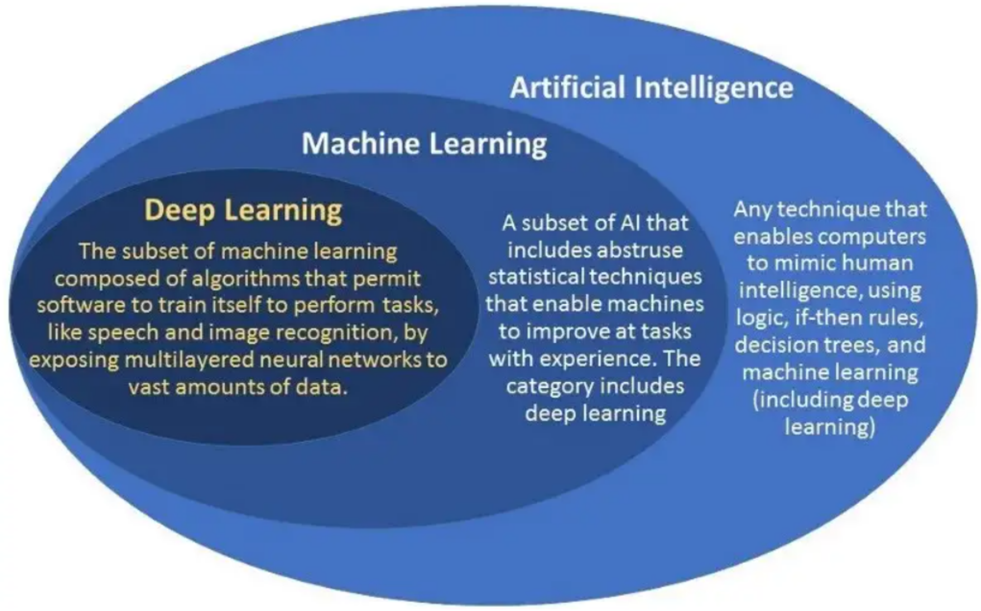
Machine Learning and Deep Learning

Machine learning and deep learning are two subfields of artificial intelligence (AI) that involve training and building models to make predictions or take actions based on data.

Machine learning refers to the development of algorithms and models that can learn from and make predictions or decisions based on data. It involves extracting patterns and insights from large datasets without being explicitly programmed. Machine learning algorithms can be broadly categorised into supervised learning, unsupervised learning, and reinforcement learning.

- Supervised learning involves training a model using labelled data, where the input data is paired with corresponding target or output labels. The model learns from this labelled data to make predictions or classify new, unseen data accurately. Examples of supervised learning algorithms include decision trees, support vector machines (SVM), and neural networks.
- Unsupervised learning, on the other hand, deals with unlabeled data, where the model aims to discover patterns, relationships, or structures within the data. It involves techniques like clustering, dimensionality reduction, and anomaly detection. Unsupervised learning helps to uncover hidden insights or groupings within the data without any predefined labels.
- Reinforcement learning is a type of machine learning where an agent learns to interact with an environment and make decisions based on trial and error. The agent receives feedback in the form of rewards or penalties for its actions, and it adjusts its behaviour over time to maximise the cumulative reward.

Deep learning, on the other hand, is a subset of machine learning that focuses on training deep neural networks with multiple layers to learn and represent complex patterns and relationships within the data. Deep learning architectures, such as convolutional neural networks (CNNs) and recurrent neural networks (RNNs), are designed to process and extract features from raw input data, such as images, text, or sequential data. Deep learning has


Artificial Intelligence
Machine Learning
Deep Learning
The subset of machine learning composed of algorithms that permit software to train itself to perform tasks, like speech and image recognition, by exposing multilayered neural networks to vast amounts of data.
A subset of AI that includes abstruse statistical techniques that enable machines to improve at tasks with experience. The category includes deep learning
Any technique that enables computers to mimic human intelligence, using logic, if-then rules, decision trees, and machine learning (including deep learning)

achieved remarkable success in various domains, including computer vision, natural language processing, speech recognition, and recommendation systems.
In summary, machine learning is a broader field that encompasses the development of algorithms and models that can learn from data and make predictions. Deep learning, on the other hand, is a subset of machine learning that focuses on training deep neural networks to learn complex patterns and representations from raw data. Both machine learning and deep learning have revolutionized various industries and continue to advance the field of artificial intelligence.

Introduction to Deep Learning

Deep learning is a subfield of machine learning that focuses on training deep neural networks to learn and represent complex patterns and relationships within data. These networks, inspired by the structure and function of the human brain, are capable of automatically learning hierarchical representations from raw input data. In this document, we will explore some of the key types of deep learning architectures and their applications.

