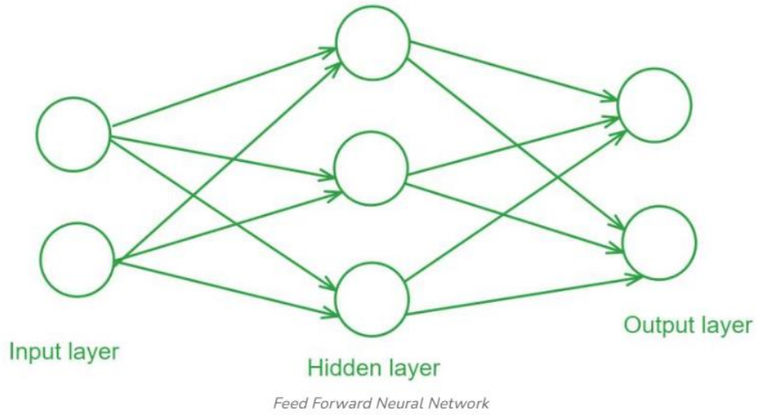
Module 3 Deep learning

Introduction to Deep learning

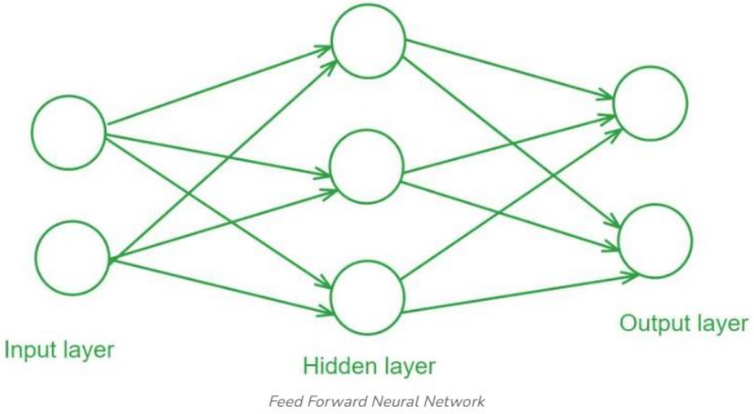
Deep Feed Forward Networks

- Deep Feed Forward Networks (FFNs) are a type of artificial neural network that maps input data to output data through a series of layers.
- These layers are organized sequentially, with the output of each layer serving as the input to the next.
- The network consists of an input layer, one or more hidden layers, and an output layer. Information flows in one direction—from input to output—hence the name “feedforward.”
- The overall length of the chain gives the depth of the model. It is from this terminology that the name “deep learning” arises



Feed Forward Neural Network

Structure of a Feedforward Neural Network



Feed Forward Neural Network

1.Input Layer: The input layer consists of neurons that receive the input data. Each neuron in the input layer represents a feature of the input data.

2.Hidden Layers: One or more hidden layers are placed between the input and output layers. These layers are responsible for learning the complex patterns in the data. Each neuron in a hidden layer applies a weighted sum of inputs followed by a non-linear activation function. Because the training data does not show the desired output for each of these layers, these layers are called hidden layers.

3.Output Layer: The output layer provides the final output of the network. The number of neurons in this layer corresponds to the number of classes in a classification problem or the number of outputs in a regression problem.
Each connection between neurons in these layers has an associated weight that is adjusted during the training process to minimize the error in predictions

Gradient-Based Learning

- Gradient-based learning refers to a family of optimization techniques used to train machine learning models, particularly those based on neural networks.
- It relies on the concept of gradients, which measure how much a model's output changes in response to small changes in its parameters.
- The most commonly used gradient-based learning algorithm is **gradient descent**, which iteratively adjusts model parameters to minimize a loss function (a measure of error).

Key Concepts:

1.Gradient: The gradient of a function at a point represents the slope or direction of the steepest ascent. In machine learning, we compute the gradient of the loss function with respect to the model parameters.
2.Gradient Descent: This is an iterative optimization algorithm used to minimize the loss function by updating the model parameters in the direction of the negative gradient. The steps can be described as:

1. Compute the gradient of the loss function with respect to the parameters.
2. Update the parameters by moving them in the direction of the negative gradient.

3.Learning Rate: This is a hyperparameter that determines the size of the steps taken during gradient descent
•

Training Deep Models:

Training deep learning models involves the following steps:

1.Data Preparation: Collect and preprocess data, then split it into training, validation, and test sets.
2.Model Architecture: Choose or design a neural network architecture suited for the task (e.g., CNNs, RNNs, or Transformers).
3.Loss Function: Define an objective to minimize, such as cross-entropy for classification or MSE for regression.
4.Optimization: Optimization algorithms in deep learning are used to minimize the loss function and update the model's weights.Use optimizers like Adam or SGD to update model weights through backpropagation. Tune learning rates and other hyperparameters.
5.Regularization: Regularization algorithms help prevent overfitting in deep learning models by controlling model complexity.Apply techniques like dropout, L2 regularization, or batch normalization to prevent overfitting.
6.Training: Train the model over multiple epochs, monitoring performance on the validation set.
7.Evaluation: Assess the model on the test set using metrics like accuracy, F1-score, or MSE.
8.Hardware: Use GPUs/TPUs to speed up training for large datasets.

Optimization Techniques

1. ## Gradient Descent (GD)

   A fundamental optimization technique where we update model parameters in the direction of the negative gradient to minimize a loss function.

2. ## GD with Momentum

   An improvement over GD that introduces momentum to accelerate convergence by incorporating information from previous updates.

3. ## Nesterov Accelerated GD

   A more sophisticated version of GD with momentum that takes a "lookahead" step before calculating the gradient, further speeding up convergence.

4. ## Stochastic GD

   A technique that utilizes randomly selected subsets of data to compute gradients, enabling faster updates and avoiding getting stuck in local minima.

5. ## AdaGrad and RMSProp

   Adaptive learning rate methods that adjust the learning rate for each parameter based on its historical gradients, allowing for faster convergence in sparse data.

Gradient Descent (GD)

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

Concept

Gradient Descent (GD) is an iterative optimization algorithm that aims to find the minimum of a function by repeatedly updating parameters in the direction of the negative gradient.

