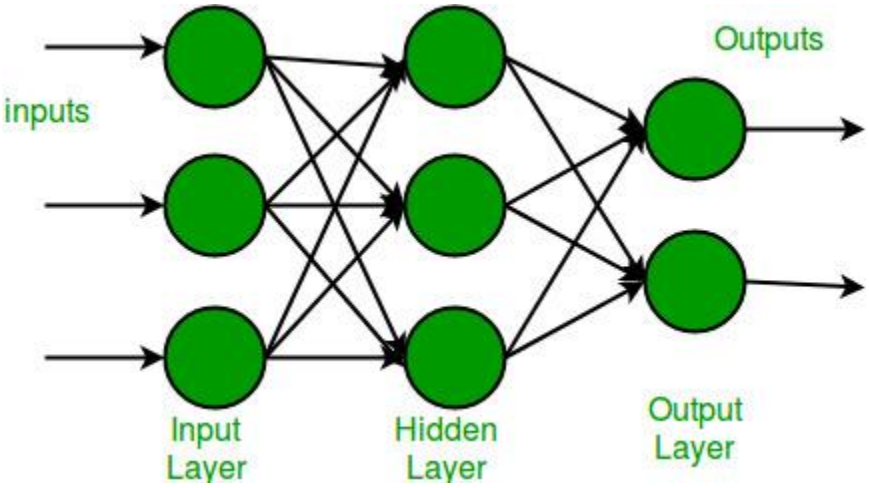
Multilayer Perceptrons (MLPs)

Multi-layer Perceptron

Multi-layer perception is also known as MLP. It is fully connected dense layers, which transform any input dimension to the desired dimension. A multi-layer perception is a neural network that has multiple layers. To create a neural network we combine neurons together so that the outputs of some neurons are inputs of other neurons.

A multi-layer perceptron has one input layer and for each input, there is one neuron(or node), it has one output layer with a single node for each output and it can have any number of hidden layers and each hidden layer can have any number of nodes. A schematic diagram of a Multi-Layer Perceptron (MLP) is depicted below.



In the multi-layer perceptron diagram above, we can see that there are three inputs and thus three input nodes and the hidden layer has three nodes. The output layer gives two outputs, therefore there are two output nodes. The nodes in the input layer take input and forward it for further process, in the diagram above the nodes in the input layer forwards their output to each of the three nodes in the hidden layer, and in the same way, the hidden layer processes the information and passes it to the output layer.

Every node in the multi-layer perception uses a sigmoid activation function. The sigmoid activation function takes real values as input and converts them to numbers between 0 and 1 using the sigmoid formula.

A Multilayer Perceptron (MLP) is a type of artificial neural network architecture that consists of multiple layers of interconnected nodes (also known as neurons) organized in a feedforward manner. It's a fundamental concept in deep learning and is used for various machine learning tasks, including classification, regression, and more complex tasks like image and text generation.

Here's a breakdown of the key components of a Multilayer Perceptron:

1. **Input Layer**: The first layer of the MLP that receives the raw input data, which could be features extracted from images, text, or any other type of data.
2. **Hidden Layers**: These are one or more layers situated between the input and output layers. Each hidden layer consists of multiple neurons that apply weighted sums of inputs, followed by activation functions. Hidden layers enable the network to learn complex and nonlinear patterns in the data.
3. **Output Layer**: The final layer of the MLP that produces the network's output. The number of neurons in this layer depends on the specific task—binary classification might require one neuron, while multi-class classification would have one neuron per class.
4. **Weights and Biases**: Each connection between neurons has an associated weight, and each neuron has a bias term. These parameters are learned during the training process to adjust the strength and influence of each connection.
5. **Activation Functions**: Neurons in hidden and output layers typically use activation functions to introduce nonlinearity into the network. Common activation functions include ReLU (Rectified Linear Unit), sigmoid, and tanh. These functions allow the network to model complex relationships in the data.

The process of training an MLP involves forward and backward passes:

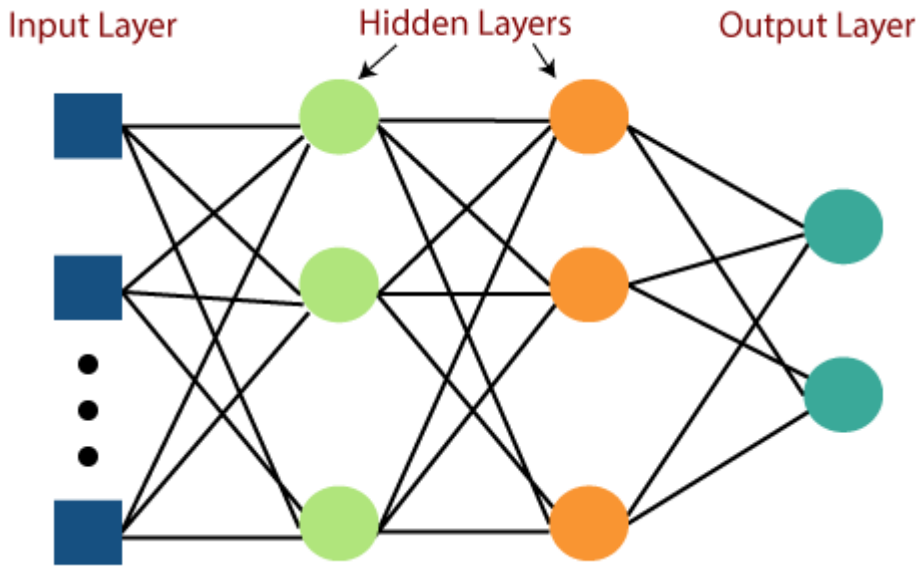
1. **Forward Pass**: During inference, data is fed through the network layer by layer, starting from the input layer. Each neuron computes a weighted sum of its inputs, applies an activation function, and passes the result to the next layer.

2. **Loss Calculation**: The output of the network is compared to the ground truth labels, and a loss function measures the difference between the predicted and actual outputs.
3. **Backward Pass (Backpropagation)**: The gradients of the loss with respect to the network's parameters (weights and biases) are computed layer by layer, starting from the output layer and moving backward. These gradients guide the update of the parameters to minimize the loss.

MLPs are capable of learning complex patterns in data and can approximate a wide range of functions, given the right architecture and sufficient training data. However, their success often depends on appropriate choices of hyperparameters (e.g., number of hidden layers, number of neurons per layer) and regularization techniques to prevent overfitting.

Multi-Layer perceptron defines the most complex architecture of artificial neural networks. It is substantially formed from multiple layers of the perceptron. If we want to understand what is a Multi-layer perceptron, we have to develop a multi-layer perceptron from scratch using Numpy.

The pictorial representation of multi-layer perceptron learning is as shown below-



MLP networks are used for supervised learning format. A typical learning algorithm for MLP networks is also called **back propagation's algorithm**.

A multilayer perceptron (MLP) is a feed forward artificial neural network that generates a set of outputs from a set of inputs. An MLP is characterized by several layers of input nodes connected as a directed graph between the input nodes connected as a directed graph between the input and output layers. MLP uses backpropagation for training the network. MLP is a deep learning method.

https://youtu.be/qw7wFGgNCSU?si=1qBNGXfupKY0hS5g

Representation Power of MLPs

https://medium.com/@HeCanThink/the-representation-power-of-perceptron-networks-mlp-1d189ad640bc

Sigmoid Neurons

https://youtu.be/X4RmokyD3U8?si=fIBrsqjl6IDfAamI

https://towardsdatascience.com/sigmoid-neuron-deep-neural-networks-a4cd35b629d7#:~:text=The%20building%20block%20of%20the,step%20functional%20output%20from%20perceptron

https://ranasinghiitkgp.medium.com/sigmoid-neuron-model-gradient-descent-with-sample-code-4919bfc9d4c4

Sigmoid neurons, also known as logistic neurons or logistic units, are a type of artificial neuron commonly used in the context of neural networks. They were one of the early activation functions used in neural network architectures. The sigmoid function itself is a mathematical function that maps any input to a value between 0 and 1.

The sigmoid activation function is defined as:

```
σ(z) = 1 / (1 + exp(-z))
```

Where:

σ(z) is the sigmoid function's output.

exp is the exponential function.

z is the weighted sum of inputs plus a bias term.

Here's a breakdown of how sigmoid neurons work:

Weighted Sum of Inputs: A sigmoid neuron receives inputs, each of which is multiplied by a corresponding weight. These weighted inputs are summed up along with a bias term. The bias allows the neuron to shift its activation threshold.

Activation Function: The weighted sum of inputs and the bias is then passed through the sigmoid activation function. This function "squashes" the input value to a range between 0 and 1. As the input to the sigmoid function becomes more positive, the output approaches 1, and as it becomes more negative, the output approaches 0.

Output Interpretation: The output of a sigmoid neuron can be interpreted as a probability. For example, in binary classification tasks, the sigmoid output can represent the probability that the input belongs to one of the classes.

Sigmoid neurons have some advantages and disadvantages:

Advantages:

Sigmoid neurons produce outputs in the range (0, 1), making them suitable for tasks where probabilities or confidence scores are needed.

They can be trained using methods that rely on gradient-based optimization, such as backpropagation.

Disadvantages:

Sigmoid neurons tend to saturate for extreme input values, leading to vanishing gradients. This can slow down learning, especially in deep networks.

The output of the sigmoid function is not centered around 0, which might lead to slow convergence during training.

Sigmoid neurons suffer from the "vanishing gradient" problem, which can make training deep networks more challenging.

Due to the disadvantages associated with sigmoid neurons, modern neural networks often use other activation functions such as the Rectified Linear Unit (ReLU) and its variants. These functions mitigate the vanishing gradient problem and promote faster training in deeper architectures. Nonetheless, sigmoid neurons played a significant role in the early development of neural networks and provided important insights into the design of activation functions.

Gradient descent neural network

Gradient Descent is known as one of the most commonly used optimization algorithms to train machine learning models by means of minimizing errors between actual and expected results. Further, gradient descent is also used to train Neural Networks.

In mathematical terminology, Optimization algorithm refers to the task of minimizing/maximizing an objective function f(x) parameterized by x. Similarly, in machine learning, optimization is the task of minimizing the cost function parameterized by the model's parameters. The main objective of gradient descent is to minimize the convex function using iteration of parameter updates. Once these machine learning models are optimized, these models can be used as powerful tools for Artificial Intelligence and various computer science applications.

In this tutorial on Gradient Descent in Machine Learning, we will learn in detail about gradient descent, the role of cost functions specifically as a barometer within Machine Learning, types of gradient descents, learning rates, etc.