- Artificial Neural Networks (ANNs): ANNs are computational models inspired by the structure and functioning of the human brain's neural networks. ANNs consist of interconnected nodes, or artificial neurons, organised in layers. ANNs have proven effective in a wide range of tasks, including pattern recognition, regression, and decision-making, making them a fundamental building block of various machine learning and artificial intelligence applications.

- Convolutional Neural Networks (CNNs): Convolutional Neural Networks are widely used for tasks involving images and computer vision. They excel at capturing spatial relationships by applying convolutional filters over the input data. CNNs can automatically learn features at different levels of abstraction, allowing them to recognize patterns, objects, and structures within images. They have found applications in image classification, object detection, image segmentation, and image generation.
- Recurrent Neural Networks (RNNs): Recurrent Neural Networks are designed to process sequential data, such as text or time series. They incorporate feedback connections to allow information to persist over time. RNNs have a memory-like capability, enabling them to capture dependencies and patterns across sequential data. They have proven effective in natural language processing tasks such as language translation, sentiment analysis, speech recognition, and text generation.
- Generative Adversarial Networks (GANs): Generative Adversarial Networks consist of two components: a generator network and a discriminator network. GANs learn through a competitive process where the generator tries to produce samples that fool the discriminator, while the discriminator aims to accurately differentiate between real and fake samples. GANs have made significant contributions to image synthesis, style transfer, and data augmentation.
- Autoencoders: Autoencoders are unsupervised learning models that aim to reconstruct input data. They consist of an encoder network that compresses the input data into a latent space representation, and a decoder network that reconstructs the input data from the latent representation. Autoencoders are primarily used for dimensionality reduction, anomaly detection, and data denoising. Variational Autoencoders (VAEs) are a variant that introduces probabilistic modeling, enabling the generation of new data samples from the learned latent space.
- Reinforcement Learning: Reinforcement Learning involves training an agent to interact with an environment and learn optimal actions through trial and error. Deep reinforcement learning combines deep neural networks with reinforcement learning algorithms. Deep Q-Networks (DQNs) are a popular form of deep reinforcement learning that has achieved groundbreaking success in game playing, robotic control, and autonomous systems.

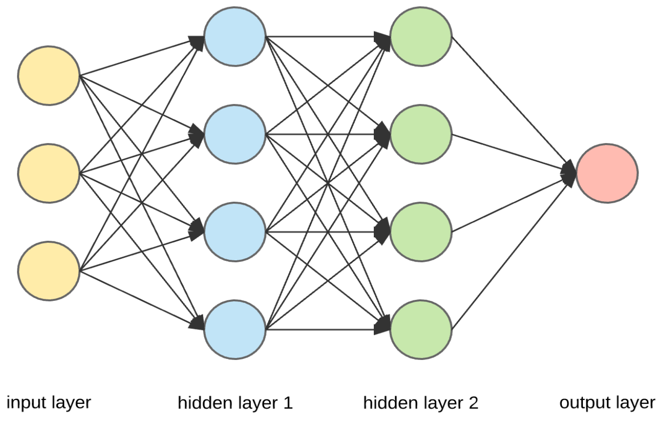
These are just a few examples of the many types of deep learning architectures that exist. Each architecture has its unique characteristics and applications, and researchers continue to explore new variations and combinations to tackle complex problems across various domains.

Neural Networks

Neural networks are a fundamental concept in the field of artificial intelligence and machine learning. Inspired by the structure and functioning of the human brain, neural networks are computational models composed of interconnected artificial neurons, also known as nodes or units. These networks are designed to learn and recognize patterns, make predictions, or perform tasks based on input data.

Components:

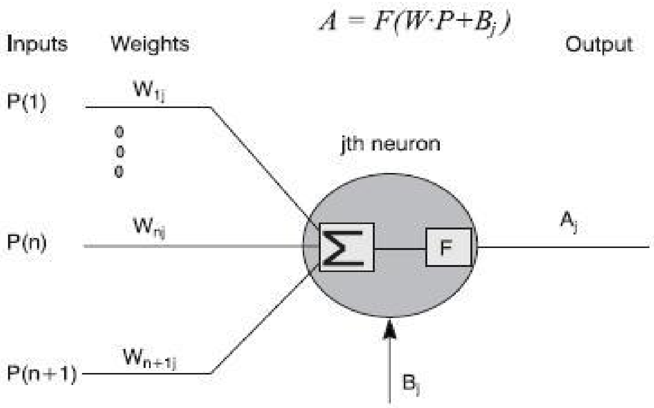
- Input Layer: The input layer receives the initial data or features and transmits them to the subsequent layers of the network. Each node in the input layer represents a feature or attribute of the input data.
- Hidden Layers: The hidden layers, positioned between the input and output layers, are responsible for extracting and learning the underlying patterns in the data. Deep neural networks have multiple hidden layers, allowing them to learn complex representations. Each node in the hidden layers performs computations based on weighted inputs and applies an activation function to introduce non-linearity into the network.
- Output Layer: The output layer produces the final results or predictions based on the computations performed in the hidden layers. The number of nodes in the output layer depends on the specific task. For instance, in a classification problem, the output layer may have nodes representing different classes, while in a regression problem, it may consist of a single node producing a continuous value.



