Rahul Verma (24BM6JP44)

Date 18-Mar-2025

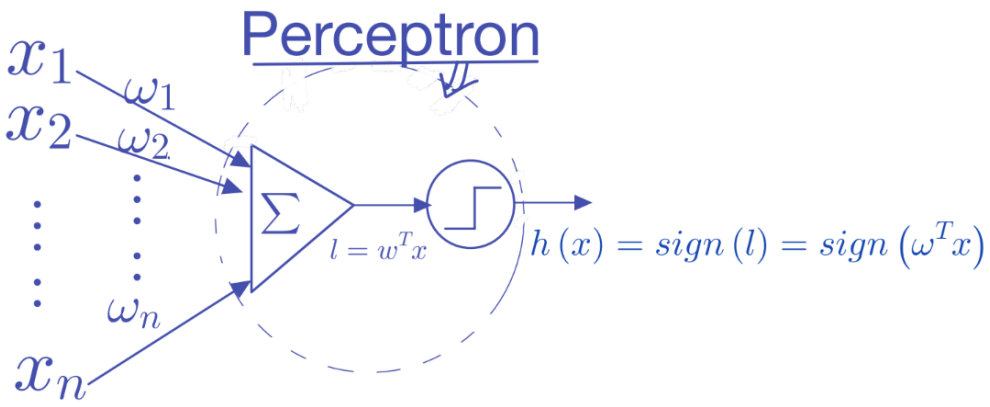
FADML Scribe Notes

Linear Models

Linear models are fundamental in machine learning and statistics. They are primarily used for classification and regression tasks. The two key types of linear models are:

1. **Classification** - Example: Perceptron, Logistic Regression (L.R)
2. **Regression** - Example: Linear Regression (L.R)

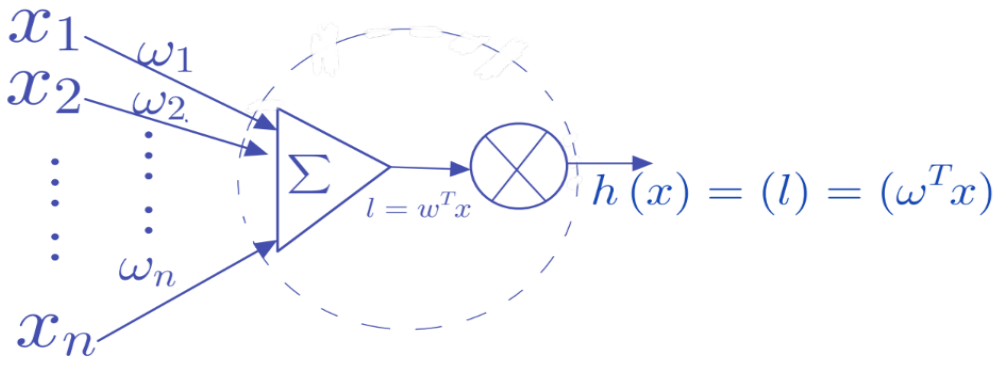
Perceptron (Classification)



The perceptron is a type of linear classifier used for binary classification problems. The perceptron works as follows:

- Inputs $x_1, x_2, \ldots, x_n$ are multiplied by their corresponding weights $w_1, w_2, \ldots, w_n$.
- The weighted sum is computed:
$$s = w^T x$$
- The activation function applies a threshold:
$$h(x) = \textbf{sign}(s) = \textbf{sign}(w^T x)$$
- If the output is positive, it belongs to one class; otherwise, it belongs to the other class.

Linear Regression (L.R)



Linear Regression is used for predicting continuous values. The model follows:

$$h(x) = w^T x$$

This represents a linear relationship between the input features and the predicted value.

Loss Function

Linear regression uses the Mean Squared Error (MSE) as its loss function:

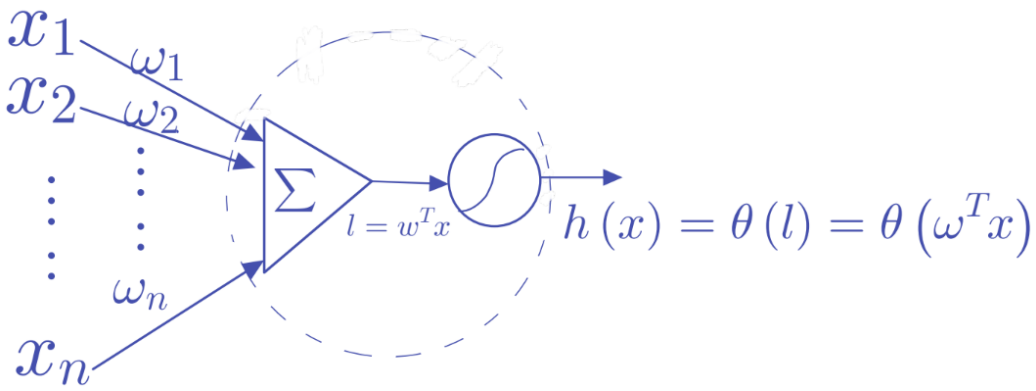
$$L = \frac{1}{N}\sum_{i=1}^{N}(y_i - w^T x_i)^2$$

A closed-form solution for **w** exists using the Normal Equation:

$$w = (X^T X)^{-1} X^T y$$

This allows direct computation of optimal weights without requiring iterative optimization methods like gradient descent.

Logistic Regression (L.R)



For logistic regression, we use the sigmoid activation function:

$$\theta(s) = \frac{e^s}{1+e^s} = \frac{1}{1+e^{-s}}$$

This function outputs values between 0 and 1, making it suitable for probability estimation.

Probability Interpretation

For classification, we model the probability of a label given an input as:

$$p(y|x) = \theta(s), \quad y = +1$$

$$p(y|x) = 1 - \theta(s), \quad y = -1$$

This forms the **basis of logistic regression.**

Loss Function

The objective in logistic regression is to maximize the likelihood function, which leads to minimizing the cross-entropy loss:

$$L = \frac{1}{N}\sum_{i=1}^{N} ln\left(1 + e^{-y_{iw}{}^{T}x_i}\right)$$

This function ensures that the predictions align with the true labels. Unlike linear regression, logistic regression does not have a closed-form solution due to the non-linearity introduced by the sigmoid function. Therefore, we rely on iterative optimization techniques such as gradient descent to find the optimal parameters.

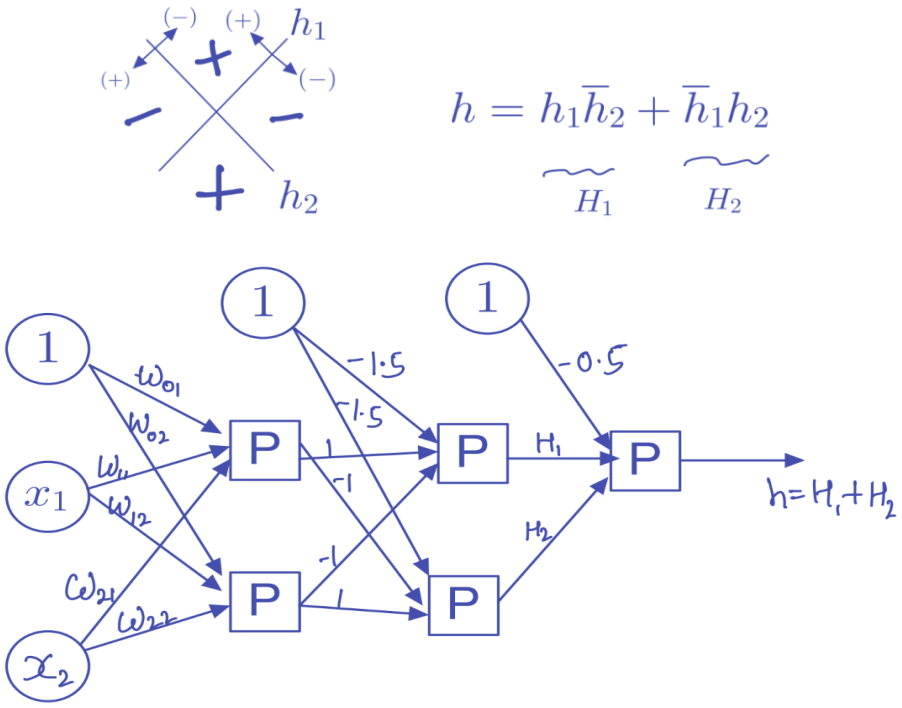
Gradient Descent

To optimize the parameters, we use stochastic gradient descent (SGD), which updates the weights iteratively to minimize the loss function.

Higher Dimensions and Non-Linear Transformation

- By moving to higher dimensions, we can separate data more effectively.
- However, this comes with a computational penalty.
- To handle non-linearly separable data, we use transformations $\phi(x)$.

