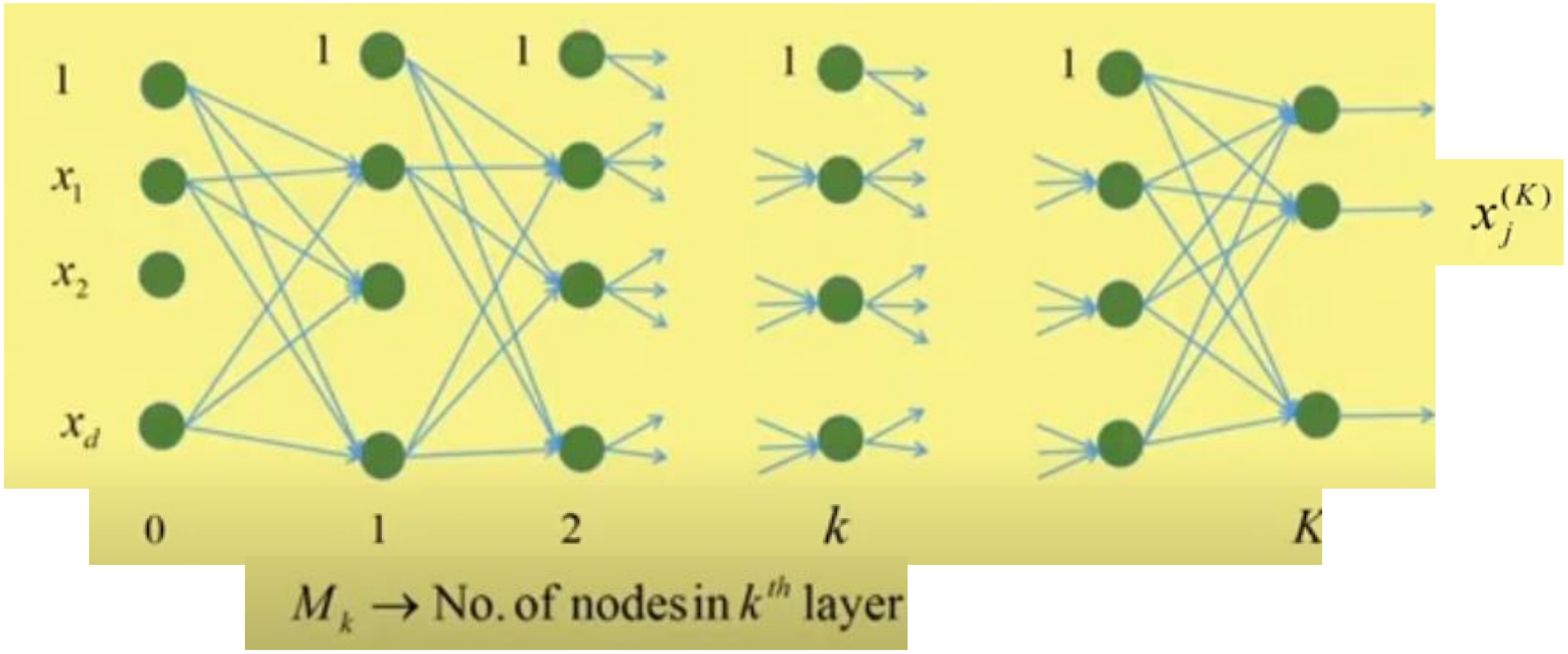
Architecture of the Multilayer Feed Forward Neural Network

- 0th layer is the input layer where neurons simply take the inputs and pass them as outputs.
- No. of hidden layers = K – 1, named as 1st, 2nd, .., kth, …, (K-1)th layer.
- Kth layer is the output layer where number of neurons = the number of classes in the dataset.



Architecture of MLFFNN

- Neurons in the input layer have the linear function of the form y=x.
- All the neurons at the hidden layers have non-linear functions to capture the nonlinearity pattern in the data.
- Neurons at output layer use nonlinear activation function to give the class belongingness.
- We assume that output of each neuron of i-th layer is passed as input to the every neurons of the (i+1)-th layer, for i=0, 1, …, K-1. Thus the connection is complete.
- Training means iteratively use the training vectors to estimate the parameters, such as weights and biases by optimizing the error function.

Architecture of MLFFNN

$M_k = No.\, of\ neurons\ in\ the\ k-th\ layer, k = 0, 1, \dots, K$

$o_j^{(k)} = output\ of\ j-th\ neuron\ at\ k-th\ layer$

$w_{ij}^{k+1} = weight\ of\ the\ connection\ between\ i-th\ neuron\ at\ k-th\ layer$
$and\ j-th\ neuron\ at\ (k+1)-th\ layer$

$Each\ training\ data\ is\ of\ the\ form\ (X_i, y_i)$ $where\ y_i = any\ one\ of\ 1, 2, \dots, M_K$

$X_i = i-th\ feature\ vector, i = 1, 2, \dots, N$

Back Propagation Algorithm

- Each neuron performs two tasks:

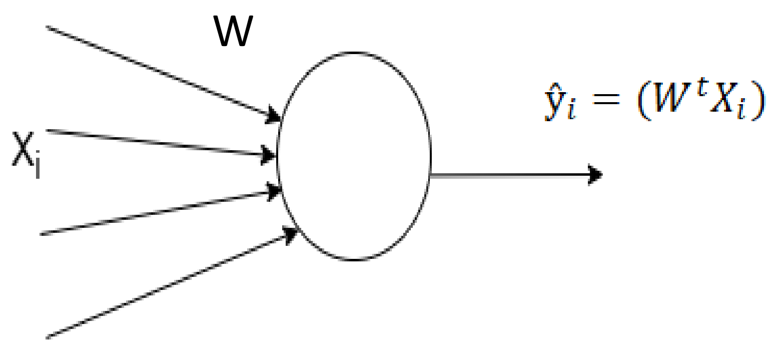
  (i) weighted sum of all inputs to the neuron

  (ii) apply non-linear function (activation function) on weighted sum to provide output of the neuron
- Thus, for each neuron in each layer, we get the output.
- For each neuron at output layer, we have the actual output.
- Thus, we can compute the error for each neuron of the output layer, from which we can compute the error function.

Back Propagation Algorithm

- But for other neurons, we cannot decide the error, as actual outputs are not known.
- So to adjust the weights and biased iteratively, we have to propagate the error from the output layer to the last hidden layer (i.e., (K-1)-th layer), from the last hidden layer to its previous hidden layer, and so on; and finally, from the 1st hidden layer to the input layer.
- This algorithm by which errors are propagated from back to previous layers to modify the weights of the connections is known as Back Propagation Learning.

Back Propagation Learning: For Single Layer Network – Single output without nonlinearity



$$E = \frac{1}{2}\sum_{i=1}^{N}(\hat{y}_i - y_i)^2 = \frac{1}{2}\sum_{i=1}^{N}(\sigma(W^t X_i) - y_i)^2$$

- Generally, mean squared error is used for regression tasks, and cross-entropy loss is used for classification tasks
- Here, Sum of square error or quadratic error is considered.
- Activation function is a linear function of the form Y = X.
- Error function E is the function of W.