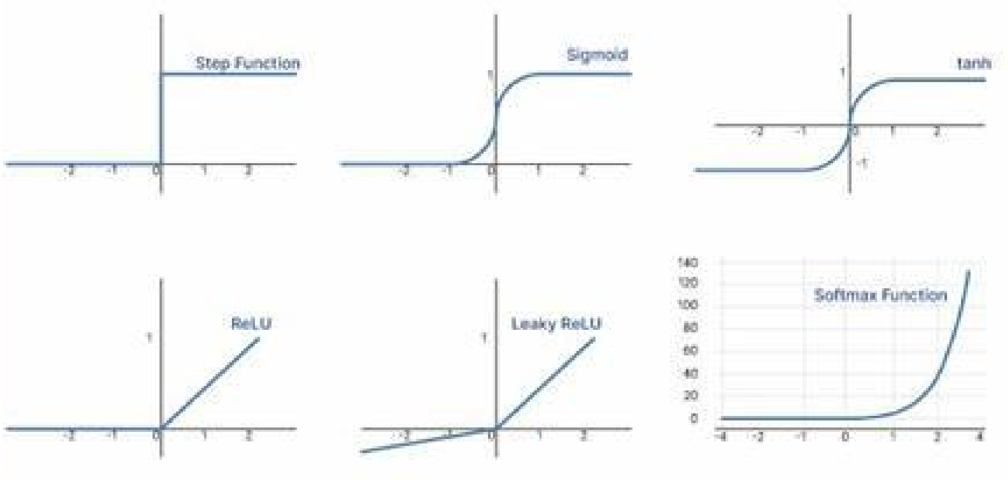
Hidden layers

These are an essential component of neural networks, situated between the input and output layers. They play a crucial role in extracting and learning complex representations and patterns from the input data. Let's delve into the details of hidden layers, including weights, biases, and activation functions.

- Weights: Weights in a neural network represent the strength or importance of the connections between neurons in different layers. Each connection between two neurons has an associated weight, which determines the contribution of the input from one neuron to the next. During the training process, these weights are adjusted iteratively using optimization algorithms, such as gradient descent, to minimize the error and optimize the network's performance. The values of the weights govern how the network learns and how the information flows through the network's layers.
- Biases: Biases in a neural network provide an additional learnable parameter for each neuron in the network, enabling it to make decisions or activations independent of the input. Biases act as an offset or threshold, allowing the network to learn and model complex relationships more effectively. Similar to weights, biases are adjusted during training to optimize the network's performance.



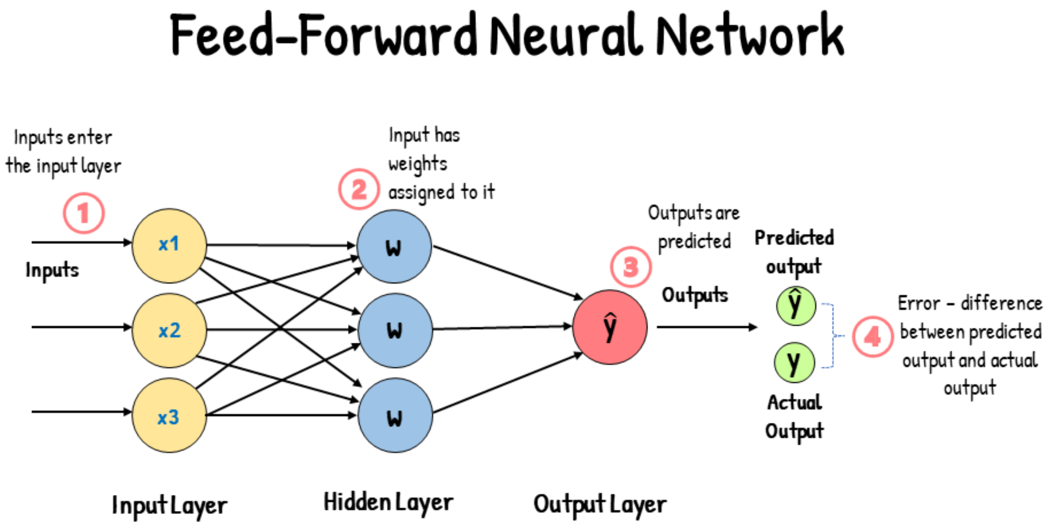
- Activation Functions: Activation functions introduce non-linearity into the neural network, allowing it to model and learn complex patterns that cannot be represented by a simple linear relationship. Each neuron in the hidden layers applies an activation function to its weighted sum of inputs (including bias) to produce an output or activation value. The choice of activation function depends on the nature of the problem and the characteristics of the data. It is crucial to select an appropriate activation function to ensure effective learning, model stability, and convergence during training. Common activation functions include:
    - Sigmoid (Logistic): Maps the input to a range between 0 and 1, smoothing out extreme values. It is commonly used in the output layer for binary classification problems.
    - Rectified Linear Unit (ReLU): Sets all negative inputs to zero and leaves positive inputs unchanged. ReLU is widely used in hidden layers as it helps in mitigating the vanishing gradient problem and speeds up training.
    - Hyperbolic Tangent (tanh): Similar to sigmoid, but maps the input to a range between -1 and 1. It is useful in scenarios where negative values are meaningful.
    - Softmax: Used in the output layer for multi-class classification problems, softmax normalizes the output values into a probability distribution, summing up to 1. It helps in selecting the most probable class.



