UNIT-3

ARTIFICIAL NEURAL NETWORK

PARADIGMS OF LEARNING

Supervised Learning

In supervised learning, the neural network is trained on a labeled dataset, where each input is paired with the correct output. The goal is to learn a mapping from inputs to outputs by minimizing the error between the predicted output and the true output. Common applications include image classification, speech recognition, and language translation.

- **Examples**: Image classification using convolutional neural networks (CNNs), language modeling with recurrent neural networks (RNNs) or transformers.

2. Unsupervised Learning

Unsupervised learning involves training a neural network on data that is not labeled. The goal is to uncover hidden patterns or structures in the data. Techniques in unsupervised learning include

clustering, dimensionality reduction, and generative models.

- **Examples: Generative adversarial networks (GANs) for creating synthetic data, autoencoders for data compression and denoising, clustering algorithms like k-means.**

3. Semi-Supervised Learning

This paradigm combines a small amount of labeled data with a large amount of unlabeled data. The idea is to leverage the large amount of unlabeled data to improve the learning accuracy of the model, which is particularly useful when labeled data is scarce or expensive to obtain.

- **Examples: Using techniques like self-training or consistency regularization to improve classification performance with limited labeled samples.**

4. Reinforcement Learning

In reinforcement learning, the neural network learns by interacting with an environment and receiving rewards or penalties. The objective is to learn a policy that maximizes cumulative rewards over time. This approach is particularly useful in scenarios involving decision-making and sequential tasks.

- **Examples: Training agents to play games like Go or chess, robotic control, autonomous driving.**

5. Self-Supervised Learning

Self-supervised learning is a type of unsupervised learning where the system generates its own labels from the data. This approach often involves creating auxiliary tasks where the model predicts parts of the data from other parts. It helps in learning useful representations from unlabeled data.

- **Examples: Contrastive learning methods where the model learns to distinguish between similar and dissimilar data samples, such as in pre-training transformers for NLP tasks.**

6. Transfer Learning

Transfer learning involves taking a neural network model that has been trained on one task and fine-tuning it for a different but related task. This approach leverages pre-trained models to improve learning efficiency and performance on new tasks.

- **Examples: Fine-tuning a pre-trained image classifier for a specific domain or adapting a language model to a new language or domain.**

7. Few-Shot Learning

Few-shot learning aims to train models that can generalize well from a very limited number of examples. This paradigm is challenging because it requires the model to leverage prior knowledge or learned representations effectively.

- **Examples: Meta-learning techniques where the model learns how to learn from few examples, or using Siamese networks to compare and classify data with minimal examples.**

Each of these paradigms addresses different types of learning problems and challenges, and the choice of paradigm often depends on the nature of the data, the task at hand, and the specific goals of the project.

USING TRAINING SAMPLES

Data Preparation

1.1 Collection: Training samples must be collected and organized into datasets. These samples can come from various sources depending on the task, such as images, text, or numerical data.

1.2 Preprocessing: Before training, the data often requires preprocessing, which may include:

- **Normalization/Standardization: Scaling features to a standard range or distribution.**
- **Augmentation: Creating variations of the training data (e.g., rotating or flipping images) to improve generalization.**
- **Tokenization: Converting text into a format suitable for the model, such as word embeddings.**

1.3 Splitting: The data is typically split into training, validation, and test sets:

- **Training Set: Used to train the model.**
- **Validation Set: Used to tune hyperparameters and evaluate the model's performance during training.**
- **Test Set: Used to assess the final performance of the trained model.**

2. Training Process

2.1 Forward Pass: During the forward pass, each training sample is fed into the neural network. The network processes the sample through its layers, applying weights and activation functions, and produces an output.

2.2 Loss Calculation: The output is compared to the true label (for supervised learning) using a loss function. The loss function calculates the difference between the predicted output and the true label, providing a measure of error.

2.3 Backward Pass (Backpropagation): The error calculated by the loss function is propagated backward through the network to update the weights. This process involves:

- **Gradient Computation: Calculating the gradient of the loss function with respect to each weight using techniques like the chain rule.**
- **Weight Update: Adjusting the weights in the direction that reduces the error, typically using optimization algorithms like Stochastic Gradient Descent (SGD) or Adam.**

2.4 Epochs: The process of forward pass, loss calculation, and backward pass is repeated for several iterations or epochs. Each epoch involves passing the entire training dataset through the network.

3. Optimization and Regularization

3.1 Optimization Algorithms:

- **Stochastic Gradient Descent (SGD): Updates weights using the gradient of a single sample or a small batch of samples.**
- **Mini-Batch Gradient Descent: Uses small batches of samples to update weights, combining the benefits of both batch and stochastic gradient descent.**
- **Advanced Optimizers: Algorithms like Adam, RMSprop, and AdaGrad adapt learning rates based on gradients' history.**

3.2 Regularization Techniques:

- **Dropout: Randomly drops neurons during training to prevent overfitting.**
- **L2 Regularization (Weight Decay): Adds a penalty to the loss function proportional to the square of the weights to discourage large weights.**
- **Early Stopping: Monitors performance on the validation set and stops training when performance begins to degrade, preventing overfitting.**

4. Validation and Hyperparameter Tuning

4.1 Validation Set: After each epoch or after a set number of epochs, the model is evaluated on the validation set to monitor its performance. This helps in tuning hyperparameters and making decisions about when to stop training.

4.2 Hyperparameter Tuning:

- **Learning Rate: Determines how much to adjust weights during training.**
- **Batch Size: The number of samples processed before updating weights.**
- **Number of Layers and Neurons: Architecture choices that affect the network's capacity and performance.**

5. Testing and Evaluation

5.1 Test Set Evaluation: Once training is complete, the final model is evaluated on the test set to assess its performance on unseen data. Metrics such as accuracy, precision, recall, F1 score, or mean squared error (depending on the task) are used to gauge performance.

5.2 Model Deployment: If the model performs well, it can be deployed in a real-world application where it makes predictions or classifications based on new data.

GRADIENT OPTIMIZATION PROCEDURE

Gradient Descent in Machine Learning

What is Gradient?

A gradient is nothing but a derivative that defines the effects on outputs of the function with a little bit of variation in inputs.

What is Gradient Descent?

Gradient Descent stands as a cornerstone orchestrating the intricate dance of model optimization. At its core, it is a numerical optimization algorithm that aims to find the optimal parameters—weights and biases—of a neural network by minimizing a defined cost function.

Gradient Descent (GD) is a widely used optimization algorithm in machine learning and deep learning that minimises the cost function of a neural network model during training. It works by iteratively adjusting the weights or parameters of the model in the direction of the negative gradient of the cost function until the minimum of the cost function is reached.

The learning happens during the backpropagation while training the neural network-based model. There is a term known as Gradient Descent, which is used to optimize the weight and biases based on the cost function. The cost function evaluates the difference between the actual and predicted outputs.

Gradient Descent is a fundamental optimization algorithm in machine learning used to minimize the cost or loss function during model training.

- **It iteratively adjusts model parameters by moving in the direction of the steepest decrease in the cost function.**
- **The algorithm calculates gradients, representing the partial derivatives of the cost function concerning each parameter.**

These gradients guide the updates, ensuring convergence towards the optimal parameter values that yield the lowest possible cost.

Gradient Descent is versatile and applicable to various machine learning models, including linear regression and neural networks. Its efficiency lies in navigating the parameter space efficiently, enabling models to learn patterns and make accurate predictions. Adjusting the learning rate is crucial to balance convergence speed and avoiding overshooting the optimal solution.