- We update the weights using gradient descent approach. So we compute gradient of E w.r.t. W as

$$\nabla_E W = \sum_{i=1}^{N}(\hat{y}_i - y_i)X_i$$

- Weight updation Rule is

$$W = W - \underset{\uparrow}{\eta}\sum_{i=1}^{N}(\hat{y}_i - y_i)X_i$$

Rate of convergence factor or learning rate

Learning Rate

- The learning rate in the back-propagation algorithm is a critical hyperparameter that influences how effectively a neural network learns from the training data.
- The learning rate, often denoted by η, scales the gradients computed during back-propagation before they are used to update the weights.
- This scaling affects how quickly or slowly a network adjusts its weights.
- A higher learning rate might allow the network to learn faster, but it can also cause the learning process to overshoot minima, leading to unstable training.
- A too-small learning rate can make the learning process too slow, which might take excessively long to converge or might get stuck in local minima.

Choosing the Learning Rate

(i) Trial and Error: Initially, we might need to experiment with different values to see which learning rate works best for your specific dataset and network architecture.

(ii) Adaptive Learning Rates: Many modern optimization algorithms, like Adam, RMSprop, and AdaGrad, adjust the learning rate dynamically during training to improve performance and stability.

(iii) **Learning Rate Schedules**: Implementing a learning rate schedule that decreases the learning rate gradually as training progresses can help the network fine-tune its weights more effectively in the later stages of learning.

Practical Tips for choosing learning rate

- **Start with a Medium Learning Rate**: Begin with a learning rate like 0.01 or 0.001 and adjust based on the observed training performance.
- **Monitor Training Progress**: Keep an eye on the training and validation loss.

  (i) If the loss fluctuates wildly or increases, the learning rate might be too high. In such cases, decreasing the learning rate can help stabilize the training process.

  (ii) If the loss decreases very slowly, the learning rate might be too low. In this case, we may increase learning rate for quick convergence.
- **Batch Size Impact**: Larger batch sizes typically require a higher learning rate because the gradient estimate is more accurate, while smaller batch sizes might need a smaller learning rate to compensate for the noisier gradient estimates.

Batch Size

- **Batch size is** the number of training examples used to calculate the gradient during one iteration of model training.
- The **batch size** is a fundamental parameter in machine learning that specifies the number of training samples to be processed before the model's internal parameters (weights) are updated.
- It plays a critical role in the training process of machine learning algorithms, particularly in the context of stochastic gradient descent and its variants used in training neural networks.

Types of Batching Strategies

- **Stochastic Gradient Descent (SGD)**: Uses a batch size of 1, meaning the model weights are updated after each training example.
- **Mini-batch Gradient Descent**: Uses a batch size greater than 1 but less than the total number of training samples. This is the most common approach used in practice as it balances the computational efficiency of batch processing with the generalization advantages of stochastic updates.
- **Batch Gradient Descent**: Uses the entire training dataset to compute the gradient and update model weights in a single iteration. This method can be computationally expensive and less practical with very large datasets.

Choosing Batch Size

- **Computational Resources**: The choice of batch size can depend on the memory limitations of the hardware used for training (like GPUs).
- **Empirical Tuning**: Similar to learning rates, batch size often requires tuning and can vary based on specific characteristics of the training data and the model architecture. Common batch sizes include 32, 64, 128, 256, etc.
- **Problem Specific**: For some problems, especially those involving time-series data or sequences, the choice of batch size can significantly affect the model's performance due to dependencies within the data.

Iteration Vs. Epoch

- In the context of neural networks, **one iteration** refers to one update of the model's weights.
- An iteration occurs every time the training process uses a batch of data to compute the gradient and update the weights. The size of this batch is defined by the **batch size**. Therefore:

- **For a batch size of 1** (stochastic gradient descent), each iteration processes one training example and updates weights based on the gradient calculated from that single example.

- **For a larger batch size** (mini-batch gradient descent), each iteration processes multiple examples (as many as the batch size), calculates the average gradient from that batch, and then updates the model’s weights accordingly.

- **For a batch size equal to the entire dataset** (batch gradient descent), each iteration processes all the training data at once to compute the gradient and update the weights once per epoch.

Iteration Vs. Epoch

- **Epoch**: A single pass through the entire dataset, which consists of many iterations, depending on the batch size.
- **Iteration**: Each time the model's weights are updated using a batch of data, which happens multiple times within an epoch.

  - For example, if we have 1,000 training examples and a batch size of 100, it would take 10 iterations to complete one epoch.

Training Process Breakdown

- The training process typically involves looping over the dataset multiple times (multiple epochs). Each epoch consists of:
- **Dividing the dataset into batches**: The data is split into several smaller sets or batches, each containing a number of examples specified by the batch size.
- **Processing each batch during an iteration**: For each batch, the model performs the following steps:
  - **Forward Pass**: Run the batch of data through the model to get the output predictions.
  - **Loss Calculation**: Calculate the loss, which is a measure of how far the model's predictions are from the actual values.
  - **Backward Pass (Gradient Calculation)**: Calculate the gradient of the loss function with respect to the model parameters.
  - **Update Model Parameters**: Adjust the parameters (weights and biases) using the gradients computed.
- **Repeating for all batches**: This process is repeated for each batch in the dataset, constituting one epoch.

Batch creation in each epoch

- Here are the most common strategies for creating batches in each epoch:

(1) Sequential Batching

- **Fixed Order**: The simplest method is to divide the dataset into batches in the order the data appears in the dataset.
- This method does not involve any randomness.
- The data is simply split into chunks of the specified batch size.
- For example, if we have 1,000 examples and a batch size of 100, the first batch will contain examples 1 to 100, the second batch will contain examples 101 to 200, and so on.

Batch creation in each epoch

(2) Random Batching

- **Shuffling**: More commonly, the entire dataset is shuffled at the beginning of each epoch, and then divided into batches. Each batch used during an iteration is likely to be different across epochs. This method helps to prevent the model from memorizing the order of the data, potentially improving generalization.

Batch creation in each epoch

(3) Stratified Batching

- **Stratified Sampling**: In classification tasks, particularly with imbalanced datasets, it’s beneficial to use stratified sampling to ensure that each batch contains a representative distribution of the different classes.
- This helps in maintaining a consistent signal for each class during training.

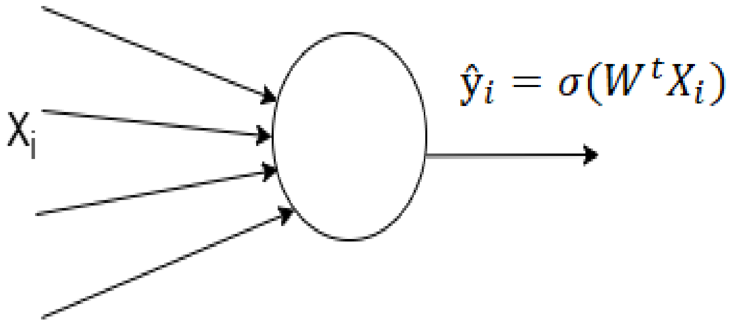
Back Propagation Learning: For Single Layer Network – Single output with linearity

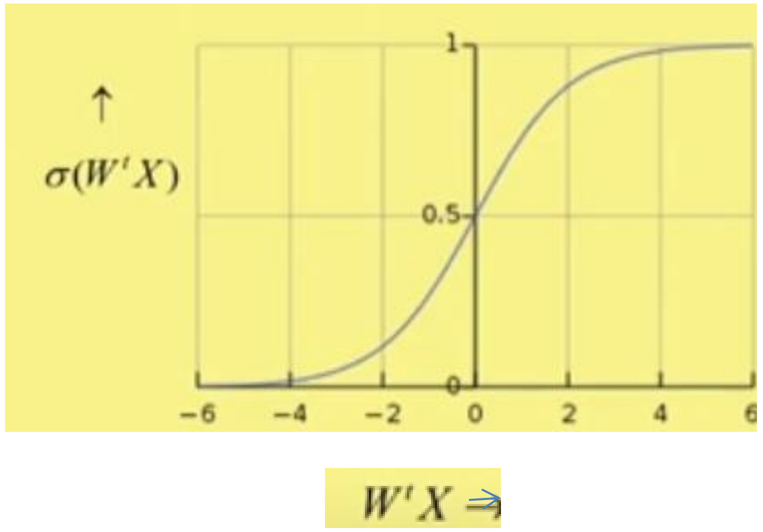
- Weight updation Rule is: $W = W - \eta \sum_{i=1}^{N} (\hat{y}_i - y_i) X_i$