Backpropagation

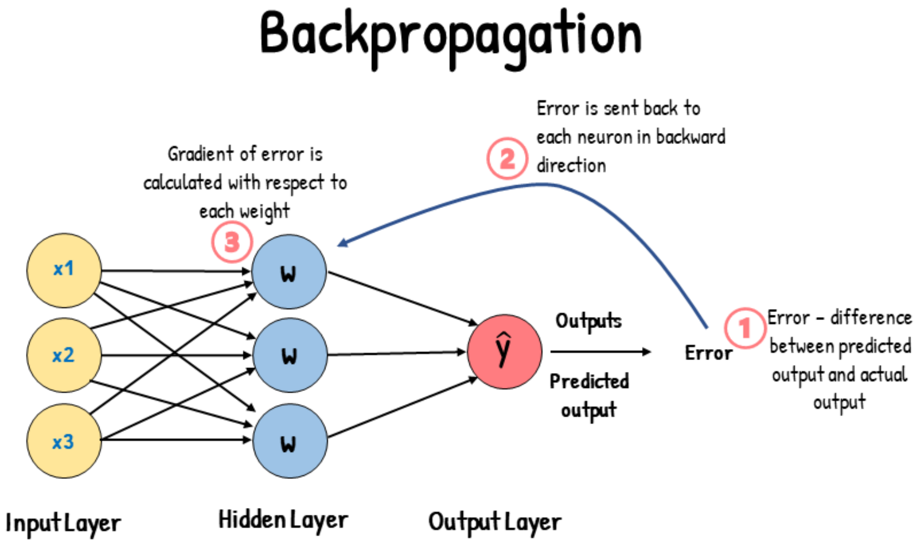
During the training phase, neural networks learn from labeled data through a process called backpropagation. It is the process of updating the weights and biases of the network based on the difference between the predicted output and the desired output.

- Forward Pass: During the forward pass, input data is fed into the neural network, and calculations are performed layer by layer. Each neuron in the network receives inputs from the previous layer, applies a weighted sum, adds a bias, and passes the result through an activation function to produce an output.


Feed-Forward Neural Network
Inputs enter the input layer
1
Inputs
x1
x2
x3
2
Input has weights assigned to it
W
W
W
Outputs are predicted
3
ŷ
Outputs
Predicted output
ŷ
y
Actual Output
4
Error - difference between predicted output and actual output
Input Layer
Hidden Layer
Output Layer

- Calculation of Error: After the forward pass, the output of the network is compared with the desired output. The difference between the predicted output and the desired output is the error.

- Backward Pass: In the backward pass (backpropagation), the error is propagated back through the network. It starts from the output layer and moves backward, calculating the contribution of each weight and bias to the overall error.


Backpropagation
Error is sent back to each neuron in backward direction
2
Gradient of error is calculated with respect to each weight
3
x1
x2
x3
w
w
w
ŷ
Outputs
Predicted output
Error
1
Error - difference between predicted output and actual output
Input Layer
Hidden Layer
Output Layer

- Weight and Bias Updates: The weights and biases are then adjusted to minimise the error. This adjustment is done by updating the weights and biases in the opposite direction of the gradient (slope) of the error with respect to each weight and bias. This is achieved using an optimization algorithm such as gradient descent.
- Iterative Process: The forward pass, error calculation, backward pass, and weight and bias updates are repeated iteratively for a set number of epochs or until the network's performance reaches a desired level.

By iteratively updating the weights and biases based on the error, the network gradually improves its ability to make accurate predictions. This process of forward pass, error calculation, and backward pass is known as backpropagation, and it is a key mechanism for training neural networks.

