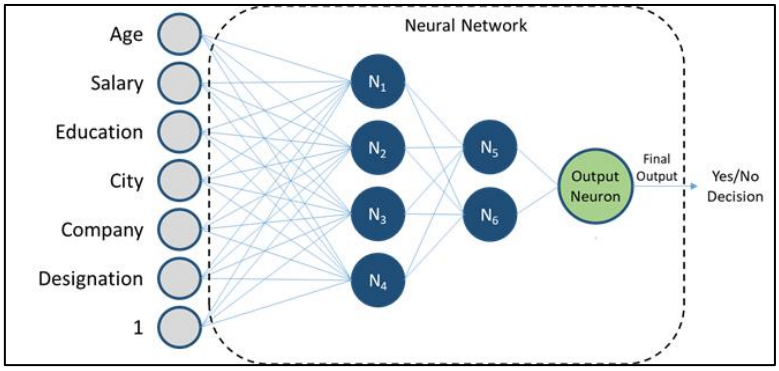
- The Perceptron receives multiple input signals, and if the sum of the input signals exceeds a certain threshold, it either outputs a signal or does not return an output. this can then be used to predict the class of a sample for a classification task.
- Perceptron is a function that maps its input $x$ which is multiplied with the learned weight coefficient; an output value $f(x)$is generated. The following example is single neuron

$$f(x) = \begin{matrix} 1 \; if \; w.x + b \; > 0 \\ 0 \; else \end{matrix}$$

- Similarly, for artificial neurons it works as follows. The more important input is given more weight and passed through an activation function to get the output. The activation function is $f(w, x)$

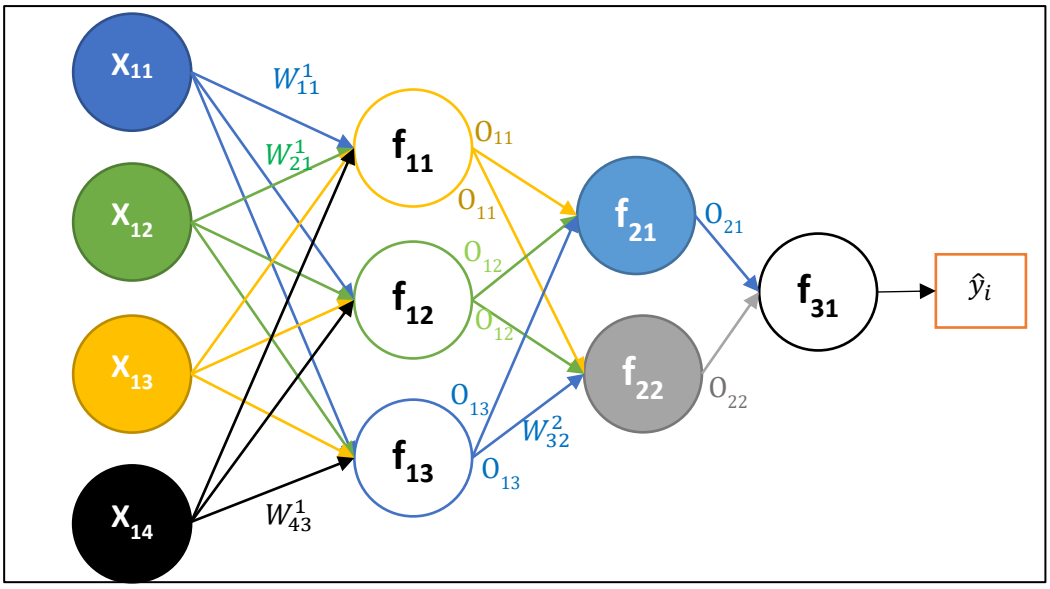



- **Multi-Layer Perceptron**

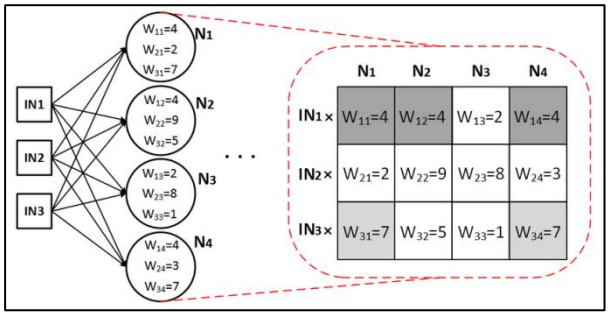



- MLP is an interconnected neuron where there are hidden layers between input & output.
- Every neuron in a layer will have a different function & it's total output will be passed to the next layer. This is parallel to how actual human brain works.
- There will be multiple functions like add, subtract, square, multiply, etc.
- Multilayer neural network is graphical way of representing multilevel function composition where each function is function of something else.
  E.g., $f_1(x) = \log x$ & $f_2(x) = a * x$ we can interpret $\log 4x$ as $f_1(f_2(4, x))$

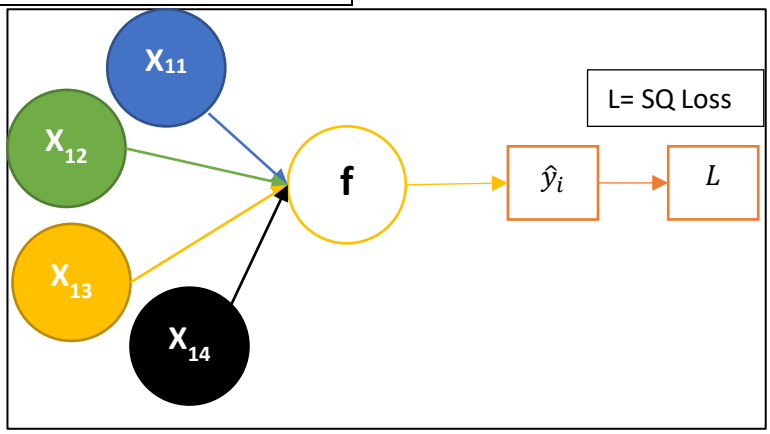
- **Notation**



- In above fig., $x$ is the input. It is passed through first layer for computation the output of this first layer will be sent to every second layer neuron as input for further computation and after this we get the final output.
- The notation are as follows input $x_{ij}$ which represents $i^{th}$ input & it's $j^{th}$ feature.
- Function $f_{11}$ will represent function of 1st layer and 1st function.
- The output $o_{11}$ will represent output of 1st layer and 1st function.
- The edges $w_{11}^{1}$ represents weight of 1st layer where point going from 1st point in input layer to 1st point in first layer
- Weight Matrix is a matrix representation of a weights between two layers




- **Training a single-neuron model**
  - Training is a basically finding best edge weights using training data
  - Problem: $\boldsymbol{D = \{x_i, y_i\}_{i=1}^{n}\ x_i \in \mathbb{R}^4,\ y_i \in \mathbb{R}}$



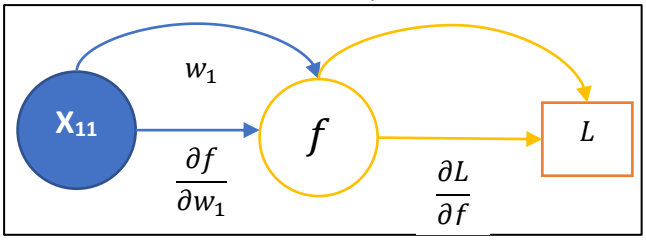

  - The above formulation is like linear regression where we have actual $y$ which is a scaler value & we will calculate the predicted $\hat{y}$ by using $W^T.X = \sum_{j=1}^{d} W_j.x_{ij}$
  - Loss for one point is $L_i = (y_i - \hat{y}_i)^2$
  - Total loss will be $\sum_{i=1}^{n} L_i$
  - The ultimate task will be to find weights such that the loss will be minimum

$$W^* = \underset{w}{argmin} \sum_{i=1}^{n} (y_i - f(W^T x_i))^2 + reg.$$

  - We solve it by using SGD
    - Initialization of random weights $W = [w_1\ w_2\ w_3\ w_4]$
    - Compute derivative $\nabla_W L = \begin{bmatrix} \frac{\partial L}{\partial w_1} \\ \frac{\partial L}{\partial w_2} \\ \frac{\partial L}{\partial w_3} \\ \frac{\partial L}{\partial w_4} \end{bmatrix}$ $as\ d = 4$
    - Vector representation: $w_{new} = w_{old} - \eta[\nabla_W L]_{w_{old}}$ $\eta = learning\ rate$
      Scalar representation: $(w_i)_{new} = (w_i)_{old} - \eta \left[\frac{\partial L}{\partial w_3}\right]_{(w_i)_{old}}$
    - By chain rule of differentiation,

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial f} * \frac{\partial f}{\partial w_1}$$

