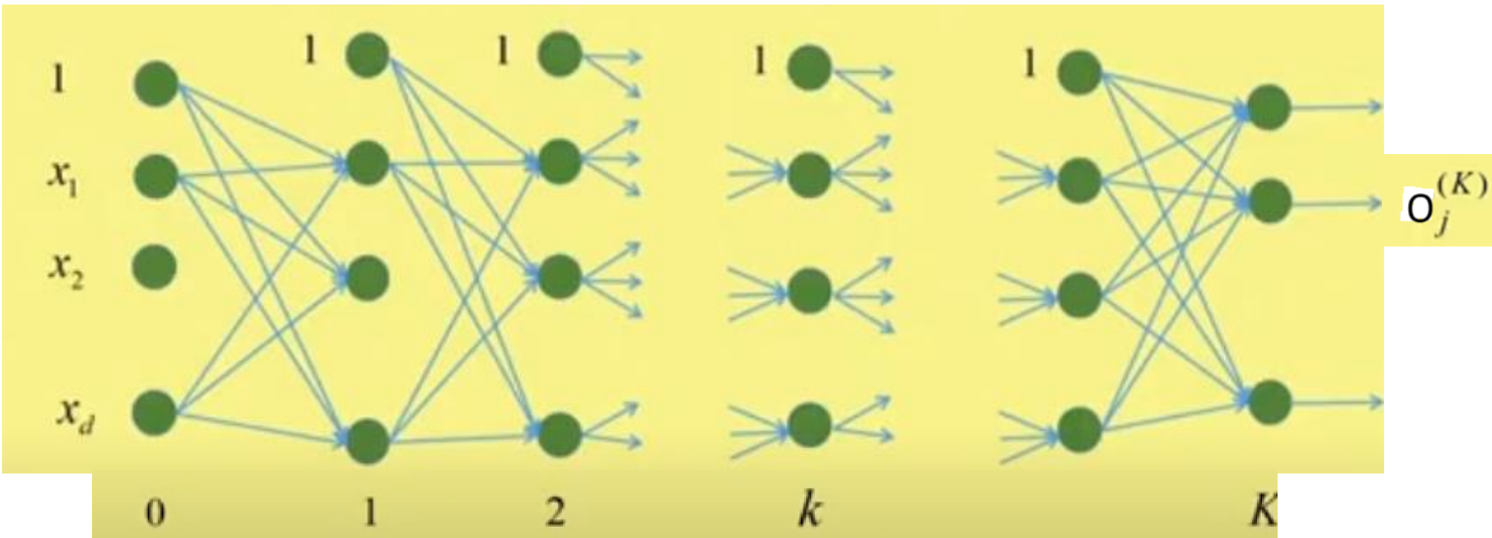
Backpropagation Learning Example



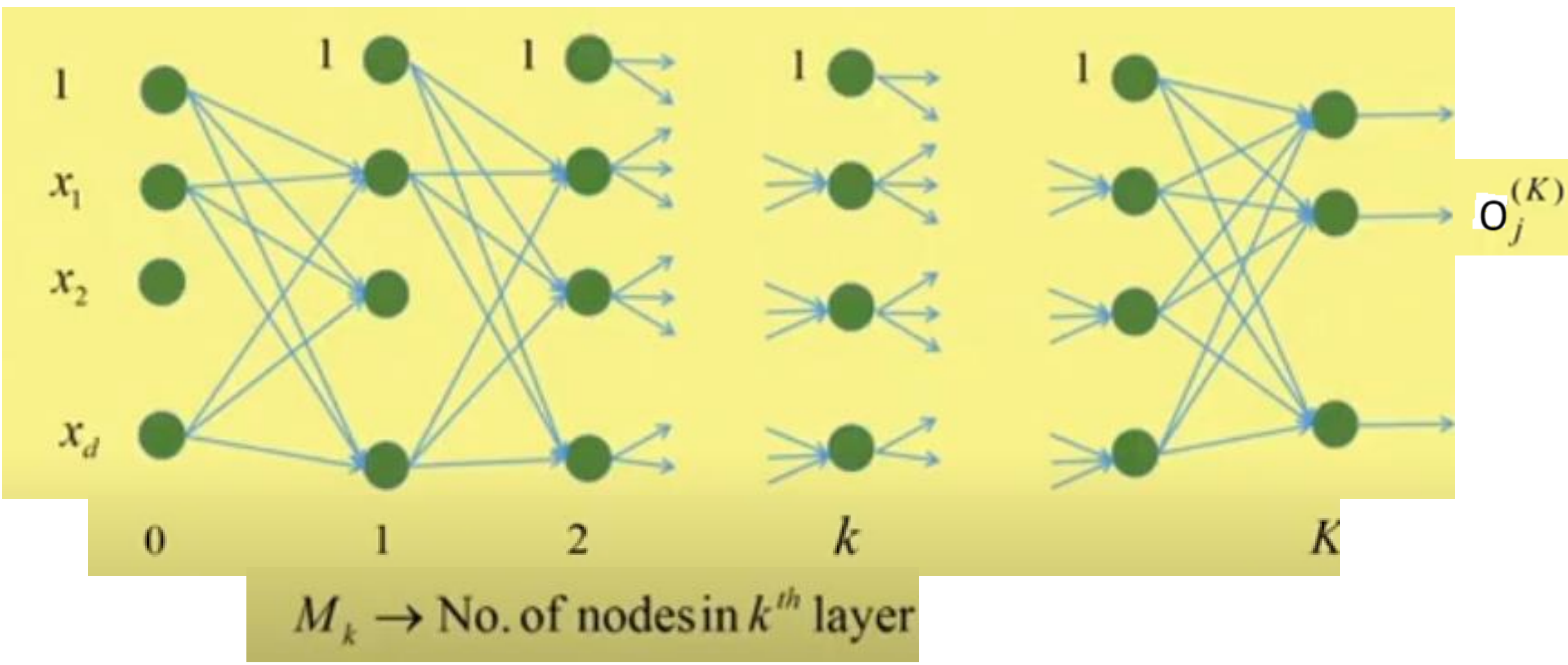
- Hidden layer nodes impose nonlinearity.
- As we increase the number of hidden layers, we can capture more and more complex form of nonlinearity.
- Finally, at the output layer, we use linear classifier.

Backpropagation Learning Example

- In real life, we have samples which are not linearly separable.