In summary, backpropagation is the process of adjusting the weights and biases of a neural network based on the error between the predicted output and the desired output. It involves propagating the error backward through the network, calculating the contribution of each weight and bias to the overall error, and updating them accordingly. Through this iterative process, the network learns to make better predictions over time.

Hyperparameters

In machine learning models, hyperparameters are adjustable settings that are determined prior to training the model. These parameters control the behavior and performance of the model during the learning process. Here are some common hyperparameters in machine learning models:

- Learning Rate: The learning rate determines the step size at each iteration during optimization. It affects how quickly the model learns and converges to an optimal solution.
- Number of Hidden Units/Layers: The number of hidden units or layers in a neural network determines the complexity and capacity of the model. It influences the model's ability to learn intricate patterns and represent complex relationships.
- Regularization Strength: Regularization is used to prevent overfitting in a model. The regularization strength hyperparameter controls the amount of regularization applied. Common types of regularization include L1 regularization (Lasso) and L2 regularization (Ridge).
    - Overfitting: When a model overfits, it essentially memorizes the training examples instead of learning the underlying relationships in the data. As a result, the model becomes too specific to the training set and performs poorly on new data, which can lead to inaccurate predictions and reduced performance.
- Batch Size: Batch size refers to the number of samples used in each iteration of training. It affects the speed and stability of the training process. Larger batch sizes can lead to faster convergence but may require more memory.
- Activation Functions: Activation functions introduce non-linearity into the model and play a crucial role in the model's ability to learn complex relationships. The choice of activation functions, such as ReLU, sigmoid, or tanh, is an important hyperparameter.
- Dropout Rate: Dropout is a regularization technique that randomly sets a fraction of the input units to zero during training. The dropout rate hyperparameter determines the proportion of units that are dropped out, helping prevent overfitting.
- Kernel Size: In convolutional neural networks (CNNs), the kernel size determines the size of the convolutional filter used for feature extraction. It affects the receptive field and the level of detail captured by the model.
- Epoch: During an epoch, the model processes each training sample, computes the loss or error between the predicted outputs and the actual labels, and updates its

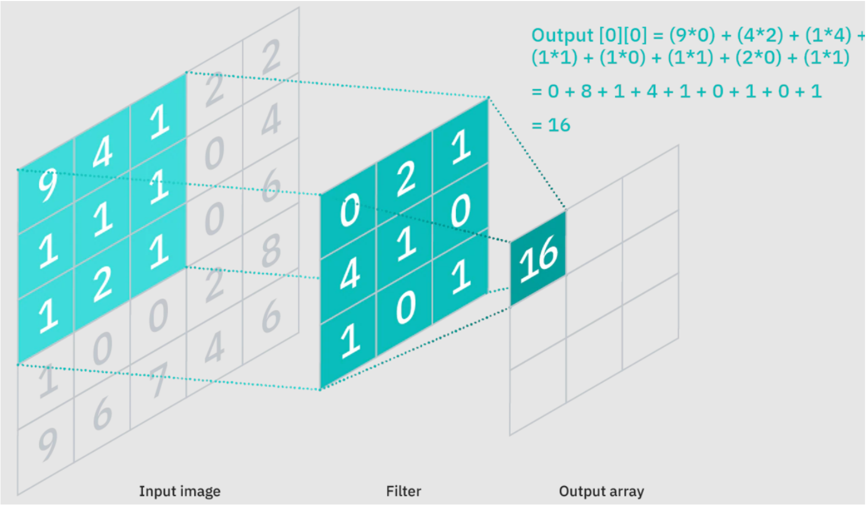
internal parameters (weights and biases) based on the optimization algorithm and the computed gradients. The number of epochs is a hyperparameter that determines how many times the model will iterate over the entire training dataset.

  - It's important to note that an epoch should not be confused with a batch. In some cases, the training data is divided into smaller batches, and one epoch consists of multiple iterations through these batches. Each batch is processed by the model, and the gradients are computed and used to update the model's parameters. Once all the batches have been processed in an epoch, the epoch is considered complete.

Convolutional Neural Networks (CNN)

Convolutional Neural Networks (CNNs) are a type of deep neural network specifically designed for processing and analysing visual data, such as images. They are widely used in computer vision tasks such as image classification, object detection, image segmentation, and more.