Process

GD starts with an initial guess for the parameters and then repeatedly updates them by moving in the direction opposite to the gradient of the loss function. This process continues until a minimum point is found or a predefined stopping criterion is met.

Advantages

GD is a relatively simple algorithm to understand and implement. It often converges to a local minimum or a saddle point.

Disadvantages

- Slow convergence
- May stuck in local Minima
- Sensitivity to learning rate

How the Gradient Descent Algorithm Works

1. **Compute Gradient**: The gradient (or slope) of the loss function with respect to the model's parameters is calculated. It shows the direction of the steepest increase in loss.
2. **Update Weights**: The model's weights are updated in the opposite direction of the gradient to minimize the loss.
    - **Formula**:
      $\theta = \theta - \eta \cdot \nabla L(\theta)$
      where θ is the model's parameters, η is the learning rate, and $\nabla L(\theta)$ is the gradient of the loss.
3. **Learning Rate**: Controls the size of each step. Too high a value can cause overshooting, while too low can result in slow convergence.

GD with Momentum

1 Concept

GD with momentum adds a term to the parameter update that incorporates the previous update direction. This helps accelerate convergence and avoid getting stuck in local minima.

2 Impact

The **momentum term** introduces a velocity term that accumulates the past gradients, helping the algorithm move faster in the relevant direction and reduce oscillations.

Advantages

- **Smoother Updates**: It dampens oscillations by averaging gradients over time.
- **Faster Convergence**: Especially effective in speeding up learning in regions with shallow gradients (long flat areas).
- **Less Sensitivity** to learning rate changes.

3 Effectiveness

GD with momentum effectively accelerates convergence, particularly in cases where the loss function has many valleys and local minima.

Key Steps:

1. **Compute Gradient**: As in normal gradient descent, compute the gradient of the loss function with respect to the parameters.
2. **Momentum Update**: Maintain a "velocity" vector that accumulates past gradients.
    - Velocity Update:
      $v_t = \beta v_{t-1} + (1 - \beta)\nabla L(\theta_t)$
      where v_t is the velocity, β is the momentum term (typically 0.9), and $\nabla L(\theta_t)$ is the current gradient.
3. **Weight Update**: Adjust the model's parameters by applying the velocity to the current parameters.
    - Weight Update:
      $\theta_t = \theta_{t-1} - \eta v_t$

Nesterov Accelerated GD

1

Concept

Nesterov Accelerated GD (NAG) is an extension of GD with momentum that incorporates a "lookahead" step before calculating the gradient. This further accelerates convergence and can improve performance compared to traditional GD with momentum.

2

Process

NAG makes a prediction about the next step and adjusts the gradient accordingly. It gives a more accurate update and can lead to faster convergence compared to standard momentum-based methods.

3

Advantages

NAG has shown to be more efficient in reaching the minimum point, particularly in cases where the loss function is highly non-convex, faster convergence and smooth updates

Steps of NAG:

1. **Predictive Step**: Make a temporary "look-ahead" step using the current velocity:
   - $\theta' = \theta_t - \beta v_{t-1}$
2. **Compute Gradient**: Calculate the gradient at this predicted position:
   - $\nabla L(\theta')$
3. **Velocity Update**: Update the velocity as in momentum-based gradient descent:
   - $v_t = \beta v_{t-1} + \eta \nabla L(\theta')$
4. **Update Parameters**: Use the new velocity to update the parameters:
   - $\theta_{t+1} = \theta_t - v_t$

Stochastic Gradient Descent

Key Steps:

1. **Random Sampling**: Instead of using the entire dataset, select one random data point (or a small batch) from the training set.
2. **Compute Gradient**: Calculate the gradient of the loss function with respect to the parameters for that single data point or mini-batch.
3. **Update Weights**: Adjust the model's parameters using the gradient and a learning rate:
   - $\theta_t = \theta_{t-1} - \eta \cdot \nabla L(\theta_t, x_i)$
   - Where θ_t is the parameter at iteration t, η is the learning rate, and $\nabla L(\theta_t, x_i)$ is the gradient computed for data point x_i.

Concept

Stochastic Gradient Descent (SGD) is a variant of GD that uses a randomly selected subset of data, called a mini-batch, to compute gradients for each update.

Benefits

SGD offers several advantages over traditional GD. It's faster and more efficient, particularly for large datasets. It can also avoid getting stuck in local minima.

Considerations

SGD can be more noisy than GD, making convergence less smooth. A proper learning rate schedule is crucial for achieving optimal results. The randomness can lead to fluctuations and may not lead directly to the global minimum.

AdaGrad

- **Adagrad (Adaptive Gradient Algorithm)** is an optimization algorithm that adapts the learning rate for each parameter individually based on how frequently it is updated during training.
- Parameters that receive frequent updates get smaller learning rates, while those that are updated less frequently get larger learning rates.
- This makes Adagrad particularly useful for dealing with sparse data.

Key Features:

1.Per-Parameter Learning Rate: Adagrad assigns different learning rates to each parameter, allowing for better handling of features with varying frequencies.
2.Learning Rate Decay: As training progresses, the learning rates for frequently updated parameters decrease over time, which helps avoid large updates and allows better convergence.

Advantages:

•**No Manual Learning Rate Tuning**: Adapting the learning rate for each parameter helps with automatic learning rate control.
•**Works Well for Sparse Data**: Features that appear less frequently (e.g., in NLP tasks) get larger learning rates, helping to learn faster for these features.

Disadvantages:
•Learning Rate Shrinking:
•No Resetting Mechanism for learning rate

Adagrad Algorithm Steps:

1. For each parameter θ_i, initialize a running total of the squared gradients, denoted $G_{t,i} = 0$.
2. For each iteration, compute the gradient $g_{t,i}$ for each parameter θ_i.
3. Update the cumulative sum of squared gradients:

$$G_{t,i} = G_{t-1,i} + g_{t,i}^2$$

4. Update the parameters using the following formula:

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,i} + \epsilon}} g_{t,i}$$

- η is the initial learning rate.
- ϵ is a small value (e.g., 10^{-8}) added to prevent division by zero.

RMSProp

- **RMSProp (Root Mean Square Propagation)** is an optimization algorithm designed to address the diminishing learning rate issue encountered in Adagrad.
- It adapts the learning rate for each parameter by keeping a moving average of the squared gradients, preventing the learning rate from decaying too quickly.