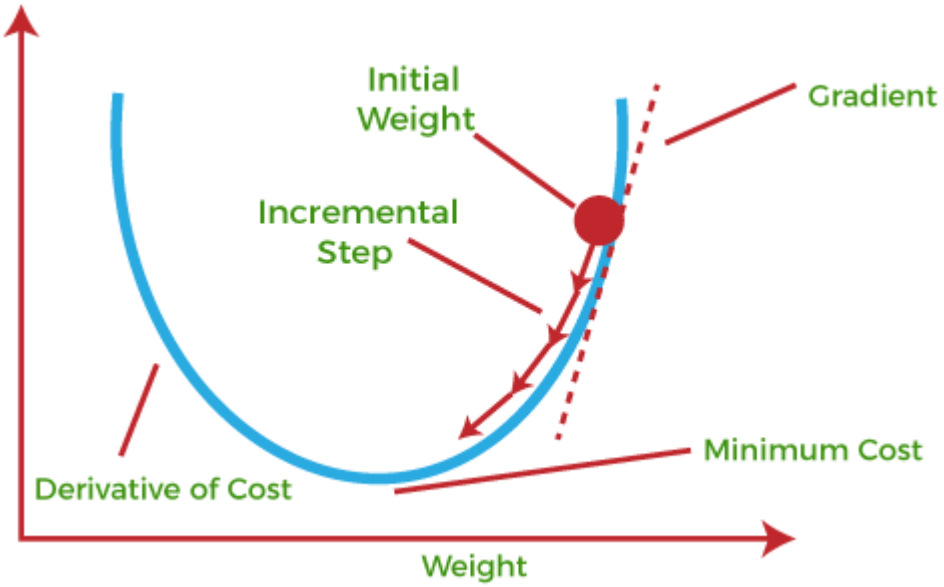
What is Gradient Descent or Steepest Descent?

Gradient descent was initially discovered by **"Augustin-Louis Cauchy"** in mid of 18th century. ***Gradient Descent is defined as one of the most commonly used iterative optimization algorithms of machine learning to train the machine learning and deep learning models. It helps in finding the local minimum of a function.***

The best way to define the local minimum or local maximum of a function using gradient descent is as follows:

- If we move towards a negative gradient or away from the gradient of the function at the current point, it will give the **local minimum** of that function.

- Whenever we move towards a positive gradient or towards the gradient of the function at the current point, we will get the **local maximum** of that function.



This entire procedure is known as Gradient Ascent, which is also known as steepest descent. ***The main objective of using a gradient descent algorithm is to minimize the cost function using iteration.*** To achieve this goal, it performs two steps iteratively:

- Calculates the first-order derivative of the function to compute the gradient or slope of that function.
- Move away from the direction of the gradient, which means slope increased from the current point by alpha times, where Alpha is defined as Learning Rate. It is a tuning parameter in the optimization process which helps to decide the length of the steps.

What is Cost-function?

The cost function is defined as the measurement of difference or error between actual values and expected values at the current position and present in the form of a single real number. It helps to increase and improve machine learning efficiency by providing feedback to this model so that it can minimize error and find the local or global minimum. Further, it continuously iterates along the direction of the negative gradient until the cost function approaches zero. At this steepest descent point, the model will stop learning further. Although cost function and loss function are considered synonymous, also there is a minor difference between them. The slight difference between the loss function and the cost function is about the error within the training of machine learning models, as loss function refers to the error of one

training example, while a cost function calculates the average error across an entire training set.

The cost function is calculated after making a hypothesis with initial parameters and modifying these parameters using gradient descent algorithms over known data to reduce the cost function.

Hypothesis:

Parameters:

Cost function:

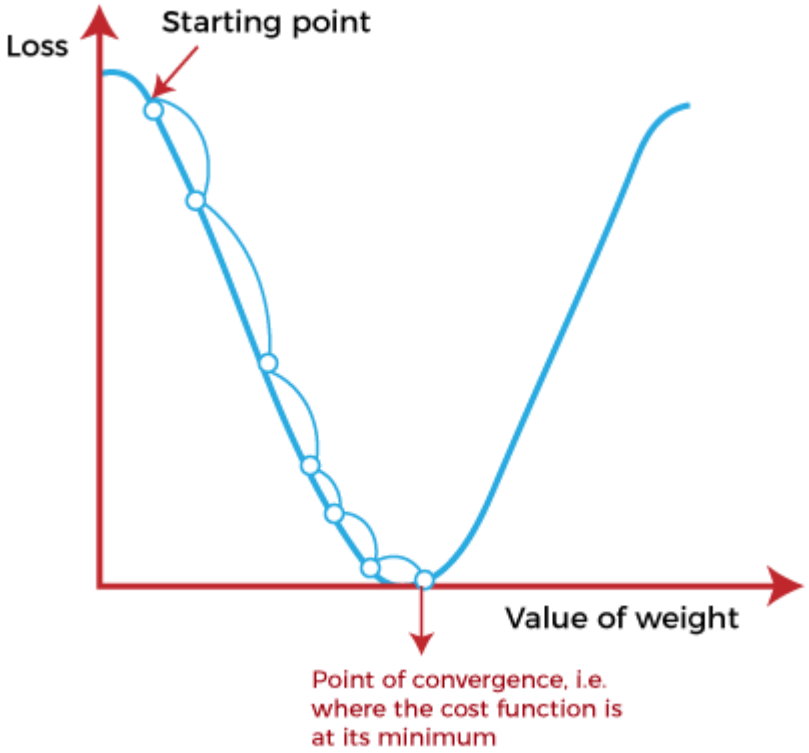
Goal:

How does Gradient Descent work?

Before starting the working principle of gradient descent, we should know some basic concepts to find out the slope of a line from linear regression. The equation for simple linear regression is given as:

1. $Y=mX+c$

Where 'm' represents the slope of the line, and 'c' represents the intercepts on the y-axis.



The starting point(shown in above fig.) is used to evaluate the performance as it is considered just as an arbitrary point. At this starting point, we will derive the first derivative or slope and then use a tangent line to calculate the steepness of this slope. Further, this slope will inform the updates to the parameters (weights and bias).

The slope becomes steeper at the starting point or arbitrary point, but whenever new parameters are generated, then steepness gradually reduces, and at the lowest point, it approaches the lowest point, which is called **a point of convergence.**

The main objective of gradient descent is to minimize the cost function or the error between expected and actual. To minimize the cost function, two data points are required:

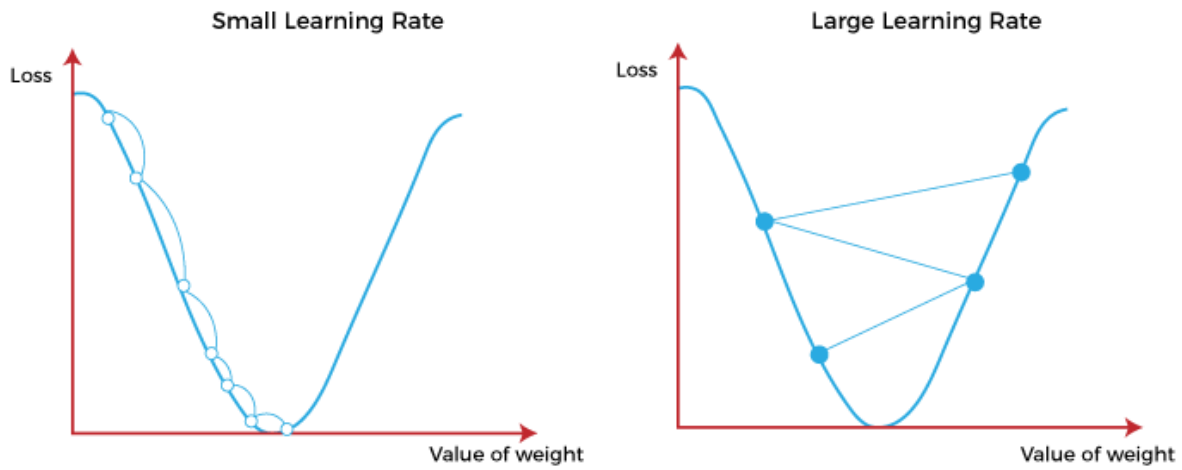
- **Direction & Learning Rate**

These two factors are used to determine the partial derivative calculation of future iteration and allow it to the point of convergence or local minimum or global minimum. Let's discuss learning rate factors in brief;

Learning Rate:

It is defined as the step size taken to reach the minimum or lowest point. This is typically a small value that is evaluated and updated based on the behavior of the cost function. If the learning rate is high, it results in larger steps but also leads to risks of

overshooting the minimum. At the same time, a low learning rate shows the small step sizes, which compromises overall efficiency but gives the advantage of more precision.



Types of Gradient Descent

Based on the error in various training models, the Gradient Descent learning algorithm can be divided into **Batch gradient descent, stochastic gradient descent, and mini-batch gradient descent.** Let's understand these different types of gradient descent:

1. Batch Gradient Descent:

Batch gradient descent (BGD) is used to find the error for each point in the training set and update the model after evaluating all training examples. This procedure is known as the training epoch. In simple words, it is a greedy approach where we have to sum over all examples for each update.

Advantages of Batch gradient descent:

- It produces less noise in comparison to other gradient descent.
- It produces stable gradient descent convergence.
- It is Computationally efficient as all resources are used for all training samples.

2. Stochastic gradient descent

Stochastic gradient descent (SGD) is a type of gradient descent that runs one training example per iteration. Or in other words, it processes a training epoch for each example within a dataset and updates each training example's parameters one at a time. As it requires only one training example at a time, hence it is easier to store in allocated memory. However, it shows some computational efficiency losses in comparison to batch gradient systems as it shows frequent updates that require more

detail and speed. Further, due to frequent updates, it is also treated as a noisy gradient. However, sometimes it can be helpful in finding the global minimum and also escaping the local minimum.

Advantages of Stochastic gradient descent:

In Stochastic gradient descent (SGD), learning happens on every example, and it consists of a few advantages over other gradient descent.

- It is easier to allocate in desired memory.
- It is relatively fast to compute than batch gradient descent.
- It is more efficient for large datasets.

3. MiniBatch Gradient Descent:

Mini Batch gradient descent is the combination of both batch gradient descent and stochastic gradient descent. It divides the training datasets into small batch sizes then performs the updates on those batches separately. Splitting training datasets into smaller batches make a balance to maintain the computational efficiency of batch gradient descent and speed of stochastic gradient descent. Hence, we can achieve a special type of gradient descent with higher computational efficiency and less noisy gradient descent.