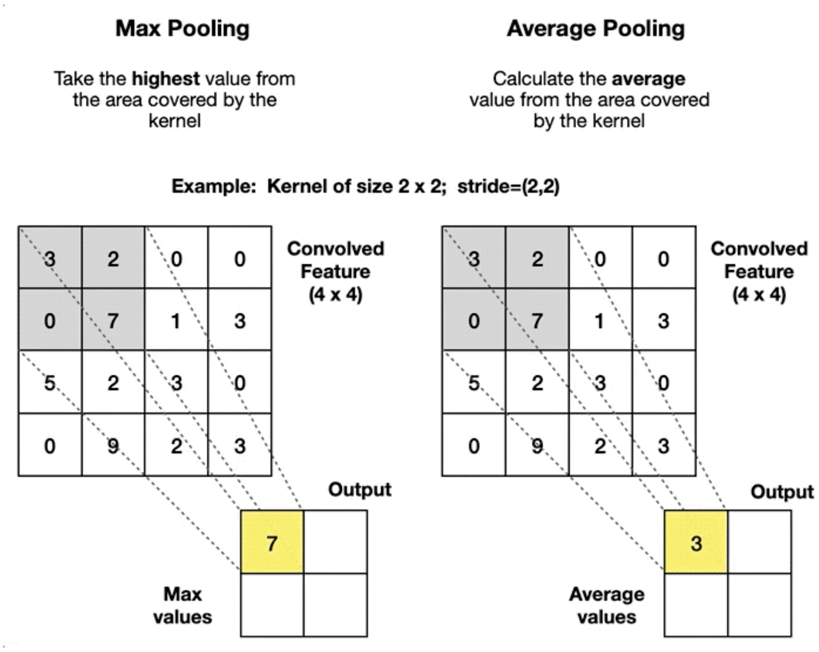
- Convolutional Layers: The primary building blocks of CNNs are convolutional layers. Each layer consists of multiple filters (also known as kernels) that slide over the input image in small strides. The filters perform element-wise multiplications and summations between their weights and the corresponding local receptive field of the input. This process produces a feature map that captures specific patterns or features at different spatial locations of the input image.



- Activation Functions: After the convolutional operation, an activation function is applied element-wise to introduce non-linearity. Common activation functions include

Rectified Linear Unit (ReLU), sigmoid, and hyperbolic tangent (tanh). ReLU is the most widely used activation function in CNNs due to its simplicity and effectiveness in preventing the vanishing gradient problem.

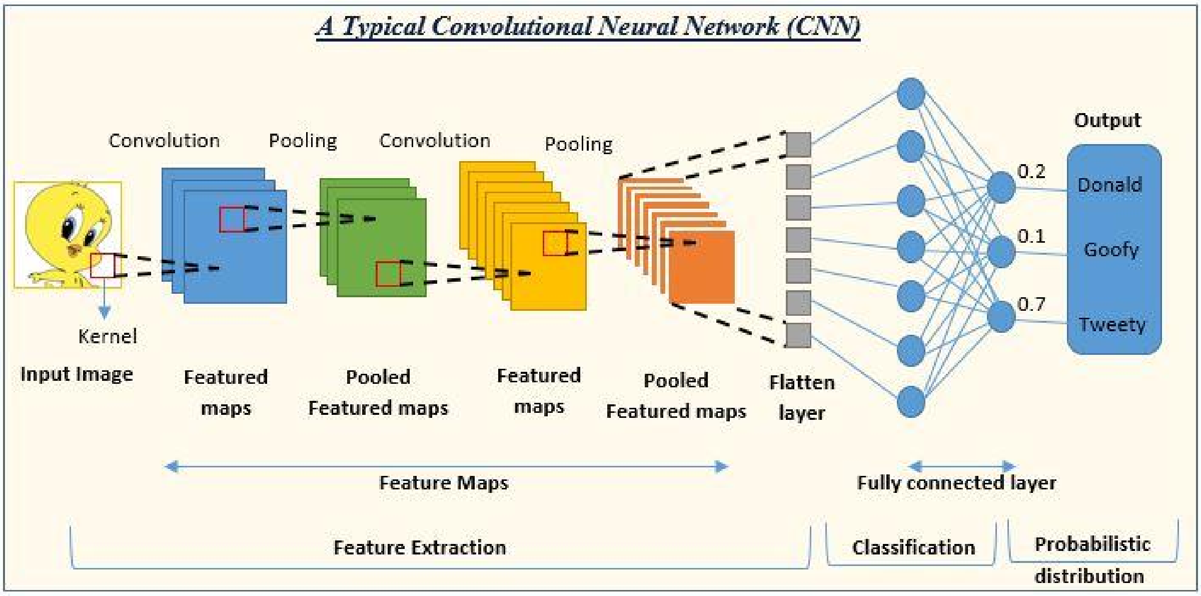
- Pooling Layers: Pooling layers help reduce the spatial dimensions of the feature maps generated by the convolutional layers, making the network more computationally efficient. The most common type of pooling is max pooling, which selects the maximum value within a pooling region. Average pooling is another variant that takes the average value instead. Pooling helps capture the most salient features and provides translational invariance to small shifts in the input.