Intuitively we are taking path from $w_1$ to $L$ through $f$



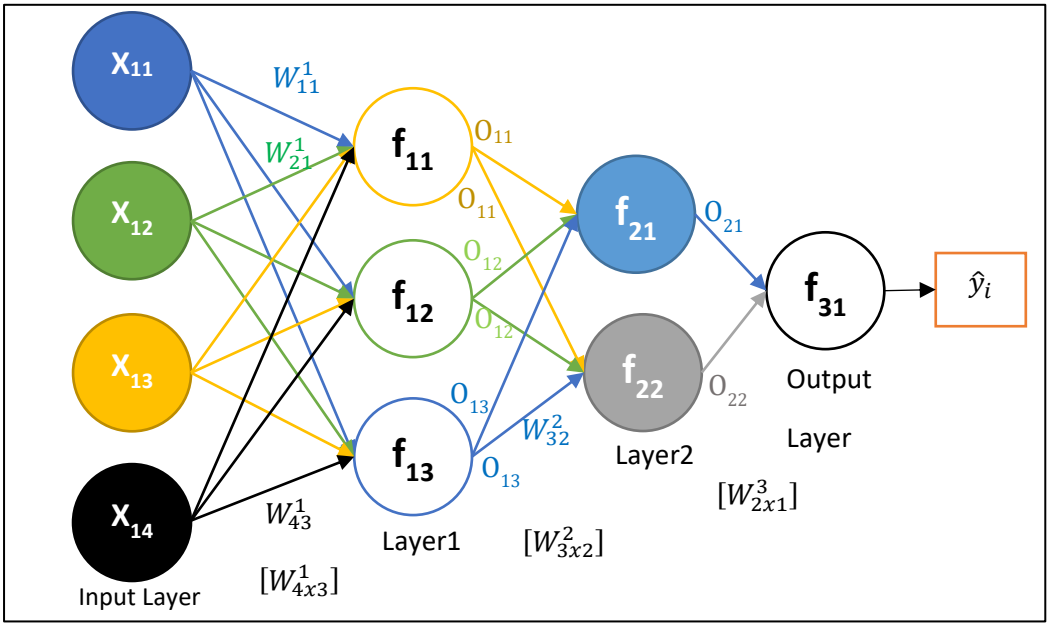

$$L = \sum_{i=1}^{n} (y_i - f(W^T x_i))^2$$

$$\frac{\partial L}{\partial f} = -\sum_{i=1}^{n} 2(y_i - f(W^T x_i))$$

$$\frac{\partial f}{\partial w_1} = x_i$$

$$\frac{\partial L}{\partial w_1} = -\sum_{i=1}^{n} 2x_i(y_i - f(W^T x_i))$$

- **Training an MLP: Chain Rule**




  - Here our main task is to find weight matrix $W_1, W_2, W_3$ such that

$$W^* = \underset{w1,w2,w3}{argmin} \sum_{i=1}^{n} (y_i - f(W^T x_i))^2 + reg.$$

    - If we consider L2 Regularization

$$reg = \sum_{i,j,k} (w_{ij}^k)^2$$

    - If we consider L1 Regularization reg term is

$$reg = \sum_{i,j,k} |w_{ij}^k|$$

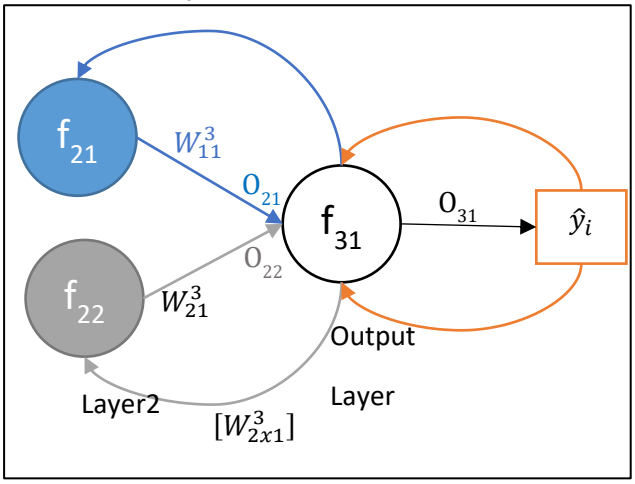
  - Let's consider optimization without regularization term, we can also represent it as

$$L = \sum_{i=1}^{n} L_i + reg$$

$$W^* = \underset{w_{ij}^k}{argmin}\ L$$

  - SGD
    - Initialize $w_{ij}^k$ randomly. Note there are more techniques for initiation
    - Update rule: $(w_{ij}^k)_{new} = (w_{ij}^k)_{old} - \eta \left[\frac{\partial L}{\partial w_{ij}^k}\right]_{(w_i)_{old}}$

- First, we will compute weight vector $W_3$

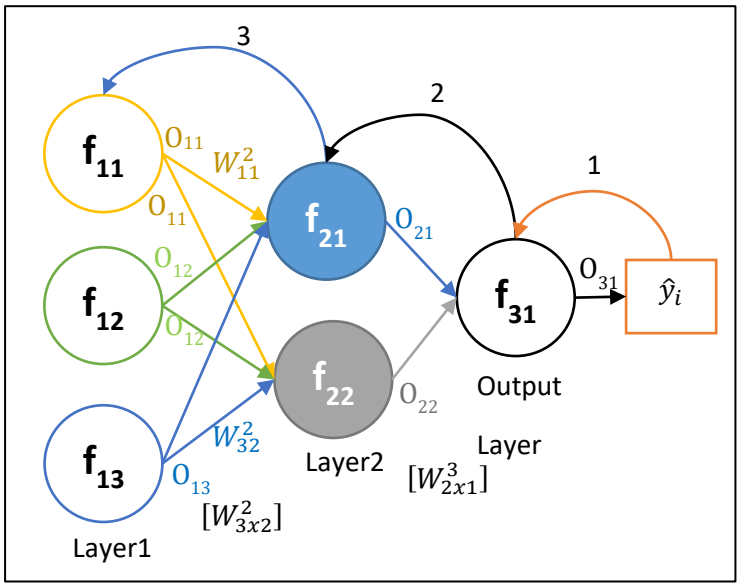



By chain rule,

$$\frac{\partial L}{\partial w_{11}^3} = \frac{\partial L}{\partial o_{31}} * \frac{\partial o_{31}}{\partial w_{11}^3}$$

$$\frac{\partial L}{\partial w_{21}^3} = \frac{\partial L}{\partial o_{31}} * \frac{\partial o_{31}}{\partial w_{21}^3}$$

- For $W_2$

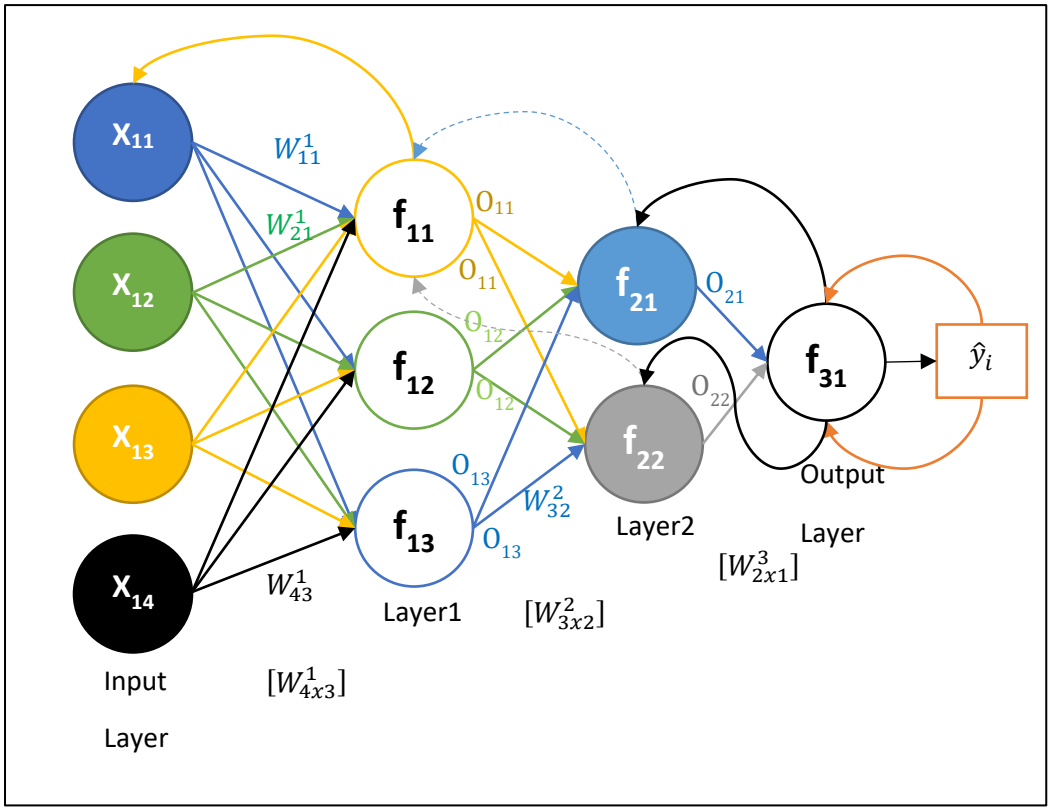



$$\frac{\partial L}{\partial w_{11}^2} = \frac{\partial L}{\partial o_{31}} * \frac{\partial o_{31}}{\partial o_{21}} * \frac{\partial o_{21}}{\partial w_{11}^2}$$

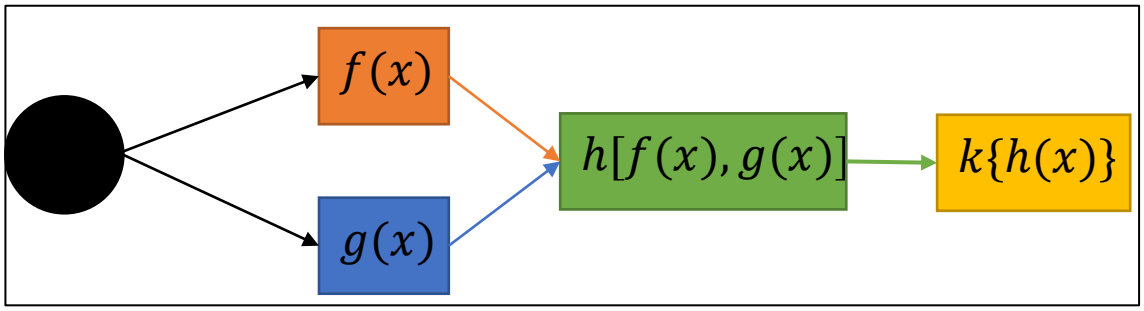
$$\frac{\partial L}{\partial w_{21}^2} = \frac{\partial L}{\partial o_{31}} * \frac{\partial o_{31}}{\partial o_{21}} * \frac{\partial o_{21}}{\partial w_{21}^2}$$

$$\frac{\partial L}{\partial w_{31}^2} = \frac{\partial L}{\partial o_{31}} * \frac{\partial o_{31}}{\partial o_{21}} * \frac{\partial o_{21}}{\partial w_{31}^2}$$