Advantages of Mini Batch gradient descent:

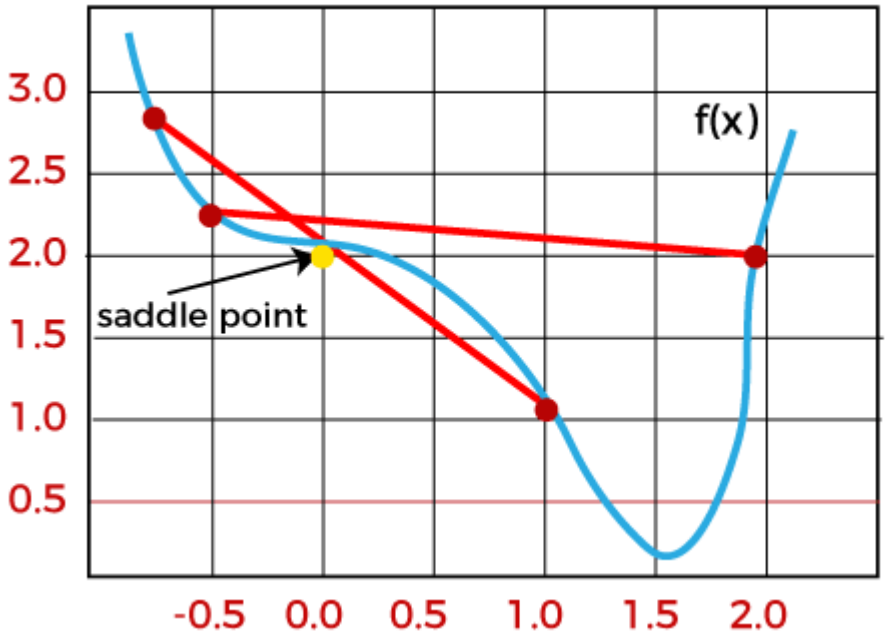
- It is easier to fit in allocated memory.
- It is computationally efficient.
- It produces stable gradient descent convergence.

Challenges with the Gradient Descent

Although we know Gradient Descent is one of the most popular methods for optimization problems, it still also has some challenges. There are a few challenges as follows:

1. Local Minima and Saddle Point:

For convex problems, gradient descent can find the global minimum easily, while for non-convex problems, it is sometimes difficult to find the global minimum, where the machine learning models achieve the best results.



Whenever the slope of the cost function is at zero or just close to zero, this model stops learning further. Apart from the global minimum, there occur some scenarios that can show this slop, which is saddle point and local minimum. Local minima generate the shape similar to the global minimum, where the slope of the cost function increases on both sides of the current points.

In contrast, with saddle points, the negative gradient only occurs on one side of the point, which reaches a local maximum on one side and a local minimum on the other side. The name of a saddle point is taken by that of a horse's saddle.

The name of local minima is because the value of the loss function is minimum at that point in a local region. In contrast, the name of the global minima is given so because the value of the loss function is minimum there, globally across the entire domain the loss function.

2. Vanishing and Exploding Gradient

In a deep neural network, if the model is trained with gradient descent and backpropagation, there can occur two more issues other than local minima and saddle point.

Vanishing Gradients:

Vanishing Gradient occurs when the gradient is smaller than expected. During backpropagation, this gradient becomes smaller that causing the decrease in the learning rate of earlier layers than the later layer of the network. Once this happens, the weight parameters update until they become insignificant.

Exploding Gradient:

Exploding gradient is just opposite to the vanishing gradient as it occurs when the Gradient is too large and creates a stable model. Further, in this scenario, model weight increases, and they will be represented as NaN. This problem can be solved using the dimensionality reduction technique, which helps to minimize complexity within the model.

https://www.ibm.com/topics/gradient-descent#:~:text=Gradient%20descent%20is%20an%20optimization,each%20iteration%20of%20parameter%20updates.

https://www.javatpoint.com/gradient-descent-in-machine-learning

https://youtu.be/7z6yXpYk7sw?si=yUj0EK3sRQ-OZeZ-

Gradient descent is a fundamental optimization algorithm used in training neural networks, including Multilayer Perceptrons (MLPs) and more complex architectures. The goal of training a neural network is to adjust its parameters (weights and biases) in a way that minimizes a predefined loss function. Gradient descent is employed to iteratively update these parameters based on the gradients of the loss function with respect to the parameters.

Here's a step-by-step overview of how gradient descent works in the context of training a neural network:

1. **Initialization**: The weights and biases of the neural network are initialized with small random values.
2. **Forward Pass**: An input data point is fed through the network using the current weights and biases. The network computes predictions for the input.
3. **Loss Calculation**: The difference between the predicted output and the actual target (ground truth) is calculated using a loss function. Common loss functions include mean squared error for regression tasks and cross-entropy for classification tasks.

4. **Backpropagation**: The gradients of the loss function with respect to the network's parameters (weights and biases) are calculated using the chain rule. This involves computing the gradients layer by layer, starting from the output layer and moving backward through the network.
5. **Gradient Update**: The gradients calculated in the previous step indicate the direction in which the loss function would decrease most rapidly. The parameters are updated in the opposite direction of the gradients to minimize the loss. The learning rate is a hyperparameter that controls the size of the steps taken during the update.
6. **Iterative Process**: Steps 2-5 are repeated for multiple iterations or epochs. In each iteration, a new batch of data (or sometimes a single data point) is processed, gradients are calculated, and parameter updates are made.
7. **Convergence**: The algorithm continues until the loss converges to a minimum or until a stopping criterion is met. This stopping criterion could be a fixed number of epochs, a threshold for the change in loss, or other conditions.

There are variations of gradient descent, such as stochastic gradient descent (SGD), mini-batch gradient descent, and more advanced optimizers like Adam and RMSprop. These variations introduce improvements to the basic gradient descent algorithm to make the training process more efficient and robust.

It's worth noting that while gradient descent is a powerful and widely used optimization algorithm, it can sometimes encounter challenges such as getting stuck in local minima or converging slowly. Techniques like learning rate schedules, momentum, and adaptive learning rates aim to address these challenges and improve the efficiency of neural network training.

Feedforward Neural Networks

https://youtu.be/svZBH0_qSt0?si=0wuFcjajGqYWwX-p

https://www.scaler.com/topics/deep-learning/introduction-to-feed-forward-neural-network/

Representation Power of Feedforward Neural Networks

Feedforward neural networks, including Multilayer Perceptrons (MLPs), possess remarkable representation power, which allows them to approximate a wide range of complex functions. This power stems from their ability to model intricate relationships between input and output data through multiple layers of interconnected neurons. Here are some key points highlighting the representation power of feedforward neural networks:

1. **Universal Approximation Theorem**: The Universal Approximation Theorem states that a feedforward neural network with a single hidden layer containing a finite number of neurons can approximate any continuous function on a bounded input domain to arbitrary accuracy, given enough neurons. This theorem underscores the neural network's capacity to represent a diverse set of functions.
2. **Hierarchy of Features**: Deep feedforward neural networks consist of multiple hidden layers, each of which can capture increasingly abstract and complex features from the input data. These layers form a hierarchy of feature extraction, enabling the network to learn hierarchical representations of data.
3. **Nonlinear Transformations**: The activation functions used in neural networks introduce nonlinearity, allowing the network to capture and model nonlinear relationships in data. This is crucial for handling complex patterns that linear models would struggle to represent.
4. **Complex Decision Boundaries**: Feedforward neural networks can learn intricate decision boundaries in classification tasks. The combination of multiple hidden layers and nonlinearity enables the network to separate classes with complex shapes, making them suitable for tasks involving image recognition, natural language processing, and more.

5. **Feature Learning**: Neural networks have the ability to learn relevant features from raw data without requiring explicit feature engineering. This is particularly advantageous when dealing with high-dimensional and unstructured data, such as images, audio, and text.
6. **Representation Transfer**: Pretrained neural network architectures trained on large datasets for general tasks (like image recognition on ImageNet) can be fine-tuned for specific tasks with smaller datasets. This highlights the fact that neural networks can capture general features that are transferable across tasks.

However, it's important to be aware of a few considerations:

- While neural networks have high representation power, successfully harnessing this power requires careful hyperparameter tuning, architecture design, and training strategies.
- Deeper networks with more layers and parameters can capture finer details in data but may also require more data and sophisticated regularization techniques to prevent overfitting.
- Neural networks are vulnerable to overfitting if not properly regularized, and they can also suffer from vanishing or exploding gradients, which affect the training process.

In summary, feedforward neural networks are powerful function approximators that can capture complex patterns and relationships in data. Their representation power, when combined with appropriate training and regularization techniques, makes them versatile tools for various machine learning tasks.

Three Classes of Deep Learning Basic Terminologies of Deep Learning

In the realm of deep learning, there are three primary classes of basic terminologies that lay the foundation for understanding the concepts and mechanisms involved. These classes are essential for grasping the fundamentals of deep learning models and their operations:

1. **Neural Networks and Architectures:**
    - **Neuron (Node):** The basic processing unit that receives input, computes a weighted sum of inputs, adds a bias, and applies an activation function to produce an output.
    - **Layer:** A collection of neurons connected in a specific arrangement, forming the building blocks of neural network architectures.
    - **Input Layer:** The initial layer that receives raw input data. Neurons in this layer often correspond to individual features of the data.
    - **Hidden Layer:** Layers situated between the input and output layers. These layers capture increasingly abstract representations of the input data.
    - **Output Layer:** The final layer that produces the network's predictions or outputs based on the learned representations.
    - **Activation Function:** A nonlinear function applied to the weighted sum of inputs in a neuron. Common activation functions include ReLU, sigmoid, and tanh.
    - **Feedforward Neural Network:** A neural network where information flows in one direction, from the input layer through hidden layers to the output layer.
2. **Training and Optimization:**
    - **Loss Function (Objective Function):** A mathematical measure that quantifies the difference between the predicted output and the actual target values. The goal is to minimize this loss during training.

- **Gradient Descent:** An optimization algorithm used to adjust the model's parameters iteratively in the direction that decreases the loss.
- **Backpropagation:** The process of computing gradients of the loss with respect to the model's parameters, starting from the output layer and moving backward through the layers.
- **Epoch:** A single pass through the entire training dataset during the training process.
- **Batch:** A subset of the training dataset used to compute gradients and update parameters during each iteration of training.
- **Learning Rate:** A hyperparameter that determines the step size for parameter updates during gradient descent.
- **Overfitting:** A phenomenon where a model learns to perform well on the training data but fails to generalize to new, unseen data.
- **Regularization:** Techniques to prevent overfitting by adding penalties or constraints to the optimization process.

3. **Deep Learning Concepts:**
    - **Deep Learning:** A subset of machine learning that focuses on neural networks with multiple hidden layers, enabling the learning of complex representations from data.
    - **Feature Extraction:** The process by which neural networks learn to automatically identify and extract relevant features from raw data.
    - **Transfer Learning:** Leveraging pre-trained neural network models on one task to boost performance on a related task.
    - **Convolutional Neural Network (CNN):** A type of neural network specialized for image and spatial data, featuring convolutional layers that capture local patterns.
    - **Recurrent Neural Network (RNN):** A type of neural network designed to process sequences of data, where information can flow both forward and backward in time.

- **Long Short-Term Memory (LSTM):** A type of RNN architecture that addresses the vanishing gradient problem and effectively captures long-range dependencies in sequences.
- **Generative Adversarial Network (GAN):** A framework consisting of a generator and a discriminator network, used to generate new data that closely resembles a given dataset.

These three classes of terminologies provide a solid foundation for understanding the concepts, mechanisms, and techniques used in the field of deep learning.