Max Pooling
Take the highest value from the area covered by the kernel
Average Pooling
Calculate the average value from the area covered by the kernel
Example: Kernel of size 2 x 2; stride=(2,2)
3 2 0 0
0 7 1 3
5 2 3 0
0 9 2 3
Convolved Feature (4 x 4)
Output
7
Max values
3 2 0 0
0 7 1 3
5 2 3 0
0 9 2 3
Convolved Feature (4 x 4)
Output
3
Average values

- Fully Connected Layers: Fully connected layers are traditionally placed at the end of the CNN. They connect every neuron in the previous layer to each neuron in the current layer, allowing for complex combinations of features. The fully connected layers process the learned features and make final predictions or classifications.
- Training and Learning: CNNs are trained using labelled training data. During training, the network learns to adjust the weights and biases associated with the filters and fully connected layers to minimise a loss function. This is typically done through an optimization algorithm, such as stochastic gradient descent (SGD) or Adam, using the backpropagation algorithm to update the weights based on the gradient of the loss function with respect to each parameter.
- Architecture Variants: CNNs can have various architectures based on the specific task or problem. Some popular architectures include AlexNet, VGGNet, GoogLeNet

(Inception), and ResNet. These architectures vary in terms of the number of layers, filter sizes, and connectivity patterns, allowing for more effective feature extraction and improved performance.

The strength of CNNs lies in their ability to automatically learn relevant features from the input data through the convolutional layers. By stacking multiple convolutional, activation, pooling, and fully connected layers, CNNs can capture increasingly complex patterns and representations. The hierarchical nature of CNNs enables them to learn local features in lower layers and global or abstract features in higher layers, leading to superior performance in visual recognition tasks.



Loss Functions and Optimization Algorithms

Loss functions measure the discrepancy between the predicted output of a CNN and the true output labels. The choice of the appropriate loss function depends on the specific task at hand, such as classification, regression, or semantic segmentation. Here are some commonly used loss functions in CNNs:

- Cross-Entropy Loss: Cross-entropy loss is widely used for classification tasks. It calculates the difference between the predicted class probabilities and the true class labels. It encourages the network to assign higher probabilities to the correct class and lower probabilities to the incorrect classes.

- Mean Squared Error (MSE) Loss: MSE loss is commonly used for regression tasks. It measures the average squared difference between the predicted continuous values and the true values. It penalizes larger errors more heavily.
- Binary Cross-Entropy Loss: Binary cross-entropy loss is used for binary classification tasks. It computes the difference between the predicted probabilities and the true binary labels.

Optimization algorithms are employed to update the weights and biases of the CNN during the training process. The objective is to find the optimal set of parameters that minimise the loss function. Here are some popular optimization algorithms used in CNNs:

- Stochastic Gradient Descent (SGD): SGD is a classic optimization algorithm that updates the parameters in the direction of the negative gradient of the loss function. It uses a small subset of training samples (a mini-batch) to compute the gradient, making it computationally efficient. Variants of SGD include Momentum, Nesterov Accelerated Gradient, and AdaGrad.
- RMSProp: RMSProp is an adaptive learning rate optimization algorithm. It adjusts the learning rate based on the magnitude of recent gradients, giving more weight to recent updates. It helps to converge faster and handle sparse data.
- Adagrad: Adagrad is another adaptive learning rate algorithm that adapts the learning rate for each parameter based on the historical gradients. It accumulates the squared gradients to give more importance to parameters with larger gradients, effectively reducing the learning rate for frequently occurring parameters.
- Adam: Adam is an adaptive optimization algorithm that computes individual learning rates for each parameter by adapting them based on past gradients. It combines the advantages of both AdaGrad and RMSProp algorithms. Adam is widely used due to its effectiveness and faster convergence.