How the Gradient Descent Algorithm Works

For the sake of complexity, we can write our loss function for the single row as below

$$J(w,b) = \frac{1}{n}(y_p - y)^2$$

$$
\begin{aligned}
J'_w &= \frac{\partial J(w,b)}{\partial w} \\
&= \frac{\partial}{\partial w}\left[\frac{1}{n}(y_p - y)^2\right] \\
&= \frac{2(y_p - y)}{n}\frac{\partial}{\partial w}\left[(y_p - y)\right] \\
&= \frac{2(y_p - y)}{n}\frac{\partial}{\partial w}\left[((xW^T + b) - y)\right] \\
&= \frac{2(y_p - y)}{n}\left[\frac{\partial(xW^T + b)}{\partial w} - \frac{\partial(y)}{\partial w}\right] \\
&= \frac{2(y_p - y)}{n}\left[x - 0\right] \\
&= \frac{1}{n}(y_p - y)[2x]
\end{aligned}
$$

Gradient of J(w,b) with respect to b

$$
\begin{aligned}
J'_b &= \frac{\partial J(w,b)}{\partial b} \\
&= \frac{\partial}{\partial b}\left[\frac{1}{n}(y_p - y)^2\right] \\
&= \frac{2(y_p - y)}{n}\frac{\partial}{\partial b}\left[(y_p - y)\right] \\
&= \frac{2(y_p - y)}{n}\frac{\partial}{\partial b}\left[((xW^T + b) - y)\right] \\
&= \frac{2(y_p - y)}{n}\left[\frac{\partial(xW^T + b)}{\partial b} - \frac{\partial(y)}{\partial b}\right] \\
&= \frac{2(y_p - y)}{n}\left[1 - 0\right] \\
&= \frac{1}{n}(y_p - y)[2]
\end{aligned}
$$

Here we have considered the linear regression. So that here the parameters are weight and bias only. But in a fully connected neural network model there can be multiple layers and multiple parameters. but the concept will be the same everywhere. And the below-mentioned formula will work everywhere.

Here,

- **= Learning rate**
- **J = Loss function**
- **= Gradient symbol denotes the derivative of loss function J**
- **Param = weight and bias There can be multiple weight and bias values depending upon the complexity of the model and features in the dataset**

In our case:

$$
\begin{aligned}
w &= w - \gamma \nabla J(w, , b) \\
b &= b - \gamma \nabla J(w, , b)
\end{aligned}
$$

1. Initialization

1.1 Weight Initialization: Before training begins, the neural network's weights are initialized, typically using random values. Proper initialization can impact the convergence and performance of the training process.

Common Initialization Methods:

- **Xavier/Glorot Initialization: For sigmoid or tanh activations, to maintain variance of activations across layers.**
- **He Initialization: For ReLU activations, to account for the fact that ReLU neurons are not always active.**

2. Forward Pass

2.1 Input Data: Training data is fed into the network. Each input sample is processed through the layers of the network, applying weights and activation functions to produce an output.

2.2 Compute Loss: The output is compared to the true target using a loss function. The loss function measures how well the network's predictions match the actual targets.

3. Backward Pass (Backpropagation)

3.1 Compute Gradients: Backpropagation involves computing the gradients of the loss function with respect to each weight in the network. This is done using the chain rule of calculus to propagate the error backward from the output layer to the input layer.

Steps in Backpropagation:

- **Calculate Gradient of Loss w.r.t Output: Determine how the loss changes with respect to the network's output.**
- **Calculate Gradient w.r.t Weights: Compute the gradients of the loss function with respect to each weight by applying the chain rule through each layer.**

4. Weight Update

4.1 Update Rules: Weights are updated using optimization algorithms based on the computed gradients. The goal is to adjust weights in a way that reduces the loss.

Common Optimization Algorithms:

- **Stochastic Gradient Descent (SGD):**
- **Stochastic Gradient Descent (SGD):**
    - **Update Rule: w←w−η∂L∂ww \leftarrow w - \eta \frac{\partial L}{\partial w}w←w−η∂w∂L**
    - **Where: www is the weight, η\etaη is the learning rate, and ∂L∂w\frac{\partial L}{\partial w}∂w∂L is the gradient of the loss LLL with respect to the weight www.**
    - **Description: Updates weights based on the gradient of a single training sample or a mini-batch.**
- **Mini-Batch Gradient Descent:**

    - **Description: Uses small batches of training samples to compute gradients and update weights. This approach balances the efficiency of batch processing with the benefits of stochastic updates.**

- **Momentum:**
    - **Update Rule: v←βv+η∂L∂wv \leftarrow \beta v + \eta \frac{\partial L}{\partial w}v←βv+η∂w∂L**
    - **Weight Update: w←w−vw \leftarrow w - vw←w−v**
    - **Where: vvv is the velocity (accumulated gradient), and β\betaβ is the momentum coefficient.**
    - **Description: Helps accelerate SGD in the relevant direction and dampen oscillations.**
- **Adam (Adaptive Moment Estimation):**
    - **Update Rule: Combines ideas from Momentum and RMSprop.**
    - **Parameters: Uses moving averages of both gradients and squared gradients to adapt the learning rate for each weight.**
    - **Description: Effective in practice and often used for a wide range of problems.**
- **RMSprop (Root Mean Square Propagation):**
    - **Update Rule: Adjusts the learning rate based on a moving average of the squared gradients.**
    - **Description: Helps to mitigate issues with varying learning rates and improves convergence.**

5. Learning Rate Adjustment

5.1 Learning Rate Schedulers:

- **Constant: The learning rate remains fixed throughout training.**
- **Decay: The learning rate decreases over time, such as exponentially or stepwise.**
- **Adaptive: The learning rate is adjusted dynamically based on training progress.**

6. Epochs and Convergence

6.1 Epochs: The process of forward pass, backward pass, and weight update is repeated for several epochs, where each epoch involves processing the entire training dataset.

6.2 Convergence: The training continues until convergence criteria are met, such as a sufficiently small loss or a specified number of epochs.

7. Validation and Regularization

7.1 Validation Set Evaluation: During training, performance is evaluated on a validation set to monitor for overfitting and adjust hyperparameters accordingly.

7.2 Regularization Techniques:

- **Dropout: Randomly drops neurons during training to prevent overfitting.**
- **L2 Regularization (Weight Decay): Adds a penalty proportional to the square of the weights to discourage large weights.**

BATCH GRADIENT DESCENT

Concept of Batch Gradient Descent

1.1 Definition: Batch gradient descent updates the weights of a neural network using the gradient of the loss function computed over the entire training dataset. This means that before performing an update, the gradient is averaged over all training samples.

1.2 Process:

1. **Compute the Gradient: Calculate the gradient of the loss function with respect to each weight using the entire dataset.**
2. **Update the Weights: Adjust the weights in the direction that reduces the average loss for the whole dataset.**

2. Mathematical Formulation

2.1 Gradient Calculation: For a given weight www, the gradient of the loss function LLL is computed as: ∇wL=1N∑i=1N∇wLi\nabla_w L = \frac{1}{N} \sum_{i=1}^{N} \nabla_w L_i∇wL=N1∑i=1N∇wLi Where:

- **NNN is the total number of training samples.**
- **LiL_iLi is the loss for the iii-th training sample.**

2.2 Weight Update Rule: The weights are updated according to: w←w−η∇wLw \leftarrow w - \eta \nabla_w Lw←w−η∇wL Where:

- **η\etaη is the learning rate.**
- **∇wL\nabla_w L∇wL is the average gradient computed over all training samples.**

3. Advantages and Disadvantages

3.1 Advantages:

- **Stable Convergence: Since the gradient is averaged over the entire dataset, updates are more stable and the convergence path is smoother.**
- **Accurate Gradient Estimates: Using the full dataset for gradient computation provides a precise estimate of the gradient, leading to potentially better optimization.**

3.2 Disadvantages:

- **Computationally Expensive: Computing the gradient over the entire dataset can be computationally intensive and memory-consuming, especially with large datasets.**
- **Slow Updates: Since the update happens only after processing the entire dataset, it may result in slower convergence compared to other methods like mini-batch gradient descent.**

4. Comparison with Other Gradient Descent Variants

4.1 Stochastic Gradient Descent (SGD):

- **Gradient Computation: Uses a single sample to compute the gradient.**
- **Updates: More frequent updates, which can lead to faster convergence but with higher variance in the updates.**

4.2 Mini-Batch Gradient Descent:

- **Gradient Computation: Uses a small, random subset of the training data (mini-batch) to compute the gradient.**
- **Updates: Provides a compromise between the stability of batch gradient descent and the efficiency of SGD. It often leads to faster convergence and can be more memory efficient.**

What is Gradient?