Multi-Layer Perceptron (MLP)



Here, P is a perceptron

- MLP is a **feedforward** and **fully connected** network, making it a type of **Artificial Neural Network (ANN).**
- It consists of multiple layers:
  - Input layer
  - Hidden layers
  - Output layer
- Each layer applies weights, biases, and activation functions.
- The perceptron-based MLP is not differentiable due to the step activation function. This limitation makes it unsuitable for gradient-based learning.
- To overcome this, alternative activation functions (such as sigmoid, ReLU, and tanh) are used, transforming the network into what we now call a **Neural Network.**

Pros and Cons of MLP

Pros:

- Can model complex patterns beyond simple linear separability.
- Enables hierarchical feature extraction through multiple layers.
- Fully connected structure allows information to propagate efficiently.

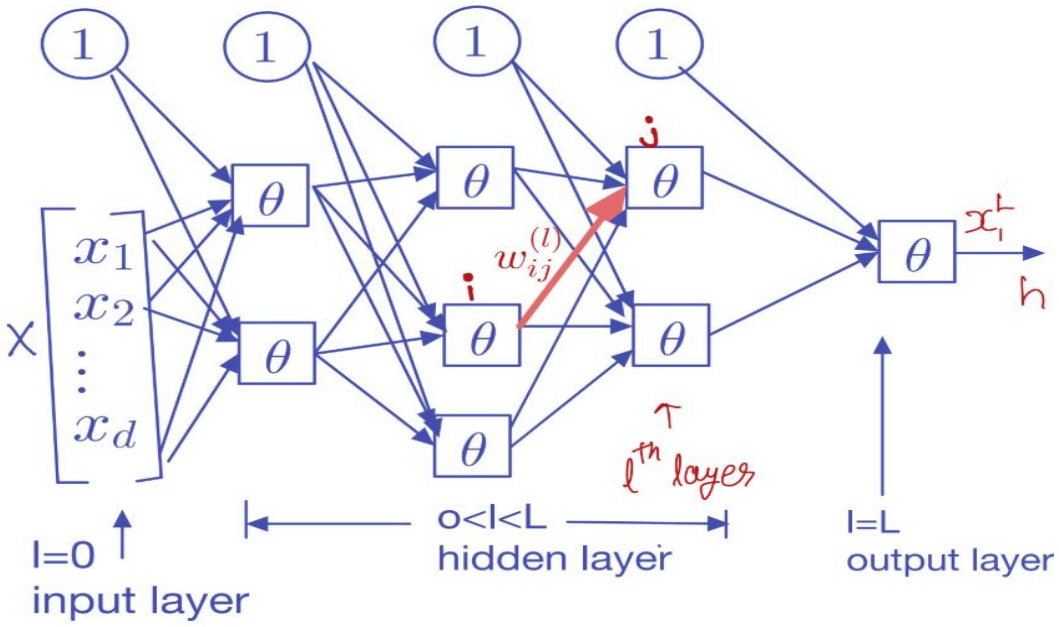
Cons:

- Perceptron-based MLP lacks differentiability, making training difficult.
- Computationally expensive due to fully connected layers.
- Prone to overfitting without proper regularization techniques

Neural Networks

Neural networks build upon MLP by introducing differentiable activation functions, allowing for efficient training using backpropagation. A neural network consists of multiple layers:

1. **Input Layer**: Accepts feature inputs.
2. **Hidden Layers**: Applies non-linear transformations through activation functions.
3. **Output Layer**: Produces final predictions.



$$Wij(l) = \{1 \le l \le L, \quad l\ layers$$
$$0 \le i \le d^{(l-1)}, \quad input\ layer$$
$$1 \le j \le d^{(l)}, \quad output\ layer$$

For every $\boldsymbol{i, j, l}$.