- Let us consider, stochastic optimization, i.e., a single training sample X_i. Then the updation rule is $W = W - \eta(\hat{y}_i - y_i) X_i$
- Let, This is a 2-class problem, with class $y_i = 0$ and $y_i=1$.
- Let for X_i, $y_i=1$. If X_i is correctly predicted then $\hat{y}_i = 1$. Then we don't need to update the weight. Let X_i is misclassified, then $\hat{y}_i$ is < 0 according to the step function. In this case, $W = W - \eta(\hat{y}_i - y_i) X_i = W + \eta\chi X_i$, i.e., we are adding fraction of X_i to the weight vector.
- Let for X_i, $y_i=0$. If X_i is correctly predicted then $\hat{y}_i = 0$. Then we don't need to update the weight. Let X_i is misclassified, then $\hat{y}_i > 0$ according to our step function. In this case, $W = W - \eta(\hat{y}_i - y_i) X_i = W - \eta\chi X_i = W + \eta\chi(-X_i)$, i.e., we are subtracting fraction of X_i, => adding fraction of negative X_i to the weight vector.

Back Propagation Learning: For Single Layer Network – Single output with nonlinearity

- Earlier one was the Single layer network without nonlinearity, i.e., activation function was linear function.
- To capture nonlinearity, we apply the nonlinear activation function.
- The nonlinear function like step function is not differentiable. So we may apply the nonlinear function like sigmoid function.





Single output with nonlinearity

- It is a differentiable function and ranges over [0, 1].
- Its derivative is very simple, i.e., for $\sigma(s) = \frac{1}{1+e^{-s}}$

$$\sigma'(s) = +\frac{1}{(1+e^{-s})^2}e^{-s} = \frac{1}{1+e^{-s}}\frac{e^{-s}}{1+e^{-s}}$$
$$= \frac{1}{1+e^{-s}}(1 - \frac{1}{1+e^{-s}}) = \sigma(s).(1-\sigma(s))$$

- Now we are considering Single Layer Network with Single output with nonlinearity function sigmoid as activation function.

- Here, the error function is $E = \frac{1}{2}(\hat{y}_i - y_i)^2 = \frac{1}{2}(\sigma(W^tX_i) - y_i)^2$

Where stochastic optimization procedure is considered.

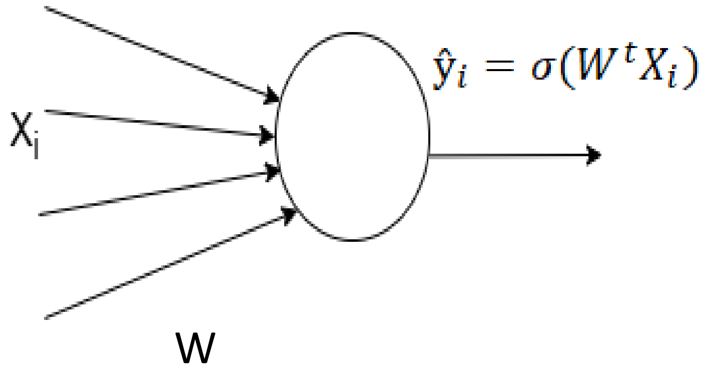
Single output with nonlinearity

- The gradient of the error function w.r.t. weight vector is $\nabla_E W = \hat{y}_i(1-\hat{y}_i)(\hat{y}_i - y_i)X_i$ since

$$\hat{y}_i = \sigma(W^t X_i)$$

- The weight updation rule is

$$W = W - \eta\hat{y}_i(1-\hat{y}_i)(\hat{y}_i - y_i)X_i$$



- Let, This is a 2-class problem, with class $y_i = 0$ and y_i=1.

Single output with nonlinearity

- Weight updation rule is : => $W = W - \eta \hat{y}_i (1 - \hat{y}_i)(\hat{y}_i - y_i) X_i$
- Let for X_i, y_i=1. If X_i is correctly predicted then $\hat{y}_i$ = high probability value, say 0.7. Then we don't need to update the weight. Let X_i is misclassified, then $\hat{y}_i$ is very low, say 0.1 according to the sigmoid function. In this case, $W = W - \eta \hat{y}_i (1 - \hat{y}_i)(\hat{y}_i - y_i) X_i = W + \eta \chi X_i$, i.e., we are adding fraction of X_i to the weight vector.
- Let for X_i, y_i=0. If X_i is correctly predicted then $\hat{y}_i$ = low value, say 0.1. Then we don't need to update the weight. Let X_i is misclassified, then $\hat{y}_i$ is high, say 0.7, according to the sigmoid function. In this case, $W = W - \eta \hat{y}_i (1 - \hat{y}_i)(\hat{y}_i - y_i) X_i = W - \eta \chi X_i$, i.e., we are subtracting fraction of X_i, => adding fraction of negative X_i to the weight vector.

Single output with linearity / nonlinearity

- With linearity, weight updation rule is:

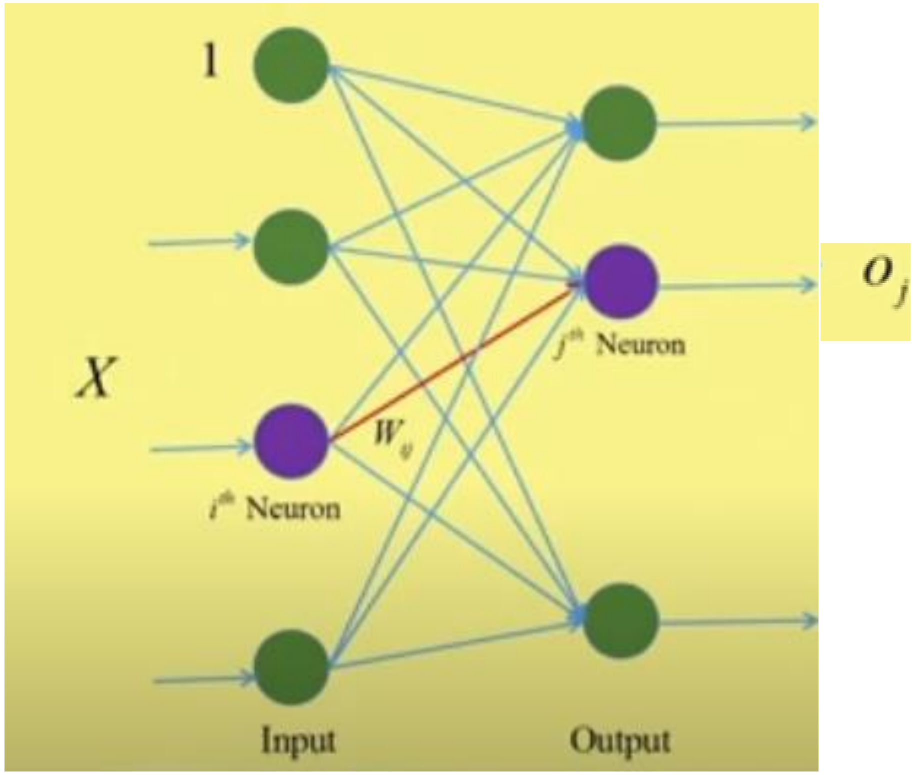
$W = W - \eta(\hat{y}_i - y_i)X_i$, where $\hat{y}_i = (W^t X_i)$

- With non-linearity, weight updation rule is:

$W = W - \eta\hat{y}_i(1 - \hat{y}_i)(\hat{y}_i - y_i)X_i$, where $\hat{y}_i = \sigma(W^t X_i)$

- Both are weight updation rule for single layer perceptron to implement a two class problem.