A gradient is nothing but a derivative that defines the effects on outputs of the function with a little bit of variation in inputs.

What is Gradient Descent?

Gradient Descent stands as a cornerstone orchestrating the intricate dance of model optimization. At its core, it is a numerical optimization algorithm that aims to find the optimal parameters—weights and biases—of a neural network by minimizing a defined cost function.

Gradient Descent (GD) is a widely used optimization algorithm in machine learning and deep learning that minimises the cost function of a neural network model during training. It works by iteratively adjusting the weights or parameters of the model in the direction of the negative gradient of the cost function until the minimum of the cost function is reached.

The learning happens during the backpropagation while training the neural network-based model. There is a term known as Gradient Descent, which is used to optimize the weight and biases based on the cost function. The cost function evaluates the difference between the actual and predicted outputs.

Gradient Descent is a fundamental optimization algorithm in machine learning used to minimize the cost or loss function during model training.

- **It iteratively adjusts model parameters by moving in the direction of the steepest decrease in the cost function.**
- **The algorithm calculates gradients, representing the partial derivatives of the cost function concerning each parameter.**

These gradients guide the updates, ensuring convergence towards the optimal parameter values that yield the lowest possible cost.

Gradient Descent is versatile and applicable to various machine learning models, including linear regression and neural networks. Its efficiency lies in navigating the parameter space efficiently, enabling models to learn patterns and make accurate predictions. Adjusting the learning rate is crucial to balance convergence speed and avoiding overshooting the optimal solution.

Concept of Stochastic Gradient Descent

Stochastic Gradient Descent updates the weights of a neural network using the gradient computed from a single training sample (or a small subset of samples) at a time, rather than the entire dataset.

1.2 Process:

1. **Random Sample Selection: Select a single training sample (or a mini-batch) from the dataset.**
2. **Compute Gradient: Calculate the gradient of the loss function with respect to the weights using this single sample (or mini-batch).**
3. **Update Weights: Adjust the weights in the direction that reduces the loss based on the computed gradient.**

2. Mathematical Formulation

2.1 Gradient Calculation: For a weight www, the gradient of the loss function LLL using a single training sample xix_ixi and lab el yiy_iyi is: ∇wLi=∂Li∂w\nabla_w L_i = \frac{\partial L_i}{\partial w}∇wLi=∂w∂Li Where:

- **LiL_iLi is the loss for the iii-th training sample.**

2.2 Weight Update Rule: The weight update rule is: w←w−η∇wLiw \leftarrow w - \eta \nabla_w L_iw←w−η∇wLi Where:

- **η\etaη is the learning rate.**
- **∇wLi\nabla_w L_i∇wLi is the gradient of the loss for the selected sample.**

3. Advantages and Disadvantages

3.1 Advantages:

- **Faster Updates: Weights are updated more frequently since updates occur after each training sample or mini-batch, leading to faster feedback and potential acceleration in convergence.**
- **Escape Local Minima: The inherent noise in the gradient estimates can help the model escape local minima and find better solutions in complex loss landscapes.**
- **Memory Efficiency: It can be more memory-efficient than batch gradient descent because it doesn't require storing the entire dataset in memory for each update.**

3.2 Disadvantages:

- **Noisy Updates: The gradient estimates are noisy due to the use of single samples or small batches, which can lead to oscillations and less stable convergence paths.**
- **Convergence: It may converge more slowly to the exact minimum compared to batch gradient descent, especially if the learning rate is not properly tuned.**

4. Variants of SGD

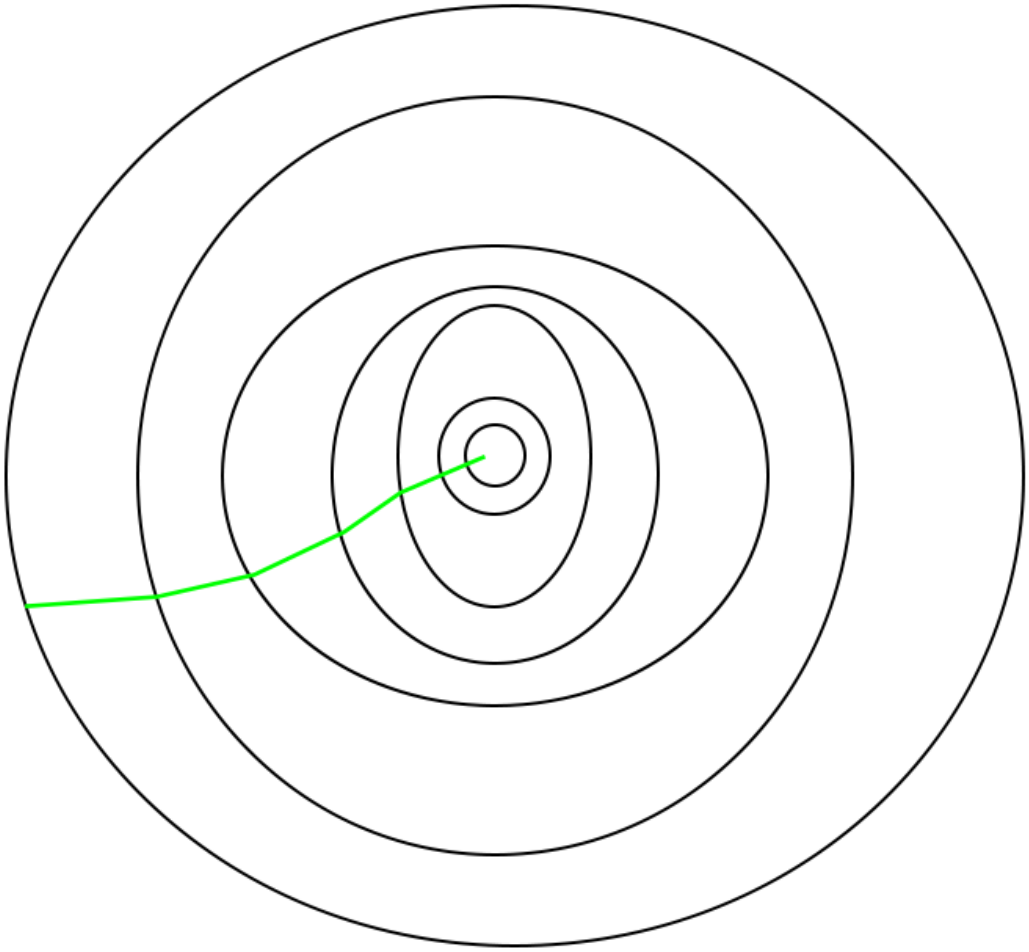
4.1 Mini-Batch Gradient Descent:

- **Definition: A compromise between batch and stochastic gradient descent, where the gradient is computed using a small, randomly chosen subset (mini-batch) of the training data.**
- **Advantages: Provides more stable updates compared to pure SGD while still benefiting from frequent updates and efficient computation.**