- The hidden layer nodes map the non-linearly separable inputs into a space where they are linearly separable.
- This space is called the latent space.
- In latent space, we may use a linear classifier to classify them.

Backpropagation Learning Example

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



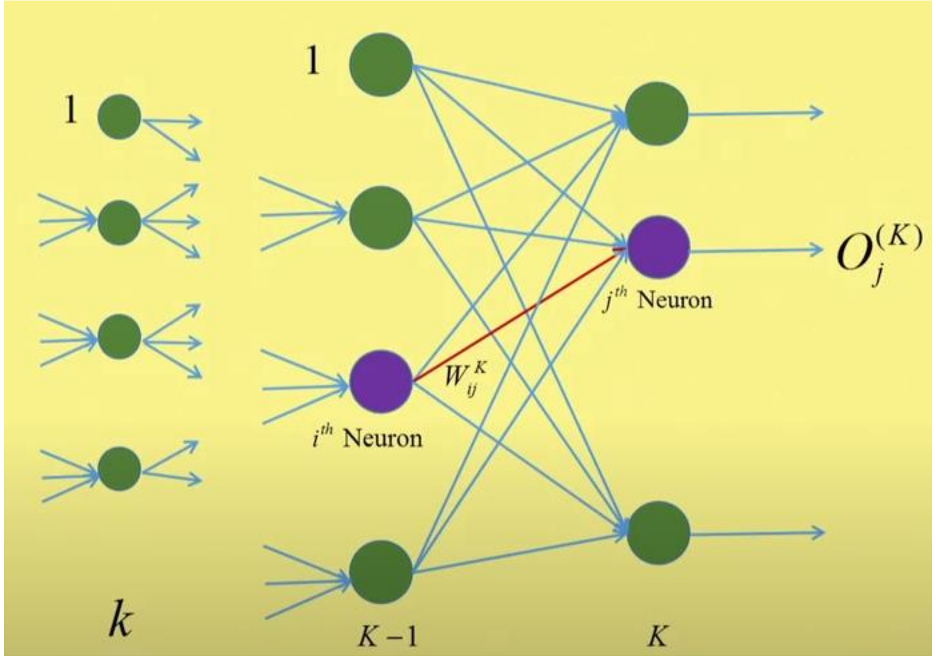
For i-th neuron of hidden layer to j-th neuron of output layer