Certainly, when categorizing deep learning concepts into three classes, you can consider the following organization:

1. **Architectures and Components:**
    - **Neural Networks:** The foundational structure of deep learning, comprising interconnected nodes (neurons) organized into layers.
    - **Convolutional Neural Networks (CNNs):** Specialized architectures for image and spatial data, utilizing convolutional and pooling layers to capture patterns and hierarchies.
    - **Recurrent Neural Networks (RNNs):** Designed for sequence data, these networks maintain memory across time steps and are suitable for tasks like language modeling and time series analysis.
    - **Long Short-Term Memory (LSTM) Networks:** A variant of RNNs that mitigates the vanishing gradient problem, enabling better modeling of long-range dependencies.
    - **Generative Models:** Models that generate new data instances that resemble the training data distribution, including Generative Adversarial Networks (GANs) and Variational Autoencoders (VAEs).
2. **Training and Optimization:**
    - **Gradient Descent and Variants:** Optimization techniques that iteratively adjust model parameters to minimize a loss function, with variations like Stochastic Gradient Descent (SGD), Mini-Batch Gradient Descent, and advanced optimizers (Adam, RMSProp).

- **Backpropagation:** A key algorithm for computing gradients and updating model parameters by propagating error signals backward through the network.
- **Loss Functions:** Metrics that quantify the difference between predicted and actual values, guiding the optimization process.
- **Regularization Techniques:** Methods like Dropout and L1/L2 regularization to prevent overfitting by adding penalties or constraints to the optimization process.
- **Learning Rate Scheduling:** Adapting the learning rate during training to improve convergence and achieve better results.

3. **Applications and Specialized Areas:**
    - **Computer Vision:** Using deep learning for tasks like image classification, object detection, segmentation, and image generation.
    - **Natural Language Processing (NLP):** Applications involving text and language, such as sentiment analysis, machine translation, and text generation.
    - **Speech Recognition and Synthesis:** Employing deep learning for tasks like speech-to-text conversion and text-to-speech synthesis.
    - **Recommendation Systems:** Utilizing deep learning to provide personalized recommendations in areas like e-commerce and content streaming.
    - **Autonomous Systems:** Enabling self-driving cars and other autonomous devices through deep learning models that interpret and react to their environment.

These three classes provide a structured way to understand the diverse aspects of deep learning, covering architectural concepts, training and optimization techniques, and the various real-world applications where deep learning has demonstrated significant impact.

Training Feedforward DNN

https://www.javatpoint.com/pytorch-feed-forward-process-in-deep-neural-network

Training a feedforward Deep Neural Network (DNN) involves several steps, including data preparation, defining the architecture, initializing parameters, selecting a loss function, and optimizing the model using gradient descent or its variants. Here's a general outline of how to train a feedforward DNN:

1. **Data Preparation:**
    - **Data Collection:** Gather a labeled dataset that is representative of the task you want the DNN to perform. The dataset should include input features and corresponding target labels.
    - **Data Preprocessing:** Normalize or standardize input features to have a similar scale, handle missing values, and perform any necessary transformations. Split the dataset into training, validation, and test sets.
2. **Defining the Architecture:**
    - **Choose the Number of Layers:** Decide on the number of hidden layers and neurons in each layer. Deeper networks can capture more complex relationships, but be cautious of overfitting.
    - **Select Activation Functions:** Choose appropriate activation functions (ReLU, sigmoid, tanh, etc.) for neurons in hidden layers. The activation function introduces nonlinearity.
    - **Output Layer Configuration:** Configure the output layer based on the task—classification (softmax activation) or regression (linear activation).
3. **Initializing Parameters:**
    - **Initialize Weights and Biases:** Initialize the weights and biases of the neurons. Common techniques include random initialization from Gaussian distributions or using specific initialization methods like Xavier/Glorot initialization.
4. **Loss Function Selection:**

- **Choose a Loss Function:** Select a loss function that matches the nature of the task—cross-entropy for classification, mean squared error for regression, etc.

5. **Optimization and Training:**
    - **Gradient Descent Variants:** Choose an optimization algorithm such as vanilla gradient descent, stochastic gradient descent (SGD), or more advanced optimizers like Adam or RMSProp.
    - **Batching:** Divide the training dataset into batches and iterate over them during training. This aids in memory efficiency and optimization.
    - **Forward Pass:** For each batch, perform a forward pass through the network to compute predictions.
    - **Loss Computation:** Calculate the loss by comparing predictions to actual target values using the chosen loss function.
    - **Backpropagation:** Compute the gradients of the loss with respect to model parameters using backpropagation.
    - **Parameter Update:** Update model parameters (weights and biases) using the computed gradients and the chosen optimization algorithm.
6. **Validation and Hyperparameter Tuning:**
    - **Validation Set:** Periodically evaluate the model's performance on the validation set during training. This helps monitor overfitting and guides hyperparameter tuning.
    - **Hyperparameter Tuning:** Adjust hyperparameters such as learning rate, batch size, and regularization strength based on validation performance.
7. **Testing and Evaluation:**
    - **Test Set Evaluation:** After training, evaluate the model's performance on the test set, which provides an unbiased estimate of its generalization performance.
    - **Metrics:** Use appropriate evaluation metrics based on the task—accuracy, precision, recall, F1-score for classification; mean squared error for regression, etc.
8. **Regularization and Fine-Tuning:**

- **Regularization Techniques:** Apply techniques like dropout, L1/L2 regularization, and early stopping to prevent overfitting and improve generalization.
- **Fine-Tuning:** Fine-tune hyperparameters or model architecture based on insights gained from validation and test set performance.

Remember that training deep neural networks can be complex and iterative. Experimentation, patience, and monitoring are crucial for achieving the best results.

Multi Layered Feed Forward Neural Network

https://www.tutorialspoint.com/understanding-multi-layer-feed-forward-neural-networks-in-machine-learning

https://www.geeksforgeeks.org/multilayer-feed-forward-neural-network-in-data-mining/

A Multi-Layered Feedforward Neural Network (MLP) is a type of artificial neural network architecture that consists of multiple layers of interconnected neurons, where information flows in one direction, from the input layer to the output layer, without any loops or cycles. MLPs are the foundation of deep learning and have been used extensively for various machine learning tasks, including classification, regression, and more complex tasks like image and text analysis.

Here's a breakdown of the key components and characteristics of a multi-layered feedforward neural network:

1. **Input Layer:** The input layer receives the raw input data, which could be features extracted from images, text, or any other type of data. The number of neurons in this layer corresponds to the number of input features.
2. **Hidden Layers:** These are the intermediate layers between the input and output layers. Each hidden layer contains multiple neurons that apply weighted sums of inputs, add bias terms, and pass the results through activation functions. Hidden layers allow the network to learn complex features and patterns from the data.
3. **Output Layer:** The output layer produces the final predictions or classifications based on the learned representations. The number of

neurons in this layer depends on the specific task. For example, a binary classification task might have one output neuron, while a multi-class classification task could have multiple output neurons, one for each class.

4. **Weights and Biases:** Each connection between neurons has an associated weight, and each neuron has a bias term. These parameters are learned during the training process and determine the strength of connections and the activation threshold.
5. **Activation Functions:** Activation functions introduce nonlinearity to the network. Common activation functions used in hidden layers include ReLU (Rectified Linear Unit), sigmoid, and tanh. The choice of activation function affects the network's ability to model complex relationships.
6. **Forward Pass:** During inference, data is fed through the network layer by layer, with each neuron computing a weighted sum of its inputs, adding the bias, and passing the result through the activation function.
7. **Loss Function:** A loss function quantifies the difference between the predicted output and the actual target values. The choice of the loss function depends on the task—mean squared error for regression, cross-entropy for classification, etc.
8. **Training:** The network is trained using optimization algorithms like gradient descent. Gradients of the loss function with respect to the parameters are computed using backpropagation, and the parameters are updated iteratively to minimize the loss.

MLPs are capable of learning complex hierarchical representations from data, which allows them to model intricate relationships and patterns. The depth and width of the network (number of layers and neurons per layer) influence its capacity to learn complex functions. However, deeper networks also require careful tuning and regularization techniques to prevent overfitting and ensure effective training.

Learning Factors

Learning factors:

The factors that improve the convergence of EBPTA are called as learning factors

The factors are as follows:

1. Initial weights
2. Steepness of activation function
3. Learning constant
4. Momentum
5. Network architecture
6. Necessary number of hidden neurons

1. Initial weights:

- The weights of the network to be trained are typically initialized at small random values.
- The initialization strongly affects the ultimate solution

2. Steepness of activation function

- The neuron's continuous activation function is characterized by its steepness factor
- Also the derivatice of the activation function serves as a multiplying factor in building components of the error signal vectors.

3. Learing constant:

- The effectiveness and convergence of the error back propagation learning algorithm depen significantly on the value of the learning constant.

4. Momentum:

- The purpose of the momentum method is ti accelerate the convergence of the error back propagation learning algorithm.
- The method involves supplementing the current weight adjustment with a fraction of the most recent weight adjustment.

5. Network architecture:

- One of the most important attributes if a layerd neural network design is choosing the architecture
- The number of input nodes is simply determined by the dimension or size of the input vector to be classified. The input vector size usually corresponds to the total number of distinct features of the input patterns.

6. Necessary number of hidden neurons:

- This problem of choice of size of the hidden layer is under intensive study with no conclusive answers available.
- One formula can be used to find out how many hidden layer neurons need to be used to achieve classification into M classes in x dimensional patterns space.

In the context of machine learning and neural networks, "learning factors" usually refer to the various components and parameters that play a crucial role in the learning process of a model. These factors influence how a model adapts and updates its parameters during training to improve its performance. Here are some key learning factors:

1. **Learning Rate:** The learning rate is a critical hyperparameter that determines the step size taken in the direction of the gradient during parameter updates. It controls how quickly the model converges to the optimal solution. Choosing an appropriate learning rate is essential—too high a value might result in overshooting the optimal point, while too low a value can slow down convergence.
2. **Batch Size:** During training, the dataset is divided into smaller batches. The batch size represents the number of training examples used in each iteration of parameter updates. Larger batch sizes can lead to more stable updates, while smaller batch sizes introduce more randomness. The choice of batch size can also impact memory usage and computational efficiency.
3. **Optimization Algorithm:** The optimization algorithm determines how the model's parameters are updated based on the computed gradients. Common optimization algorithms include stochastic gradient descent (SGD), Adam, RMSProp, and more. Each algorithm has its own advantages and may be more suitable for specific scenarios.
4. **Initialization Methods:** The initial values of the model's parameters (weights and biases) can significantly affect training. Proper initialization

can help avoid issues like vanishing or exploding gradients. Common initialization methods include random initialization from a Gaussian distribution and Xavier/Glorot initialization.