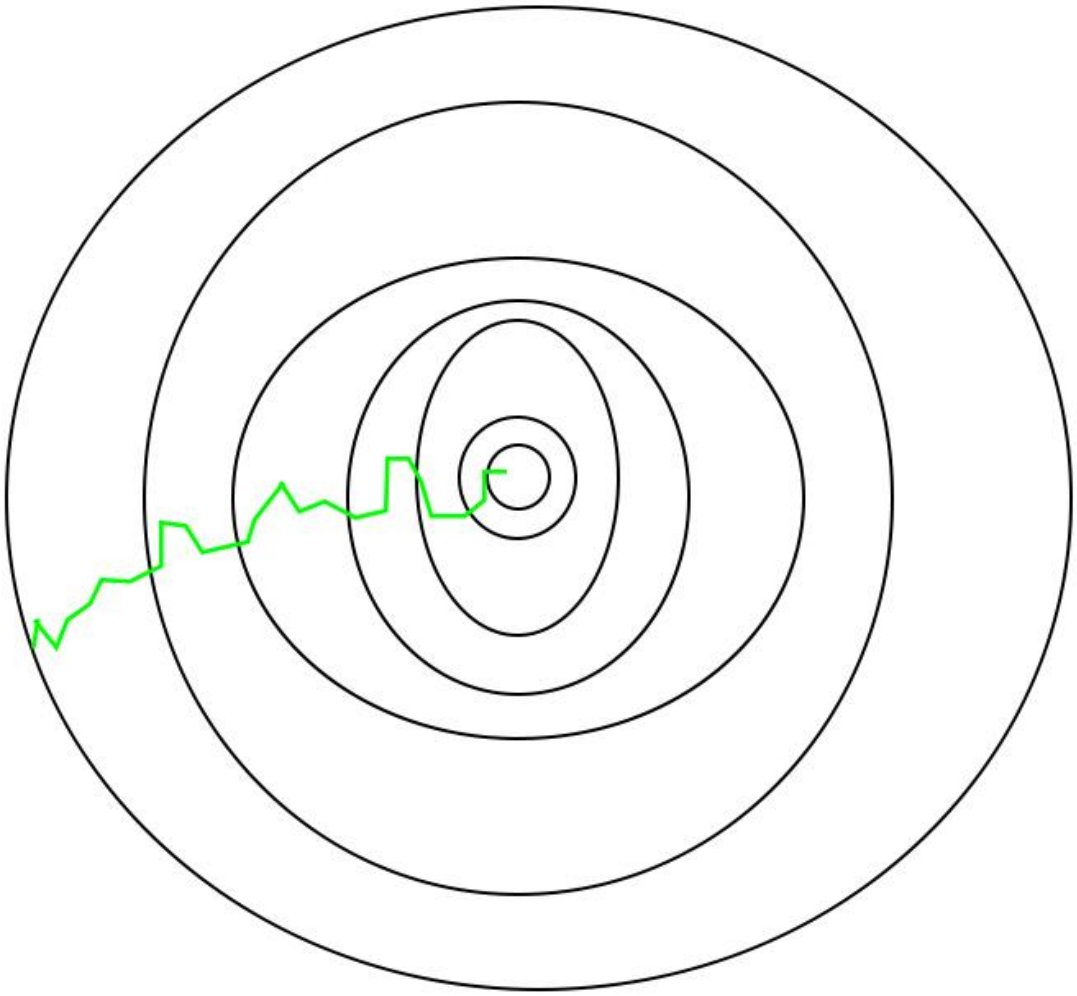
Momentum:

- **Definition: Adds a velocity term to the weight update to accumulate the past gradients and smooth out updates.**
- **Update Rule: v←β**
- **stochastic gradient descent algorithm**
- **Stochastic Gradient Descent (SGD):**
- **Stochastic Gradient Descent (SGD) is a variant of the Gradient Descent algorithm that is used for optimizing machine learning models. It addresses the computational inefficiency of traditional Gradient Descent methods when dealing with large datasets in machine learning projects.**
- **In SGD, instead of using the entire dataset for each iteration, only a single random training example (or a small batch) is selected to calculate the gradient and update the model parameters. This random selection introduces randomness into the optimization process, hence the term "stochastic" in stochastic Gradient Descent**
- **The advantage of using SGD is its computational efficiency, especially when dealing with large datasets. By using a single example or a small batch, the computational cost per iteration is significantly reduced compared to traditional Gradient Descent methods that require processing the entire dataset.**
- **Stochastic Gradient Descent Algorithm**
- **Initialization: Randomly initialize the parameters of the model.**
- **Set Parameters: Determine the number of iterations and the learning rate (alpha) for updating the parameters.**
- **Stochastic Gradient Descent Loop: Repeat the following steps until the model converges or reaches the maximum number of iterations:**
- **Shuffle the training dataset to introduce randomness.**
- **Iterate over each training example (or a small batch) in the shuffled order.**
- **Compute the gradient of the cost function with respect to the model parameters using the current training**
  example (or batch).
- **Update the model parameters by taking a step in the direction of the negative gradient, scaled by the learning rate.**
- **Evaluate the convergence criteria, such as the difference in the cost function between iterations of the gradient.**

- **Return Optimized Parameters: Once the convergence criteria are met or the maximum number of iterations is reached, return the optimized model parameters.**
- **In SGD, since only one sample from the dataset is chosen at random for each iteration, the path taken by the algorithm to reach the minima is usually noisier than your typical Gradient Descent algorithm. But that doesn't matter all that much because the path taken by the algorithm does not matter, as long as we reach the minimum and with a significantly shorter training time.**
- **The path taken by Batch Gradient Descent is shown below:**



- ***Batch gradient optimization path***
- **A path taken by Stochastic Gradient Descent looks as follows –**



-
- ***stochastic gradient optimization path***
- **One thing to be noted is that, as SGD is generally noisier than typical Gradient Descent, it usually took a higher number of iterations to reach the minima, because of the randomness in its descent. Even though it requires a higher number of iterations to reach the minima than typical Gradient Descent, it is still computationally much less expensive than typical Gradient Descent. Hence, in most scenarios, SGD is preferred over Batch Gradient Descent for optimizing a learning algorithm.**
- **Difference between Stochastic Gradient Descent & batch Gradient Descent**
- **The comparison between Stochastic Gradient Descent (SGD) and Batch Gradient Descent are as follows:**

• Aspect	• Stochastic Gradient Descent (SGD)	• Batch Gradient Descent
• Dataset Usage	• Uses a single random sample or a small batch of samples at each iteration.	• Uses the entire dataset (batch) at each iteration.
• Computational Efficiency	• Computationally less expensive per iteration, as it processes fewer data points.	• Computationally more expensive per iteration, as it processes the entire dataset.
• Convergence	• Faster convergence due to frequent updates.	• Slower convergence due to less frequent updates.
• Noise in Updates	• High noise due to frequent updates with a single or few samples.	• Low noise as it updates parameters using all data points.
• Stability	• Less stable as it may oscillate around the optimal solution.	• More stable as it converges smoothly towards the optimum.
• Memory Requirement	• Requires less memory as it processes fewer data points at a time.	• Requires more memory to hold the entire dataset in memory.
• Update Frequency	• Frequent updates make it suitable for online learning and large datasets.	• Less frequent updates make it suitable for smaller datasets.
• Initialization Sensitivity	• Less sensitive to initial parameter values due to frequent updates.	• More sensitive to initial parameter values.

Hebbian Learning

Basic Principle: Hebbian learning is often summarized by the phrase "cells that fire together, wire together." The idea is that the connection between two neurons is strengthened if they are activated simultaneously. This principle reflects a form of associative learning where the co-activation of neurons leads to an increase in synaptic strength.

1.2 Mathematical Formulation: The Hebbian learning rule can be expressed as: Δwij=η·xi·yj\Delta w_{ij} = \eta \cdot x_i \cdot y_jΔwij=η·xi·yj Where:

- **Δwij\Delta w_{ij}Δwij is the change in the weight connecting neuron iii to neuron jjj.**
- **η\etaη is the learning rate.**
- **xix_ixi is the activation of neuron iii.**
- **yjy_jyj is the activation of neuron jjj.**

2. Hebbian Learning in Neural Networks

2.1 Synaptic Weight Update: In the context of neural networks, Hebbian learning involves updating the weights of connections based on the product of the activations of the connected neurons. If two neurons are active at the same time, the connection between them is strengthened.