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

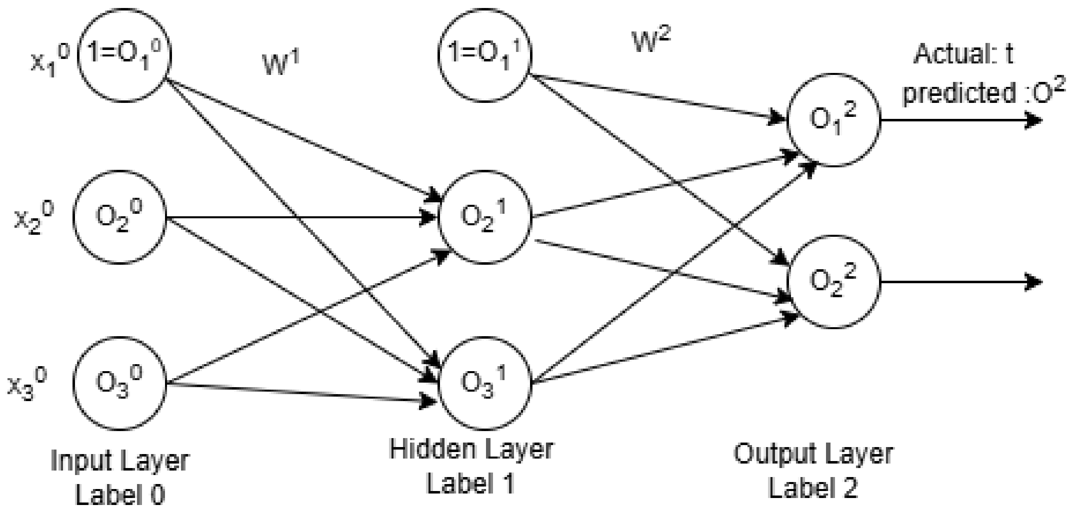
- Sigmoid function: $O_j{}^K = \dfrac{1}{1 + e^{-\theta_j{}^K}}$

- Error function:

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



Example with 1 hidden layer



$$O^2 = \begin{bmatrix} O_1{}^2 \\ O_2{}^2 \end{bmatrix}$$

$$t = \begin{bmatrix} t_1 \\ t_2 \end{bmatrix}$$

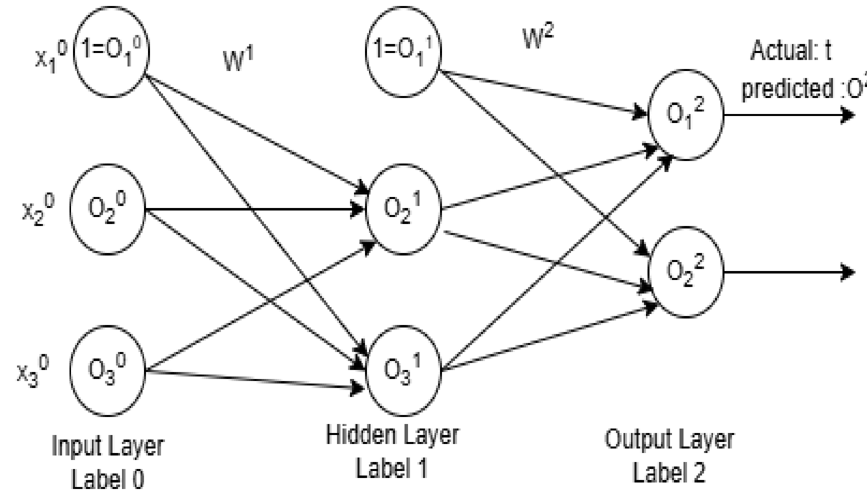
$$W^1 = \begin{bmatrix} W_{01}{}^1 & W_{11}{}^1 & W_{21}{}^1 \\ W_{02}{}^1 & W_{12}{}^1 & W_{22}{}^1 \end{bmatrix} \quad W^2 = \begin{bmatrix} W_{01}{}^2 & W_{11}{}^2 & W_{21}{}^2 \\ W_{02}{}^2 & W_{12}{}^2 & W_{22}{}^2 \end{bmatrix}$$