Key Features:

1.Moving Average of Squared Gradients: RMSProp uses an exponentially decaying average of past squared gradients to adjust the learning rate for each parameter.
2.Effective for Non-Stationary Problems: It's especially good for tasks where gradients change rapidly, such as deep learning and recurrent neural networks (RNNs).

Advantages:

•**Prevents Learning Rate Decay**: Unlike Adagrad, RMSProp prevents the learning rate from shrinking too much by using a moving average of gradients instead of accumulating all past gradients.
•**Handles Non-Stationary Problems Well**: The algorithm adapts to changing gradients effectively, making it useful for tasks like RNNs where gradients vary significantly over time.
•**Efficient for Deep Networks**: RMSProp is commonly used in deep learning tasks due to its ability to deal with noisy or fast-changing gradients.

Disadvantages:
•Sensitive to Hyperparameters: The learning rate and decay factor need careful tuning for optimal performance.

RMSProp Algorithm Steps:

1. At each iteration, the squared gradient is computed for each parameter.
2. The EMA of the squared gradients is updated using a decay factor (denoted by `β`, typically around 0.9).

$$E[g^2]_t = \beta E[g^2]_{t-1} + (1 - \beta) g_t^2$$

   Here, $E[g^2]_t$ is the moving average of the squared gradients, and g_t is the gradient at step t.

3. The parameter updates are then scaled by the square root of this EMA:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

   - θ_t represents the model parameters at time t.
   - η is the learning rate.
   - ϵ is a small value (e.g., 10^{-8}) added to avoid division by zero.

Difference between AdaGrad and RMSProp

Algorithm	Concept	Advantages
AdaGrad	Adaptively scales the learning rate for each parameter based on the sum of squared gradients, giving more weight to parameters with smaller gradients.	Effectively handles sparse datasets and avoids getting stuck in shallow local minima.
RMSProp	Similar to AdaGrad, RMSProp also adaptively scales the learning rate based on a moving average of squared gradients. However, it uses an exponentially decaying average, which helps prevent the learning rate from becoming too small.	Offers a more stable and efficient convergence compared to AdaGrad, especially in scenarios where the gradients are highly variable.

Adam Optimization

- **Adam (Adaptive Moment Estimation)** is a popular optimization algorithm in deep learning that combines the advantages of two other extensions of stochastic gradient descent: **AdaGrad** and **RMSProp**.
- It adapts the learning rate for each parameter based on both the first moment (mean) and the second moment (uncentered variance) of the gradients, providing a powerful and efficient optimization method.

Key Features:

1.Adaptive Learning Rates: Adam computes individual adaptive learning rates for different parameters from estimates of first and second moments of the gradients.

2.Bias Correction: Adam incorporates bias correction for the moving averages of the gradients to ensure they are accurate, especially in the early stages of training.

Steps:

1.Initialize Parameters:
2.Compute Gradients:
3.Update biased first and second moment estimates:
4.Bias Correction:
5.Update Parameters:

Advantages:

•Efficiency:
•Adaptability:
•Robustness:

Disadvantages:

•Hyperparameter Sensitivity:
•Potential Overfitting:

Advantages of Adam Optimization

Speed

Adam is known for its fast convergence rate, often outperforming other optimization algorithms.

Stability

Adam is relatively stable and less prone to oscillations or getting stuck in local minima, even with noisy gradients.

Adaptability

Adam adapts the learning rate for each parameter individually, allowing it to handle diverse data distributions effectively.

Efficiency

Adam is computationally efficient, requiring only a few parameters to be updated during optimization.

Regularization Techniques

- Regularization is a set of methods for reducing overfitting in machine learning models.
- Regularization techniques work by adding a penalty term to the loss function during training.
- This penalty discourages the model from becoming too complex or having large parameter values

Lasso regression or L1 regression

•Lasso regression (Least Absolute Shrinkage and Selection Operator) is a type of linear regression that performs both variable selection and regularization to enhance prediction accuracy and interpretability.
•It works by introducing a penalty equal to the absolute value of the magnitude of the coefficients.
•This regularization term causes some coefficients to shrink towards zero, effectively eliminating less important features from the model.
•This is useful in
    •Feature selection in high-dimensional datasets.
    •Models where interpretability is crucial, and only the most important variables are desired.
Lasso is particularly useful when there are many features, and you want to remove irrelevant ones.

Key features of Lasso regression:

1. **Regularization**: Lasso adds a penalty term $\lambda \sum_{i=1}^{n} |\beta_i|$ to the cost function, where λ controls the amount of shrinkage.
2. **Feature selection**: The Lasso penalty can force some feature coefficients to be exactly zero, which simplifies the model by excluding irrelevant variables.
3. **Bias-variance trade-off**: As the penalty increases, it introduces more bias but reduces variance, which helps prevent overfitting.

Lasso regression cost function:

$$\text{Cost function} = \text{RSS} + \lambda \sum_{i=1}^{n} |\beta_i|$$

Where:

- RSS = Residual Sum of Squares (the typical cost function for linear regression)
- λ = regularization parameter
- β_i = coefficients

Lasso Regression (L1)

- LASSO stands for(Least Absolute Shrinkage and Selection Operator) regression.
- L1 is a popular regularization technique in machine learning,
- It offers a powerful approach to mitigate overfitting and perform feature selection in regression modeling.
- This regularization technique is particularly beneficial when dealing with high-dimensional datasets
- Traditional regression models may struggle when dealing with high-dimensional datasets containing many irrelevant features,
- leading to poor predictive performance and model interpretability.
- It helps simplify the model and improve interpretability by focusing on the most influential predictors.
- Lasso Regression adds the "absolute value of magnitude" of the coefficient as a penalty term to the loss function(L).
- Lasso regression also helps us achieve feature selection by penalizing the weights to approximately equal to zero if that feature does not serve any purpose in the model.

$$\text{Cost} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 + \lambda \sum_{i=1}^{m} |w_i|$$

where,

- ***m*** *– Number of Features*
- ***n*** *– Number of Examples*
- ***y_i*** *– Actual Target Value*
- ***y_i(hat)*** *– Predicted Target Value*

L2 L2 regularization or Ridge regression

L2 regularization, also known as **Ridge regularization**, is a technique used in machine learning and statistical modeling to prevent overfitting by penalizing large coefficients in a model.
It works by adding a regularization term to the cost function that is proportional to the sum of the squared values of the model parameters (weights or coefficients).