2.2 Application: Hebbian learning is typically used in unsupervised learning scenarios where there is no explicit target output. It is often applied in:

- **Associative Memory Networks: Where the network learns to associate patterns with one another.**
- **Self-Organizing Maps (SOMs): Where neurons organize themselves to represent input patterns effectively.**

3. Hebbian Learning Variants

3.1 Delta Rule (Oja's Rule): Oja's rule is a modification of the Hebbian learning rule that incorporates a normalization term to prevent the weights from growing without bound: Δwij=η·xi·(yj−yj2·wij)\Delta w_{ij} = \eta \cdot x_i \cdot (y_j - y_j^2 \cdot w_{ij})Δwij=η·xi·(yj−yj2 ·wij) Where yjy_jyj is the output of neuron jjj, and wijw_{ij}wij is the weight. This rule maintains stability by normalizing the weight updates.

3.2 Spike-Timing-Dependent Plasticity (STDP): STDP is a more biologically realistic variant of Hebbian learning that considers the timing of spikes. It updates the weights based on the precise timing of the pre-synaptic and post-synaptic spikes:

- **Positive Timing Difference: If a presynaptic spike occurs shortly before a postsynaptic spike, the weight is increased.**
- **Negative Timing Difference: If the postsynaptic spike occurs before the presynaptic spike, the weight is decreased.**

4. Advantages and Limitations

4.1 Advantages:

- **Biological Plausibility: Reflects a simple and biologically plausible mechanism for learning in neural networks.**

- **Pattern Learning: Useful for unsupervised learning tasks, pattern recognition, and associative memory.**

4.2 Limitations:

- **Unbounded Growth: Without modifications like Oja's rule, weights can grow indefinitely, leading to instability.**
- **Lack of Supervision: The Hebbian rule does not use target values, which can limit its application in supervised learning tasks.**

5. Implementation Steps

5.1 Initialization:

- **Initialize the weights of the connections between neurons.**

5.2 Learning Loop:

- **Activation: For each training example, compute the activation of neurons in the network.**
- **Weight Update: Apply the Hebbian learning rule to update the weights based on the activations of the neurons.**

5.3 Normalize Weights (if needed):

- **Apply a normalization technique (like Oja's rule) to ensure weights remain bounded.**

5.4 Iterate:

- **Repeat the learning process over multiple examples and epochs to reinforce learned associations.**

6. Applications

6.1 Associative Memory:

- **Used in networks like Hopfield networks where the goal is to store and recall patterns based on associative learning.**

6.2 Feature Learning:

- **In self-organizing maps and certain unsupervised learning models, Hebbian learning helps in discovering and organizing features in the input data.**

- **Hebbian Learning Rule with Implementation of AND Gate**

Hebbian Learning Rule, also known as Hebb Learning Rule, was proposed by Donald O Hebb. It is one of the first and also easiest learning rules in the neural network. It is used for pattern classification. It is a single layer neural network, i.e. it has one input layer and one output layer. The input layer can have many units, say n. The output layer only has one unit. Hebbian

rule works by updating the weights between neurons in the neural network for each training sample.

Hebbian Learning Rule Algorithm :

1. **Set all weights to zero, $w_i = 0$ for i=1 to n, and bias to zero.**
2. **For each input vector, S(input vector) : t(target output pair), repeat steps 3-5.**
3. **Set activations for input units with the input vector $X_i = S_i$ for i = 1 to n.**
4. **Set the corresponding output value to the output neuron, i.e. y = t.**
5. **Update weight and bias by applying Hebb rule for all i = 1 to n:**

$$w_i(new) = w_i(old) + x_i y$$

$$b(new) = b(old) + y$$

Implementing AND Gate :

INPUT				TARGET	
	x_1	x_2	b		y
X_1	-1	-1	1	Y_1	-1
X_2	-1	1	1	Y_2	-1
X_3	1	-1	1	Y_3	-1
X_4	1	1	1	Y_4	1

There are 4 training samples, so there will be 4 iterations. Also, the activation function used here is Bipolar Sigmoidal Function so the range is [-1,1].

Step 1 :

Set weight and bias to zero, $w = [\ 0\ 0\ 0\ ]^T$ and b = 0.

Step 2 :

Set input vector $X_i = S_i$ for i = 1 to 4.

$X_1 = [\ -1\ -1\ 1\ ]^T$

$X_2 = [\ -1\ 1\ 1\ ]^T$

$X_3 = [\ 1\ -1\ 1\ ]^T$

$X_4 = [\ 1\ 1\ 1\ ]^T$

Step 3 :

Output value is set to y = t.

Step 4 :

Modifying weights using Hebbian Rule:

First iteration –

$w(new) = w(old) + x_1y_1 = [\ 0\ 0\ 0\ ]^T + [\ -1\ -1\ 1\ ]^T . [\ -1\ ] = [\ 1\ 1\ -1\ ]^T$

For the second iteration, the final weight of the first one will be used and so on.

Second iteration –

$w(new) = [\ 1\ 1\ -1\ ]^T + [\ -1\ 1\ 1\ ]^T . [\ -1\ ] = [\ 2\ 0\ -2\ ]^T$

Third iteration –

$w(new) = [\ 2\ 0\ -2]^T + [\ 1\ -1\ 1\ ]^T . [\ -1\ ] = [\ 1\ 1\ -3\ ]^T$

Fourth iteration –

$w(new) = [\ 1\ 1\ -3]^T + [\ 1\ 1\ 1\ ]^T . [\ 1\ ] = [\ 2\ 2\ -2\ ]^T$

So, the final weight matrix is $[\ 2\ 2\ -2\ ]^T$

DELTA LEARNING RULE

The Delta Learning Rule, also known as the Widrow-Hoff rule, is a fundamental algorithm used for training neural networks, particularly in the context of single-layer perceptrons and linear models. It is a specific case of supervised learning, often used to minimize the error between predicted and actual outputs. Here's a detailed look at the Delta Learning Rule:

1. Concept of the Delta Learning Rule

1.1 Definition: The Delta Learning Rule adjusts the weights of a neural network to minimize the difference between the predicted output and the actual target value. It is based on gradient descent, aiming to reduce the error by making small adjustments to the weights.

1.2 Mathematical Formulation: For a single neuron, the weight update is given by: Δwi=η·δ·xi\Delta w_i = \eta \cdot \delta \cdot x_iΔwi=η·δ·xi Where:

- **Δwi\Delta w_iΔwi is the change in weight wiw_iwi.**
- **η\etaη is the learning rate, a small positive constant that controls the size of the weight updates.**
- **δ\deltaδ is the error term (or delta) for the neuron.**
- **xix_ixi is the input value associated with weight wiw_iwi.**

2. Error Term (Delta)

2.1 Definition: The error term, δ\deltaδ, represents how much the output of the neuron deviates from the desired output. For a neuron with a linear activation function, δ\deltaδ is computed as: δ=t−y\delta = t - yδ=t−y Where:

- **ttt is the target output (desired value).**
- **yyy is the actual output produced by the neuron.**

2.2 For Non-Linear Activation Functions: If the neuron uses a non-linear activation function (e.g., sigmoid or tanh), the error term is modified to account for the derivative of the activation function: δ=(t−y)⋅f′(z)\delta = (t - y) \cdot f'(z)δ=(t−y)⋅f′(z) Where:

- **f′(z)f'(z)f′(z) is the derivative of the activation function with respect to the net input zzz.**

3. Delta Learning Rule in Practice

3.1 Steps for Weight Update:

1. **Forward Pass: Calculate the output of the neuron by applying the current weights to the inputs.**
2. **Compute Error: Determine the error between the predicted output and the target output.**
3. **Calculate Delta: Compute the delta term based on the error and the activation function.**
4. **Update Weights: Adjust the weights using the delta learning rule.**

3.2 Example Calculation: Assume a single-layer perceptron with weights w1w_1w1 and w2w_2w2, inputs x1x_1x1 and x2x_2x2, and a learning rate η\etaη:

1. **Compute the output y=w1⋅x1+w2⋅x2y = w_1 \cdot x_1 + w_2 \cdot x_2y=w1⋅x1+w2⋅x2.**
2. **Calculate the error δ=t−y\delta = t - yδ=t−y.**
3. **Update weights:**
    - **Δw1=η⋅δ⋅x1\Delta w_1 = \eta \cdot \delta \cdot x_1Δw1=η⋅δ⋅x1**
    - **Δw2=η⋅δ⋅x2\Delta w_2 = \eta \cdot \delta \cdot x_2Δw2=η⋅δ⋅x2**

4. Advantages and Disadvantages

4.1 Advantages:

- **Simplicity: The Delta Learning Rule is straightforward to implement and understand.**
- **Convergence: It works well for linear problems and simple neural network architectures.**

4.2 Disadvantages:

- **Linear Limitation: The rule is primarily suitable for linear models and single-layer networks. It struggles with more complex, non-linear problems.**
- **Learning Rate Sensitivity: The choice of learning rate η\etaη is crucial. If too high, the training may become unstable; if too low, convergence may be very slow.**

5. Extension to Multi-Layer Networks

For multi-layer networks, the Delta Learning Rule is extended through the backpropagation algorithm. Backpropagation applies the Delta Rule in a layer-wise fashion, propagating errors backward through the network and updating weights accordingly.

5.1 Backpropagation Steps:

1. **Forward Pass: Compute outputs for all layers.**
2. **Backward Pass: Calculate error terms for each neuron in the output layer and propagate these errors backward to compute gradients for hidden layers.**
3. **Weight Update: Apply the Delta Rule to adjust weights in all layers based on computed gradients.**

6. Implementation Steps

6.1 Initialize Weights:

- **Start with small random values for weights.**

6.2 Training Loop:

- **For each training example:**
    1. **Compute Output: Perform a forward pass to get the output.**
    2. **Calculate Error: Compute the difference between predicted and target outputs.**
    3. **Compute Delta: Calculate the delta for each neuron.**
    4. **Update Weights: Adjust weights according to the Delta Rule.**

CONVERGENCE AND LOCAL MINIMA

the concepts of convergence and local minima are critical to understanding how effectively a model learns and generalizes. Here's an in-depth look at these concepts:

---

1. Convergence

1.1 Definition: Convergence in the context of neural networks refers to the process by which the optimization algorithm finds a set of weights that minimizes the loss function, indicating that the training process is complete or has reached a satisfactory solution.

1.2 Indicators of Convergence:

- **Loss Function: The training and validation loss should stabilize or decrease consistently over epochs.**
- **Accuracy: Training and validation accuracy should reach a plateau or continue improving within an acceptable margin.**
- **Gradient Magnitude: Gradients of the loss function with respect to the weights should become small, indicating that changes in weights have minimal impact on the loss.**

1.3 Factors Affecting Convergence:

- **Learning Rate: A learning rate that is too high can cause overshooting and instability, while a learning rate that is too low can lead to very slow convergence.**
- **Optimization Algorithm: Different algorithms (SGD, Adam, RMSprop) have varying effects on the speed and reliability of convergence.**
- **Initialization: Proper weight initialization can impact convergence speed and stability.**
- **Batch Size: Larger batch sizes can provide more stable gradients, while smaller batch sizes may offer faster updates but with more variance.**

1.4 Convergence Criteria:

- **Fixed Number of Epochs: Training stops after a predefined number of epochs.**
- **Early Stopping: Training halts if the validation performance stops improving for a certain number of epochs.**
- **Gradient Norm: Training can be stopped if the norm of the gradients falls below a threshold, indicating minimal change.**

---

2. Local Minima

2.1 Definition: Local minima are points in the loss landscape where the loss function has a value lower than its immediate surroundings but is not the lowest value overall (i.e., not a global minimum). In a complex loss surface with many parameters, local minima can present challenges during optimization.

2.2 Types of Minima:

- **Local Minimum: A point where the loss function value is lower than at neighboring points but not necessarily the lowest possible value.**
- **Global Minimum: The absolute lowest point in the loss landscape, representing the optimal solution.**
- **Saddle Point: A point where the gradient is zero but is not a local minimum. The loss function curves upwards in some directions and downwards in others.**

2.3 Impact on Training:

- **Stuck in Local Minima: The optimization process may converge to a local minimum that is not the best possible solution, leading to suboptimal performance.**
- **Overfitting: If the network finds a local minimum that fits the training data too well, it may not generalize well to new, unseen data.**

2.4 Strategies to Address Local Minima:

- **Learning Rate Adjustments: Using learning rate schedules or adaptive learning rates can help escape local minima by making larger steps in the parameter space.**
- **Momentum: Incorporating momentum helps to smooth out the gradient updates and can help escape local minima by adding a velocity term.**

- **Random Restarts: Running the optimization multiple times with different initializations can help in finding better minima.**
- **Stochastic Gradient Descent (SGD): The inherent noise in SGD can help in escaping local minima by introducing variability in the gradient estimates.**
- **Regularization: Techniques such as dropout or weight decay can prevent overfitting to local minima and improve generalization.**

2.5 Advanced Techniques:

- **Simulated Annealing: A probabilistic technique that gradually decreases the likelihood of accepting worse solutions as the algorithm progresses.**
- **Genetic Algorithms: Evolutionary algorithms that use selection, crossover, and mutation to explore the parameter space.**

---

3. Visualization of Loss Landscapes

3.1 Loss Surface Visualization: Understanding the loss surface helps in visualizing how local minima and convergence behaviors affect training. Techniques include:

- **2D/3D Plots: For low-dimensional cases, plotting the loss surface can provide insight into the location of minima.**
- **Contour Maps: Useful for understanding how the loss function changes with respect to different parameters.**

3.2 Challenges:

- **High Dimensionality: Neural networks often have a high-dimensional parameter space, making it difficult to visualize and analyze the loss landscape fully.**

REPRESENTATION POWER OF FEED FORWARD NETWORK

The representation power of feedforward neural networks refers to their ability to model and approximate complex functions or patterns from data. This concept is crucial for understanding the capability and limitations of feedforward neural networks in various applications, such as classification, regression, and more complex tasks.

Here's a detailed overview of the representation power of feedforward neural networks:

1. Fundamental Concepts

1.1 Definition: Feedforward neural networks (FNNs) are a class of artificial neural networks where connections between nodes do not form cycles. Information moves in one direction—from input to output—without looping back.

1.2 Components:

- **Input Layer: Receives the input features.**
- **Hidden Layers: Intermediate layers that perform transformations and feature extraction.**

- **Output Layer: Produces the final prediction or output.**

1.3 Activation Functions: Activation functions in hidden layers introduce non-linearity into the network, which is essential for modeling complex patterns. Common activation functions include:

- **Sigmoid: σ(x)=11+e−x\sigma(x) = \frac{1}{1 + e^{-x}}σ(x)=1+e−x1**
- **Tanh: tanh⁡(x)=ex−e−xex+e−x\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}tanh(x)=ex+e−xex−e−x**
- **ReLU (Rectified Linear Unit): ReLU(x)=max⁡(0,x)\text{ReLU}(x) = \max(0, x)ReLU(x)=max(0,x)**

2. Representation Power

2.1 Universal Approximation Theorem: The Universal Approximation Theorem states that a feedforward neural network with a single hidden layer, given a sufficient number of neurons and appropriate activation functions, can approximate any continuous function on a compact subset of Rn\mathbb{R}^nRn to an arbitrary degree of accuracy.