- For W1




If $x$ is passed through 2 functions & then outputs of these functions are passed through 3rd function and subsequently this 3rd output is passed through 4th function (Loss function)




In such case, chain rule says

$$\frac{\partial k}{\partial x} = \frac{\partial k}{\partial h} * \frac{\partial h}{\partial x}$$

$$\frac{\partial h}{\partial x} = \frac{\partial h}{\partial f} * \frac{\partial f}{\partial x} + \frac{\partial h}{\partial g} * \frac{\partial g}{\partial x}$$

$$\frac{\partial k}{\partial x} = \frac{\partial k}{\partial h} * \left\{\frac{\partial h}{\partial f} * \frac{\partial f}{\partial x} + \frac{\partial h}{\partial g} * \frac{\partial g}{\partial x}\right\}$$

In our problem, the equation shall be

$$\frac{\partial L}{\partial w_{11}^1} = \frac{\partial L}{\partial o_{31}} * \frac{\partial o_{31}}{\partial w_{11}^1}$$

$$\frac{\partial o_{31}}{\partial w_{11}^1} = \frac{\partial o_{31}}{\partial o_{21}} * \frac{\partial o_{21}}{\partial o_{11}} * \frac{\partial o_{11}}{\partial w_{11}^1} + \frac{\partial o_{31}}{\partial o_{22}} * \frac{\partial o_{22}}{\partial o_{11}} * \frac{\partial o_{11}}{\partial w_{11}^1}$$

$$\frac{\partial L}{\partial w_{11}^1} = \frac{\partial L}{\partial o_{31}} * \left\{\frac{\partial o_{31}}{\partial o_{21}} * \frac{\partial o_{21}}{\partial o_{11}} * \frac{\partial o_{11}}{\partial w_{11}^1} + \frac{\partial o_{31}}{\partial o_{22}} * \frac{\partial o_{22}}{\partial o_{11}} * \frac{\partial o_{11}}{\partial w_{11}^1}\right\}$$

- **Training an MLP: Memoization**
  - In computing, memoization is an optimization technique used to speed up computer programs by storing the results of expensive function calls and returning the stored result when the same inputs occur again. E.g., In above formulas lots of derivatives are common. We can compute them once & reuse it

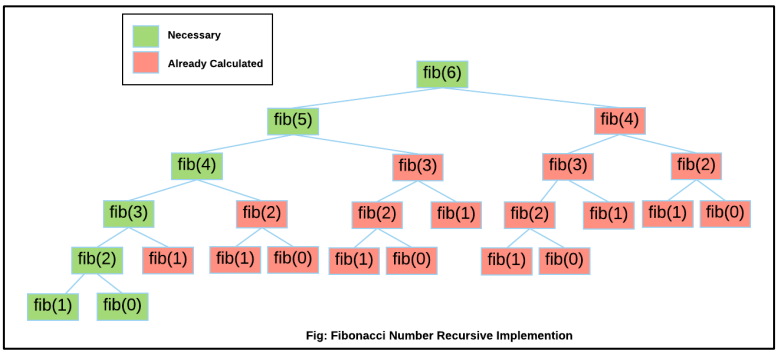


Fig: Fibonacci Number Recursive Implemention

- **Backpropagation**
  - Backpropagation is combination of memoization & chain rule. The procedure is as follows
    - Initialize $w_{ij}^{k}$
    - For each $x_i$ in $D$
      - ⇨ Forward Propagation: Pass $x_i$ forward through the network
      - ⇨ Compute Loss function $L(y_i, \hat{y}_i)$
      - ⇨ Compute all derivatives using chain rule & memoization
      - ⇨ Update the weights again from end to the start such that loss will be minimized
    - Repeat step2 till convergence
  - **Note:** backpropagation works only if activation functions are differentiable. If these functions are easily or fast differentiable it speeds up the training & Neural Network.
- **Activation Function - sigmoid**
  - The activation function is a non-linear transformation of the input before sending it to the next layer of neurons. The two most popular ones are **sigmoid** & **tanh**.
  - The sigmoid function is appealing because for any input value, output value will be between 0 and 1, which is useful to perform binary classification.
  - The sigmoid function is $\boxed{\sigma = \frac{1}{1+e^{-z}} = \frac{e^{z}}{1+e^{z}}}$ where $\boxed{z = \sum_i w_i . x_i = W^T X}$
  - Differentiation of sigmoid

    Sigmoid function is a case of $\sigma(g(z))$

    where $g(z) = 1 + e^{-z}$ and $\sigma(g) = (1 + e^{-z})^{-1} = (g)^{-1}$

    applying chain rule $\frac{\partial \sigma}{\partial z} = \frac{\partial \sigma}{\partial g} * \frac{\partial g}{\partial z} = \frac{\partial \sigma(g)}{\partial g} * \frac{\partial g(z)}{\partial z}$

    ⇨ $\frac{\partial}{\partial g}(g)^{-1} * \frac{\partial}{\partial z}(1 + e^{-z})$

    ⇨ $\frac{-1}{g^2} * \left[\frac{\partial}{\partial z}1 + \frac{\partial}{\partial z}e^{-z}\right]$

    ⇨ $\frac{\partial}{\partial z}e^{-z} = \frac{\partial}{\partial h}e^{h} * \frac{\partial}{\partial z} - z$ where $h = -z$

    ⇨ $\frac{\partial}{\partial z}e^{-z} = e^{h} * -1$

    ⇨ $\frac{-1}{g^2} * [0 + -e^{-z}]$

    ⇨ $\frac{-1}{g^2} * -e^{-z}$

    ⇨ $\frac{-e^{-z}}{g^2}$

    ⇨ $\frac{e^{-z}}{(1+e^{-z})^2}$

    ⇨ This can also be written as

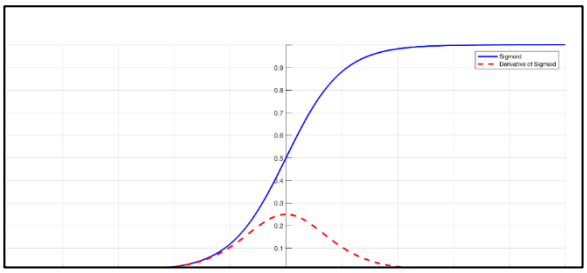
    ⇨ $\frac{1+e^{-z}-1}{(1+e^{-z})^2}$

    ⇨ $\frac{1+e^{-z}}{(1+e^{-z})^2} - \frac{1}{(1+e^{-z})^2}$

    ⇨ $\frac{1}{(1+e^{-z})} - \frac{1}{(1+e^{-z})^2}$

⇨ $\sigma(g) - (\sigma(g))^2$

⇨ $\sigma(g)(1 - \sigma(g))$



- The important property of sigmoid function is its output value lie between 0 to 1 & it's derivative always lie between 0 to 0.25

- **Activation Function -tanh**
  - The tanh function and sigmoid activation function look very similar. While sigmoid function output values will between 0 and 1, Tanh output values will between -1 and 1.
  - $a = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$
  - ⇨ We can represent tanh as $(e^z - e^{-z}) * (e^z + e^{-z})^{-1}$
  - ⇨ By product rule of derivation
  - ⇨ $\frac{\partial}{\partial x} f(x) * g(x) = \left(\frac{\partial}{\partial x} f(x)\right) * g(x) + \left(\frac{\partial}{\partial x} g(x)\right) * f(x)$
  - ⇨ $\frac{\partial a}{\partial z} = \left\{\left[\frac{\partial}{\partial z}(e^z - e^{-z})\right] * (e^z + e^{-z})^{-1}\right\} + \left\{\left[\frac{\partial}{\partial z}(e^z + e^{-z})^{-1}\right] * (e^z - e^{-z})\right\}$

  Solving the first Part

  ⇨ $\left[\frac{\partial}{\partial z}(e^z - e^{-z})\right] * (e^z + e^{-z})^{-1}$

  ⇨ $[e^z - * (-e^{-z})] * (e^z + e^{-z})^{-1}$

  ⇨ $\frac{e^z + e^{-z}}{e^z + e^{-z}} = 1$

  To Solving the second Part, we can apply chain rule where,

  $g(z) = e^z + e^{-z}$ and $f(g) = (e^z + e^{-z})^{-1} = g^{-1}$

  ⇨ $\left[\frac{\partial}{\partial z}(e^z + e^{-z})^{-1}\right] * (e^z - e^{-z})$

  ⇨ $\left[\frac{\partial}{\partial g} g^{-1} * \frac{\partial}{\partial z} g(z)\right] * (e^z - e^{-z})$

  ⇨ $\left[\frac{\partial}{\partial g} g^{-1} * \frac{\partial}{\partial z} e^z + e^{-z}\right] * (e^z - e^{-z})$

  ⇨ $\left[-1g^{-2} * \left(e^z + (-e^{-z})\right)\right] * (e^z - e^{-z})$

  ⇨ $[-1g^{-2} * (e^z - e^{-z})] * (e^z - e^{-z})$

  ⇨ $\left[\frac{-1}{g^2} * (e^z - e^{-z})\right] * (e^z - e^{-z})$

  ⇨ As $g = e^z + e^{-z}$

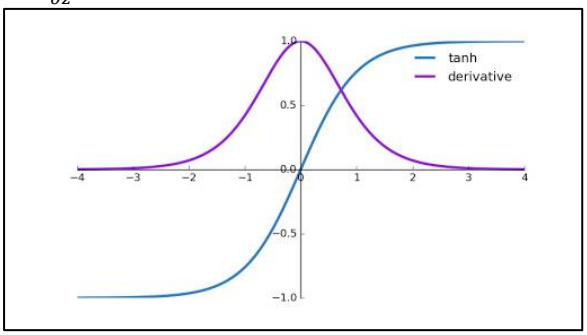
  ⇨ $\left[\frac{-1}{(e^z + e^{-z})^2} * (e^z - e^{-z})\right] * (e^z - e^{-z})$