Back Propagation Learning: For Single Layer Network – Multiple output with nonlinearity



$$o_j = \frac{1}{1+e^{-\theta_j}} \qquad \theta_j = \sum_{i=1}^{D} W_{ij} x_i$$

$$E = \frac{1}{2}\sum_{j=1}^{M}(o_j - t_j)^2$$

$$\frac{\partial E}{\partial W_{ij}} = \frac{\partial E}{\partial o_j}\cdot\frac{\partial o_j}{\partial \theta_j}\cdot\frac{\partial \theta_j}{\partial W_{ij}}$$

$$= (o_j - t_j)o_j(1-o_j)x_i$$

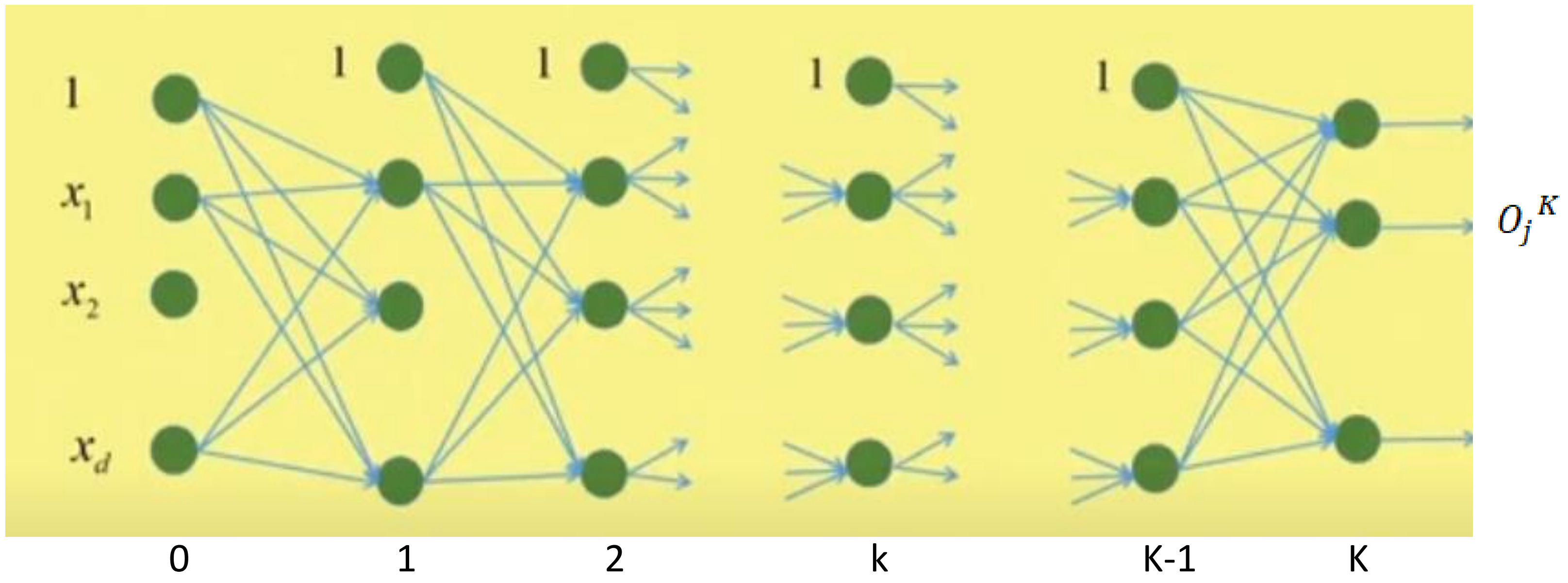
Weight updation rule ⇒

$$W_{ij} \leftarrow W_{ij} - \eta(o_j - t_j)o_j(1-o_j)x_i$$

- D= dimension of training samples = number of input layer neurons.
- M = Number of classes = no. of output layer neurons.
- If we compute w_{ij} for every i and j, then all the components of the weight matrix, $W = (w_{ij})_{DXM}$ will be updated.

Architecture of the Multilayer Feed Forward Neural Network

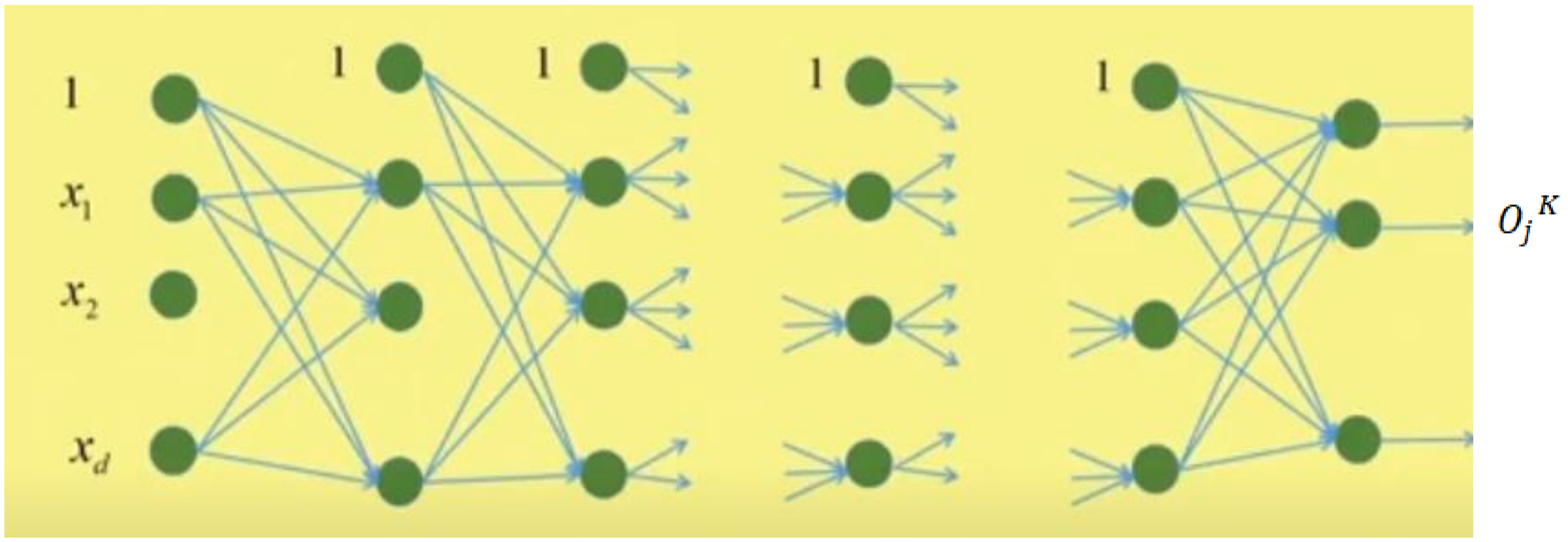
- 0th layer is the input layer where neurons simply take the inputs and pass them as outputs.
- No. of hidden layers = K – 1, named as 1st, 2nd, .., kth, …, (K-1)th layer.
- Kth layer is the output layer where number of neurons = the number of classes in the dataset.