5. **Regularization Techniques:** Regularization helps prevent overfitting and improves generalization. Techniques like L1/L2 regularization, dropout, and early stopping constrain the model's complexity or add noise to training data.
6. **Activation Functions:** The choice of activation functions in the neurons of the network impacts the nonlinearity and expressive power of the model. Common choices include ReLU, sigmoid, and tanh, each with its own characteristics and benefits.
7. **Data Augmentation:** In image-based tasks, data augmentation involves applying various transformations (rotations, flips, cropping) to training images. This technique introduces more diversity into the training data and helps the model become more robust.
8. **Learning Schedule:** Some optimization algorithms use learning rate schedules that dynamically adjust the learning rate during training. Techniques like learning rate annealing or learning rate decay can improve convergence.
9. **Hyperparameter Tuning:** Many of these learning factors are hyperparameters, which need to be tuned to achieve the best model performance. Hyperparameter tuning involves experimenting with different values and strategies to find the configuration that yields optimal results.

The interaction of these learning factors can significantly impact the effectiveness and efficiency of training a machine learning model. Fine-tuning and experimentation are often necessary to strike the right balance and achieve the best performance.

Activation functions

https://www.geeksforgeeks.org/activation-functions-neural-networks/

https://youtu.be/7LcUkgzx3AY?si=QJ04Umm_uxEHOI1S

Activation functions play a crucial role in artificial neural networks by introducing nonlinearity to the network's transformations. They determine the output of a neuron given its weighted sum of inputs and bias. Activation functions enable neural networks to capture complex relationships and make them capable of learning and approximating arbitrary functions. Here are some commonly used activation functions:

1. **Sigmoid Function (Logistic Activation):**
    - Formula: $\sigma(z) = 1 / (1 + \exp(-z))$
    - Output Range: (0, 1)
    - Characteristics: It produces a smooth S-shaped curve. It's used in the past for binary classification, but is less commonly used now due to its vanishing gradient problem.
2. **Hyperbolic Tangent (Tanh) Function:**
    - Formula: $\tanh(z) = (2 / (1 + \exp(-2z))) - 1$
    - Output Range: (-1, 1)
    - Characteristics: Similar to the sigmoid, but with an output range from -1 to 1. It also suffers from the vanishing gradient problem.
3. **Rectified Linear Unit (ReLU):**
    - Formula: $\text{ReLU}(z) = \max(0, z)$
    - Output Range: $[0, \infty)$
    - Characteristics: It outputs the input directly if it's positive, and zero otherwise. ReLU is widely used because of its simplicity and effectiveness in mitigating the vanishing gradient problem.
4. **Leaky ReLU:**
    - Formula: $\text{LeakyReLU}(z) = z$ if $z > 0$, else αz where α is a small positive constant (usually around 0.01).

- Output Range: (-∞, ∞)
- Characteristics: It's a variant of ReLU that allows a small gradient when the input is negative, helping to address the "dying ReLU" problem.

Linear Activation Function:

The linear activation function is one of the simplest activation functions. It computes the weighted sum of the inputs without any transformation. Mathematically, for a single neuron, it can be represented as:

f(x) = ax

Here, 'x' represents the weighted sum of inputs, 'a' is a constant (weight), and 'f(x)' is the output. This function simply scales the input by a constant 'a'. As a result, it's a linear transformation, and stacking multiple layers of neurons with linear activation functions would still result in a linear transformation. Because of this, linear activation functions are rarely used in hidden layers of neural networks. They are primarily used in the output layer for regression tasks, where the network needs to output a continuous value.

These activation functions can be chosen based on the specific characteristics of the problem, the architecture of the network, and the challenges associated with each function. Activation functions are a critical component in designing effective neural network architectures.

Loss Functions in NLP

Loss functions in natural language processing (NLP) play a critical role in training machine learning models for various NLP tasks. The choice of an appropriate loss function depends on the specific NLP task you are working on, such as text classification, sequence labeling, machine translation, or text generation. Here are some commonly used loss functions in NLP:

1. **Cross-Entropy Loss (Log Loss)**:
    - **Binary Cross-Entropy Loss (Binary Log Loss)**: Used for binary classification tasks, where the model predicts a binary outcome (e.g., sentiment analysis, spam detection). It measures the dissimilarity between predicted probabilities and true binary labels.

- **Categorical Cross-Entropy Loss (Multiclass Log Loss)**: Used for multiclass classification tasks, where there are more than two classes (e.g., topic classification, part-of-speech tagging). It measures the dissimilarity between predicted class probabilities and true class labels.

2. **Mean Squared Error (MSE) Loss**:
    - Used for regression tasks in NLP, where the model predicts continuous values (e.g., predicting a numerical value like sentiment score or price). It measures the squared difference between predicted values and true target values.

Loss function parameters are another essential component of natural language processing (NLP) models. They are used to evaluate how well the model is performing by comparing the predicted output with the actual output. The loss function measures the difference between the predicted and actual output and is used to update the model's parameters during training.

In simpler terms, loss function parameters can be thought of as **a measure of how well a model is doing at its task**. Just like how a teacher grades a student's performance, loss function parameters grade a model's performance. The goal of the model is to minimize the loss function and improve its performance.

Cross-entropy Loss

One example of a loss function used in NLP models is the cross-entropy loss. This loss function is commonly used for classification tasks, where the model is trained to predict a class label for a given input. The cross-entropy loss measures the difference between the predicted probability distribution and the actual probability distribution of the class labels.

Mean Squared Error (MSE) Loss

Another example of a loss function used in NLP models is the mean squared error (MSE) loss. This loss function is used for regression tasks, where the model is trained to predict a

continuous output value. The MSE loss measures the difference between the predicted output and the actual output.

https://machinelearningmastery.com/loss-and-loss-functions-for-training-deep-learning-neural-networks/

Choosing output function and loss function

Choosing the right output function and loss function is a critical step in designing and training a machine learning model for a specific task. The choice of these functions depends on the nature of the task, the type of data, and the overall model architecture. Here are some considerations for choosing the appropriate output function and loss function:

1. **Task Type**:
    - **Regression Tasks**: If your task involves predicting continuous values (e.g., predicting house prices), you typically use a linear output function (identity function) and the Mean Squared Error (MSE) loss function.
    - **Binary Classification**: For tasks where the output is binary (e.g., spam detection, sentiment analysis), you typically use a sigmoid or softmax output function and the Binary Cross-Entropy Loss or Categorical Cross-Entropy Loss, depending on the number of classes.
    - **Multiclass Classification**: If there are more than two classes (e.g., topic classification, part-of-speech tagging), you often use a softmax output function along with Categorical Cross-Entropy Loss.
    - **Sequence-to-Sequence**: In tasks like machine translation or text summarization, you use a sequence output function along with Sequence Cross-Entropy Loss or similar sequence-based losses.
    - **Ranking Tasks**: For tasks related to ranking (e.g., search ranking), you might use ranking-specific loss functions like pairwise ranking loss or listwise ranking loss.
2. **Model Architecture**:

- **Neural Networks**: For most deep learning models, you'll use standard output functions like softmax, sigmoid, or linear, depending on the task. The choice of the loss function follows the task type and output function.
- **Recurrent Neural Networks (RNNs)**: When working with sequential data, such as text or time series, you might use specialized loss functions like CTC loss for speech recognition or sequence-to-sequence tasks.
- **Transformers**: Transformers are widely used in NLP. They often employ softmax output functions and Categorical Cross-Entropy Loss for classification tasks. For sequence-to-sequence tasks, they use Sequence Cross-Entropy Loss.

3. **Evaluation Metrics**:
    - Consider the evaluation metrics you plan to use to assess your model's performance. The choice of loss function should align with the evaluation metric. For example, if you are optimizing for accuracy, the Cross-Entropy Loss might be suitable.
4. **Data Characteristics**:
    - Understand the nature of your data. Are there class imbalances? Are the data points in different classes equally important? These factors can influence the choice of loss function and any weighting applied to it.
5. **Objective**:
    - Clearly define the objective of your task. For instance, in some cases, you may want to optimize for precision, recall, or F1-score instead of accuracy. In such cases, you might use a custom loss function that directly optimizes the desired metric.
6. **Regularization**:
    - Depending on your model's complexity and the risk of overfitting, you might incorporate regularization terms into your loss function, such as L1 or L2 regularization.
7. **Domain Knowledge**:

- Sometimes, domain-specific knowledge can guide your choice of loss function. For instance, in medical diagnosis, false negatives may be more costly than false positives, affecting your choice of loss.

8. **Experimentation**:
    - Don't hesitate to experiment with different output and loss functions to see what works best for your specific problem. It's often a trial-and-error process.

In summary, selecting the right output function and loss function is a crucial part of designing a machine learning model. It involves considering the task, model architecture, evaluation metrics, data characteristics, and any domain-specific knowledge to make an informed decision.

Optimization Learning with backpropagation

Optimization learning with backpropagation is a fundamental technique used to train artificial neural networks, including deep learning models. Backpropagation, short for "backward propagation of errors," is an iterative optimization algorithm that adjusts the model's parameters to minimize a specified loss or cost function. Here's a step-by-step overview of how optimization learning with backpropagation works:

1. **Initialize Model Parameters**:

Start by initializing the weights and biases of the neural network with small random values. These parameters represent the model's ability to map input data to desired output.

2. **Forward Pass**:
    - Input data is passed through the network layer by layer, from the input layer to the output layer.
    - Each neuron in each layer computes a weighted sum of its inputs and applies an activation function to produce an output.
    - The output of the final layer is the network's prediction for the given input.

3. **Compute Loss**:

Compare the network's prediction to the actual target values using a loss function. Common loss functions include mean squared error (MSE) for regression tasks and cross-entropy loss for classification tasks. The loss quantifies how far off the model's predictions are from the true values.

4. **Backward Pass (Backpropagation)**:

Backpropagation involves calculating the gradient of the loss with respect to each model parameter. This is done by applying the chain rule of calculus, which allows us to compute how changes in each parameter affect the overall loss.

- Start with the output layer and calculate the gradient of the loss with respect to the output layer's activations and parameters.
- Propagate these gradients backward through the network, layer by layer, by calculating the gradients at each layer with respect to its inputs and parameters.
- Update the model's parameters in the opposite direction of the gradient to minimize the loss. This update can be performed using various optimization algorithms, with stochastic gradient descent (SGD) being one of the most common.

5. **Repeat**:

The forward pass, loss computation, and backward pass steps are repeated for multiple iterations or epochs. During each iteration, the model's parameters are updated to reduce the loss further.

6. **Convergence**:

Continue the training process until one or more stopping criteria are met. Common stopping criteria include a maximum number of epochs, a threshold on the loss value, or early stopping based on a validation dataset's performance.

7. **Evaluate**:

Once training is complete, evaluate the trained model's performance on a separate test dataset to assess its generalization ability. Common evaluation metrics depend on the specific task and may include accuracy, F1-score, mean squared error, etc.

Key considerations during optimization learning with backpropagation:

- Learning Rate: The learning rate is a hyperparameter that controls the size of parameter updates in each iteration. It should be chosen carefully as it can affect the convergence and stability of training.
- Batch Size: Training can be performed using mini-batches of data rather than the entire dataset at once. This approach, known as mini-batch gradient descent, can lead to faster convergence.
- Regularization: Techniques like L1 and L2 regularization can be applied to prevent overfitting during training.
- Initialization: Proper initialization of weights can speed up convergence. Techniques like Xavier/Glorot initialization and He initialization are commonly used.

