Introduction to Deep Learning: Part II

June 23, 2025

How to Train a Deep Neural Network?

Training an artificial neural network (ANN) aims to optimize its parameters so that the model achieves the best possible performance based on a specific evaluation criterion. This process involves adjusting the network's weights and biases to minimize the difference between predicted and actual outputs.

The general steps for training a deep neural network can be summarized as follows:

1. **Initializing Parameters**: Set up the initial weights and biases of the network, typically using random initialization techniques.

2. **Forward Propagation**: Pass input data through the network to compute predictions based on current parameters.

3. **Loss Function**: Measure the difference between the network's predictions and the actual target values.

4. **Backward Propagation (Gradient Descent)**: Compute gradients of the loss function with respect to the parameters and update them to minimize the error.

Steps 2–4 are repeated iteratively until the model converges to an optimal solution, ensuring it generalizes well to new data.

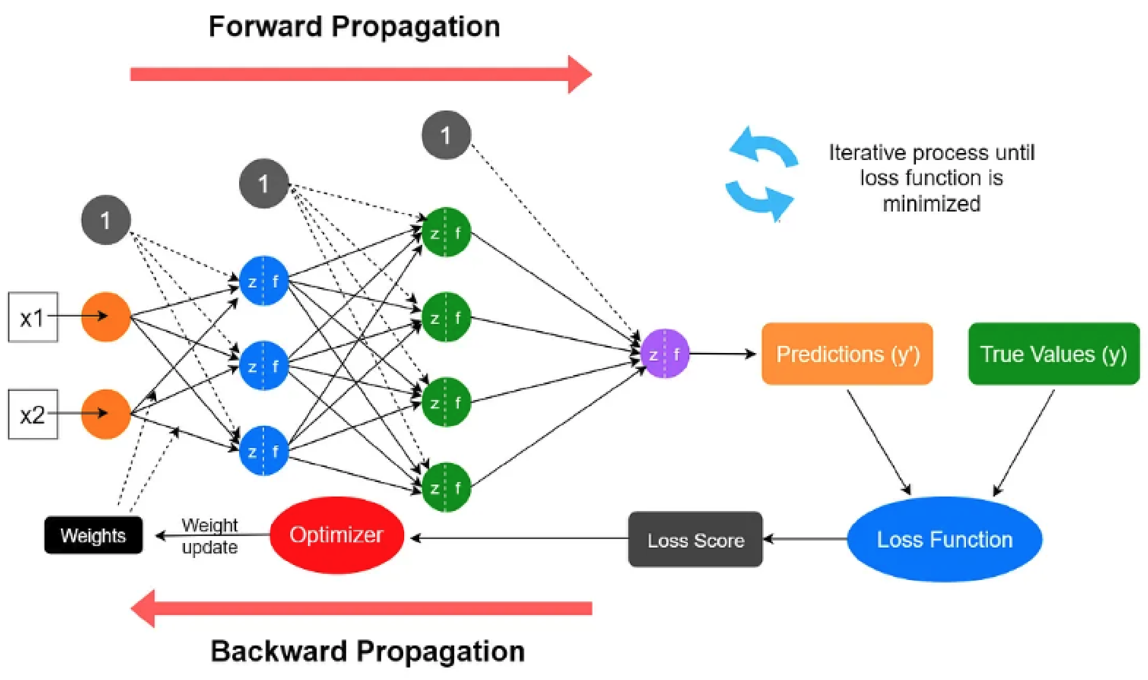


Figure 1: Training a Deep Neural Network.

0.1 Initialization of Network Parameters

The first step in training a deep neural network is to initialize the weights w and biases b with appropriate values. Proper initialization ensures efficient learning, prevents vanishing or exploding gradients, and improves convergence speed.

There are several methods for initializing network parameters:

1. **Zero Initialization**
   - Sets all weights and biases to zero.
   - **Not recommended** for deep learning because it causes all neurons to learn the same features, leading to ineffective training.

2. **Constant Initialization**
   - Assigns a fixed constant value to all weights and biases.
   - **Rarely used** as it suffers from the same symmetry issues as zero initialization.

3. **Uniform Initialization**
   - Draws weights from a **uniform distribution**.
   - **Less commonly used** compared to random initialization with normal distributions.

4. **Normal Initialization**
   - Weights and biases are sampled randomly from a **normal distribution**.
   - **Most commonly used** as it helps break symmetry and allows diverse learning across neurons.