- This adds the "*squared magnitude*" of the coefficient as a penalty term to the loss function(L). It addresses multicollinearity (correlation between independent variables) and overfitting by adding a regularization term to the cost function
- The core principle is to balance the trade-off between bias and variance.
- By adding a penalty term proportional to the square of the coefficients to the loss function, Ridge regression effectively shrinks the coefficient estimates towards zero while still allowing them to be non-zero.
- This regularization technique is particularly beneficial when dealing with multicollinear datasets, where it helps stabilize the model by reducing the sensitivity of coefficient estimates to small changes in the training data.
- As λ increases, the coefficients shrink more aggressively, and the model becomes simpler.

$$\text{Cost function (with L2 regularization)} = \text{RSS} + \lambda \sum_{i=1}^{n} \beta_i^2$$

$$\text{Cost} = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 + \lambda \sum_{i=1}^{m} w_i^2$$

Where:

- RSS = Residual Sum of Squares (error between predicted and actual values)
- β_i = coefficients of the model
- λ = regularization parameter (determines the strength of regularization)

The term $\lambda \sum_{i=1}^{n} \beta_i^2$ is the L2 penalty, where each coefficient β_i is squared, so larger coefficients are penalized more heavily.

Effects of L2 Regularization:

•**Prevents overfitting**: By shrinking the magnitude of coefficients, L2 regularization helps the model generalize better to new data, especially when dealing with highly collinear or high-dimensional datasets.

•**Reduces variance**: L2 regularization controls the variance of the model by discouraging overly complex models that fit too closely to the training data.

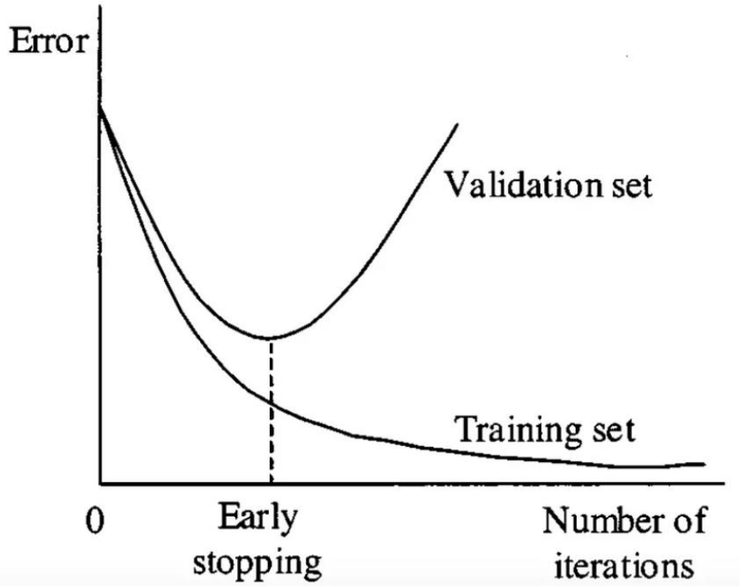
•**Retains all features**: Unlike Lasso (L1 regularization), which can set some coefficients to zero and effectively remove features, L2 regularization shrinks all coefficients but does not eliminate any features.

Regularization Type	Advantages	Disadvantages
L1	1. Performs feature selection by shrinking less important feature weights to zero.	1. Not effective for datasets with many important features.
	2. Can be used for high-dimensional datasets with many irrelevant features.	2. The solution may not be unique, which can lead to instability in the model.
L2	1. Provides a smooth solution and improves the generalization performance of the model.	1. May not perform well for datasets with many irrelevant features.
	2. Can handle datasets with many important features.	2. Does not perform feature selection, and all features are included in the model with non-zero weights.

L1 Regularisation	L2 Regularisation
Sum of absolute value of weights	Sum of square of weights
Sparse solution is the outcome	Non-sparse (more segregated) solution
Multiple solutions are possible	Only one solution
Built-in feature selection in the penalty term	No specific feature selection mechanism
Robust to outliers	Not robots to outliers due to square term
Used in datasets with sparse features	Used in dataset with complex features

Early Stopping

- Early stopping is a form of regularization used to prevent overfitting in machine learning and deep learning models.
- It involves stopping the training process before the model starts to overfit.
- The idea is to monitor the model's performance on a validation set during the training process and stop training when the performance starts to degrade, which is an indication that the model is beginning to overfit the training data.



How Early Stopping Works

1. **Validation Set:** During training, a portion of the training data is set aside as a validation set. This set is used to evaluate the model's performance at each epoch (iteration over the entire dataset).
2. **Performance Monitoring:** The model's performance on the validation set is monitored at each epoch. Common metrics used for this purpose include accuracy, loss, or any other metric relevant to the problem at hand.
3. **Early Stopping Criterion:** If the performance on the validation set starts to degrade (e.g., the loss increases or the accuracy decreases), it's an indication that the model is beginning to overfit the training data. At this point, early stopping is triggered, and the training process is halted.
4. **Model Selection:** Since the training is stopped before overfitting occurs, the model at the point of early stopping is typically the best model, as it has not yet learned the noise in the training data

Benefits of Early Stopping

Early stopping offers several benefits in deep learning:
•Regularization: Early stopping acts as a regularization technique by preventing the model from overfitting to the training data. It encourages the model to find a balance between fitting the training data and generalizing to unseen examples.

•Computational Efficiency: By stopping the training process early, we can save computational resources and time. Instead of training the model for an excessive number of epochs, early stopping allows us to find the optimal stopping point efficiently.

•Automated Tuning: Early stopping automates the process of determining the optimal number of training epochs. It eliminates the need for manual intervention and helps find the sweet spot for stopping the training.

•Robustness: Early stopping makes the model more robust to variations in the training data and hyperparameters. It reduces the model's sensitivity to noise and outliers in the training set

Considerations and Limitations

While early stopping is a powerful technique, there are a few considerations and limitations to keep in mind:
•Validation Set Size: The effectiveness of early stopping depends on the quality and size of the validation set. If the validation set is too small or not representative of the true data distribution, early stopping may not provide accurate guidance.