Optimization learning with backpropagation is the foundation of training neural networks and has been instrumental in the success of deep learning in various domains, including computer vision, natural language processing, and reinforcement learning.

https://www.edureka.co/blog/backpropagation/#:~:text=The%20Backpropagation%20algorithm%20looks%20for,solution%20to%20the%20learning%20problem.

https://youtu.be/ayOOMlgb320?si=hO8AwHtdm8G4xIQe

Gradient Descent

https://towardsdatascience.com/gradient-descent-algorithm-a-deep-dive-cf04e8115f21#:~:text=Gradient%20descent%20(GD)%20is%20an,e.g.%20in%20a%20linear%20regression).

https://www.ibm.com/topics/gradient-descent

Refer ppt too

Gradient descent is a fundamental optimization algorithm used in deep learning to train neural networks. It's the process of iteratively adjusting the model's parameters (weights and biases) to minimize a loss or cost function. Here's an overview of how gradient descent works in deep learning:

1. **Initialization**:
    - Start by initializing the model's parameters randomly or using specific initialization techniques. These parameters are the knobs that the algorithm will adjust during training.
2. **Forward Pass**:
    - In the forward pass, input data is passed through the neural network.
    - Each neuron computes a weighted sum of its inputs and applies an activation function to produce an output.
    - The final layer's output is compared to the ground truth to calculate the loss, which quantifies how well the model is performing on the task.
3. **Backward Pass (Backpropagation)**:
    - Backpropagation is the heart of gradient descent. It computes the gradient of the loss function with respect to each model parameter.
    - The gradients represent the direction and magnitude of the change needed in each parameter to minimize the loss.
    - The chain rule of calculus is used to compute these gradients layer by layer, starting from the output layer and going backward.
4. **Update Parameters**:
    - Once the gradients are computed, they are used to update the model's parameters.
    - The parameters are adjusted in the opposite direction of the gradients to decrease the loss.
    - A hyperparameter called the learning rate controls the step size of the parameter updates. A small learning rate may slow down convergence, while a large one can cause divergence.
5. **Repeat**:

- Steps 2 to 4 are repeated for multiple iterations or epochs. During each iteration, the model makes incremental improvements in its ability to make accurate predictions.

6. **Convergence**:
    - The training process continues until one or more stopping criteria are met. Common stopping criteria include a maximum number of epochs, a threshold on the loss value, or early stopping based on validation performance.
7. **Evaluation**:
    - After training is complete, the model's performance is evaluated on a separate test dataset to assess its ability to generalize to new, unseen data.

Key Concepts and Considerations:

- **Loss Function**: The choice of the loss function depends on the task (e.g., mean squared error for regression, cross-entropy for classification). It measures the error between predictions and ground truth.
- **Gradients**: Gradients provide information on how sensitive the loss is to changes in each parameter. They guide the parameter updates in the direction of steepest decrease in the loss.
- **Learning Rate**: The learning rate determines the step size for parameter updates. It's a crucial hyperparameter that affects the convergence and stability of training.
- **Batch Size**: Training can be performed using mini-batches of data rather than the entire dataset at once. This approach, known as mini-batch gradient descent, can lead to faster convergence.
- **Regularization**: Techniques like L1 and L2 regularization can be applied to prevent overfitting during training.

Gradient descent is a core concept in deep learning and serves as the foundation for training neural networks to perform a wide range of tasks, including image classification, natural language processing, and more.

Learning Parameters: Gradient Descent (GD), Stochastic and Mini Batch GD, Momentum Based GD, Nesterov Accelerated GD, AdaGrad, Adam, RMSProp

Learning parameters in deep learning involves finding the optimal values for a model's weights and biases to minimize a specified loss function. Various optimization algorithms are used for this purpose. Here's a brief overview of some common optimization algorithms used in deep learning:

1. **Gradient Descent (GD)**:
    - GD is the fundamental optimization algorithm used in deep learning.
    - It computes the gradient (derivative) of the loss function with respect to all model parameters.
    - Updates all parameters simultaneously in the opposite direction of the gradient to minimize the loss.
    - Can be slow for large datasets and high-dimensional parameter spaces.
2. **Stochastic Gradient Descent (SGD)**:
    - Instead of using the entire dataset in each iteration (as in GD), SGD updates parameters using a single randomly selected data point at a time.
    - This introduces noise into the parameter updates, which can help escape local minima and converge faster.
    - SGD is often used for training large-scale models.
3. **Mini-Batch Gradient Descent**:
    - Mini-Batch GD strikes a balance between GD and SGD by updating parameters using a small, random subset (mini-batch) of the training data in each iteration.
    - This approach leverages the advantages of both GD and SGD, making it efficient for both convergence and computational resources.

4. **Momentum-Based Gradient Descent**:
    - Momentum-based methods add a momentum term to the parameter updates, which helps accelerate convergence.
    - They accumulate past gradients and use them to influence the current update direction.
    - The momentum term helps overcome oscillations in the loss landscape.
5. **Nesterov Accelerated Gradient Descent (NAG)**:
    - Nesterov Accelerated GD is an improvement over regular momentum-based GD.
    - It calculates the gradient at a point ahead of the current position by using the momentum term.
    - This anticipatory update can lead to faster convergence and better performance.
6. **AdaGrad (Adaptive Gradient Descent)**:
    - AdaGrad adapts the learning rate for each parameter based on the historical gradient information.
    - It gives smaller learning rates to frequently updated parameters and larger learning rates to infrequently updated ones.
    - Well-suited for sparse data but may result in overly aggressive learning rate decay.
7. **RMSProp (Root Mean Square Propagation)**:
    - RMSProp addresses the aggressive learning rate decay issue of AdaGrad.
    - It uses a moving average of squared gradients to adaptively scale the learning rates.
    - Helps maintain a balance between learning fast and avoiding divergence.
8. **Adam (Adaptive Moment Estimation)**:
    - Adam combines the ideas of momentum-based GD and RMSProp.

- It uses both the first-moment (like momentum) and second-moment (like RMSProp) estimates of the gradients.
- Adam is widely used and often exhibits fast convergence and good generalization.

The choice of optimization algorithm depends on factors such as the dataset size, model architecture, and specific problem. It often involves empirical experimentation to determine which algorithm works best for a given deep learning task.

Stochastic Gradient Descent (SGD)

Stochastic Gradient Descent (SGD) is a variant of the gradient descent optimization algorithm used in machine learning and deep learning. It is a widely used optimization technique for training neural networks and other models, especially when dealing with large datasets. SGD differs from the standard gradient descent in how it updates model parameters. Here's how SGD works:

1. **Initialization**:
    - Start by initializing the model's parameters (weights and biases) with random values or using specific initialization techniques.
2. **Data Shuffling**:
    - Before each training epoch (a full pass through the dataset), the training data is often shuffled randomly. Shuffling ensures that the data points are presented to the model in a random order during training.
3. **Iterative Updates**:
    - In each training iteration (or batch), instead of computing gradients for the entire dataset (as in standard gradient descent), SGD computes gradients for a single data point or a small subset of the data, called a mini-batch.
    - The mini-batch size is typically a hyperparameter and can range from a single data point (true stochastic gradient descent) to a small fraction of the dataset.

- The gradients are computed based on the loss for the current mini-batch.

4. **Parameter Updates**:
    - After computing the gradients for the mini-batch, the model's parameters (weights and biases) are updated using the gradients.
    - The update rule is typically of the form: **parameter = parameter - learning_rate * gradient**, where the learning rate is a hyperparameter that controls the step size of the updates.
    - The parameter updates are performed separately for each mini-batch.
5. **Repeat**:
    - Steps 3 and 4 are repeated for a specified number of training iterations (epochs).
    - The entire dataset is processed in mini-batches, and each mini-batch contributes to the gradual improvement of the model's parameters.
6. **Convergence**:
    - SGD continues until a stopping criterion is met. Common stopping criteria include reaching a maximum number of epochs or achieving a desired level of convergence in the loss.

Key Characteristics and Benefits of SGD:

- **Efficiency**: By using mini-batches, SGD can significantly speed up the training process compared to standard gradient descent, especially when dealing with large datasets.
- **Stochasticity**: The use of randomly selected data points in each iteration introduces a level of noise into the optimization process. This noise can help the model escape local minima and explore different parts of the loss landscape.
- **Parallelism**: SGD can be parallelized effectively, making it suitable for distributed computing and GPU acceleration.

- **Hyperparameters**: Important hyperparameters in SGD include the learning rate, mini-batch size, and the number of training epochs. Proper tuning of these hyperparameters is crucial for achieving good performance.

It's important to note that the choice of mini-batch size and learning rate can significantly affect the convergence and stability of SGD. Additionally, SGD may exhibit more variability in the training process compared to other optimization algorithms, but this variability can often be mitigated with proper hyperparameter tuning and regularization techniques.

Mini Batch GD

Mini-Batch Gradient Descent (Mini-Batch GD) is a variant of the gradient descent optimization algorithm used in machine learning and deep learning. It lies between two extremes: Stochastic Gradient Descent (SGD), where the gradient is computed for each individual data point, and Batch Gradient Descent (BGD), where the gradient is computed for the entire training dataset. In Mini-Batch GD, the gradient is computed based on a random subset of the training data in each iteration. Here's how Mini-Batch GD works:

1. **Initialization**:
    - Start by initializing the model's parameters (weights and biases) with random values or using specific initialization techniques.
2. **Data Shuffling and Partitioning**:
    - Before training begins, the training data is often shuffled randomly to ensure that the data points are presented to the model in a random order.
    - The training dataset is divided into smaller subsets called mini-batches. The mini-batch size is a hyperparameter that determines how many data points are included in each mini-batch.
3. **Iterative Updates**:
    - In each training iteration (or batch), a mini-batch of data is randomly selected from the training dataset.

- The gradients of the loss function with respect to the model parameters are computed based on the loss for the current mini-batch.
- The parameter updates are performed using these mini-batch gradients. The update rule is typically of the form: **parameter = parameter - learning_rate * gradient**, where the learning rate is a hyperparameter that controls the step size of the updates.

4. **Repeat**:
    - Steps 3 and 4 are repeated for a specified number of training iterations (epochs).
    - The entire dataset is processed in mini-batches, and each mini-batch contributes to the gradual improvement of the model's parameters.
5. **Convergence**:
    - The training process continues until one or more stopping criteria are met. Common stopping criteria include reaching a maximum number of epochs or achieving a desired level of convergence in the loss.

Key Characteristics and Benefits of Mini-Batch GD:

- **Efficiency**: Mini-Batch GD strikes a balance between the computational efficiency of Batch Gradient Descent (BGD) and the stochasticity of Stochastic Gradient Descent (SGD). It allows for faster convergence than BGD while reducing the noise introduced by SGD.
- **Generalization**: Mini-Batch GD introduces some level of noise into the optimization process, which can act as a form of regularization. This regularization can help improve the model's generalization to unseen data.
- **Parallelism**: Mini-Batch GD can be parallelized effectively, making it suitable for distributed computing and GPU acceleration.
- **Hyperparameters**: Important hyperparameters in Mini-Batch GD include the mini-batch size and the learning rate. Proper tuning of these hyperparameters is crucial for achieving good performance.