5. **Xavier (Glorot) Initialization**
   - Weights are drawn from a **normal distribution with mean** 0 **and variance** $\frac{1}{n}$, where n is the number of neurons in the previous layer.
   - **Best suited for sigmoid and tanh activation functions** as it maintains the variance of activations across layers.

6. **He Initialization**
   - Weights are drawn from a normal distribution with mean 0 and variance $\frac{2}{n}$, where n is the number of neurons in the previous layer.
   - **Recommended for ReLU and Leaky ReLU activation functions** to prevent the vanishing gradient problem.

7. **Orthogonal Initialization**
   - Weights are initialized using an **orthogonal matrix**, preserving gradient norms during backpropagation.

- **Useful in deep architectures** where maintaining numerical stability is critical.

Proper weight initialization is fundamental to stabilizing the learning process and ensuring smooth gradient flow during backpropagation. Choosing the right strategy depends on the activation function and network architecture.

0.2 Forward Propagation

Forward propagation is the process in which input data is passed through the layers of a neural network to compute the output. Mathematically, it can be expressed as:

$$\hat{y} = NN(x;\theta) = W^{(l)} f^{(l)} \odot \left(W^{(l-1)} \odot f^{(l-1)}(\cdots) + b_{l-1} \right) + b_l \tag{1}$$

where:

- $\hat{y}$ is the predicted output of the network.
- θ represents all trainable parameters, i.e., $\theta = \{W^{(l)}, b^{(l)}, \ldots, W^{(1)}, b^{(1)}\}$.
- $W^{(l)}$ and $b^{(l)}$ are the weights and biases of layer l.
- $f^{(l)}$ is the activation function applied at layer l.
- $\odot$ represents element-wise multiplication.

Steps of Forward Propagation:

1. **Input Layer**: The input data x is fed into the network.
2. **Hidden Layers**: Each hidden layer applies a linear transformation followed by a non-linear activation function:

   $$z^{(l)} = W^{(l)} h^{(l-1)} + b^{(l)}, \quad \text{where} \quad h^{(l)} = f^{(l)}(z^{(l)}),$$

   where $h^{(l)}$ is the activation at layer l.
3. **Output Layer**: The final layer produces the predicted output $\hat{y}$, which depends on the chosen activation function (e.g., softmax for classification, linear for regression).
4. **Loss Calculation**: The predicted output is compared to the actual target y using a loss function, which quantifies the model's error.

Forward propagation is crucial for making predictions, as it computes how input features transform into outputs based on the current model parameters. The loss computed at the end of forward propagation is used in the next step—backward propagation—to update the weights and improve the model's performance.

0.3 Loss Function (or Cost Function)

The loss function quantifies how well a neural network performs a given task. It measures the discrepancy between the predicted output $\hat{y}$ and the actual target y. The goal of training is to minimize this loss, ensuring that the network's predictions are as accurate as possible.

At a high level, loss is computed as follows:

1. Pass each training example (x_i, y_i) through the network $NN(x; \theta)$ to obtain a prediction $\hat{y}_i = NN(x_i; \theta)$.
2. Compare $\hat{y}_i$ with the actual label y_i using a loss function.
3. Compute the average loss over all training examples:

$$\mathcal{L}(\theta) = \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2$$

   where θ represents all the trainable parameters in the network.

The choice of loss function depends on the type of problem: **regression** (predicting continuous values) or **classification** (predicting discrete labels).

0.3.1 Regression Loss Functions

1. **Mean Squared Error (MSE or L2 Loss)**: MSE calculates the average squared difference between the predicted and actual values:

$$\mathcal{L} = \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2$$

   - Squaring the error amplifies larger deviations, making MSE sensitive to outliers.
   - It is differentiable, making it suitable for gradient-based optimization.
   - Ideal when the target outputs follow a normal (Gaussian) distribution.

2. **Mean Absolute Error (MAE or L1 Loss)**: MAE computes the average absolute difference between predictions and actual values:

$$\mathcal{L} = \frac{1}{N} \sum_{i=1}^{N} |y_i - \hat{y}_i|$$

   - Less sensitive to outliers compared to MSE.
   - Not differentiable at $y_i = \hat{y}_i$, which can make optimization trickier.