Back Propagation Learning: For Multi Layer Feed Forward Network

- Let M_k is the number of nodes in the k-th layer, for k=0, 2, …, (K-1), K. So, M_0 is the dimension of the input feature vector.
- $o_j{}^k$ = Output of the j-th neuron of the k-th layer
- $o_j{}^K$ = Output of the j-th neuron of the output layer

Multi Layer Perceptron

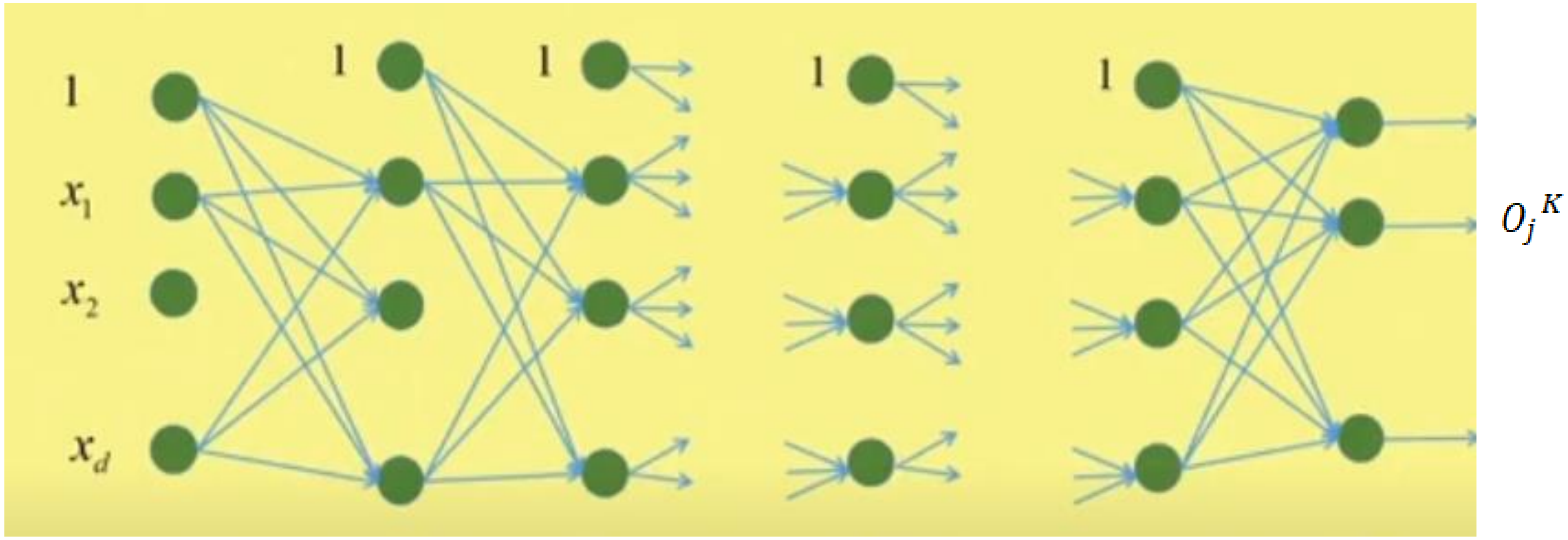


• Each neuron in a hidden/output layer performs two tasks:

(i) Collects inputs from all the neurons of the previous layer and compute the weighted sum of the coming inputs

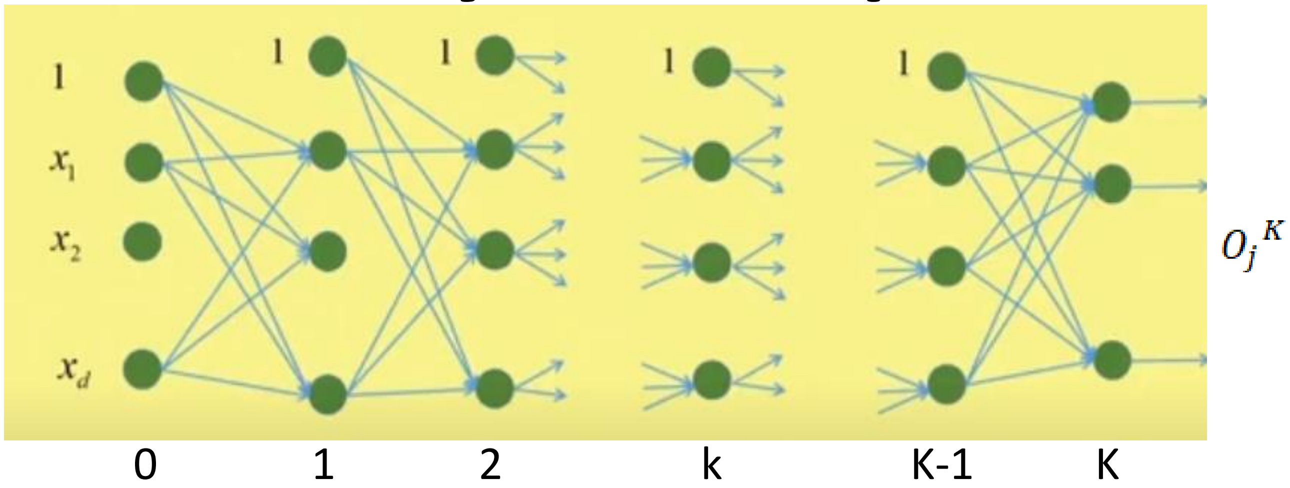
(ii) Apply activation function, like RELU, Sigmoid etc. to compute the output.

Multi Layer Perceptron



- Thus every neuron at k-th layer receives inputs from all neurons in (k-1)-th layer and passes the computed output to each neuron of the (k+1)-th layer, for k= 1, 2, …, K.
- When k=K, it is the output layer. So it does not pass the values to any other layer. These values are the predicted values of the class.
- o_j^K is the predicted output of the j-th neuron of the output layer (i.e., K-th layer). Actual value for j-th neuron is t_j.

Multi Layer Perceptron



- Thus, we may compute the error function for the output layer. So, the weights of the connections between each neuron of last hidden layer (i.e., (K-1)-th layer) and each neuron of the output layer (i.e., (K-th layer) can be directly updated by the previous weight updation rule.
- But we cannot compute the error function for hidden layers as the actual output for hidden layer neurons are not known. So we cannot directly update the weights of the connections between neurons in two consecutive hidden layers and between input layer and first hidden layer.

Multilayer Perceptron

- In case of Multilayer Feed Forward neural network, the training of hidden layer neurons is slightly different from the training of the output layer neurons.
- It is because, the actual outputs of the neurons of the output layer are known but these are unknown for the hidden layer neurons.
- We can compute the error at the output layer as we have the predicted values as well as the true values of the neurons of the output layer. These error expression provides us the error function associated with the output layer.
- We can backpropagate the error through the gradient descent approach to the connection weights in between the neurons in the (K-1)-th layer (i.e., the last hidden layer) and the K-th layer (i.e., the output layer).
- So we can backpropagate the error for updation of the weight vectors from (K-1)-th layer to K-th layer.