$$O^0 = \begin{bmatrix} x_1{}^0 \\ x_2{}^0 \\ x_3{}^0 \end{bmatrix} = \begin{bmatrix} O_1{}^0 \\ O_2{}^0 \\ O_3{}^0 \end{bmatrix} \qquad O^1 = \begin{bmatrix} O_1{}^1 \\ O_2{}^1 \\ O_3{}^1 \end{bmatrix}$$

- Let, feature vector is X = [0.7   1.2]
- Category is class 1 and binary class problem.
- So output of node 1 is 1, and output of node 2 is 0

Feed Forward learning

- Here, $O^0 = \begin{bmatrix} 1 \\ 0.7 \\ 1.2 \end{bmatrix}$ and $t = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$
- Let, $W^1 = \begin{bmatrix} 0.5 & 1.5 & 0.8 \\ 0.8 & 0.2 & -1.6 \end{bmatrix}$

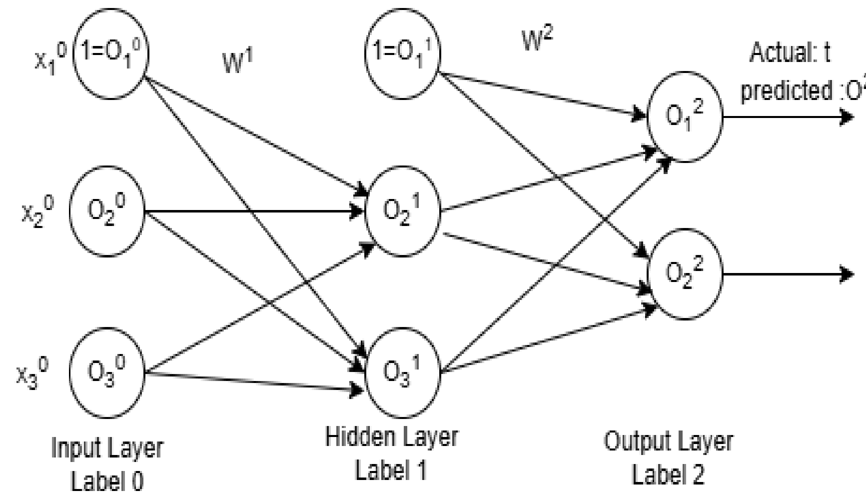


$$\theta^1 = W^1 O^0 = \begin{bmatrix} 0.5 & 1.5 & 0.8 \\ 0.8 & 0.2 & -1.6 \end{bmatrix} \begin{bmatrix} 1 \\ 0.7 \\ 1.2 \end{bmatrix} = \begin{bmatrix} 2.51 \\ -9.8 \end{bmatrix} \quad where, \theta^1 = \begin{bmatrix} \theta_2{}^1 \\ \theta_3{}^1 \end{bmatrix}$$

- Using sigmoid function, we get $\begin{bmatrix} O_2{}^1 \\ O_3{}^1 \end{bmatrix} = \begin{bmatrix} 0.92 \\ 0.27 \end{bmatrix}$
- So, output matrix at hidden layer is: $O^1 = \begin{bmatrix} O_1{}^1 \\ O_2{}^1 \\ O_3{}^1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0.92 \\ 0.27 \end{bmatrix}$

Feed Forward learning

- Here, $O^1 = \begin{bmatrix} O_1{}^1 \\ O_2{}^1 \\ O_3{}^1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0.92 \\ 0.27 \end{bmatrix}$ and $t = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$
- Let, $W^2 = \begin{bmatrix} 0.9 & -1.7 & 1.6 \\ 1.2 & 2.1 & -0.2 \end{bmatrix}$