- **Key Implication: This theorem indicates that FNNs have the theoretical capability to approximate a wide range of functions, making them highly versatile.**

2.2 Practical Considerations:

- **Hidden Layer Size: While the theorem guarantees approximation, the number of neurons required in practice can be very large, depending on the complexity of the function and the network architecture.**
- **Training and Generalization: The ability to represent functions does not guarantee good performance on unseen data. Training techniques, regularization, and validation are essential to ensure that the network generalizes well and does not overfit.**

3. Expressive Power

3.1 Linear Separability:

- **Single-Layer Perceptrons: Can only represent linearly separable functions. They are limited to problems where a linear boundary suffices.**
- **Multi-Layer Perceptrons (MLPs): With one or more hidden layers, FNNs can represent non-linear decision boundaries, which allows them to solve more complex problems.**

3.2 Approximation of Complex Functions:

- **Polynomials and Trigonometric Functions: FNNs can approximate polynomial functions, trigonometric functions, and other complex patterns through non-linear activation functions and multiple layers.**
- **Piecewise Functions: FNNs can approximate piecewise continuous functions, which are useful for tasks like image recognition and speech processing.**

4. Network Depth and Width

4.1 Depth (Number of Layers):

- **Shallow Networks: Networks with only one or two hidden layers can approximate many functions but may require an impractically large number of neurons to do so effectively.**
- **Deep Networks: Deep networks with multiple hidden layers (Deep Neural Networks) can represent more complex functions and learn hierarchical features, making them suitable for tasks like image and speech recognition.**

4.2 Width (Number of Neurons per Layer):

- **Broader Layers: Increasing the number of neurons in each hidden layer improves the network's ability to capture finer details and complexities in the data.**
- **Trade-offs: While wider layers can improve expressiveness, they also increase computational requirements and the risk of overfitting.**

HYPOTHESIS SEARCH SPACE AND INDUCTIVE BIAS

Hypothesis Search Space

1. **Definition: The hypothesis search space refers to the set of all possible models or hypotheses that a learning algorithm can consider when trying to fit data. It represents the range of functions or patterns that the algorithm can use to make predictions.**
2. **Importance: The size and nature of this space have a significant impact on the learning process. A larger hypothesis space might allow for more complex and accurate models but can also make learning more computationally expensive and prone to overfitting. Conversely, a smaller hypothesis space might be computationally more manageable but may not capture the underlying patterns in the data well.**
3. **Example: In a linear regression problem, the hypothesis space is the set of all possible linear functions that can be used to predict the target variable. For more complex models like neural networks, the hypothesis space includes various network architectures, activation functions, and weight configurations.**

Inductive Bias

1. **Definition: Inductive bias refers to the set of assumptions a learning algorithm makes to generalize from the training data to unseen data. It influences how the algorithm makes predictions on new data and helps constrain the hypothesis search space to more plausible solutions.**
2. **Importance: Inductive bias is essential because it guides the learning process by narrowing down the hypothesis space to those models that are more likely to generalize well. Without some form of inductive bias, learning algorithms would struggle to make predictions beyond the training data.**
3. **Example: In the case of linear regression, the inductive bias is the assumption that the relationship between input features and output is linear. For decision trees, the bias is that the data can be partitioned into regions with similar responses. Neural networks might have biases related to the assumption that complex patterns can be captured through layers of non-linear transformations.**

Relationship Between Hypothesis Search Space and Inductive Bias

- **Guiding Learning: Inductive bias helps to efficiently explore the hypothesis search space by incorporating domain knowledge or assumptions about the nature of the data. For instance, if you know your data has a linear relationship, a linear model is a good starting point.**
- **Balancing Act: There's often a trade-off between the complexity of the hypothesis search space and the strength of the inductive bias. A very strong bias can lead to underfitting, where the model is too simple to capture the underlying patterns. Conversely, a very weak bias can lead to overfitting, where the model captures noise in the training data as if it were a true signal.**
- **Adaptation: Different learning algorithms and models come with their own types of inductive biases, which affect how they navigate their hypothesis search space. For example, regularization techniques in machine learning impose biases that help in managing the trade-off between bias and variance.**

GENERALIZATION

1. **Definition: Generalization refers to the ability of a neural network to make accurate predictions on data that it has not encountered during the training phase. A well-generalized model effectively captures the underlying patterns in the data and is not overly fitted to the training examples.**
2. **Importance: Good generalization ensures that the model's performance on real-world, unseen data is as good as its performance on the training data. Poor generalization indicates that the model might be overfitting or underfitting the training data.**

Factors Influencing Generalization

1. **Model Complexity:**
    - **Capacity: Neural networks with too many layers or parameters can model very complex patterns, but they might also memorize the training data, leading to overfitting. On the other hand, very simple models might not capture the necessary patterns, leading to underfitting.**
    - **Architecture: The choice of network architecture (e.g., number of layers, types of layers) affects how well the model can generalize. More complex architectures can capture more intricate patterns but may require careful tuning and regularization.**
2. **Training Data:**
    - **Quantity: Larger datasets generally help in better generalization because they provide more examples of the underlying data distribution, reducing the risk of overfitting.**
    - **Quality: High-quality data with fewer errors or noise leads to better generalization. If the data is representative of the real-world scenarios where the model will be applied, generalization improves.**
3. **Regularization Techniques:**

- **Dropout: This technique randomly drops units (and their connections) from the neural network during training, which helps prevent overfitting by making the network more robust.**
- **Weight Decay (L2 Regularization): Adds a penalty to the loss function based on the size of the weights, discouraging overly complex models.**
- **Early Stopping: Stops training when performance on a validation set starts to deteriorate, preventing the model from overfitting the training data.**

4. **Training Procedures:**
   - **Cross-Validation: Using techniques like k-fold cross-validation helps in understanding how well the model generalizes to unseen data by training and validating it on different subsets of the data.**
   - **Hyperparameter Tuning: Proper tuning of hyperparameters (like learning rate, batch size, etc.) helps in achieving a balance between fitting the training data and maintaining generalization.**
5. **Data Augmentation:**
   - **Techniques: Techniques such as rotation, scaling, and cropping for images, or adding noise for other types of data, increase the diversity of training examples, which helps the model generalize better.**
6. **Ensemble Methods:**
   - **Bagging and Boosting: Combining predictions from multiple models can improve generalization by reducing variance and bias. Techniques like Random Forests and Gradient Boosting Machines are common examples.**

Measuring Generalization

1. **Validation and Test Sets: Evaluating the model's performance on a separate validation set and test set helps gauge its generalization capability. The validation set is used during training to tune hyperparameters, while the test set provides an unbiased estimate of generalization.**
2. **Learning Curves: Plotting training and validation loss or accuracy over epochs helps visualize whether the model is overfitting or underfitting. A large gap between training and validation performance often indicates overfitting.**
3. **Performance Metrics: Metrics like accuracy, precision, recall, F1 score, and area under the ROC curve are used to evaluate generalization, depending on the task (classification, regression, etc.).**

OVERFITTING AND STOPPING CRITERION -ERROR FUNCTIONS

Indicators of Overfitting:

- **Performance Gap: A significant discrepancy between training and validation performance (e.g., accuracy or loss) suggests overfitting. Training performance may be very high while validation performance remains low.**

- **High Variance: The model's performance fluctuates significantly when exposed to different subsets of the data.**

Causes of Overfitting:

- **Model Complexity: Using a very complex model with too many parameters relative to the amount of training data.**
- **Insufficient Data: Training on a limited dataset may not provide a representative sample of the underlying data distribution.**
- **Noise in Data: Including noisy or irrelevant features in the training data.**

Mitigation Strategies:

1. **Regularization:**
    - **L1 and L2 Regularization: Penalize large weights to prevent the model from fitting noise in the data.**
    - **Dropout: Randomly drop neurons during training to prevent the network from becoming too reliant on any specific neuron.**
2. **Early Stopping: Monitor validation performance and stop training when it starts to degrade, indicating that the model is beginning to overfit the training data.**
3. **Cross-Validation: Use techniques like k-fold cross-validation to evaluate the model's performance on multiple subsets of the data, providing a better estimate of its generalization ability.**
4. **Data Augmentation: Increase the diversity of the training data by applying transformations such as rotations or scaling (for images) to help the model generalize better.**
5. **Simplifying the Model: Reduce the complexity of the neural network by decreasing the number of layers or parameters.**

Stopping Criteria

Definition: Stopping criteria are conditions or rules used to determine when to halt the training process of a neural network. Proper stopping is essential to avoid overfitting and ensure that the model converges to a good solution.