3. **Huber Loss (Smooth L1 Loss)**: Huber loss combines the advantages of MSE and MAE by being quadratic for small errors and linear for large errors:

$$\mathcal{L}_\delta(y_i, \hat{y}_i) = \begin{cases} \frac{1}{2}(y_i - \hat{y}_i)^2, & \text{if } |y_i - \hat{y}_i| \leq \delta \\ \delta \cdot (|y_i - \hat{y}_i| - \frac{1}{2}\delta), & \text{otherwise} \end{cases}$$

- Robust to outliers while maintaining differentiability.
- The hyperparameter δ determines the transition point between MSE-like and MAE-like behavior.

0.3.2 Classification Loss Functions

For classification problems, loss functions measure how well a model distinguishes between classes.

1. **Binary Cross-Entropy (Log Loss)**: Used for **binary classification** tasks where the model outputs a probability between 0 and 1.

$$\mathcal{L} = -\frac{1}{N}\sum_{i=1}^{N}\left(y_i \log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i)\right)$$

- Penalizes incorrect predictions and encourages high confidence in correct ones.
- Works well when the labels are binary (e.g., spam vs. not spam).
- The logarithmic penalty heavily punishes confident incorrect predictions.

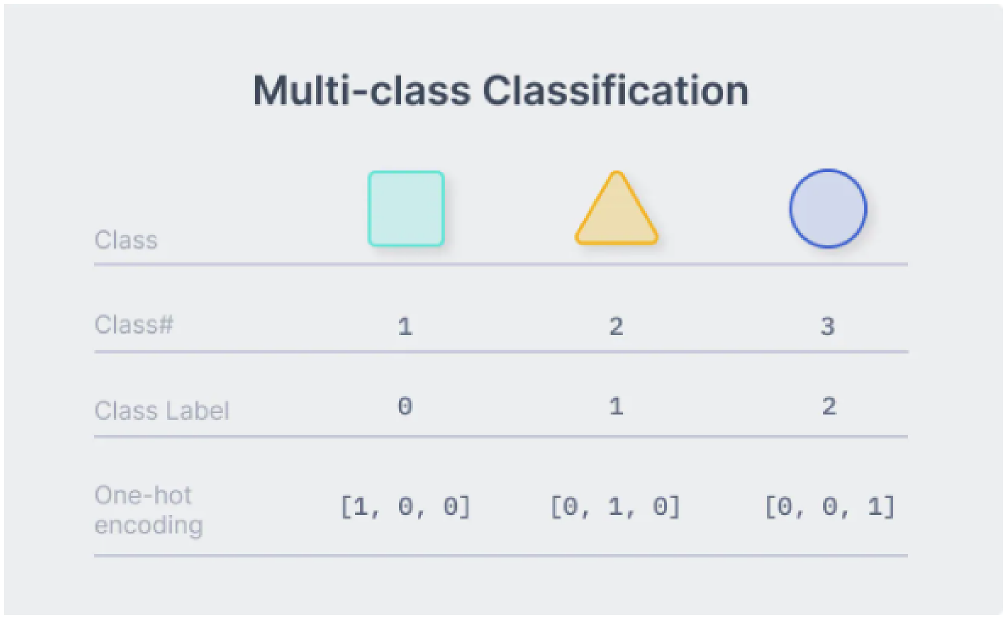
2. **Categorical Cross-Entropy (Softmax Loss)**: Used for **multi-class classification**, where the model predicts a **probability distribution** over multiple classes. The **softmax function** ensures that all predicted class probabilities sum to 1.

$$\mathcal{L}(y_i, \hat{y}_i) = -\sum_{j}^{C} t_j \log\left(\frac{e^{s_p}}{\sum_j^C e^{s_j}}\right) = -\log\left(\frac{e^{s_p}}{\sum_j^C e^{s_j}}\right),$$

where

$$y_i = (t_1, \cdots, t_C), \quad \hat{y}_i = (s_1, \cdots, s_C)$$

- C is the number of classes.
- s_j represents the predicted score for class j.
- s_p is the score of the correct class.
- One-hot encoded target vectors ensure only the correct class contributes to the loss, i.e., $t_i = 0, \forall i \neq p$ and $t_p = 1$.
- The model is trained to assign **higher probability to the correct class**.
- Works well when labels are **one-hot encoded** (only one class per example).