The choice of mini-batch size can impact the convergence speed and stability of Mini-Batch GD. Smaller mini-batches introduce more noise but can lead to faster convergence, while larger mini-batches reduce noise but may slow down convergence. Therefore, the selection of an appropriate mini-batch size is often an empirical process and depends on factors such as the dataset size and model architecture.

Momentum Based GD

Momentum-Based Gradient Descent is an optimization algorithm used in machine learning and deep learning to accelerate convergence during the training of neural networks and other models. It's an enhancement of the standard gradient descent algorithm that aims to overcome some of its limitations, such as slow convergence in certain cases and the tendency to get stuck in shallow minima. Here's an overview of how Momentum-Based Gradient Descent works:

1. **Initialization**:
    - Start by initializing the model's parameters (weights and biases) with random values or using specific initialization techniques.
2. **Introduction of Momentum Term**:
    - In standard gradient descent, the parameter updates are based solely on the gradient of the loss function at the current iteration.
    - Momentum-Based GD introduces a momentum term, usually denoted as "beta" (β), which is a hyperparameter between 0 and 1.
    - The momentum term accumulates a moving average of past gradients to influence the direction and speed of parameter updates.
3. **Iterative Updates**:
    - In each training iteration, calculate the gradient of the loss function with respect to the model parameters based on the loss for the current mini-batch (as in Mini-Batch GD).
    - Update the moving average of past gradients:
        - **v = β * v + (1 - β) * gradient**

- Update the model parameters using the moving average of gradients:
    - **parameter = parameter - learning_rate * v**
- Here, "v" represents the velocity or accumulated gradient, "β" is the momentum hyperparameter, and "learning_rate" is the learning rate hyperparameter.

4. **Repeat**:
    - Continue the iterative updates using mini-batches, accumulating the gradient's moving average and updating the parameters for a specified number of training iterations (epochs).
5. **Convergence**:
    - The training process continues until one or more stopping criteria are met. Common stopping criteria include reaching a maximum number of epochs or achieving a desired level of convergence in the loss.

Key Characteristics and Benefits of Momentum-Based Gradient Descent:

- **Acceleration**: The momentum term allows the optimization process to accumulate speed in directions where the gradients consistently point. This helps the algorithm to move faster through flat regions and overcome local minima.
- **Smoothing**: Momentum acts as a smoothing mechanism by reducing oscillations in the optimization path. It helps prevent the model from making abrupt and jittery updates.
- **Hyperparameter Tuning**: The momentum hyperparameter "β" needs to be tuned, usually in the range of 0.8 to 0.99. Proper tuning of this hyperparameter can have a significant impact on the optimization process.
- **Parallelism**: Momentum-Based GD can be parallelized effectively, making it suitable for distributed computing and GPU acceleration.
- **Improved Convergence**: In many cases, Momentum-Based GD converges faster than standard gradient descent, making it a popular choice for training deep neural networks.

While Momentum-Based GD offers several advantages, it's essential to carefully select the momentum hyperparameter and monitor convergence during training to avoid overshooting the optimal solution. It is often used in conjunction with other techniques like learning rate schedules and adaptive methods to improve optimization stability and performance.

Nesterov Accelerated GD

https://youtu.be/rKG9E6rce1c?si=rclTjRa_yrjSV2eO

Nesterov Accelerated Gradient Descent (NAG), also known as Nesterov Momentum or Nesterov Accelerated Momentum, is an optimization algorithm used in machine learning and deep learning. It's an improvement over standard Momentum-Based Gradient Descent, aiming to enhance convergence speed and stability. NAG was developed by Yurii Nesterov in the late 1980s and is particularly effective for training deep neural networks. Here's how Nesterov Accelerated GD works:

1. **Initialization**:
    - Start by initializing the model's parameters (weights and biases) with random values or using specific initialization techniques.
2. **Introduction of Velocity Term**:
    - Like Momentum-Based GD, NAG introduces a momentum term, usually denoted as "beta" (β), which is a hyperparameter between 0 and 1.
    - The momentum term accumulates a moving average of past gradients, influencing the direction and speed of parameter updates.
3. **Lookahead Update (Lookahead Step)**:
    - Before computing the gradient update for the model parameters, NAG performs a lookahead or preview update. This lookahead step estimates where the parameters would end up if they continued in the current momentum-based direction.
    - To do this, NAG adjusts the model parameters as if they had already moved with the momentum term:

        - **parameter_lookahead = parameter - learning_rate * β * v**
    - Here, "parameter_lookahead" represents the updated parameters as if the model had moved with the momentum.
4. **Gradient Calculation**:
    - With the lookahead parameters in place, NAG calculates the gradient of the loss function using the lookahead parameters rather than the current parameters.
5. **Iterative Updates**:
    - Update the moving average of past gradients:
        - **v = β * v + (1 - β) * gradient_lookahead**
    - Update the model parameters using the moving average of gradients:
        - **parameter = parameter - learning_rate * v**
    - Here, "v" represents the velocity or accumulated gradient, "β" is the momentum hyperparameter, and "learning_rate" is the learning rate hyperparameter.
6. **Repeat**:
    - Continue the iterative updates using mini-batches, accumulating the gradient's moving average, and updating the parameters based on the lookahead step for a specified number of training iterations (epochs).
7. **Convergence**:
    - The training process continues until one or more stopping criteria are met. Common stopping criteria include reaching a maximum number of epochs or achieving a desired level of convergence in the loss.

Key Characteristics and Benefits of Nesterov Accelerated GD:

- **Faster Convergence**: Nesterov Accelerated GD often converges faster than standard Momentum-Based GD by more precisely estimating the direction in which to update the model parameters.

- **Improved Stability**: The lookahead update in NAG helps reduce oscillations and overshooting, making the optimization process more stable.
- **Parallelism**: Nesterov Accelerated GD can be parallelized effectively, making it suitable for distributed computing and GPU acceleration.
- **Hyperparameter Tuning**: The momentum hyperparameter "β" in NAG also requires proper tuning, typically in the range of 0.8 to 0.99.

Nesterov Accelerated GD is a widely used optimization algorithm for training deep neural networks due to its faster convergence properties and improved stability compared to standard momentum-based methods. It is often used in conjunction with other techniques like learning rate schedules and adaptive methods to further enhance optimization performance.

AdaGrad

AdaGrad, short for Adaptive Gradient Descent, is an optimization algorithm used in machine learning and deep learning to adapt the learning rate for each parameter during training. It was developed to address some of the challenges posed by traditional gradient descent methods, such as choosing a fixed learning rate that may not be suitable for all parameters. AdaGrad adapts the learning rate individually for each parameter based on their historical gradients. Here's how AdaGrad works:

1. **Initialization**:
    - Start by initializing the model's parameters (weights and biases) with random values or using specific initialization techniques.
2. **Initialization of Sum of Squared Gradients**:
    - For each parameter in the model, initialize a sum of squared gradients variable. This variable keeps track of the sum of the squares of the gradients encountered for that parameter.
3. **Iterative Updates**:
    - In each training iteration (or batch), compute the gradient of the loss function with respect to the model parameters for the current mini-batch.

- For each parameter (weight or bias), update the sum of squared gradients by adding the squared magnitude of the current gradient:
    - **sum_of_squared_gradients += gradient^2**
- Calculate the effective learning rate for each parameter using the accumulated sum of squared gradients:
    - **effective_learning_rate = learning_rate / sqrt(sum_of_squared_gradients + epsilon)**
    - Here, "learning_rate" is a hyperparameter that controls the global learning rate, and "epsilon" is a small constant (e.g., 1e-7) added to avoid division by zero.
- Update each parameter using the effective learning rate and the gradient:
    - **parameter = parameter - effective_learning_rate * gradient**

4. **Repeat**:
    - Continue the iterative updates using mini-batches, updating the sum of squared gradients and parameters for a specified number of training iterations (epochs).
5. **Convergence**:
    - The training process continues until one or more stopping criteria are met. Common stopping criteria include reaching a maximum number of epochs or achieving a desired level of convergence in the loss.

Key Characteristics and Benefits of AdaGrad:

- **Adaptive Learning Rates**: AdaGrad adapts the learning rates for each parameter individually based on their historical gradients. Parameters that have steep gradients receive smaller effective learning rates, while those with small gradients receive larger effective learning rates.
- **Automatic Scaling**: AdaGrad effectively scales the learning rates for each parameter based on the history of gradients. This reduces the need for manual tuning of learning rates, making it particularly useful for models with a large number of parameters.

- **Effective on Sparse Data**: AdaGrad is effective when dealing with sparse data, as it tends to give smaller learning rates to parameters that are frequently updated and larger learning rates to infrequently updated parameters.

However, there are some limitations to AdaGrad:

- **Accumulation of Squared Gradients**: The accumulation of squared gradients over time can lead to diminishing learning rates, which may slow down convergence in the later stages of training. To address this, other algorithms like RMSProp and Adam have been introduced.
- **Hyperparameter Tuning**: While AdaGrad automates learning rates, it still requires the tuning of the initial learning rate and the epsilon value.
- **Memory Usage**: AdaGrad stores the sum of squared gradients for each parameter, which can be memory-intensive for models with a large number of parameters.

AdaGrad is a useful optimization algorithm, especially in scenarios where manual tuning of learning rates is challenging. However, for certain deep learning tasks, more advanced optimization algorithms like RMSProp and Adam have become popular choices due to their improved handling of certain issues that AdaGrad encounters.

https://youtu.be/GSmW59dM0-o?si=btnCQgfuECFcNNo4

Adam

Adam, short for Adaptive Moment Estimation, is a popular optimization algorithm used in machine learning and deep learning to efficiently and effectively update model parameters during training. Adam combines the ideas of both momentum-based methods and adaptive learning rate methods to provide fast and stable convergence. It was introduced by Diederik P. Kingma and Jimmy Ba in 2014. Here's how the Adam algorithm works:

1. **Initialization**:
    - Start by initializing the model's parameters (weights and biases) with random values or using specific initialization techniques.
2. **Initialization of Moments**:

- For each parameter in the model, initialize two moment variables:
    - The first moment, "m," is an exponentially moving average of past gradients.
    - The second moment, "v," is an exponentially moving average of past squared gradients.
- Initialize two additional hyperparameters:
    - "beta1" (typically close to 1) controls the exponential decay rate for the first moment.
    - "beta2" (typically close to 1) controls the exponential decay rate for the second moment.
    - "epsilon" (a small constant, e.g., 1e-7) is added to prevent division by zero.
- Initialize a time step variable, "t," which keeps track of the number of training iterations.