•Patience Setting: The choice of the patience parameter can impact the performance of early stopping. Setting the patience too low may lead to premature stopping, while setting it too high may result in unnecessary training. Experimenting with different patience values can help find the optimal setting.

•Noisy Validation Loss: In some cases, the validation loss may exhibit noisy or fluctuating behavior, making it challenging to determine the optimal stopping point. Techniques such as smoothing the validation loss curve or using a moving average can help mitigate this issue.

•Generalization Gap: Early stopping relies on the assumption that the validation set is representative of the true data distribution. If there is a significant difference between the validation set and the actual test set, the model's performance may not generalize well to the test set.

Dataset Augmentation

- Data augmentation is a technique used to artificially increase the size of a training dataset by applying various transformations to the existing data.
- This technique is commonly used in machine learning and deep learning tasks, especially in computer vision, to improve the generalization and robustness of the trained models.

Why Use Data Augmentation?

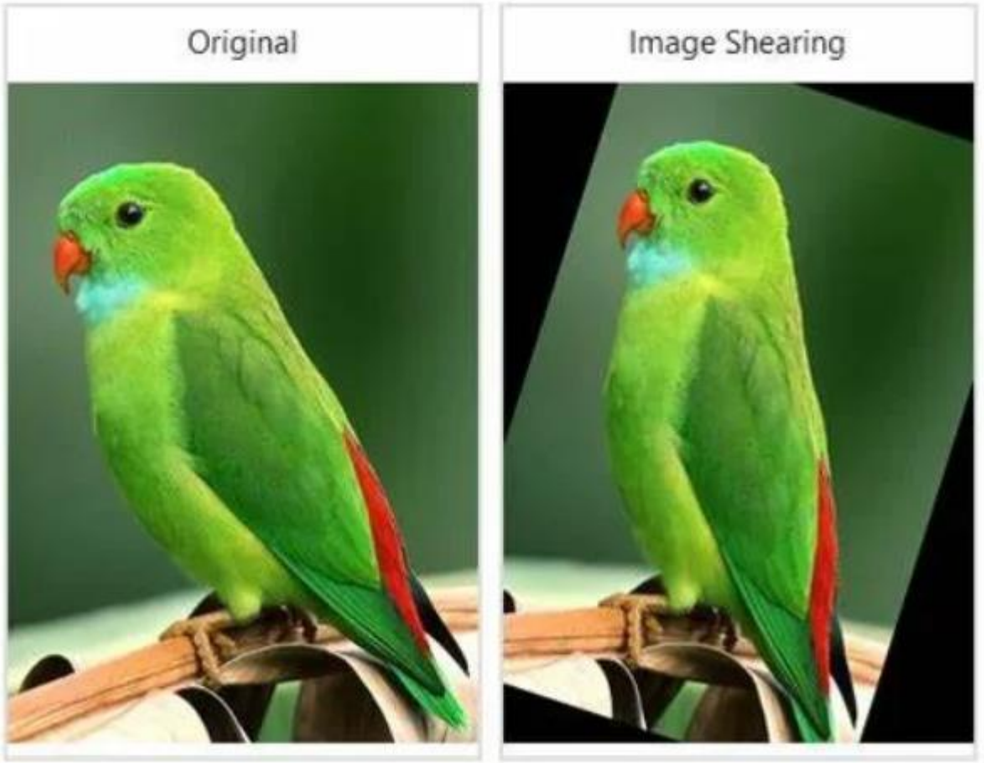
1.Increased Dataset Size: By creating new samples from the existing data, data augmentation effectively increases the size of the dataset, which can lead to better model performance.

2.Regularization: Data augmentation introduces additional variations in the data, which can help prevent overfitting by providing the model with a more diverse set of examples.

3.Improved Generalization: By exposing the model to a wider range of variations in the data, data augmentation helps the model generalize better to unseen examples.

Common Data Augmentation Techniques

1.Rotation: Rotate the image by a certain angle (e.g., 90 degrees, 180 degrees).
2.Translation: Shift the image horizontally or vertically by a certain distance.
3.Scaling: Enlarge or shrink the image by a certain factor.
4.Flipping: Flip the image horizontally or vertically.
5.Shearing: Skew the image along the x or y-axis.
6.Zooming: Zoom in or out of the image.
7.Brightness Adjustment: Increase or decrease the brightness of the image.
8.Contrast Adjustment: Increase or decrease the contrast of the image.
9.Noise Addition: Add random noise to the image.



Advanced Regularization Techniques

Parameter Sharing

Sharing the same parameters across multiple layers or parts of the network, often used in convolutional neural networks to extract features from different regions of an image.

Noise Injection

Adding noise to the input or hidden layers to make the model more robust to noise and variations in data, reducing overfitting.

Ensemble Methods

Combining multiple models trained on different subsets of data or with different hyperparameters to achieve a better prediction performance and improve generalization.

Dropout

A technique that randomly drops out units (neurons) during training, preventing co-adaptation and encouraging the model to learn more robust features.

Parameter Sharing & Tying

- Deep learning models are known for their ability to learn complex relationships between inputs and outputs. However, as the number of parameters in a model grows, so does the risk of overfitting and the computational cost of training. One technique to reduce the number of parameters and improve the generalization of a model is parameter sharing and tying.
- Parameter sharing refers to the idea of using the same set of parameters for multiple parts of a model. For example, in convolutional neural networks (CNNs), the same set of weights is used to convolve over different patches of an image. By doing so, the model can learn features that are translation invariant, meaning they can recognize patterns regardless of their position in the input.
- Parameter tying, on the other hand, refers to the practice of constraining different parts of a model to share the same parameter values. This can be useful in cases where we want to encourage certain properties of the model, such as symmetry or sparsity. For example, in autoencoders, the encoder and decoder can share the same set of weights, leading to a more symmetric architecture and reduced number of parameters.
- Parameter sharing and tying can be applied in different ways across different types of models. In CNNs, for example, parameter sharing is often used in the convolutional layers to learn local features, while parameter tying can be used in the fully connected layers to reduce the number of parameters. In recurrent neural networks (RNNs), parameter sharing can be used to learn shared representations across different time steps, while parameter tying can be used to enforce certain properties of the model, such as weight symmetry.