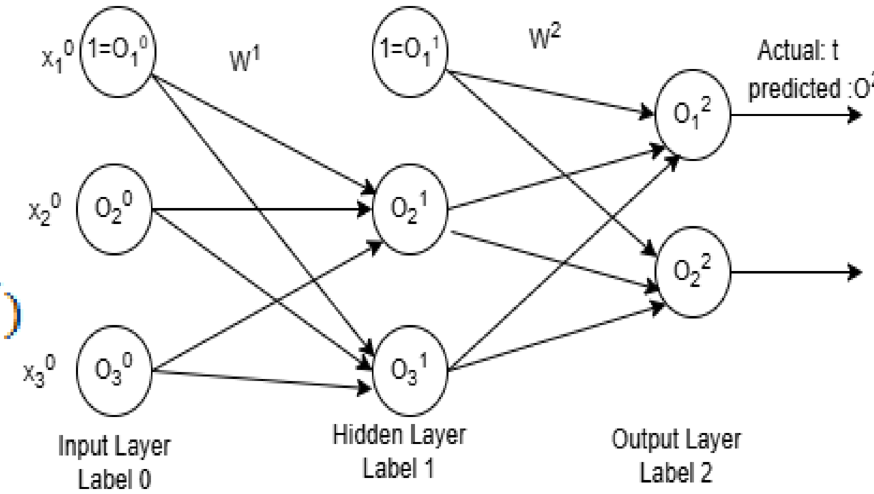


$$\theta^2 = W^2 O^1 = \begin{bmatrix} 0.9 & -1.7 & 1.6 \\ 1.2 & 2.1 & -0.2 \end{bmatrix} \begin{bmatrix} 1 \\ 0.92 \\ 0.27 \end{bmatrix} = \begin{bmatrix} -0.232 \\ 3.057 \end{bmatrix}$$

$$where, \theta^2 = \begin{bmatrix} \theta_1{}^2 \\ \theta_2{}^2 \end{bmatrix}$$

- Using sigmoid function, we get $O^2 = \begin{bmatrix} O_1{}^2 \\ O_2{}^2 \end{bmatrix} = \begin{bmatrix} 0.44 \\ 0.95 \end{bmatrix}$
- This is the predicted output at output layer.
- Actual output for X = [0.7   1.2] is $t = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$, so X is misclassified
- This is all about one iteration of feed forward learning for a single feature vector. We need backprpagation learning.

Backpropagation for weight update



- We know that weight updation rule for output layer is:

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

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

- Here, K=2. So matrix form of weight updation rule is:

$$W^2 = W^2 - \eta \delta^2 O^1$$

- Where, $W^2 = \begin{bmatrix} W_{01}^{2} & W_{11}^{2} & W_{21}^{2} \\ W_{02}^{2} & W_{12}^{2} & W_{22}^{2} \end{bmatrix}$, $\delta^2 = \begin{bmatrix} \delta_1^{2} \\ \delta_2^{2} \end{bmatrix}$ and $O^1 = [O_1^{1} \quad O_2^{1} \quad O_3^{1}]$

- So, $W^2 = W^2 - \eta \begin{bmatrix} \delta_1^{2} \\ \delta_2^{2} \end{bmatrix} [O_1^{1} \quad O_2^{1} \quad O_3^{1}] = W^2 - \eta \begin{bmatrix} \delta_1^{2} O_1^{1} & \delta_1^{2} O_2^{1} & \delta_1^{2} O_3^{1} \\ \delta_2^{2} O_1^{1} & \delta_2^{2} O_2^{1} & \delta_2^{2} O_3^{1} \end{bmatrix}$

- So, $\begin{bmatrix} W_{01}^{2} & W_{11}^{2} & W_{21}^{2} \\ W_{02}^{2} & W_{12}^{2} & W_{22}^{2} \end{bmatrix} = \begin{bmatrix} W_{01}^{2} & W_{11}^{2} & W_{21}^{2} \\ W_{02}^{2} & W_{12}^{2} & W_{22}^{2} \end{bmatrix} - \eta \begin{bmatrix} \delta_1^{2} O_1^{1} & \delta_1^{2} O_2^{1} & \delta_1^{2} O_3^{1} \\ \delta_2^{2} O_1^{1} & \delta_2^{2} O_2^{1} & \delta_2^{2} O_3^{1} \end{bmatrix}$

- Since, all values are known, so weigh vector W^2 is modified.