  ⇨ $\left[\frac{-1 * (e^z - e^{-z}) * (e^z - e^{-z})}{(e^z + e^{-z})^2}\right]$

  ⇨ $\left[\frac{-(e^z - e^{-z})^2}{(e^z + e^{-z})^2}\right]$

  ⇨ $-\tanh(z)^2 = -a^2$

  ⇨ $\frac{\partial a}{\partial z} = 1 - a^2 = 1 - \tanh z^2$

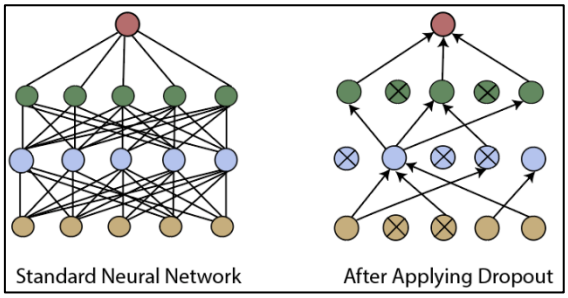



  - The property of tanh curve is that the tanh value lies between -0 & 1 but its derivative value lies between 0 & 1

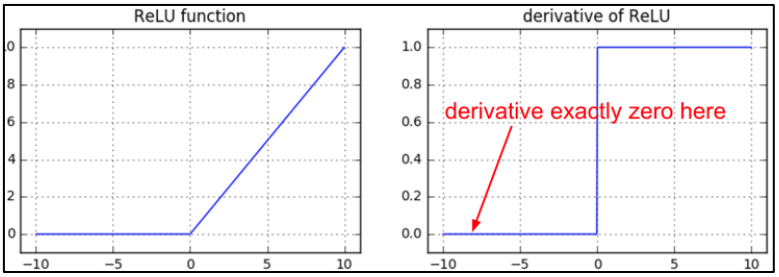
- **The Vanishing/Exploding Gradient**

- In a network of $n$ hidden layers, $n$ derivatives will be multiplied together. If the derivatives are large then the gradient will increase exponentially as we propagate down the model until they eventually explode, and this is what we call the problem of exploding gradient.
- Alternatively, if the derivatives are small then the gradient will decrease exponentially as we propagate through the model until it eventually vanishes, and this is the vanishing gradient problem.

- **Bias Variance Trade Off**
  - As we have lot of weights to train, the model will overfit. It will have high variance.
  - To overcome this overfitting problem, we add regularization term
- **Dropout layers & Regularization**
  - Dropout is a simple method for regularizing in order to avoid overfitting**.** Through dropout we can reduce the model capacity so that our model can achieve lower generalization error.
  - In Random Forest, we have decision trees trained by subset of features. We take majority vote for final prediction.Although, each of these base learners are overfit but by randomization of features & majority vote we are regularizing this overfitting
  - The concept of randomization for regularization can also be applied to Neural Networks

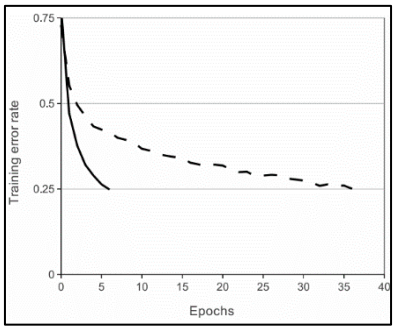



  - In dropout we don't use all neurons but only turn on some neuron in each training iteration wit probability $p$.
  - Dropout rate is number of neurons will be dropped out of total.
  - This idea can be connected to random forest such that at any neuron we only have random subset of features as input.
  - At train time the dropout will be applied with probability of edge being present. At test time all neurons will be connected except the output weights will be multiplied with probability from train.
  - If we have case of very deep network where no of weights is more than no of data points. There will be higher chances of overfitting. In such case we should keep dropout rate large
- **Rectified Linear Units (ReLU)**
  - The major problem with sigmoid & tanh is vanishing gradient & due to that, convergence slows down as the difference between updated weights & new weights is less.
    - After training we get
    - $W^T.X = \sum_{j=1}^{d} W_j.x_{ij} = \mathbb{Z}$
    - ReLU is defined as $f(\mathbb{Z}) = \mathbb{Z}^+ = \max(0, \mathbb{Z}) = \begin{cases} 0 \; if \; \mathbb{Z} \leq 0 \\ else \; \mathbb{Z} \end{cases}$



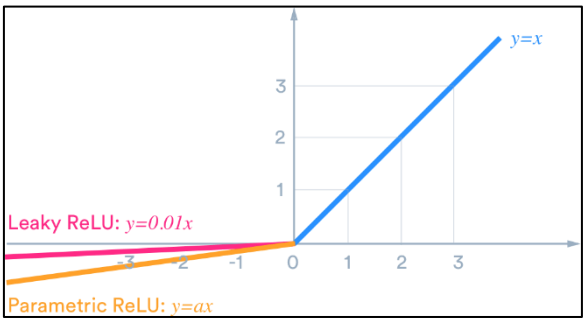

  - Here the derivative will be either 1 or 0 and this solves the problem of exploding or vanishing gradient.
  - If $\mathbb{Z}$ is negative its derivative becomes zero & ultimately the chain rule output is zero which implies that there will be no difference between weights. This is a dead activation problem. The solution is to keep count of dead activation functions out of total and keep an eye on it.

- **Why use ReLU?**

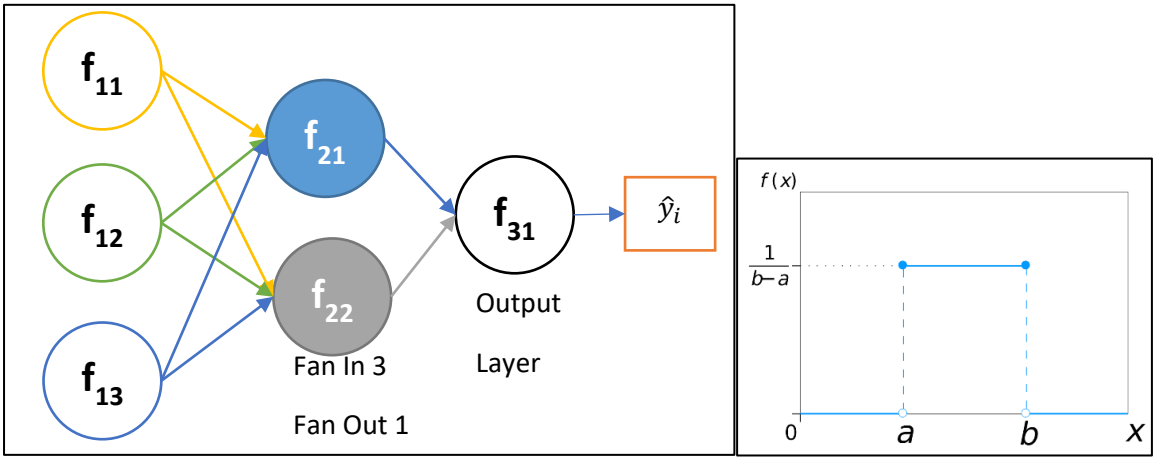



- A four-layer CNN with ReLUs (solid line) reaches a 25% training error rate on CIFAR-10 six times faster than a same network with tanh neurons (dashed line). The learning rates for each network were chosen independently.
- Networks with ReLUs consistently learn several times faster than equivalents with saturating neurons.
- It Speeds up convergence
- ReLU function & derivative functions are easy to compute.
- Sparsity in ReLU arises when$\mathbb{Z} \leq 0$. If more units have $\mathbb{Z} \leq 0$, the result representation will be sparser
- While as Sigmoid always generate some non-zero value resulting in dense representations. Sparse representations are more beneficial than dense representations.

- **Parametric & Leaky ReLU**
  - In a parametric ReLU, instead of making derivative 0 when $\mathbb{Z}$ is negative, We will make it some $a * \mathbb{Z}$ where $a$ is hyper parameter.
  - For leaky ReLU, $a = 0.01$




- **Weight Initialization**
  - The constraints that we need to follow while initiating the weights are
    - Weights should neither be too small nor too large
    - All weights should not be same cause if the activation functions are same its output will be same
    - There should be good variance in the weights
  - **Gaussian/Normal Initialization**: weights come from set $N(0, \sigma)$ 0 mean & a small standard deviation.
    - Here the variance will be $\sigma^2$
    - But some values will be positive & Negative
  - **Uniform Initialization**: weights belongs to uniform distribution of $\left(\frac{-1}{\sqrt{fan\ in}}, \frac{1}{\sqrt{fan\ in}}\right)$ Uniform initialization works well for sigmoid function

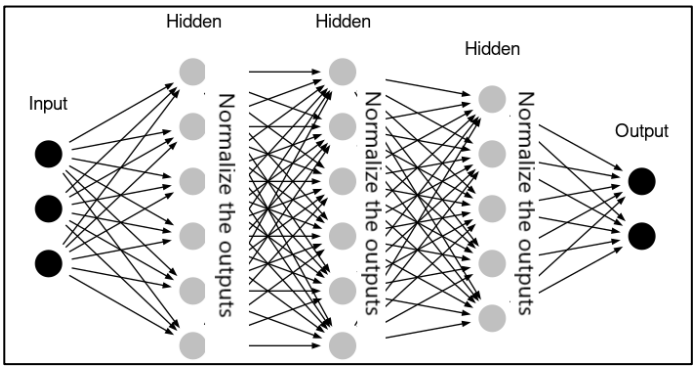



  - **Xavier glorot initialization**

- **Uniform distribution**: Value in weights is in $(-x, x)$ where $x = \sqrt{\left(\frac{6}{fan\ in + fan\ out}\right)}$
- **Normal distribution:** Value in weights is in $N(0, \sigma)$ and $\sigma = \sqrt{\left(\frac{2}{fan\ in + fan\ out}\right)}$
- Works well with sigmoid function