3. **Iterative Updates**:
    - In each training iteration (or batch), compute the gradient of the loss function with respect to the model parameters for the current mini-batch.
    - Update the first moment "m" and second moment "v" for each parameter using exponential moving averages:
        - **m = beta1 * m + (1 - beta1) * gradient**
        - **v = beta2 * v + (1 - beta2) * (gradient^2)**
    - Correct for bias in the moving averages:
        - **m_hat = m / (1 - beta1^t)**
        - **v_hat = v / (1 - beta2^t)**
    - Calculate the effective learning rate for each parameter using the corrected moving averages and "epsilon":
        - **effective_learning_rate = learning_rate / (sqrt(v_hat) + epsilon)**

- Update each parameter using the effective learning rate and the first moment "m":
    - **parameter = parameter - effective_learning_rate * m_hat**

4. **Repeat**:
    - Continue the iterative updates using mini-batches, updating the moments and parameters for a specified number of training iterations (epochs).
5. **Convergence**:
    - The training process continues until one or more stopping criteria are met. Common stopping criteria include reaching a maximum number of epochs or achieving a desired level of convergence in the loss.

Key Characteristics and Benefits of Adam:

- **Adaptive Learning Rates**: Adam adapts the learning rates for each parameter individually based on the historical gradients and squared gradients. It provides large learning rates for parameters with small updates and smaller learning rates for parameters with large updates.
- **Momentum**: Adam includes a momentum-like term through the first moment "m," which helps accelerate convergence by accumulating past gradients.
- **Bias Correction**: Adam corrects for bias in the moving averages, which is especially important at the beginning of training when the moving averages are biased towards zero.
- **Efficiency**: Adam is computationally efficient and can handle a wide range of machine learning tasks.

Adam is a widely used optimization algorithm for training deep neural networks and other machine learning models. It often converges faster than traditional gradient descent methods and requires minimal hyperparameter tuning. However, the choice of hyperparameters such as "beta1," "beta2," "epsilon," and the initial learning rate can impact performance, and these should be tuned based on the specific task and dataset.

https://youtu.be/N5AynalXD9g?si=znDDzDuZfUJXDRgY

RMSProp

Overview of Overfitting

Overfitting is a common challenge in deep learning and machine learning in general. It occurs when a model learns to perform exceptionally well on the training data but struggles to generalize its performance to unseen or new data. In essence, the model becomes too specialized in capturing the noise and nuances in the training data, which can lead to poor performance on real-world, out-of-sample data. Here's an overview of overfitting in deep learning:

1. **Training vs. Test Performance**:
    - During the training process, a deep learning model is exposed to a labeled dataset to learn patterns and relationships between inputs and targets. The goal is to minimize a loss function by adjusting model parameters (weights and biases).
    - Overfitting typically becomes evident when there is a significant difference between the model's performance on the training data and its performance on a separate test dataset (or validation dataset) that it has not seen before.
2. **Overly Complex Models**:
    - Overfitting often occurs when the model is overly complex relative to the size of the training dataset.
    - Deep neural networks with many layers and a large number of parameters are particularly prone to overfitting when training data is limited.
3. **Effects of Overfitting**:
    - Models that have overfit the training data may exhibit the following symptoms:
        - High accuracy or low loss on the training data.
        - Poor generalization performance, with significantly lower accuracy or higher loss on the test data or new, unseen data.

- High sensitivity to small variations in the training data.
- Instability and inconsistent performance.

4. **Causes of Overfitting**:
    - Overfitting can be caused by several factors, including:
        - Having too few training examples, making it difficult for the model to learn meaningful patterns.
        - Using an excessively complex model architecture with many layers or parameters.
        - Training for too many epochs, allowing the model to continue learning noise in the training data.
        - Lack of regularization techniques (e.g., dropout, weight decay) to prevent overfitting.
5. **Preventing Overfitting**:
    - Several strategies can help prevent or mitigate overfitting:
        - Increasing the size of the training dataset, when possible, to provide the model with more diverse examples.
        - Simplifying the model architecture by reducing the number of layers or parameters.
        - Applying regularization techniques to constrain the model's capacity and encourage generalization.
        - Using techniques like cross-validation to assess model performance more accurately.
        - Early stopping: Monitoring the model's performance on a validation set and stopping training when performance starts to degrade.
6. **Validation and Test Sets**:
    - To diagnose and combat overfitting, it's crucial to have separate datasets for training, validation (used for hyperparameter tuning and model selection), and testing (used to assess final model performance).

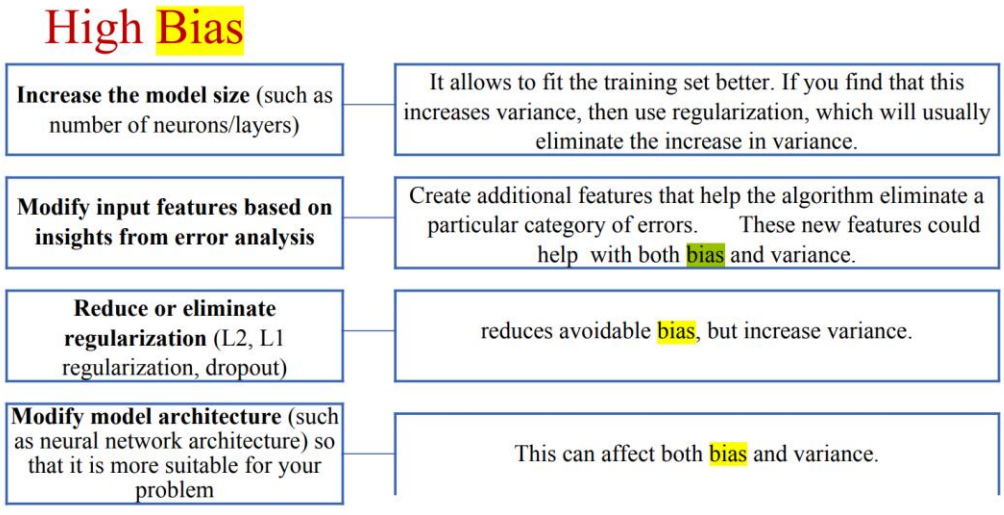
- The test set should be held out and not used during model development to provide an unbiased evaluation of generalization.

7. **Balancing Act**:
    - Finding the right balance between a model's complexity and its ability to generalize is a fundamental challenge in machine learning.
    - Deep learning practitioners often iterate on model architectures, regularization techniques, and hyperparameters to strike this balance.
8. **Monitoring Loss Curves**:
    - Monitoring the training and validation loss curves during training can provide insights into whether overfitting is occurring. A rising validation loss, while the training loss continues to decrease, is a common indicator of overfitting.

Overfitting is a pervasive issue in deep learning and machine learning. Addressing it effectively involves a combination of data management, model design, regularization, and monitoring techniques to ensure that a trained model generalizes well to unseen data.

Bias

The term "bias" refers to a systematic error or deviation from the true value or reality in a measurement, estimation, or prediction. Bias can arise from various sources and can impact the accuracy and fairness of data, models, and decisions.



In deep learning, various types of biases can affect the model's training, predictions, and overall performance. These biases can arise from different sources and have distinct implications. Here are some common types of biases in deep learning:

1. **Data Bias**:
    - **Definition**: Data bias, also known as dataset bias, occurs when the training data is not representative of the real-world population or contains skewed or unbalanced samples.
    - **Example**: In a facial recognition system trained on predominantly light-skinned individuals, the model may perform poorly on darker-skinned faces due to data bias.
2. **Label Bias**:
    - **Definition**: Label bias arises when the training data contains incorrect or biased labels, leading the model to learn incorrect associations.
    - **Example**: If an image dataset contains mislabeled cats as dogs, a deep learning model trained on this data may struggle to correctly identify cats.
3. **Algorithm Bias**:
    - **Definition**: Algorithm bias refers to inherent biases within the design or architecture of a deep learning algorithm that can influence its predictions or decisions.
    - **Example**: Some machine learning algorithms may have a built-in bias toward certain classes or may not handle imbalanced data well.
4. **Sampling Bias**:
    - **Definition**: Sampling bias occurs when the process used to collect or sample data systematically favors certain types of data points or regions of the data space.
    - **Example**: In autonomous vehicle testing, if data collection primarily occurs in a specific city with mild weather conditions, the

model may not perform well in other environments due to sampling bias.

5. **Cultural Bias**:
    - **Definition**: Cultural bias arises when the training data reflects cultural or regional biases, leading the model to make assumptions based on these biases.
    - **Example**: Language models trained on text from the internet may inadvertently learn and perpetuate cultural biases present in the data, such as gender or racial stereotypes.
6. **Measurement Bias**:
    - **Definition**: Measurement bias occurs when the data collection process introduces systematic errors or inaccuracies.
    - **Example**: In healthcare, if medical sensors used for data collection are biased toward one gender, it can result in measurement bias in health-related deep learning models.
7. **Response Bias**:
    - **Definition**: Response bias refers to a situation where the way data is collected influences the responses provided by individuals, leading to inaccurate or biased data.
    - **Example**: In online surveys, users may be hesitant to provide certain information, leading to response bias in the collected data.
8. **Temporal Bias**:
    - **Definition**: Temporal bias occurs when the training data does not adequately represent changes or trends over time.
    - **Example**: A deep learning model trained on historical stock market data may not perform well in predicting future market behavior due to temporal bias.
9. **Interaction Bias**:
    - **Definition**: Interaction bias happens when the way users or agents interact with a system introduces bias into the collected data.

- **Example**: In recommender systems, if users receive recommendations that align with their existing preferences, interaction bias can reinforce existing biases and limit exposure to diverse content.

10. **User Bias**:
    - **Definition**: User bias can occur when user behavior or preferences introduce bias into the data, such as user-generated content or interactions.
    - **Example**: On social media platforms, the content that users engage with or share can introduce user bias into recommendation algorithms.

Mitigating biases in deep learning involves careful data collection, preprocessing, model design, and evaluation. Techniques such as bias-aware algorithms, diverse and representative datasets, and fairness audits can help address and reduce these biases, ensuring that deep learning models make predictions and decisions that are fair, accurate, and unbiased.

Bias Variance Tradeoff

https://www.geeksforgeeks.org/ml-bias-variance-trade-off/

https://towardsdatascience.com/understanding-the-bias-variance-tradeoff-165e6942b229

Regularization

https://youtu.be/4xRonrhtkzc?si=rCXMpCE04vHBqCNK

https://www.analyticsvidhya.com/blog/2018/04/fundamentals-deep-learning-regularization-techniques/#h-what-is-regularization

https://www.codingninjas.com/studio/library/parameter-sharing-and-tying

https://www.geeksforgeeks.org/parameter-sharing-and-typing-in-machine-learning/

https://www.linkedin.com/advice/3/how-do-you-compare-weight-decay-other#:~:text=What%20is%20weight%20decay%3F,them%20from%20growing%20too%20large.

https://towardsdatascience.com/this-thing-called-weight-decay-a7cd4bcfccab

https://www.analyticsvidhya.com/blog/2021/03/introduction-to-batch-normalization/

https://youtu.be/agGUR06jM_g?si=Yf_SVRbDoSwsIQe-