Multilayer Perceptron

- But when we try to update the weight vectors in between two hidden layers or between input layer and first hidden layer, we can't compute the error, as we don't know the true output of the hidden layer neurons.
- So we have to backpropagate the effect of the output error (that we have computed at the output layer) to the hidden layers. For that we use the gradient descent approach to update the weight vectors between (k-1)-th layer and k-th layer, for k=K-1, K-2, …, 1.
- So, we first update the weight vectors between (K-1)-th layer and K-th layer directly using error function using gradient descent approach.
- Then we backpropagate the errors in a order as follows to update the weight vectors:

step 1) between (K-2)-th layer and (K-1)-th layer

step 2) between (K-3)-th layer and (K-2)-th layer

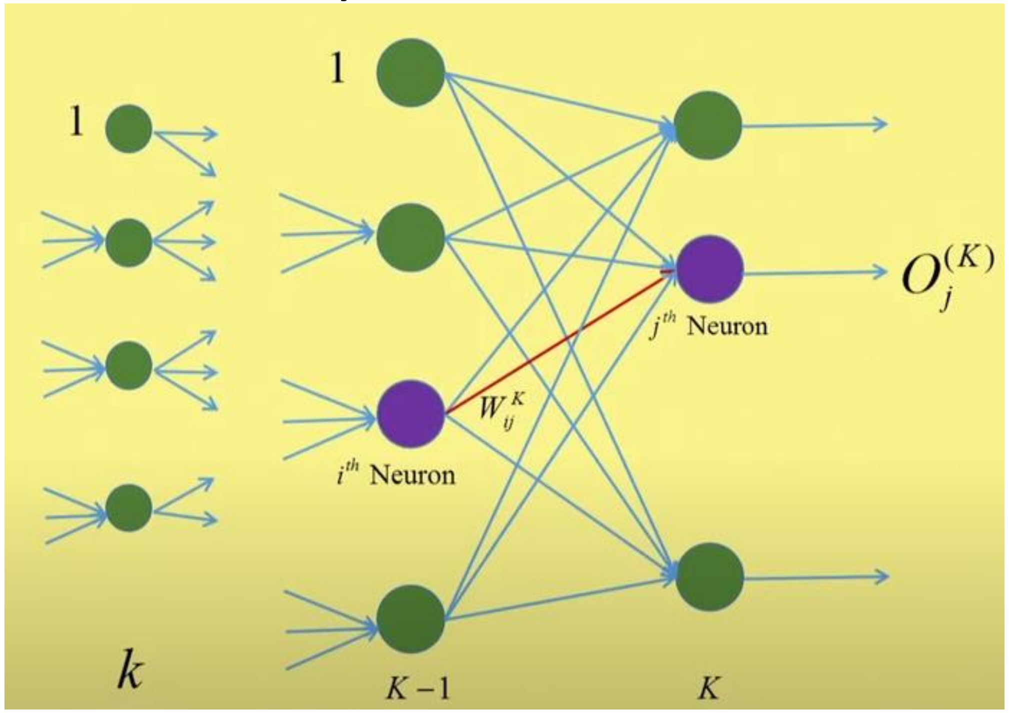
……….. ……… ……….

step K-2) between 1-th layer and 2-th layer

step K-1) between 0-th layer and 1-th layer

Back Propagation Learning : Output Layer

- Let last hidden layer = the (K-1)-th layer and so output layer is the K-th layer of the neural network.



- j-th neuron performs 2 tasks:

i) Compute the weighted sum of all the inputs coming from the outputs of the neurons of (K-1)-th layer.

$$\theta_j^K = \sum_{i=1}^{M_{K-1}} W_{ij}^K O_i^{K-1}$$

ii) Apply the activation function (here, sigmoidal function on the weighted sum to get the output of the j-th neuron.

$$O_j^K = \frac{1}{1+e^{-\theta_j^K}}$$

- The error at the output layer is:

$$E = \frac{1}{2}\sum_{j=1}^{M_K}(O_j^K - t_j)^2$$

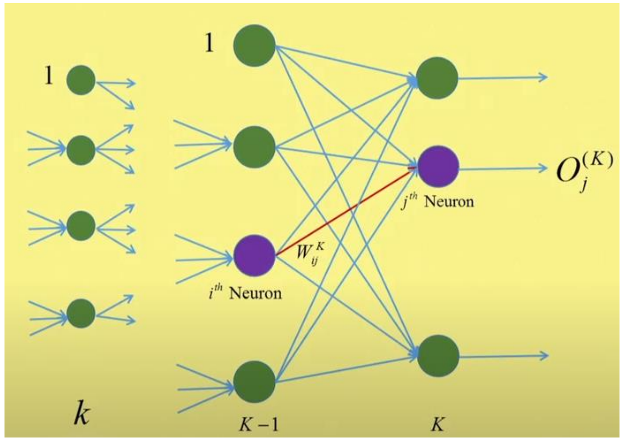
- t_j is the true output of the j-th neuron at the K-th layer.

Back Propagation Learning : Output Layer

- In back propagation learning, we will estimate the weight vector W_{ij}^{K} that minimizes $E = \frac{1}{2}\sum_{j=1}^{M_K}(O_j^{K} - t_j)^2$
- In gradient descent approach we need to compute the gradient of this error function, i.e., we need to take the partial derivative of E w.r.t. the weight vector W_{ij}^{K} , i.e., we need to compute $\frac{\partial E}{W_{ij}^{K}}$
- By chain rule, we get $\frac{\partial E}{\partial W_{ij}^{K}} = \frac{\partial E}{\partial O_j^{K}} \cdot \frac{\partial O_j^{K}}{\partial \theta_j^{K}} \cdot \frac{\partial \theta_j^{K}}{\partial W_{ij}^{K}}$

Back Propagation Learning : Output Layer

- Since, $E = \frac{1}{2}\sum_{j=1}^{M_K}(O_j{}^K - t_j)^2$ so $\frac{\partial E}{\partial O_j{}^K} = O_j{}^K - t_j$
- Since, $O_j{}^K = \frac{1}{1+e^{-\theta_j{}^K}}$ which is a sigmoidal function, so $\frac{\partial O_j{}^K}{\partial \theta_j{}^K} = O_j{}^K(1 - O_j{}^K)$
- Since, $\theta_j{}^K = \sum_{i=1}^{M_{K-1}} W_{ij}{}^K O_i{}^{K-1}$ so $\frac{\partial \theta_j{}^K}{\partial W_{ij}{}^K} = O_i{}^{K-1}$



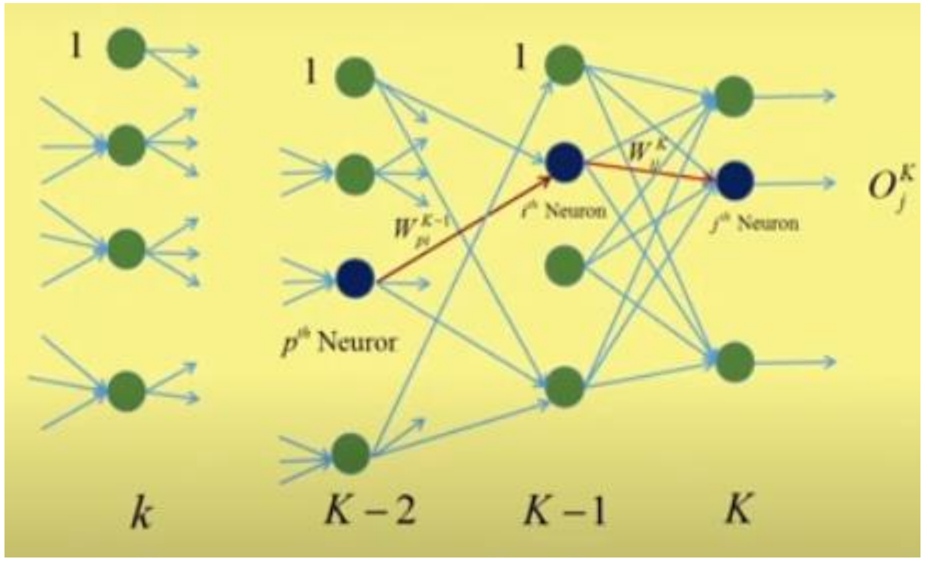
- Therefore, $\frac{\partial E}{\partial W_{ij}{}^K} = \frac{\partial E}{\partial O_j{}^K}\cdot\frac{\partial O_j{}^K}{\partial \theta_j{}^K}\cdot\frac{\partial \theta_j{}^K}{\partial W_{ij}{}^K} = (O_j{}^K - t_j)\, O_j{}^K(1 - O_j{}^K)O_i{}^{K-1}$
- Let, $\delta_j{}^K = (O_j{}^K - t_j)\, O_j{}^K(1 - O_j{}^K)$, so $\frac{\partial E}{\partial W_{ij}{}^K} = \delta_j{}^K\, O_i{}^{K-1}$
- Therefore, weight updation rule at the output layer is:

$$W_{ij}{}^K = W_{ij}{}^K - \eta\, \delta_j{}^K\, O_i{}^{K-1}$$

Back Propagation learning : Hidden layer

- We can compute the error only at the output layer. But when we are updating the weights at the hidden layer, we cannot compute the error at the hidden layer. So we cannot define any error function at the hidden layer as we don't know what is the target output at hidden layer.
- So the error computed at the output layer is back propagated to the hidden layer and check what is the effect of this error in weight updation at hidden layers. We will also check it using gradient descent approach.
- Initially, we assume that we have updated the weights between last [i.e., (K-1)-th] layer and the output [i.e., K-th] layer, which we have discussed earlier.
- Now we will check how we can update the weights between (K-2)-th hidden layer and the (K-1)-th hidden layer. Then we can generalize it for every two consecutive hidden layers.
- We also use it to update the weights between input layer (i.e., 0-th layer) and first hidden layer. In this case, the input X_t of t-th neuron in input layer is directly passes as the output X_t of the same neuron. So we consider X_t as output of the t-th neuron of the 0-th layer, i.e., O_t^0 .

Back Propagation learning : Hidden layer



- We are considering the (K-2)-th layer and (K-1)-th layer, between which we want to update the weights.
- Let o_p^{K-2} is the output of p-th neuron at (K-2)-th layer.
- W_{pi}^{K-1} is the weight between p-th neuron of (K-2)-th layer and i-th neuron of (K-1)-th layer

- Now we are considering p-th neuron of (K-2)-th layer, i-th neuron of (K-1)-th layer, and j-th neuron of K-th layer.
- So, the output o_p^{K-2} of p-th neuron of (K-2)-th layer is passed to i-th neuron of (K-1)-th layer with connection weight W_{pi}^{K-1} .
- And the output o_i^{K-1} of i-th neuron of (K-1)-th layer is passed to j-th neuron of K-th layer with connection weight W_{ij}^{K} .

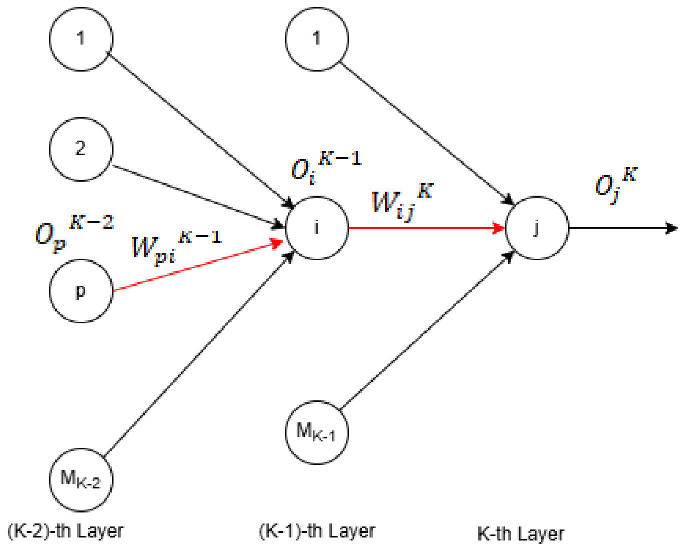
Back Propagation learning : Hidden layer

- We have the only output function at the output layer, which is $E = \frac{1}{2}\sum_{j=1}^{M_K}(O_j^K - t_j)^2$, where M_K is the number output layer neurons. W_{pi}^{K-1}
- In order to update the weight W_{pi}^{K-1} , we have to take the derivative of the loss function E w.r.t. W_{ij}^{K} , not w.r.t. .
- Again we have to apply chain rule to know how the error E varies with the variation of W_{pi}^{K-1} , when j-th neuron is considered.
- Given $E = \frac{1}{2}\sum_{j=1}^{M_K}(O_j^K - t_j)^2$ find $\frac{\partial E_j}{\partial W_{pi}^{K-1}} = ?$
- We have

$$O_j^K = \sigma(\theta_j^K) = \frac{1}{1+e^{-\theta_j^K}}$$

$$\theta_j^K = \sum_{i=1}^{M_{K-1}} W_{ij}^K O_i^{K-1} \qquad O_i^{K-1} = \sigma(\theta_i^{K-1}) = \frac{1}{1+e^{-\theta_i^{K-1}}}$$

$$\theta_i^{K-1} = \sum_{p=1}^{M_{K-2}} W_{pi}^{K-1} O_p^{K-2}$$



Back Propagation learning : Hidden layer

- Using chain rule: $\frac{\partial E^{j}}{\partial W_{pi}{}^{K-1}} = \frac{\partial E}{\partial O_j{}^K} \frac{\partial O_j{}^K}{\partial \theta_j{}^K} \frac{\partial \theta_j{}^K}{\partial O_i{}^{K-1}} \frac{\partial O_i{}^{K-1}}{\partial \theta_i{}^{K-1}} \frac{\partial \theta_i{}^{K-1}}{\partial W_{pi}{}^{K-1}}$

$$E = \frac{1}{2}\sum_{j=1}^{M_K}(O_j{}^K - t_j)^2 \quad \Rightarrow \quad \frac{\partial E}{\partial O_j{}^K} = O_j{}^K - t_j$$

$$O_j{}^K = \frac{1}{1+e^{-\theta_j{}^K}} \quad \Rightarrow \quad \frac{\partial O_j{}^K}{\partial \theta_j{}^K} = O_j{}^K(1 - O_j{}^K)$$

$$\theta_j{}^K = \sum_{i=1}^{M_{K-1}} W_{ij}{}^K O_i{}^{K-1} \quad \Rightarrow \quad \frac{\partial \theta_j{}^K}{\partial O_i{}^{K-1}} = W_{ij}{}^K$$

$$O_i{}^{K-1} = \sigma\left(\theta_i{}^{K-1}\right) = \frac{1}{1+e^{-\theta_i{}^{K-1}}} \quad \Rightarrow \quad \frac{\partial O_i{}^{K-1}}{\partial \theta_i{}^{K-1}} = O_i{}^{K-1}(1 - O_i{}^{K-1})$$

Back Propagation learning : Hidden layer

- Using chain rule: $$\frac{\partial E^j}{\partial W_{pi}{}^{K-1}} = \frac{\partial E}{\partial O_j{}^K}\frac{\partial O_j{}^K}{\partial \theta_j{}^K}\frac{\partial \theta_j{}^K}{\partial O_i{}^{K-1}}\frac{\partial O_i{}^{K-1}}{\partial \theta_i{}^{K-1}}\frac{\partial \theta_i{}^{K-1}}{\partial W_{pi}{}^{K-1}}$$

$$\theta_i{}^{K-1} = \sum_{p=1}^{M_{K-2}} W_{pi}{}^{K-1} O_p{}^{K-2} \quad => \quad \frac{\partial \theta_i{}^{K-1}}{\partial W_{pi}{}^{K-1}} = O_p{}^{K-2}$$