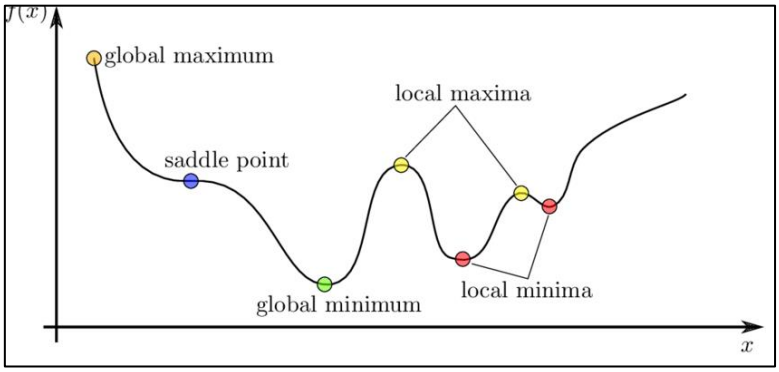
  - **He initialization**
    - **Uniform distribution**: Value in weights is in $(-x, x)$where $x = \sqrt{\left(\frac{6}{fan\ in}\right)}$
    - **Normal distribution:** Value in weights is in $N(0, \sigma)$and $\sigma = \sqrt{\left(\frac{2}{fan\ in}\right)}$
    - Works well with ReLU function

- **Batch Normalization**

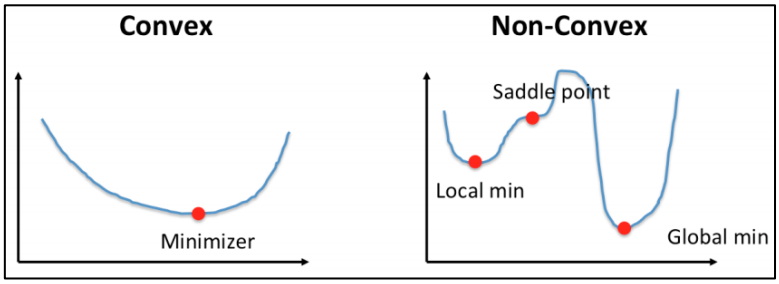



  - Batch Normalization converts interlayer outputs into a standard format by normalizing, by subtracting the batch mean, and then dividing by the batch's standard deviation.
  - It effectively resets the distribution of previous layer's output to be more efficiently processed by the subsequent layer.
  - This leads to faster learning as normalization ensures that no activation value is too high or too low, also allowing each layer to learn independently of the others.
    - Normalize output from activation function
    $$z = \frac{x - m}{s}$$
    - Multiply by arbitrary parameter $g$
    $$z * g$$
    - Add an arbitrary parameter $b$
    $$(z * g) + b$$
  - These parameters are trainable and optimized
  - Advantages:
    - Reduces internal covariant shift.
    - Reduces the dependence of gradients on the scale of the parameters or their initial values.
    - Regularizes the model and reduces the need for dropout, photometric distortions, local response normalization and other regularization techniques.
    - Allows use of saturating non-linearities and higher learning rates.

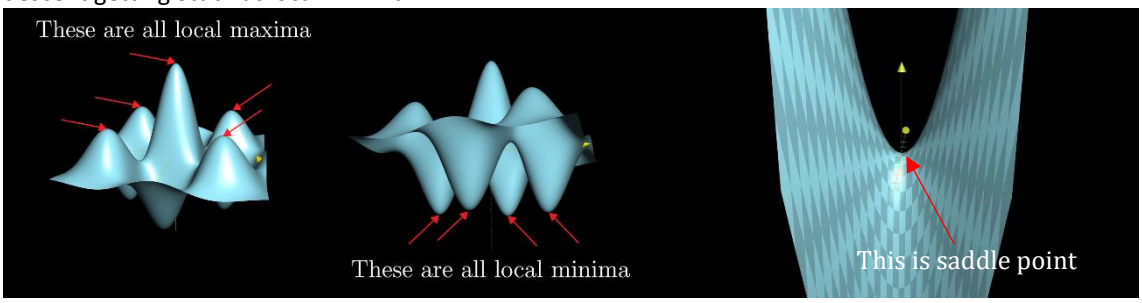
- **Optimizers: Hill-descent analogy in 2D**
  - In SGD or GD, we update the weights till there is no change. if derivative is zero there will be no update in weights
  - The derivative for a function will become zero at minima, maxima and saddle point



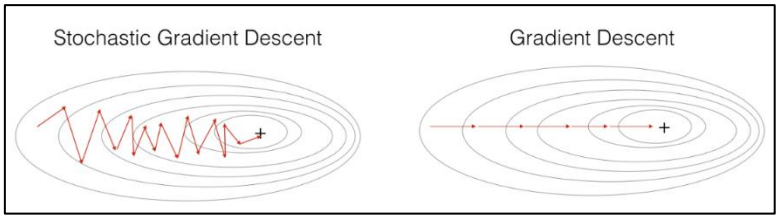

  - There are two types of functions: convex & non convex




  - Convex functions have one minima or maxima. Here there is no local minima
  - Non-Convex Function has One global minimum and multiple local minima.
  - The loss function for Linear regression, Logistic Regression, SVM is a convex function e.g. MSE, Hinge loss
  - The Loss function for DL algorithms such as MLP is a non convex function. Here,there are chances of gradient descent getting Stuck at local minima




- **SGD vs GD Convergence**




  - In both gradient descent (GD) and stochastic gradient descent (SGD), we update a set of parameters in an iterative manner to minimize an error function.
  - While in GD, we run through ALL the samples in training set to do a single update for a parameter in a particular iteration.
  - In SGD, we use ONLY ONE or SUBSET of training sample for the update. If we use SUBSET, it is called Mini batch Stochastic gradient Descent.
  - Thus, if the number of training samples are very large, gradient descent will take too long for update. While as SGD will be faster because it uses only one training sample and it starts improving itself right away from the first sample.
  - SGD often converges faster compared to GD but the error function is not as well minimized as compared to GD.
  - Also, the number of iterations will be more in case of SGD as compared to GD.

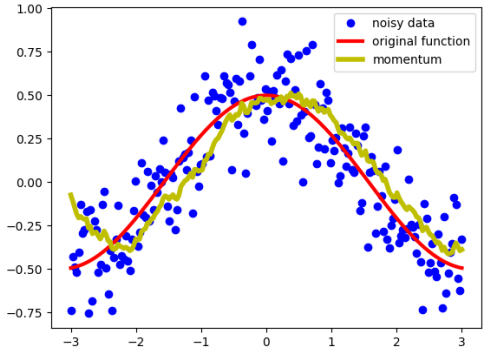
- Often in most cases, the close approximation that we get in SGD for the parameter values are enough because they reach the optimal values and keep oscillating there.
- For an example of this with a practical case, check cs229-notes
  1. What is the reason for the zig zag path for convergence in SGD?

  ⇨ It is because as we are using only batches of data in SGD, the path of convergence goes in zig-zag fashion. When we choose GD, the path of convergence will be smoother but the speed of convergence is slow.

  2. Why are we worried about the intermediate updates to the Weights in SGD when we finally approximately converge to optimum W?

  ⇨ The reason why we are worried about intermediate updates is that the scale of the features gets changed and the path of convergence goes in zig-zag way, it will give faster update as compared to GD but takes a greater number of epochs. Hence, we perform batch normalization to get all the features onto the same scale and, we go with variants of SGD in order to make the path smoother. (Note: The path of convergence in variants of SGD is smoother than that of simple batch SGD, but not as smooth as that of GD)

- **Stochastic Gradient Descent with momentum**
  - SGD with momentum is method which helps accelerate gradients vectors in the right direction, thus leading to faster converging. Here we use Exponentially weighed averages for updating weights**.**
  - Exponentially weighed averages deal with sequences of numbers. Suppose, we have some **sequence S** which is noisy and If we plot cosine function and added some Gaussian noise.
  - Although these dots seem very close to each over, none of them share $x$ coordinate. It is a unique number for each point. That's the number that defines the index of each point in our **sequence S**.
  - Instead of using original data, we want some kind of 'moving' average which would 'denoise' the data and bring it closer to the original function.




  - The moving average gives a **smoother line** which is closer to original function instead of noised data. Exponentially weighed averages define a new sequence $v_t$

$$v_t = 1 * a_t + \gamma * a_{t-1} + \gamma^2 * a_{t-2} + \gamma^3 * a_{t-3} + \gamma^4 * a_{t-4} + \cdots$$
$$where\ v_t = denoised\ estimate\ \&\ 0 \leq \gamma \leq 1$$

  - In the above equation we are giving more weightage to the recent points as compared to the other ones. In practice, typically $\gamma = 0.9$
  - An alternate explanation will be **blue** points are the points which are output of SGD and ideally it should've been like **red curve** which can be thought as GD. After applying momentum, we get the **yellow curve** which is closer to the GD.
  - Derivation

⇨ $w_t = w_{t-1} - \eta \left(\frac{\partial L}{\partial w}\right)_{w_{t-1}}$

⇨ Let's call $g_t = \left(\frac{\partial L}{\partial w}\right)_{w_{t-1}}$

⇨ $w_t = w_{t-1} - \eta * g_t$

Applying exponential weighing concept,

⇨ $v_t = \gamma * v_{t-1} + \eta * g_t$

⇨ $w_t = w_{t-1} - v_t$

⇨ **Case1:** $\gamma = 0$ (Vanilla SGD)

⇨ $v_t = 0 * v_{t-1} + \eta * g_t$

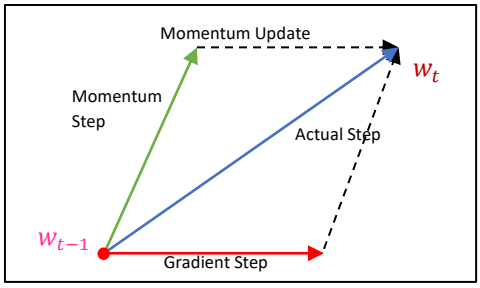
⇨ $w_t = w_{t-1} - \eta * g_t$

⇨ **Case2:** $\gamma = 0.9$

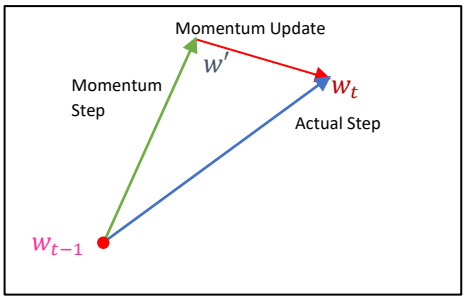
⇨ $v_t = 0.9 * v_{t-1} + \eta * g_t$

⇨ $w_t = w_{t-1} - 0.9 * v_{t-1} - \eta * g_t$

- Here $\eta * g_t$ is a gradient term while $0.9 * v_{t-1}$ is a momentum term.
  $v_t = 1 * \eta * g_t + \gamma * \eta * g_{t-1} + \gamma^2 * \eta * g_{t-2} + \gamma^3 * \eta * g_{t-3}$
- SGD with momentum will speed up the convergence with significant reduction in no of epochs
- Intuitively, gradient step says move in direction of red arrow & momentum says move in direction of green arrow. The actual step will be resultant of these two vectors.