Multi-class Classification

Class	■	▲	●
Class#	1	2	3
Class Label	0	1	2
One-hot encoding	[1, 0, 0]	[0, 1, 0]	[0, 0, 1]

Figure 2: The multi-class classification

0.3.3 A summary of loss functions

Task Type	Common Loss Functions	Strengths	Weaknesses
Regression	MSE (L2), MAE (L1), Huber	MSE is smooth and differentiable; MAE is robust to outliers; Huber balances both.	MSE is sensitive to outliers; MAE is non-differentiable at zero.
Binary Classification	Binary Cross-Entropy	Encourages confident and correct predictions.	Strongly penalizes uncertain predictions.
Multi-Class Classification	Categorical Cross-Entropy	Works well with softmax activation for probability distribution.	Not suitable for multi-label classification.

0.4 Backward Propagation (Optimization with Gradient Descent)

Backpropagation, short for **Backward Propagation of Errors**, is a fundamental algorithm used to train artificial neural networks. Its primary objective is to minimize the discrepancy between the model's predicted output and the actual output by iteratively adjusting the network's **weights** and **biases**. This is achieved by computing the gradient of the loss function with respect to each parameter and updating them using an optimization technique, typically **gradient descent**.

The training process aims to minimize the following loss function:

$$\theta^* = \arg\min_{\theta} \mathcal{L}(\theta) = \frac{1}{N}\sum_{i=1}^{N}(y_i - NN(x_i;\theta))^2$$

where:

- $\mathcal{L}(\theta)$ represents the loss function,
- N is the total number of training samples,
- y_i is the true output,
- $NN(x_i;\theta)$ is the neural network's predicted output given input x_i and parameters θ.

0.4.1 Understanding Backpropagation

Backpropagation is a crucial technique in deep learning, primarily for training neural network models. It is an **iterative process** that seeks to **minimize the cost function by updating model parameters** in the direction of the negative gradient. The algorithm computes the gradient of the loss function with respect to each parameter using the **chain rule of calculus** to propagate the error backward through the network. The chain rule is applied layer by layer, starting from the output layer and moving backward to the input layer. This enables the computation of the partial derivatives of the loss function with respect to each parameter efficiently. Finally, it updates the network's parameters by modifying weights and biases using gradient descent or its variants.

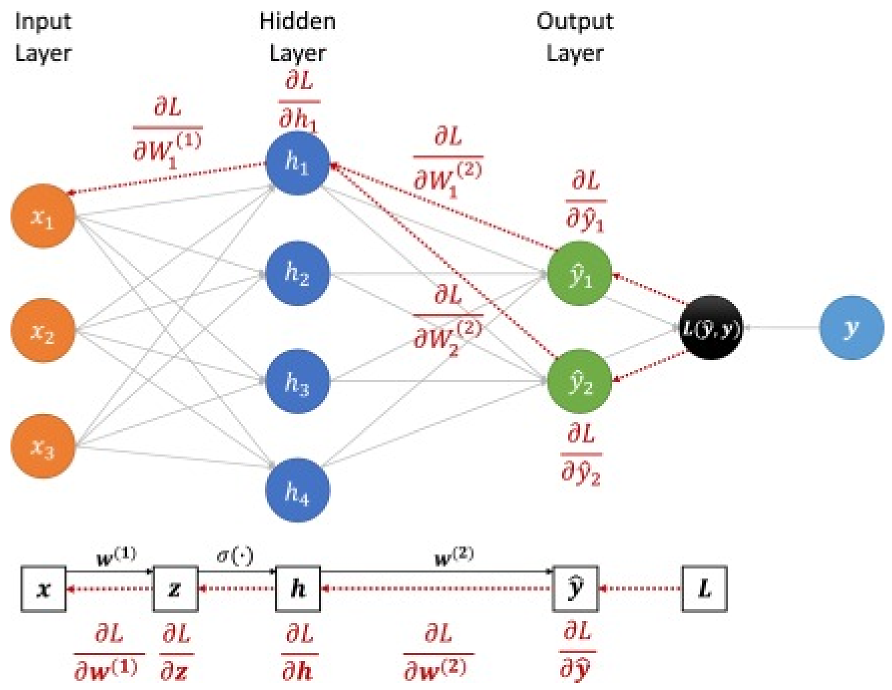

Input Layer
Hidden Layer
Output Layer
$\frac{\partial L}{\partial W_1^{(1)}}$
$\frac{\partial L}{\partial h_1}$
$\frac{\partial L}{\partial W_1^{(2)}}$
$\frac{\partial L}{\partial \hat{y}_1}$
$\frac{\partial L}{\partial W_2^{(2)}}$
$\frac{\partial L}{\partial \hat{y}_2}$
x_1
x_2
x_3
h_1
h_2
h_3
h_4
$\hat{y}_1$
$\hat{y}_2$
$L(\hat{y}, y)$
y
$w^{(1)}$
$\sigma(\cdot)$
$w^{(2)}$
x
z
h
$\hat{y}$
L
$\frac{\partial L}{\partial w^{(1)}}$
$\frac{\partial L}{\partial z}$
$\frac{\partial L}{\partial h}$
$\frac{\partial L}{\partial w^{(2)}}$
$\frac{\partial L}{\partial \hat{y}}$