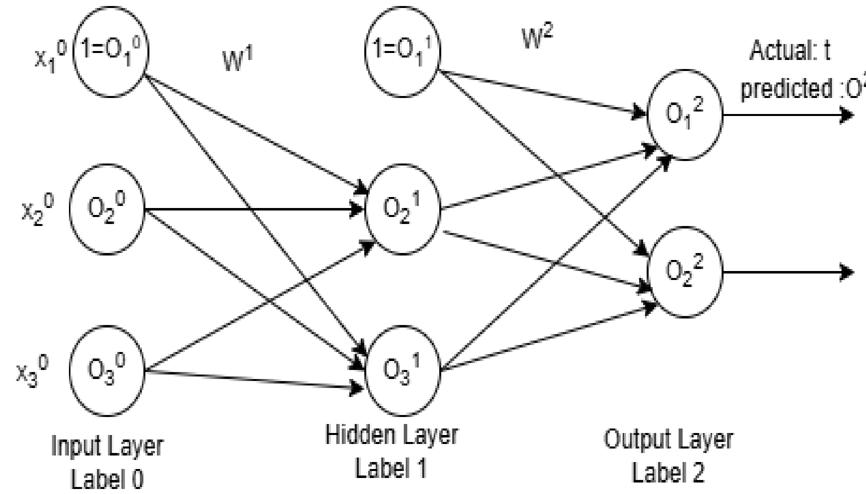
Weight updation rule in other layer

- Weight updation rule is:

$$W_{pi}^{K-1} = W_{pi}^{K-1} - \eta\, \partial_i^{K-1}\, O_p^{K-2}$$

- Where,

$$\partial_i^{K-1} = O_i^{K-1}(1 - O_i^{K-1}) \sum_{j=1}^{M_K} \delta_j^K W_{ij}^K$$



- As K=2, we get modified W^1. Thus all the weights are updated.
- So for a feature vector X, one iteration is complete.
- Similar process is repeated for same X and weights are updated until the predicted value for output neuron -1 is equal or closer to 1 and neuron-2 equal or closer to 0.
- Then training of neural network by X is complete.

Batch Training

- During training, instead of processing one feature vector, say X, at a time, multiple feature vectors (a batch) can be processed simultaneously.

1. Shared weights:

- The neural network starts with a set of initial weights, which are the same for every feature vector in that batch.
- These weights are usually initialized randomly or using specific strategies (like Xavier or He initialization) before training begins.

2. **Forward Pass**:

- During the forward pass, all feature vectors in the batch are passed through the network using these shared weights to compute their outputs.

Batch Training

3. Loss Calculation:

- After obtaining the predictions for all feature vectors, the loss is calculated based on the entire batch.

4. Backpropagation:

- The gradients are computed based on the cumulative loss from the batch, and these gradients are used to update the shared weights.

5. Weight Update:

- After processing the entire batch, the weights are updated once based on the averaged gradients from all feature vectors in that batch.

Training of next Batch

1. Weight Persistence:

- After the neural network processes a batch and updates its weights based on the gradients calculated from that batch, these new weights become the current weights of the network.

2. Next Batch Processing:

- When the next batch is processed, the network uses these updated weights for the forward pass.
- This means that each batch builds upon the learning from the previous batch, allowing the model to incrementally improve its performance.

3. Continuous Updates:

- This cycle continues through all epochs of training, where the weights are continuously updated after processing each batch, progressively refining the model’s understanding of the data.

Parallel Processing

- Training a neural network can involve parallel processing, especially when using multiple feature vectors.
- **Batch Processing**: During training, instead of processing one feature vector at a time, multiple feature vectors (a batch) can be processed simultaneously. This is often done using matrix operations, which are highly optimized for parallel computation on GPUs (Graphics Processing Unit) or TPUs (Tensor Processing Unit).
- **Data Parallelism**: In larger models or datasets, data parallelism can be employed, where the training dataset is split across multiple devices (like multiple GPUs). Each device processes a different subset of the data in parallel, computes the gradients, and then aggregates the results to update the weights.
- **Model Parallelism**: In cases where the model itself is very large and doesn’t fit on a single device, model parallelism can be used. Here, different parts of the model are placed on different devices, and computations for a single input are spread across them.
- **Asynchronous Updates**: In some advanced setups, you can have multiple workers processing different batches and updating the model weights asynchronously, which can further speed up training.

THANK YOU