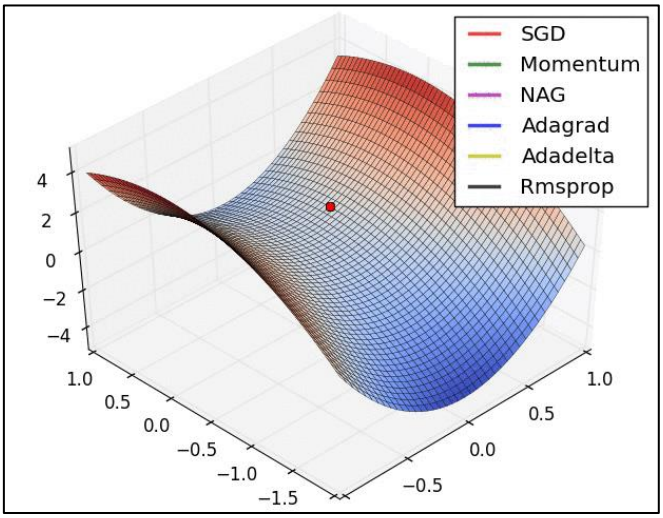

- **Nesterov Accelerated Gradient**




- NAG says first move into momentum direction & then compute the gradient at $w'$
- Mathematically,
  - ⇨ $w_t = w_{t-1} - (\gamma * v_{t-1} + \eta * g') \; ; \; g' \neq g_t$
  - ⇨ $g' = \left(\frac{\partial L}{\partial w}\right)_{(w')}$
  - ⇨ $w' = w_{t-1} - \gamma * v_{t-1}$
- **AdaGrad (Adaptive Gradients)**
  - In SGD, learning rate was constant for each weight update. Although if features are sparse having constant learning rate is not recommended.
  - Adagrad adapts the learning rate for every weight at every iteration
    - ⇨ $w_t = w_{t-1} - \eta'_t * g_t$
    - ⇨ $\eta'_t = \frac{\eta}{\sqrt{\alpha_t + \epsilon}}$
    - ⇨ $\eta$ = constant learning rate = 0.01
    - ⇨ $\alpha_t = \sum_{i=1}^{t-1} g_i^2$
    - ⇨ $g_t = \left(\frac{\partial L}{\partial w}\right)_{w_{t-1}}$
    - ⇨ $\epsilon$ = small positive number added to avoid division by zero problem
  - Intuitively, at each iteration as no of iterations increase $\alpha_t$ increases and which decreases $\eta'_t$.
  - The problem with Adagrad is that with every iteration $\alpha_t$ becomes very large & impacts $\eta'_t$ . since learning rate is small, there is not much change in $w_t$. This makes Adagrad to converge slower
  - Q: we are changing the learning rate ($\eta'$). but nothing changes in $g_t$. suppose we have a saddle point. there $g_t = 0$, in that case $\eta'_t * g_t = 0$. that means we get stuck at saddle point. then how adagrad or even adadelta fix saddle point problem?
  - ⇨ The primary reason we use Adagrad is to have per-weight update which considers the feature sparsity into account. We can even get simple SGD to avoid Saddle points by adding a very simple randomization as follows: $w_{new} = w_{old} - n * \left(\frac{dL}{dw}\right) + \epsilon$ where $\epsilon$ is a small random normal $N(0,1)$ value. This ensures that a random movement in some direction even when $\frac{dL}{dw} = 0$ at the saddle point. The same trick is used in other variants of SGD also. This is explained very well in this blog.
- **AdaDelta**
  - Adadelta is an extension of Adagrad that seeks to reduce its aggressive, monotonically decreasing learning rate. Instead of accumulating all past squared gradients, Adadelta restricts the window of accumulated past gradients to some fixed size $w$

- Instead of inefficiently storing all previous squared gradients, the sum of gradients is recursively defined as an exponential decaying average($eda_t$) of all past squared gradients.
  - ⇨ $\eta'_t = \frac{\eta}{\sqrt{eda_t + \epsilon}}$
  - ⇨ $eda_t = \gamma * eda_{t-1} + (1-\gamma) {g_t}^2$
  - ⇨ Typically, $\gamma = 0.95$
  - ⇨ $0.95 * eda_{t-1} + (0.05) {g_t}^2$
  - ⇨ $0.95 * [0.95 * eda_{t-2} + (0.05) {g_{t-2}}^2] + (0.05) {g_{t-1}}^2$

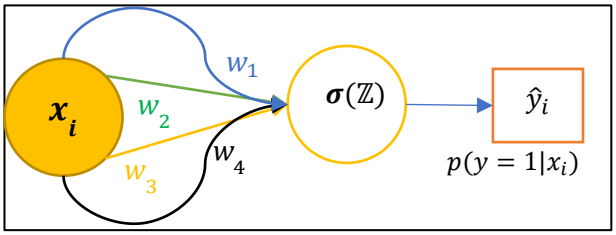
- **Adam (adaptive moment estimation)**
  - In Adadelta/Adagrad, we considered $g_t^2$ to calculate $\eta'_t$. In addition, Adam uses $g_t$
  - In statistics mean is considered as 1st order moment & variance as 2nd order moment
    - ⇨ eda of mean at time t: $m_t = \beta_1 * m_{t-1} + (1-\beta_1) * g_t \; ; 0 \leq \beta_1 \leq 1$
    - ⇨ eda variance at time t: $v_t = \beta_2 * v_{t-1} + (1-\beta_2) * {g_t}^2 \; ; 0 \leq \beta_1 \leq 1$
    - ⇨ $\hat{m}_t = \frac{m_t}{1-(\beta_1)^t}$
    - ⇨ $\hat{v}_t = \frac{v_t}{1-(\beta_2)^t}$
    - ⇨ $w_t = w_{t-1} - \alpha * \frac{\hat{m}_t}{\sqrt{\hat{v}_t + \epsilon}}$
    - ⇨ $\beta_1 = 0.9\,,\ \beta_2 = 0.99,\ \alpha = 1$
- **Comparison between all optimizers**