Figure 3: Backpropagation Process

Mathematical Formulation of Backpropagation

For a single neuron in a layer, the output is given by:

$$z = w_1x_1 + w_2x_2 + \cdots + w_nx_n + b$$

where x_i are the inputs, w_i are the corresponding weights, b is the bias term, and z is the weighted sum passed into the activation function. The activation function $f(z)$ produces the neuron's output:

$$a = f(z)$$

To compute the gradients for weight updates, we use the chain rule to determine how changes in weights affect the loss $\mathcal{L}$:

$$\frac{\partial \mathcal{L}}{\partial w_i} = \frac{\partial \mathcal{L}}{\partial a} \cdot \frac{\partial a}{\partial z} \cdot \frac{\partial z}{\partial w_i}$$

Similarly, for the bias:

$$\frac{\partial \mathcal{L}}{\partial b} = \frac{\partial \mathcal{L}}{\partial a} \cdot \frac{\partial a}{\partial z} \cdot \frac{\partial z}{\partial b}$$

Using these gradients, the weights and biases are updated using gradient descent:

$$w_i \leftarrow w_i - \eta \frac{\partial \mathcal{L}}{\partial w_i}, \quad b \leftarrow b - \eta \frac{\partial \mathcal{L}}{\partial b}$$

where η is the learning rate.

Key Benefits of Backpropagation

- **Efficient Weight Updates**: The algorithm computes gradients using the chain rule, making it possible to update weights systematically.
- **Scalability**: Backpropagation is highly scalable and works well with deep networks comprising multiple layers.
- **Automated Learning**: It automates the process of learning from data, enabling the model to optimize its performance with minimal manual intervention.

0.4.2 Gradient Descent: The Optimization Algorithm

What is Gradient Descent?

Gradient Descent is a fundamental optimization algorithm used in machine learning to minimize a given cost function by iteratively adjusting model parameters. The core idea is to move in the direction of the **negative gradient of the loss function** to find its minimum. The parameter update rule in gradient descent is given by:

$$\theta_{t+1} = \theta_t - \eta \frac{\partial \mathcal{L}(\theta_t)}{\partial \theta_t} \tag{1}$$

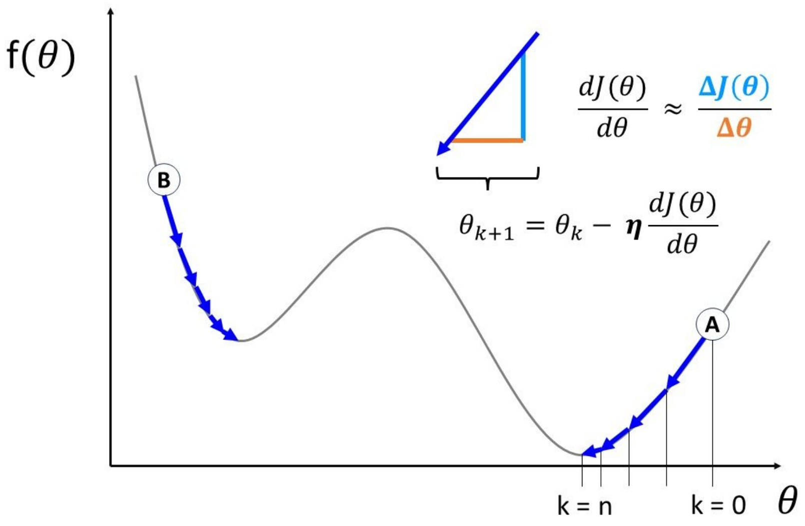
where:

- θ_t represents the model parameters at iteration t,
- η (learning rate) controls the step size of each update,
- $\frac{\partial \mathcal{L}(\theta_t)}{\partial \theta_t}$ is the gradient of the loss function with respect to θ_t.

The Importance of Learning Rate

In (1), the step size is evaluated and updated according to the behavior of the cost function $\mathcal{L}$. The higher the gradient of the cost function, the steeper the slope and the faster a model can learn. Therefore, a high learning rate η results in a higher step value, and a lower learning rate η results in a lower step value. If the gradient of the cost function is zero, the model stops learning.