- Therefore, $$\frac{\partial E^j}{\partial W_{pi}{}^{K-1}} = \frac{\partial E}{\partial O_j{}^K}\frac{\partial O_j{}^K}{\partial \theta_j{}^K}\frac{\partial \theta_j{}^K}{\partial O_i{}^{K-1}}\frac{\partial O_i{}^{K-1}}{\partial \theta_i{}^{K-1}}\frac{\partial \theta_i{}^{K-1}}{\partial W_{pi}{}^{K-1}}$$

$$= \left(O_j{}^K - t_j\right) O_j{}^K(1 - O_j{}^K)\; W_{ij}{}^K\; O_i{}^{K-1}(1 - O_i{}^{K-1})\; O_p{}^{K-2}$$

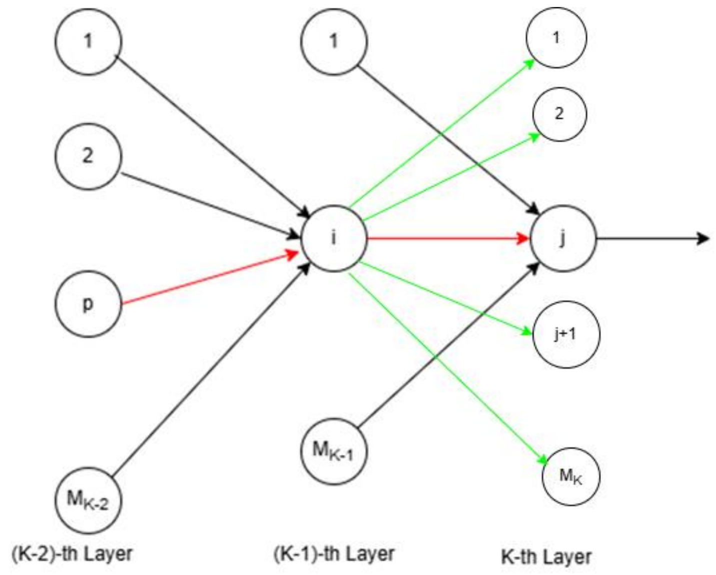
$$= O_i{}^{K-1}(1 - O_i{}^{K-1})\; O_p{}^{K-2}\; \delta_j{}^K\, W_{ij}{}^K$$, where

$\delta_j{}^K = (O_j{}^K - t_j)\, O_j{}^K(1 - O_j{}^K)$, obtained earlier.

- This gradient is calculated using only j-th output layer neuron.

Back Propagation learning : Hidden layer

- But the error is propagated to the p-th neuron at (K-2)-th layer from all the output layer neurons via i-th neuron of the (K-1)-th hidden layer.



•So, the actual gradient should be the obtained by summation over all output layer neuron.

•Therefore,

$$\frac{\partial E}{\partial {W_{pi}}^{K-1}} = \sum_{j=1}^{M_K} \frac{\partial E^j}{\partial {W_{pi}}^{K-1}} = {O_i}^{K-1}(1 - {O_i}^{K-1}){O_p}^{K-2} \sum_{j=1}^{M_K} {\delta_j}^K {W_{ij}}^K$$

$$Let, {\partial_i}^{K-1} = {O_i}^{K-1}(1 - {O_i}^{K-1}) \sum_{j=1}^{M_K} {\delta_j}^K {W_{ij}}^K \qquad Then, \frac{\partial E}{\partial {W_{pi}}^{K-1}} = {\partial_i}^{K-1}\, {O_p}^{K-2}$$

So, the weight updation rule is: $${W_{pi}}^{K-1} = {W_{pi}}^{K-1} - \eta\, {\partial_i}^{K-1}\, {O_p}^{K-2}$$

Back Propagation learning : Hidden layer

- Instead of last hidden layer, i.e., (K-1)-th layer, if we consider k-th layer, then replace K by k+1 and we get, $W_{ij}{}^{k} = W_{ij}{}^{k} - \eta\, \partial_j{}^{k}\, O_i{}^{k-1}$, where

$$\partial_j{}^{k} = O_j{}^{k}(1 - O_j{}^{k}) \sum_{t=1}^{M_{k+1}} \delta_t{}^{k+1} W_{jt}{}^{k+1}$$

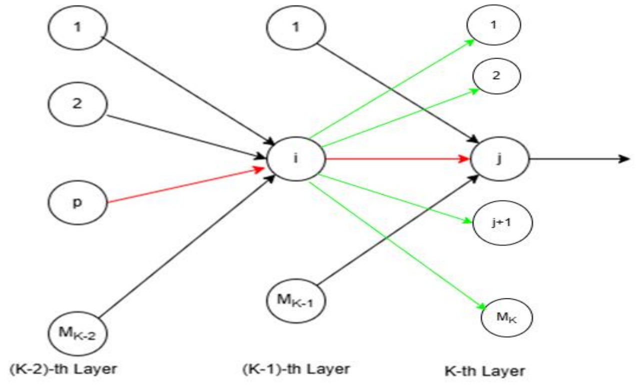
- Thus varying k from 1 to K-1, we will get all weight updation rules for all possible i and j, which gives us the new weights of the connections between input layer and first hidden layer, and between all consecutive hidden layers.

Back Propagation learning

So the weight updation rule for hidden layer:

$$W_{ij}{}^{k} = W_{ij}{}^{k} - \eta\, \partial_j{}^{k}\, O_i{}^{k-1}$$ where

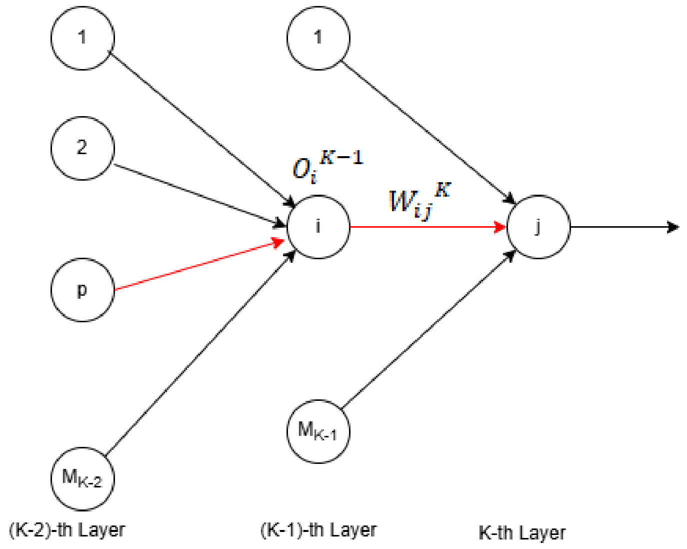
$$\partial_j{}^{k} = O_j{}^{k}(1 - O_j{}^{k}) \sum_{t=1}^{M_{k+1}} \delta_t{}^{k+1} W_{jt}{}^{k+1}$$



- So the weight updation rule for output layer:

$$W_{ij}{}^{K} = W_{ij}{}^{K} - \eta\, \delta_j{}^{K}\, O_i{}^{K-1}$$ where,

$$\delta_j{}^{K} = (O_j{}^{K} - t_j)\, O_j{}^{K}(1 - O_j{}^{K})$$



THANK YOU