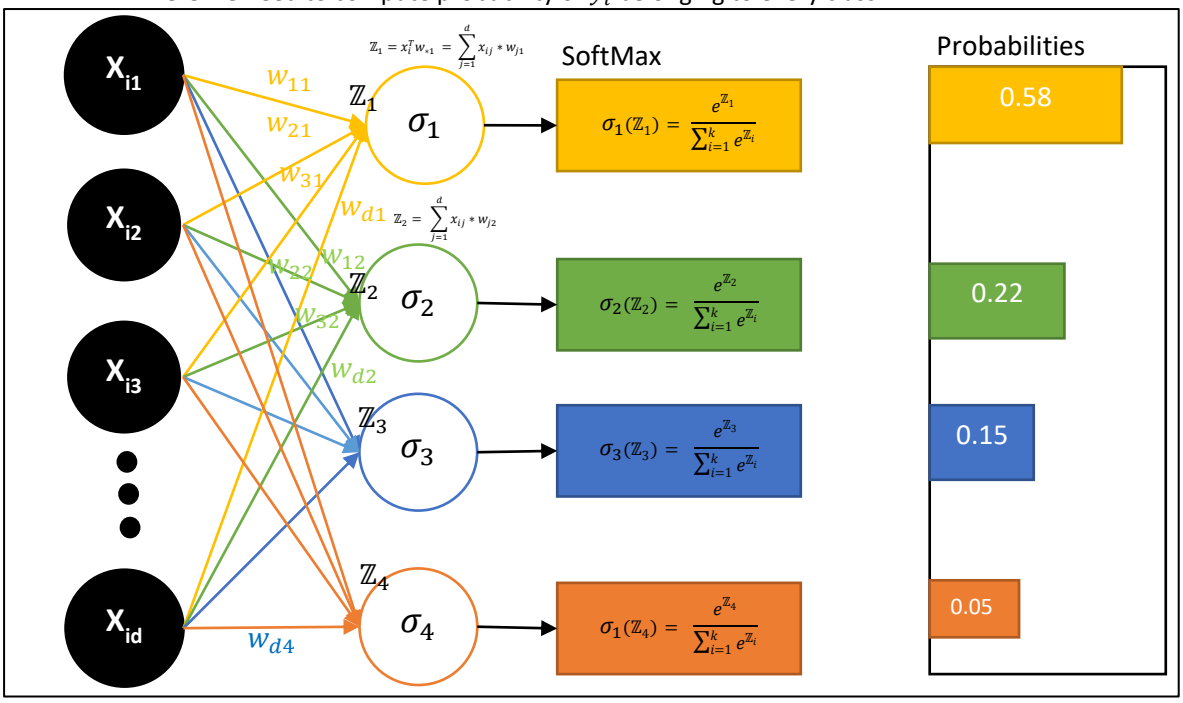

  - Adadelta is faster compared to Adagrad as learning rate $\eta$ does not become too low but there is no guarantee either of them will be able to get out of saddle point.
  - In most cases, ADAM performs better as it combines adaptive learning and momentum methods. But for some datasets, adaptive learning algorithms don't work well.
  - Momentum methods also helps in getting out of saddle point as it affects the direction of descent but it may tend to oscillate as larger $\eta$ may lead to jump over the minima/valley like seen in linear regression. But otherwise, it's faster.
  - Q: if there are no saddle points is Momentum based methods faster than adaptive learning-based methods?
  - ⇨ It seems true as momentum add additional descent that too with less zig zag motion or to say shortest path. On the other hand, adaptive learning keeps on shortening descent increments for each iteration.
- **Gradient Clipping**
  - Gradient clipping is a technique that tackles exploding gradients. If the gradient gets too large, we rescale it to keep it small.
  - $G$ is a vector of gradients.
    - ⇨ $G_{new} = \frac{G}{\|G\|_2} * \tau$
    - ⇨ $\|G\|_2 = \sqrt{G_1^2 + G_2^2 + G_3^2 + \cdots}$
  - where $\tau$ is a hyper parameter, since $\frac{\boldsymbol{G}}{\|\boldsymbol{G}\|_2}$ is a unit vector, after rescaling the new $\boldsymbol{G}$ will have norm $\tau$. Note that if $\|\boldsymbol{G}\| < \tau$, then we don't need to do anything.

- **SoftMax and Cross-entropy for multi-class classification**

- Logistic regression solves binary classification problem. For multi class problems we use one vs all.
- SoftMax is generalization of Logistic Regression to solve multi class problems
- In Logistic regression




  - $\mathbb{Z} = W^t X$
  - $p(y = 1|x_i) = \hat{y} = \sigma(\mathbb{Z})$
  - $\sigma(\mathbb{Z}) = \frac{1}{1+e^{-\mathbb{Z}}} = \frac{e^{\mathbb{Z}}}{e^{\mathbb{Z}}+1}$
- In SoftMax,
  - $D = \{x_i, y_i\}\ where\ y_i \in \{1,2,3,4, \dots, k\}$
  - Here we need to compute probability of $y_i$ belonging to every class

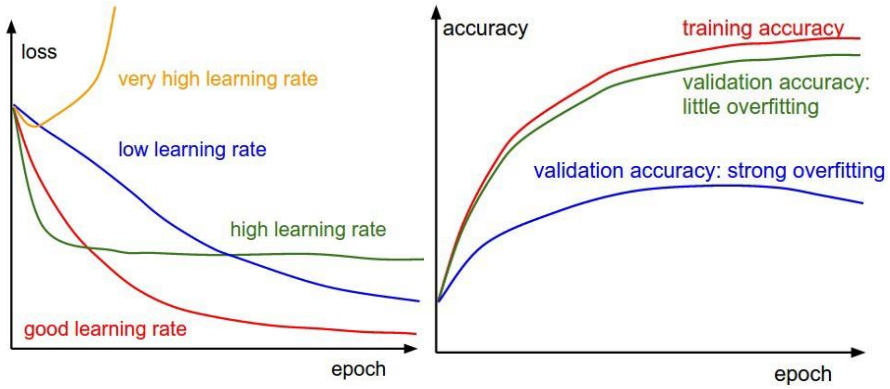



  - Each of the points will be going through $\sigma_1, \sigma_2, \sigma_3, \dots, \sigma_k$ which is a function of $\mathbb{Z}_k$
  - $\mathbb{Z}_1 = \sum_{j=1}^{d} x_{ij} * w_{j1}$ , $\mathbb{Z}_2 = \sum_{j=1}^{d} x_{ij} * w_{j2}$
  - $\sigma_1(\mathbb{Z}_1) = \frac{e^{\mathbb{Z}_1}}{\sum_{i=1}^{k} e^{\mathbb{Z}_i}}$
- As logistic Regression minimizes log loss. similarly, SoftMax minimizes multi class log loss

$$multi\text{-}class\ Log\ loss = \frac{-1}{N}\sum_{i=1}^{N}\sum_{j=1}^{K} y_{ij} * \log\left(p_{ij}\right)$$

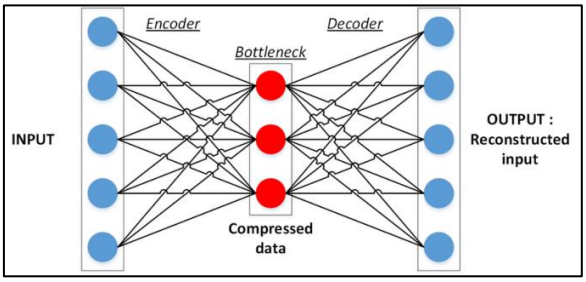
- **How to train a Deep MLP?**
  - Pre-process: Normalize the data
  - Weight Initialization:
    - Xavier/glorot for sigmoid / tanh
    - He Initialization for ReLU
    - Gaussian with small variance $\sigma$
  - Choose activation function: ReLU (Best as of 2018)
  - Batch Normalization
  - Dropout
  - Optimizer: Adam (Best as of 2018)
  - Hyper parameter tuning

        - Architecture: No of layers & no of neurons in each layer: Number of neurons in any layer is a hyper parameter to be tuned. One rule of thumb is to have fewer neurons in later layers as compared to earlier layers.
        - Dropout rate
        - Adam: $\beta_1, \beta_2, \alpha$
    - Loss Function
        - Binary classification: Log Loss
        - Multi Class Classification: Multi Class Log Loss
        - Regression: SQ Loss
    - Keep monitoring gradients & perform clipping
    - Plots: Epoch vs Loss, Epoch Vs Accuracy

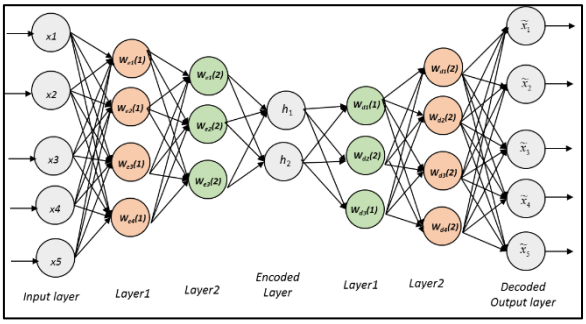



    - Avoid Overfitting: It is so easy for MLP to overfit

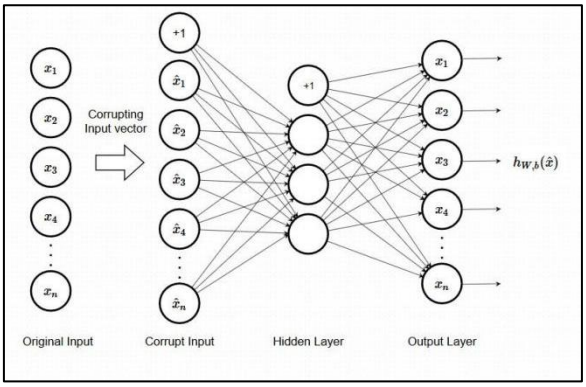
- **Auto Encoders**
    - An auto encoder is a type of ANN used to learn efficient coding of unlabelled data.
    - The encoding is validated and refined by attempting to regenerate the input from the encoding. The auto encoder learns a representation (encoding) for a set of data, typically for dimensionality reduction, by training the network to ignore insignificant data (“noise”).



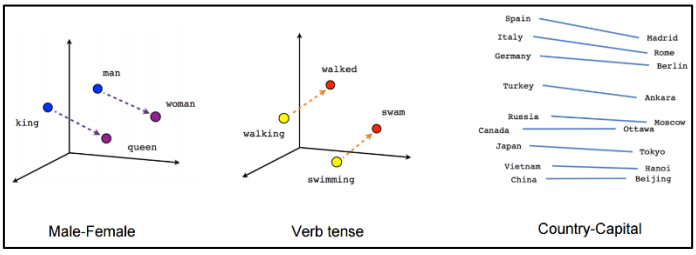

        - $D = \{x_i\}_{i=1}^{n}\ ;\ x_i \in R^d$
        - $D' = \{x'_i\}_{i=1}^{n}\ ;\ x_i \in R^{d'} : d' \ll d$
        - The new dataset is constructed such that it preserves some information from original dataset.
        - $x'_i$ is a compressed form of d’ representation of $x_i$ and then recreate the d dimensional vector
        - In the above example d = 6, d’ =3 & output is again of d = 6. We want the output layer very close to the input
    - Deep/MLP Auto encoders: here we have multiple layers



- **Denoising Autoencoders**




  - Denoising autoencoders corrupts the input data on purpose, adding noise or masking some of the input values
  - They can be trained on both noise-free or partially-noisy training-data.
  - For noise-free data, we are adding noise and building a representation robust to noise which could be observed in the future. This will make AE work well on test-points which may be partially-noisy.
  - For partially-noisy data, we still add some small noise to all the points as we do not know which data points are actually corrupted and by what extent. Adding some noise to all training points is a good hack in this context.
- **Sparse Auto Encoder**
  - The number of neurons in the hidden layer can be even greater than the size of the input layer and still auto encoder can learn interesting patterns provided some additional constraints are imposed on learning.
  - One such constraint is the sparsity constraint and the resulting encoder is known as sparse auto encoder.
  - In sparsity constraint, we try to control the number of hidden layer neurons that become active, that is to produce output close to 1, for any input.
  - E.g., we have 100 hidden neurons. If sparsity = 10%, we restrict only 10 hidden neurons to be active for an input vector.
  - The way to include the sparsity constraint is to modify the error criterion by adding a penalty term to reflect deviation from desired sparsity and then apply backpropagation incorporating the sparsity penalty. Experience with different levels of sparsity indicates an inverse relationship between the level of sparsity and the nature of relationships captured in the training data. Higher level of sparsity tends to capture more local features and vice a versa.
  - if we use sparse autoencoder with one hidden layer then the motive is average activation for most of the neurons in the hidden layer should be 0 for all training example. By this we are putting a constraint and taking away freedom from the NN model to learn anything, so it can be considered as regularization
  - Sparsity on outputs is a way of regularizing the model to ensure that we are coming up with more interesting/useful featurization without losing out on the representation power of the model which we would lose out on if we used a smaller model.
- **Word2Vec: CBOW (Continuous Bag of Words)**
  - W2V input is a text corpus and its output is a set of vectors. The outputs are based on semantic meaning of words.