Common Stopping Criteria:

1. **Early Stopping:**
    - **Validation Loss: Training is stopped when the validation loss stops improving for a predefined number of epochs (patience). This prevents overfitting by halting training before the model starts to memorize the training data.**
    - **Performance Metrics: Monitoring metrics such as validation accuracy and stopping when there is no significant improvement.**
2. **Fixed Number of Epochs:**
    - **Training is terminated after a predefined number of epochs. This approach is less adaptive and might require tuning to balance underfitting and overfitting.**

3. **Convergence Criteria:**
    - **Gradient Magnitude: Training stops when the gradient magnitude (change in weights) falls below a certain threshold, indicating that the model has converged to a solution.**
    - **Change in Loss: Training stops when changes in the loss function between epochs fall below a certain threshold.**

Error Functions

Error functions (or loss functions) are used to quantify how well the neural network is performing. They measure the difference between the predicted values and the true values, guiding the training process.

Common Error Functions:

1. **Mean Squared Error (MSE):**
    - **Usage: Commonly used for regression problems.**
    - **Definition: Measures the average squared difference between predicted and actual values.**
    - **Formula: MSE=1N∑i=1N(yi−y^i)2\text{MSE} = \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2MSE=N1∑i=1N(yi−y^i)2**
    - **Interpretation: Lower MSE indicates better performance.**
2. **Cross-Entropy Loss:**
    - **Usage: Commonly used for classification problems.**
    - **Definition: Measures the difference between two probability distributions – the true labels and the predicted probabilities.**
    - **Formula: For binary classification: Binary Cross-Entropy=−1N∑i=1N[yilog⁡(y^i)+(1−yi)log⁡(1−y^i)]\text{Binary Cross-Entropy} = -\frac{1}{N} \sum_{i=1}^{N} [y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)]Binary Cross-Entropy=−N1∑i=1N[yilog(y^i)+(1−yi)log(1−y^i)] For multi-class classification: Categorical Cross-Entropy=−∑i=1N∑j=1Cyijlog⁡(y^ij)\text{Categorical Cross-Entropy} = -\sum_{i=1}^{N} \sum_{j=1}^{C} y_{ij} \log(\hat{y}_{ij})Categorical Cross-Entropy=−∑i=1N∑j=1Cyijlog(y^ij)**
    - **Interpretation: Lower cross-entropy indicates better performance in classifying instances correctly.**
3. **Hinge Loss:**
    - **Usage: Commonly used for binary classification with Support Vector Machines (SVMs).**
    - **Definition: Measures the error between the predicted and true labels, particularly in cases where margins are important.**

- Interpretation: Lower hinge loss indicates better performance.
- ERROR MINIMIZING PROCEDURES-HEBBIAN LEARNING

Error Minimizing Procedures

Error minimizing procedures are methods used to adjust the parameters of a model to reduce the difference between the predicted outputs and the true outputs (i.e., to minimize the error or loss function). These procedures are crucial for training neural networks effectively.

1. Gradient Descent

Definition: Gradient descent is an optimization algorithm used to minimize the error function by iteratively adjusting model parameters in the direction of the steepest decrease in error.

- **Basic Idea: Calculate the gradient (partial derivatives) of the error function with respect to each parameter and update the parameters in the opposite direction of the gradient.**
- **Formula: For a parameter θ\thetaθ, the update rule is: θ←θ−η·∂L∂θ\theta \leftarrow \theta - \eta \cdot \frac{\partial L}{\partial \theta}θ←θ−η·∂θ∂L where η\etaη is the learning rate, and LLL is the loss function.**

Variants:

- **Batch Gradient Descent: Uses the entire training dataset to compute gradients.**
- **Stochastic Gradient Descent (SGD): Uses a single training example to compute the gradient and update parameters.**
- **Mini-Batch Gradient Descent: Uses a small subset (mini-batch) of the training data to compute gradients and update parameters.**

Advantages:

- **Can be adapted to various types of loss functions and models.**
- **Widely used and well-understood.**

Disadvantages:

- **Convergence can be slow, especially for large datasets.**
- **Requires tuning of the learning rate.**

2. Momentum

Definition: Momentum is an extension of gradient descent that helps accelerate convergence and smooth out updates by taking into account the previous gradients.

- **Basic Idea: Accumulate a moving average of past gradients and use this to update parameters.**
- **Formula: vt=βvt−1+(1−β)·∂L∂θv_t = \beta v_{t-1} + (1 - \beta) \cdot \frac{\partial L}{\partial \theta}vt=βvt−1+(1−β)·∂θ∂L θ←θ−η·vt\theta \leftarrow \theta - \eta \cdot v_tθ←θ−η·vt where vtv_tvt is the velocity term, and β\betaβ is the momentum coefficient.**

Advantages:

- **Reduces oscillations and speeds up convergence.**
- **Helps in escaping local minima.**

Disadvantages:

- **Requires tuning of the momentum coefficient.**

3. Adam (Adaptive Moment Estimation)

Definition: Adam combines ideas from momentum and RMSprop to adaptively adjust the learning rates of parameters based on their past gradients.

- **Basic Idea: Use estimates of first-order (mean) and second-order (uncentered variance) moments of the gradients to adjust learning rates.**
- **Formula: mt=β1mt−1+(1−β1)·∂L∂θm_t = \beta_1 m_{t-1} + (1 - \beta_1) \cdot \frac{\partial L}{\partial \theta}mt=β1mt−1+(1−β1)·∂θ∂L vt=β2vt−1+(1−β2)·(∂L∂θ)2v_t = \beta_2 v_{t-1} + (1 - \beta_2) \cdot \left(\frac{\partial L}{\partial \theta}\right)^2vt=β2 vt−1+(1−β2)·(∂θ∂L)2 m^t=mt1−β1t\hat{m}_t = \frac{m_t}{1 - \beta_1^t}m^t=1−β1tmt v^t=vt1−β2t\hat{v}_t = \frac{v_t}{1 - \beta_2^t}v^t=1−β2tvt θ←θ−η·m^tv^t+ϵ\theta \leftarrow \theta - \eta \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}θ←θ−η·v^t +ϵm^t where mtm_tmt and vtv_tvt are estimates of the first and second moments, β1\beta_1β1 and β2\beta_2β2 are hyperparameters, and ϵ\epsilonϵ is a small constant to prevent division by zero.**

Advantages:

- **Adaptively adjusts learning rates for each parameter.**
- **Generally works well in practice with minimal tuning.**

Disadvantages:

- **Computationally more expensive than basic gradient descent.**

Hebbian Learning

Definition: Hebbian learning is a learning rule based on the principle that neurons that fire together, wire together. It's a biologically inspired learning mechanism and is foundational in understanding how learning and memory might work in biological systems.

Basic Principle:

- **Hebb's Rule: The change in the connection strength (weight) between two neurons is proportional to the product of their activations.**
- **Formula: Δwij=η·xi·yj\Delta w_{ij} = \eta \cdot x_i \cdot y_jΔwij=η·xi·yj where wijw_{ij}wij is the weight between neuron iii and neuron jjj, xix_ixi is the activation of neuron iii, yjy_jyj is the activation of neuron jjj, and η\etaη is the learning rate.**

Advantages:

- **Simple and biologically plausible.**

- **Can help in unsupervised learning scenarios and associative memory.**

Disadvantages:

- **Not typically used in modern deep learning due to limitations in handling complex tasks and lacking a mechanism for error correction.**
- **Does not inherently incorporate a concept of error minimization; learning is based purely on correlations.**

Applications:

- **Often used in models of associative memory and unsupervised learning algorithms.**
- **Forms the basis for more sophisticated learning rules, like those used in spiking neural networks.**