Error Computation and Weight Updates

For a given network, the total error is defined as:

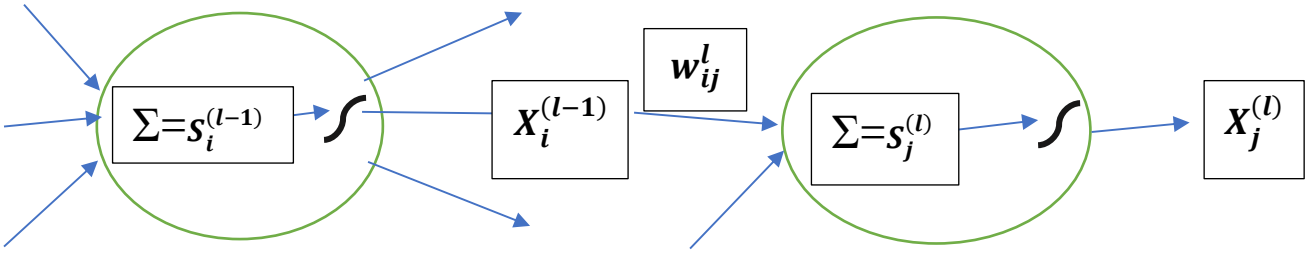
$e(w) =$ error (or loss) function

The choice of the **error function** (or loss function) depends on the type of problem being solved. Some commonly used error functions include:

- **Mean Squared Error (MSE):** Used for regression problems
- **Huber Loss:** Used for robust regression, combining MSE and MAE.
- **Hinge Loss:** Used in SVM-based classification.

To minimize the error, we adjust the weights iteratively using optimization techniques. This requires calculating how the error changes with respect to the weights, which is done by taking the derivative of the loss function.

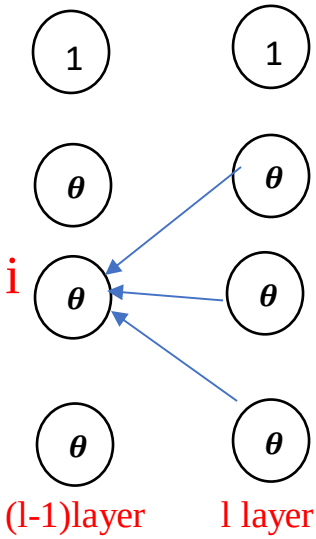
To understand this process, we first analyze it for a **single neuron**:



$$X_j^{(l)} = \theta(S_j^{(l)}) = \theta\left(\sum_{i=0}^{d^{l-1}} X_i^{(l-1)} w_{ij}^{l}\right)$$

$$\frac{\partial e(w)}{\partial w_{ij}^{(l)}} = \frac{\partial e(w)}{\partial S_j^{(l)}} \times \frac{\partial S_j^{(l)}}{\partial w_{ij}^{(l)}} = \delta_j^{(l)} \times X_i^{(l-1)}$$

To facilitate a clearer understanding and generalize the above formula, we first analyze a **partial structure of the network**, focusing on the relationship between layers:



$$\boldsymbol{\delta_i^{(l-1)} = f(\delta_j^{(l)})}$$

$$\boldsymbol{\delta_i^{(l-1)} = \frac{\partial e(w)}{\partial S_i^{(l-1)}} = \sum_{j=1}^{d^l} \frac{\partial e(w)}{\partial S_j^{(l)}} \times \frac{\partial S_j^{(l)}}{\partial x_i^{(l-1)}} \times \frac{\partial X_i^{(l-1)}}{\partial S_i^{(l-1)}} = \sum_{j=1}^{d^l} \delta_j^{(l)} \times w_{ij}^{(l)} \times \theta'(S_i^{(l-1)})}$$

This formulation helps generalize the **backpropagation process**, ensuring efficient weight updates across all layers of the network.

To update weights, we compute gradients using backpropagation:

As we had propagated through one previous layer, so by applying recursion, we efficiently update weights across all layers. This process follows a structured approach:

1. Compute gradients at the output layer using the chain rule.
2. Propagate these gradients backward through hidden layers.
3. Recursively apply the weight update formula until all layers are updated.

This recursive approach ensures efficient parameter tuning, making deep learning models scalable and trainable regardless of depth.

Steps for Backpropagation

1. **Initialize Weights and Biases**
   - Assign small random values to weights and biases.
2. **Loop Through Epochs**
   - Repeat the process multiple times for better optimization.
3. **Select an Input Sample**
   - Choose one input and feed it into the neural network.
4. **Forward Propagation**
   - Compute weighted sums and apply activation functions.
5. **Compute Loss**
   - Compare predicted and actual values using an error function.
6. **Compute Gradients (Backpropagation)**
   - Calculate how the error changes with respect to each weight.
7. **Update Weights Using Gradient Descent**
   - Update rule:
   $$w_{new} = w_{old} + \eta \frac{\partial L}{\partial w}$$
   - η is the learning rate.
8. **Recursive Weight Updates for Deeper Networks**
   - Extend weight updates using recursion for any depth.