The **learning rate** η is a tuning parameter that determines the step size at each iteration of gradient descent. It plays an important part in ensuring a balance between optimization time and accuracy. A **small** η means a small step size which usually leads to slow convergence. However, if η is **too large**, we could miss the minimum point completely, which leads to the risk of overshooting the minimum.



$f(\theta)$
$\frac{dJ(\theta)}{d\theta} \approx \frac{\Delta J(\theta)}{\Delta\theta}$
$\theta_{k+1} = \theta_k - \boldsymbol{\eta}\frac{dJ(\theta)}{d\theta}$
B
A
k = n
k = 0
θ

Figure 4: The gradient descent (Figure from `https://www.makerluis.com/gradient-descent/`)

Variants of Gradient Descent

- **1. Stochastic Gradient Descent (SGD)**: This variant suggests model update using a **single training example at a time**, which does not require a large amount of computation and therefore is suitable for large datasets. Thus, they are stochastic and can produce noisy updates, and therefore may require a careful selection of learning rates.

$$\theta_{t+1} = \theta_t - \eta\frac{\partial \mathcal{L}_i(\theta_t)}{\partial \theta_t}, \quad \text{where} \quad \mathcal{L}_i(\theta_t) = (y_i - NN(x_i;\theta))^2$$

  - **Pros**: Faster updates, suitable for large datasets; Can escape local minima due to noisy updates.
  - **Cons**: High variance in updates, may lead to instability.

- **2. Mini-Batch Gradient Descent**: Mini-batch GD updates parameters using a **small batch** of training examples, rather than a single example (SGD) or the full dataset (batch GD). It aims to achieve a balance between the amount of time and precision. Mini-batch GD converges faster than SGD and is used widely in practice to train many deep-learning models.

$$\theta_{t+1} = \theta_t - \eta \frac{\partial \mathcal{L}_b(\theta_t)}{\partial \theta_t}, \quad \text{where} \quad \mathcal{L}_b(\theta_t) = \frac{1}{m}\sum_{i=1}^{m}(y_i - NN(x_i;\theta))^2$$

  where $m \ll n$ is the mini-batch size.

  - **Pros**: Reduces variance while still being computationally efficient.

- **3. Momentum-Based Gradient Descent**: Momentum improves SGD by **adding the information of the preceding steps** of the algorithm to the next step. Adding a portion of the current update vector to the previous update enables the algorithm to **penetrate through flat areas and noisy gradients** to help minimize the time to train and find convergence.

$$\theta_{t+1} = \theta_t - \eta \frac{\partial \mathcal{L}_i(\theta_t)}{\partial \theta_t} + \alpha \Delta\theta_t$$

  where $\Delta\theta_t$ indicates the weight change at the previous step and $\alpha \in (0, 1)$ is the momentum coefficient (typically 0.9).

  - **Pros**: Helps escape local minima; Reduces oscillations and speeds up convergence.

0.5 Summary: Training Deep Neural Networks

Training a deep neural network (DNN) involves several key steps, from defining the model and loss function to optimizing the parameters using backpropagation and gradient descent. This process ensures that the network learns to make accurate predictions by iteratively adjusting its weights and biases.

- **(1) Parameter Initialization**
  - Initializing network's parameters with suitable initialization techniques, such as Normal Initialization and Xavier (Glorot) Initialization.

- **(2) Forward Propagation**
  - In forward propagation, input data is passed through multiple layers of the network.

  - Each neuron applies a linear transformation followed by a nonlinear activation function to generate outputs.
  - The final layer produces predictions, which are then compared to the actual target values.

- **(3) Loss Function**
  - The loss function measures how far the model's predictions are from the actual values.
  - Common loss functions include Mean Squared Error (MSE) for regression and Cross-Entropy Loss for classification.
  - The objective of training is to minimize this loss function by updating the model parameters.

- **(4) Backpropagation (Optimization with Gradient Descent)**
  - Backpropagation calculates the gradient of the loss function with respect to each model parameter using the chain rule of calculus.
  - These gradients are then used to update the parameters in the direction that minimizes the loss.
  - This optimization is typically performed using gradient descent, which iteratively updates parameters using:

    $$\theta_{t+1} = \theta_t - \eta \frac{\partial \mathcal{L}(\theta_t)}{\partial \theta_t}$$

    where η is the learning rate.