- One of the main benefits of parameter sharing and tying is the reduction in the number of parameters required to train a model. This can lead to faster training times, reduced memory requirements, and better generalization performance, especially in cases where the data is limited.
- However, parameter sharing and tying can also have some drawbacks. For example, tying parameters can lead to a loss of expressiveness and restrict the model's ability to learn complex relationships. In addition, parameter sharing can introduce dependencies between different parts of the model, leading to potential performance issues if the shared parameters are not carefully chosen.

Injecting noise at input

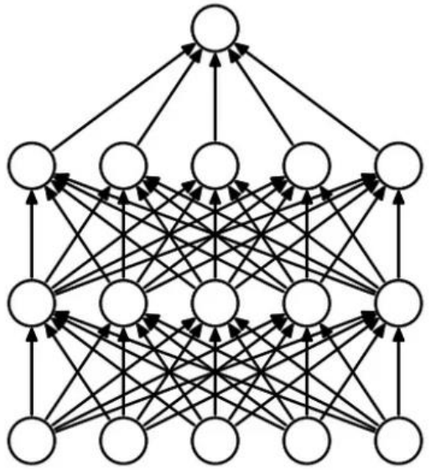
- Injecting noise at the input is a common technique used in machine learning to improve the robustness and generalization of models.
- By adding noise to the input data, the model is forced to learn the underlying patterns in the data rather than simply memorizing specific instances.
- This helps to prevent overfitting and improves the model's ability to generalize to new, unseen data.
- By exposing the model to different types of noise, it can learn to recognize patterns in the input data that are more invariant to variations in the data.
- There are different types of noise that can be injected at the input of a deep learning model.
- Each type of noise has its advantages and disadvantages, and the choice of which type to use depends on the specific problem and the characteristics of the input data.

Gaussian noise:

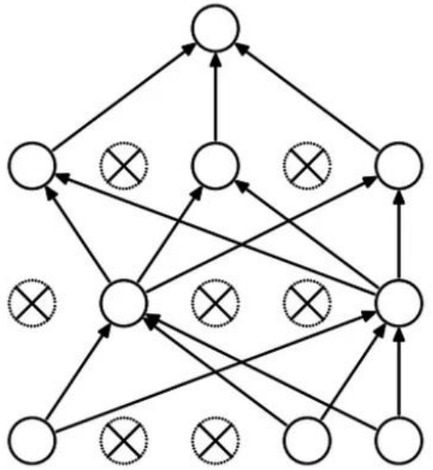
- It is a type of noise that is commonly used in deep learning to inject random variations into the input data.
- It is also known as normal noise, and it is generated by adding random values sampled from a Gaussian distribution to the input data. This distribution, also known as the normal distribution, is a probability distribution that is widely used in statistics. It is a continuous probability distribution that is characterized by two parameters: the mean and the standard deviation.
- The mean represents the center of the distribution, while the standard deviation represents the spread of the distribution. And this noise can be added to the input data in several ways.
- One common approach is to add Gaussian noise to the input images by adding random values to each pixel in the image.
- Another way to use Gaussian noise is to add it to the weights of the model during training. This technique is known as weight decay or L2 regularization, and it involves adding a penalty term to the loss function of the model that is proportional to the square of the weights.
- This helps prevent the weights from becoming too large and encourages the model to learn more robust and generalizable features.

Dropout:

- Dropout is a regularization technique that is commonly used in deep learning to prevent overfitting.
- It involves randomly dropping out (setting to zero) a certain percentage of neurons in a neural network during training.
- Dropout is a regularization technique that is commonly used in deep learning to prevent overfitting.
- It involves randomly dropping out (setting to zero) a certain percentage of neurons in a neural network during training.
- During training, dropout is applied to each layer of the network with a certain probability (typically around 0.5).
- This means that each neuron in the layer has a 50% chance of being dropped out during each training iteration.
- At test time, no neurons are dropped out, and the full network is used for making predictions.
- The dropout technique can be viewed as a form of ensemble learning, where multiple neural networks with different architectures are trained on the same data. In this case, each iteration of the training process corresponds to a different subset of the neurons in the network being activated.



(a) Standard Neural Net



(b) After applying dropout.

Salt-and-pepper noise:

- Salt-and-pepper noise is a type of noise that can occur in digital images or signals.
- It is also known as impulse noise, as it appears as bright and dark pixels randomly scattered throughout the image.
- It can occur due to various factors, such as transmission errors, sensor malfunctions, or data corruption.
- It is called salt-and-pepper noise because it looks like someone sprinkled salt and pepper on the image. In the context of image processing and deep learning, salt-and-pepper noise is often used as a benchmark for evaluating the robustness of image denoising algorithms.
- The goal is to remove the noise from the image while preserving the underlying structure and details.
- There are various techniques for removing salt-and-pepper noise from images, such as median filtering, adaptive median filtering, and morphological filtering. These techniques work by replacing the noisy pixels with the median or the mode of the neighboring pixels.
- Deep learning techniques can also be used to remove salt-and-pepper noise from images. For example, denoising autoencoders, convolutional neural networks (CNNs), and recurrent neural networks (RNNs) have been used to denoise images corrupted with salt-and-pepper noise. These techniques can learn to extract meaningful features from the noisy input images and generate clean output images.

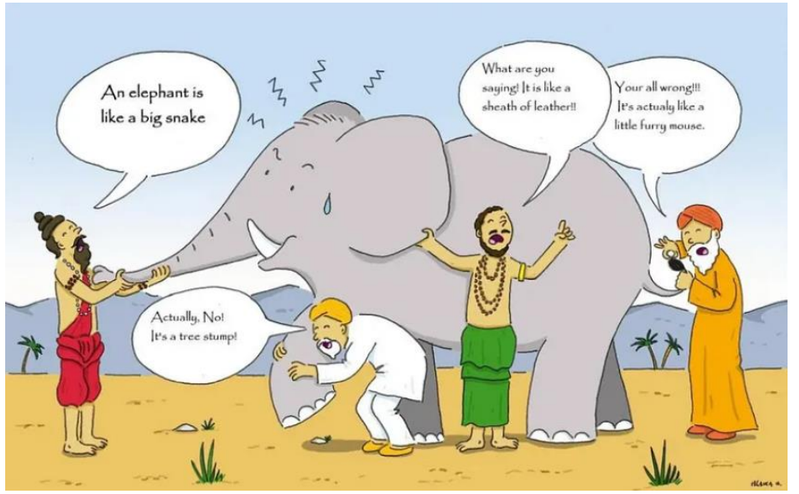
Random rotation, scaling or translation:

- Random rotation, scaling, or translation are common data augmentation techniques used in deep learning to artificially increase the size and diversity of the training dataset.
- These techniques involve applying random geometric transformations to the input data during training, such as rotating, scaling, or translating the images.
- The primary benefit of using these data augmentation techniques is to improve the robustness and generalization of the deep learning models.
- By exposing the model to a larger and more diverse set of training examples, it becomes more adept at recognizing patterns in the input data and is less likely to overfit to the training data.
- Random rotation involves rotating the input image by a random angle, usually within a specified range. This can help the model learn to recognize objects in different orientations and angles.
- Random scaling involves scaling the input image by a random factor, usually within a specified range. This can help the model learn to recognize objects of different sizes and scales.
- Random translation involves shifting the input image by a random amount, usually within a specified range. This can help the model learn to recognize objects in different positions and locations.
- Other types of geometric transformations can also be used as data augmentation techniques, such as shearing, flipping, and warping. The choice of which technique to use depends on the specific problem and the characteristics of the input data.

Ensemble methods

Ensemble learning is essentially a form of meta-learning, where the goal is to leverage the collective knowledge of several models to achieve better results than any single model could on its own.

The main principle behind the ensemble model is that a group of weak learners come together to form a strong learner, thus increasing the accuracy of the model. When we try to predict the target variable using any machine learning technique, the main causes of difference in actual and predicted values are noise, variance, and bias. Ensemble helps to reduce these factors (except noise, which is irreducible error).

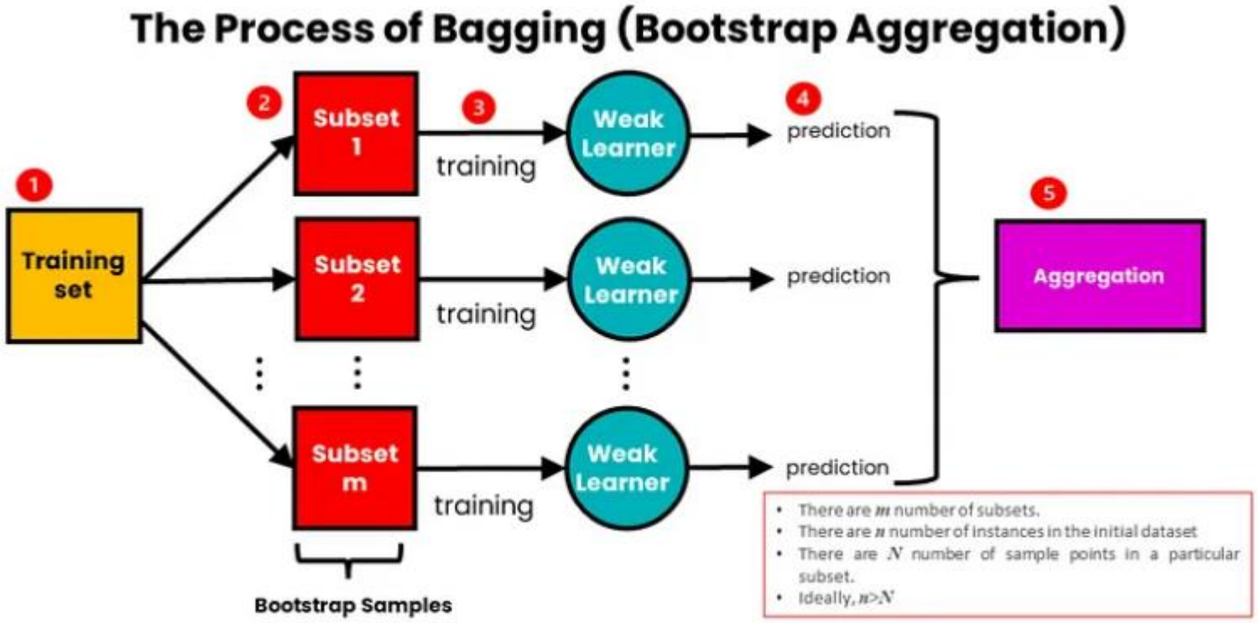


Remember the story of the blind men and the elephant? Every blind man offered a different account of what the elephant looked like. While all of the descriptions were accurate, it would have been preferable to gather and talk about their knowledge before making a decision. This narrative encapsulates the Ensemble learning approach perfectly.

In the machine learning paradigm known as "*ensemble learning*," several models — often referred to as "weak learners" — are taught to tackle a single problem and then integrated to produce better outcomes.

Ensemble approaches

1.Bagging, also known as bootstrap aggregating
Is the process of training several models concurrently on a random subset of the data (with replacement), then averaging the predictions from each model. Another well-known example of bagging is Random Forest.

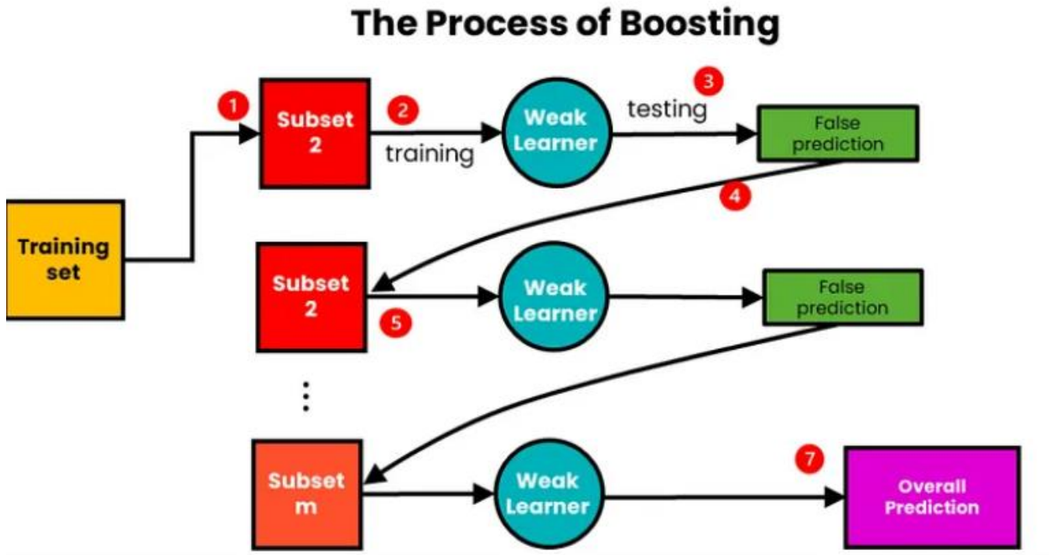


Steps:
These are the steps involved in bagging:

An initial training dataset with n instances is available to us.
We take the training set and divide it into m-number subgroups.
For every subset, we select a subset of N sample points from the original dataset.
We take each subset and replace it. This implies that multiple samples can be taken from a single data point.
We train the associated weak learners independently for each subgroup of the data.
Since these models are all of the same type, they are homogeneous.
Every model projects a future state.
One forecast is created by combining all of the predictions. Either maximum voting or average are applied in this case.

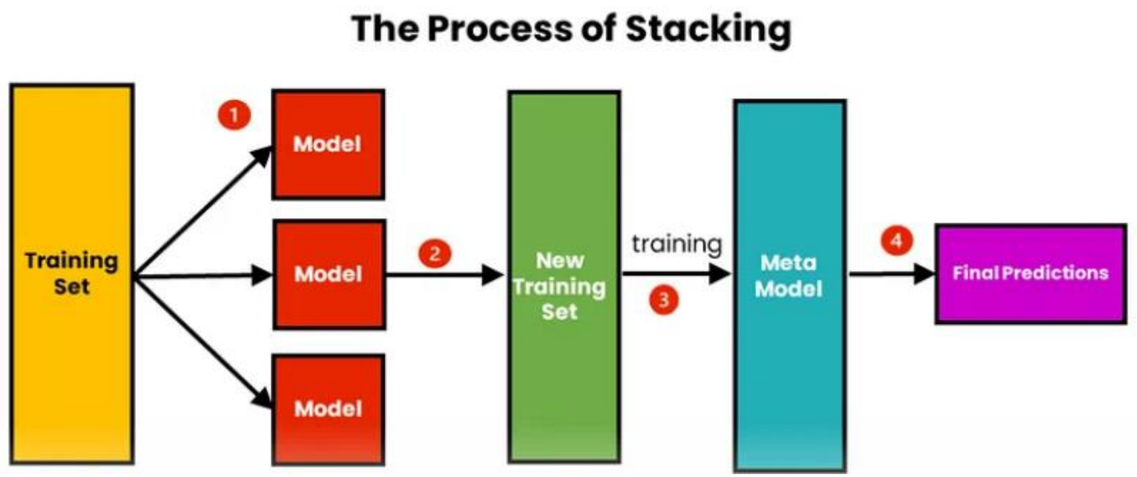
2. Boosting

- Boosting involves successively training models so that each one tries to fix the mistakes of the one before it. Predictions are derived from the weighted sum of the models, which are ranked according to how accurate they are. AdaBoost, gradient boosting, and XGBoost are a few examples.
- Boosting involves sequentially training weak learners. Here, each subsequent learner improves the errors of previous learners in the sequence. A sample of data is first taken from the initial dataset.
- This sample is used to train the first model, and the model makes its prediction. The samples can either be correctly or incorrectly predicted. The samples that are wrongly predicted are reused for training the next model. In this way, subsequent models can improve on the errors of previous models.
- Unlike bagging, which aggregates prediction results at the end, boosting aggregates the results at each step. They are aggregated using weighted averaging.
- Using weighted averaging, each model is assigned a different weight based on how well it predicts the future. Stated differently, it assigns greater weight to the model possessing the highest predictive power. This is due to the fact that the learner deemed most significant is the one with the most predictive power.



3. Stacking (stacked generalization)

Stacking, also known as stacked generalization, is the process of training several models on the same set of data, followed by the training of a meta-model to pr prediction that is based on the earlier models' predictions.



Here are the steps involved in stacking:

We train m-number of algorithms using the original training set of data.
Every algorithm's output is used to generate a fresh training set.
We develop a meta-model algorithm with the fresh training set.
Our ultimate forecast is based on the meta-model's findings. Weighted a utilized to aggregate the outcomes.

4. Voting

Voting: To arrive at a final prediction, a majority vote (in classification problems) or an average (in regression problems) is used to aggregate the predictions of multiple independently trained models. This technique makes use of the differences between the models to enhance performance overall.

When to use Bagging vs Boosting vs Stacking?

	Bagging	Boosting	Stacking
Purpose	Reduce Variance	Reduce Bias	Improve Accuracy
Base Learner Types	Homogeneous	Homogeneous	Heterogeneous
Base Learner Training	Parallel	Sequential	Meta Model
Aggregation	Max Voting, Averaging	Weighted Averaging	Weighted Averaging

Parameter Initialization

Initializing the parameters of a deep neural network is an important step in the training process, as it can have a significant impact on the convergence and performance of the model. Here are some common parameter initialization techniques used in deep learning:

1.Zero Initialization: Initialize all the weights and biases to zero. This is not generally used in deep learning as it leads to symmetry in the gradients, resulting in all the neurons learning the same feature.

2.Random Initialization: Initialize the weights and biases randomly from a uniform or normal distribution. This is the most common technique used in deep learning.

3.Xavier Initialization: Initialize the weights with a normal distribution with mean 0 and variance of sqrt(1/n), where n is the number of neurons in the previous layer. This is used for the sigmoid activation function.

4.He Initialization: Initialize the weights with a normal distribution with mean 0 and variance of sqrt(2/n), where n is the number of neurons in the previous layer. This is used for the ReLU activation function.

5.Orthogonal Initialization: Initialize the weights with an orthogonal matrix, which preserves the gradient norm during backpropagation.

6.Uniform Initialization: Initialize the weights with a uniform distribution. This is less commonly used than random initialization.

7.Constant Initialization: Initialize the weights and biases with a constant value. This is rarely used in deep learning.

It’s important to note that there is no one-size-fits-all initialization technique, as the optimal initialization may vary depending on the specific architecture and problem being tackled. Therefore, it’s often a good idea to experiment with different initialization techniques to see which works best for a given task.