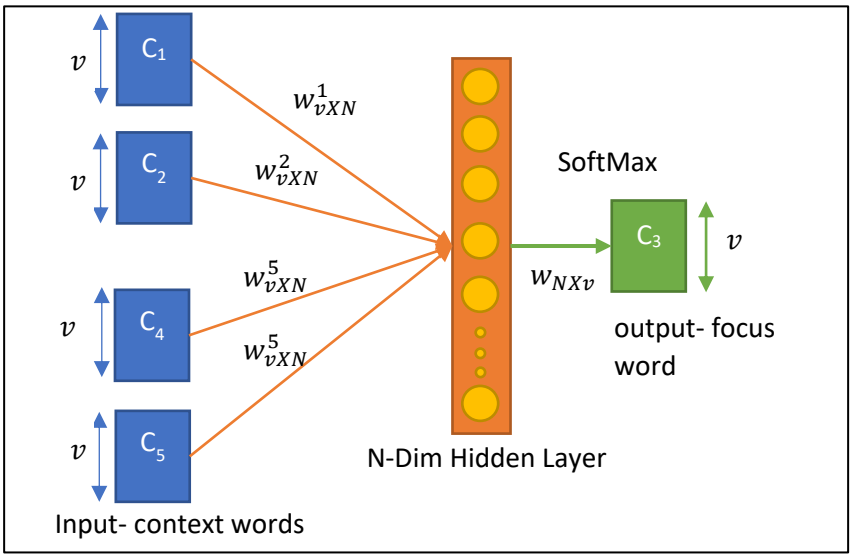

  - The word2vec algorithm uses a neural network model to learn word associations from a large corpus of text. Once trained, such a model can detect synonymous words or suggest additional words for a partial sentence. CBOW & Skip grams are two algorithms used for computing W2V
  - Definitions:

$$\underbrace{\text{The cat}}_{\text{context}}\ \underbrace{\text{sat}}_{\text{focus word}}\ \underbrace{\text{on the wall}}_{\text{context}}$$

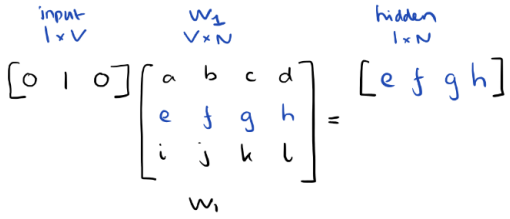
- The context words are very useful for understanding focus words
- Let's say we have dictionary $V$ with length $v$ and we use one hot encoding (binary vector of length v) to represent every word

$$w_i \in \mathbb{R}^v$$

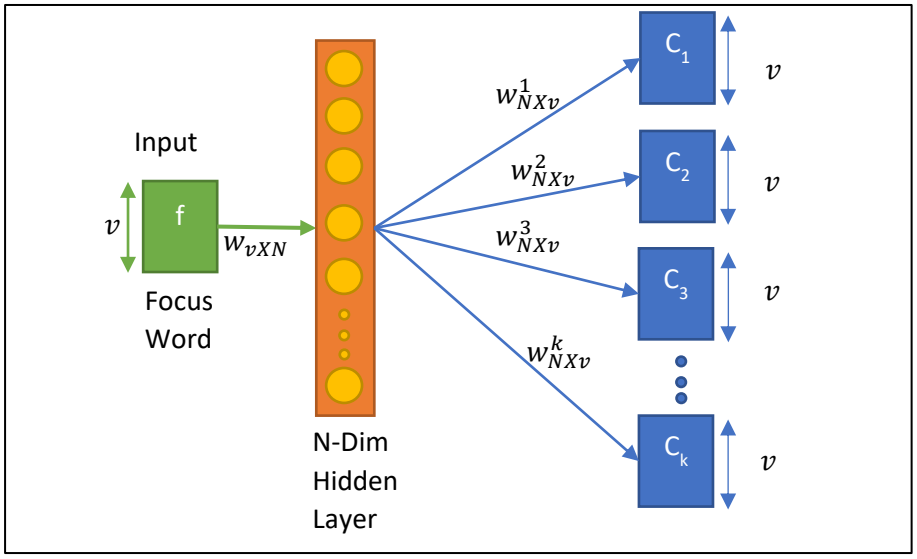
- We will represent vector of context words as input & then try to predict the focus word.
- It is connected to a linear activation unit of N-dim. Linear activation unit is $f(z) = z$
- This can also be thought as predicting multi class classification problem. Hence, we run SoftMax over it predict the focus word.
- CBOW is trained to predict a single word from a fixed window size of context words, whereas Skip-gram does the opposite, and tries to predict several context words from a single input word.
- CBOW learn better syntactic relationships between words while Skip-gram is better in capturing semantic relationships.




- In above figure we are trying to predict central word with window size of 5.
- Train CBOW: take all of the text and create combination of focus word & context words
- $w_{NXv}$ matrix will be of N rows & v columns. So, each of our words will be represented as a vector of size N

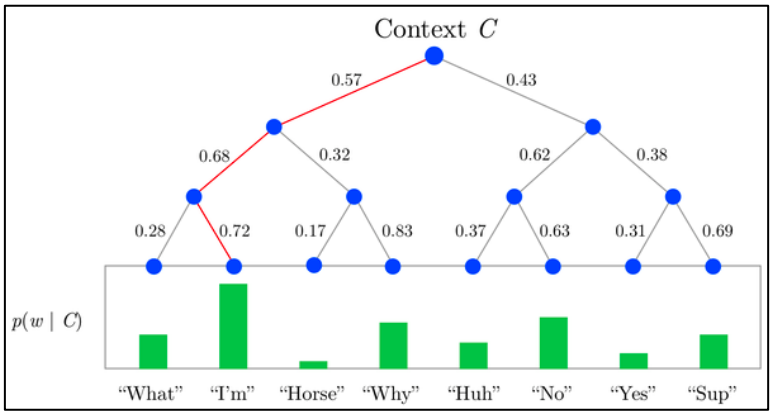



- **Skip Gram**
  - The Skip-gram tries to achieve the reverse of what the CBOW model does. It tries to predict the source context words (surrounding words) given a focus word (the centre word).
  - Considering our simple sentence from earlier, "the quick brown fox jumps over the lazy dog". If we used the CBOW model, we get pairs of (context words, focus word) where if we consider a context window of size 2, we have examples like ([quick, fox], brown), ([the, brown], quick), ([the, dog], lazy) and so on.
  - Now considering that the skip-gram model's aim is to predict the context from the target word, the model typically inverts the contexts and targets, and tries to predict each context word from its target word.
  - Hence the task becomes to predict the context [quick, fox] given target word 'brown'. Thus, the model tries to predict the context words based on the focus word.




- No of weights to compute are same as CBOW
  $k(NXV) + NXV = (k+1)(NXV)$
- In CBOW, we have only one SoftMax to train while as in Skip gram we have to train k SoftMax. Hence skip gram is computationally more expensive
- CBOW is faster than skip gram as it has less no of SoftMax to train
- CBOW is Better for frequently occurring words
- Skip grams work well with smaller amount of training data
- Skip grams works well for infrequently occurring words
- Intuitively, CBOW is an easy problem sort of like fill in the blanks. Skip gram is harder as we have to predict k words from one focus word.
- As k increases N dimensional representation is better & also as N increases, we can have more clear representation of a word

- **Word2Vec: Algorithmic Optimizations (Hierarchical SoftMax)**
  - Hierarchical SoftMax uses a binary tree to represent all words in the vocabulary. The words themselves are leaves in the tree. For each leaf, there exists a unique path from the root to the leaf, and this path is used to estimate the probability of the word represented by the leaf. “We define this probability as the probability of a random walk starting from the root ending at the leaf in question.”
  - Hierarchical SoftMax is a is an alternative to SoftMax that is faster to evaluate
  - The SoftMax is trying to solve a v class classification where v is no of words in dictionary.
  - Let’s say our dictionary has 8 words and we are trying to solve an 8-class classification problem
  - it is $O(\log_2 v)$ time to evaluate compared to $o(v)$ for SoftMax. It utilises a multi-layer binary tree, where the probability of a word is calculated through the product of probabilities on each edge on the path to that node.
  - See the Figure to the right for an example of where the product calculation would occur for the word "I'm".




- **Word2Vec: Algorithmic Optimizations (Negative Sampling)**
  - Negative Sampling is that we only update a sample of output words per iteration.
  - The target output word should be kept in the sample and gets updated, and we add to this a few (non-target) words as negative samples.

- "A probabilistic distribution is needed for the sampling process, and it can be arbitrarily chosen… One can determine a good distribution empirically."
- Mikolov et al. also use a simple subsampling approach to counter the imbalance between rare and frequent words in the training set (for example, "in", "the", and "a" provide less information value than rare words). Each word in the training set is discarded with probability $P(w_i)$ where,

$$p(w_i) = 1 - \sqrt{\frac{t}{f(w_i)}}$$

$f(wi)$ = frequency of word wi
$t$ = threshold, typically around 10-5.