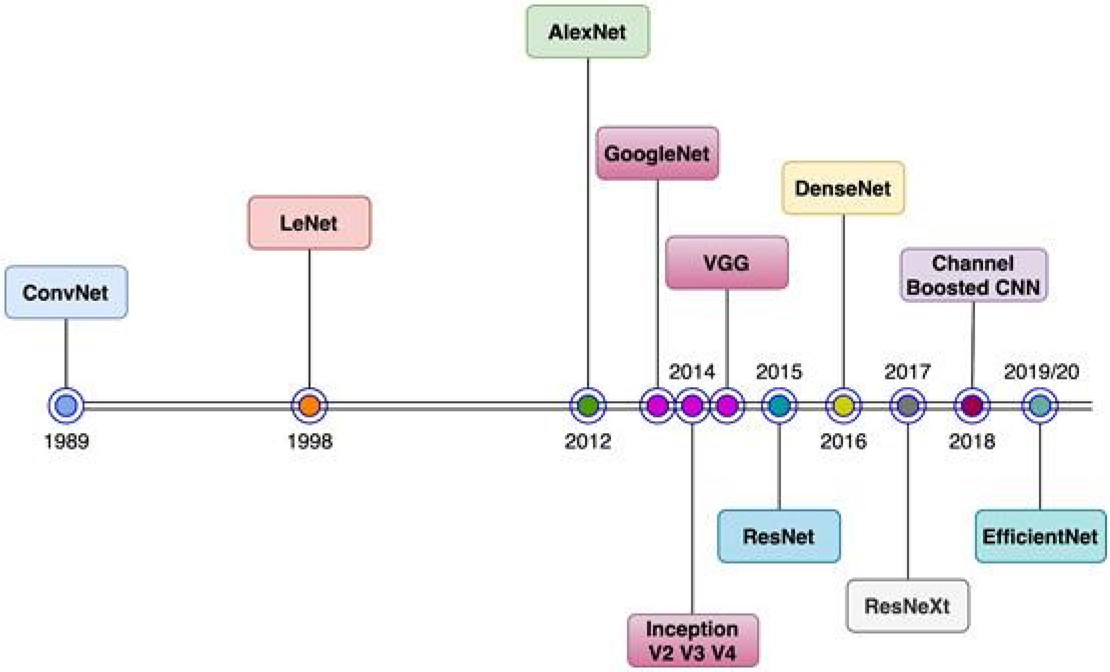
These optimization algorithms aim to find the optimal set of weights and biases that minimize the loss function, thereby improving the performance and accuracy of the CNN during training. It's important to note that the selection of loss functions and optimization algorithms depends on the specific task, dataset characteristics, and network architecture. Experimentation and fine-tuning of these choices are often required to achieve optimal results.

Image Classification

Several CNN architectures have been developed specifically for image classification tasks, achieving state-of-the-art performance on benchmark datasets like ImageNet. Here are some popular CNN architectures for image classification:

- AlexNet: AlexNet was one of the first deep CNN architectures to gain significant attention. It consists of five convolutional layers, max-pooling layers, and three fully connected layers. AlexNet used the ReLU activation function and dropout regularization.
- VGGNet: VGGNet, proposed by the Visual Geometry Group, is known for its simplicity and uniform architecture. It consists of multiple stacked 3x3 convolutional layers, followed by max-pooling layers. VGGNet offers different variations with varying depths, such as VGG16 and VGG19.
- GoogLeNet (Inception): GoogLeNet introduced the concept of the Inception module, which performs parallel convolutions of different sizes and concatenates their outputs. This architecture reduces computational complexity while capturing multi-scale features effectively. Notable versions include GoogLeNet v1, v2, and v3.
- ResNet: ResNet (Residual Network) is a groundbreaking architecture that introduced residual connections. ResNet enables the training of very deep networks (50 or more layers) by mitigating the vanishing gradient problem. ResNet architectures include ResNet50, ResNet101, and ResNet152, among others.
- DenseNet: DenseNet employs dense connections between layers, where each layer receives inputs from all preceding layers. This connectivity pattern encourages feature reuse and facilitates gradient flow, leading to better parameter efficiency and performance. DenseNet variants include DenseNet121, DenseNet169, and DenseNet201.
- EfficientNet: EfficientNet focuses on achieving optimal model scaling by scaling the depth, width, and resolution of the network simultaneously. It utilizes compound scaling to balance model size and performance. EfficientNet models, such as EfficientNetB0, EfficientNetB7, have demonstrated excellent performance with efficient resource utilization.
- MobileNet: MobileNet architectures are designed for mobile and embedded devices, prioritizing computational efficiency and model size reduction. MobileNet employs depth-wise separable convolutions to reduce the number of parameters and

operations while maintaining accuracy. MobileNetV2 and MobileNetV3 are notable versions.



These architectures have pushed the boundaries of image classification accuracy and efficiency. However, it is essential to consider the specific requirements and constraints of your application when choosing a CNN architecture, as different architectures offer trade-offs in terms of model size, inference speed, and accuracy.

PyTorch

PyTorch is a widely-used open-source deep learning framework known for its simplicity, flexibility, and efficiency. It provides a Python-based interface that allows developers and researchers to build and train neural networks with ease. Whether you are a beginner or an experienced deep learning practitioner, PyTorch offers a seamless and intuitive experience for developing state-of-the-art models.

Features

- Dynamic Computational Graph: This feature allows users to build and modify computational graphs on the fly, providing more flexibility and enabling dynamic control flow. It simplifies debugging and makes experimentation faster and more straightforward.

- Easy-to-Use API: PyTorch's API is designed to be user-friendly and intuitive. Its syntax closely resembles standard Python, making it easy to learn and understand.
- Extensive Community and Ecosystem: PyTorch boasts a vibrant and active community of researchers, developers, and enthusiasts.
- Deep Learning Research Framework: PyTorch has gained significant popularity among researchers due to its seamless integration with Python scientific libraries such as NumPy and SciPy. It provides an ideal platform for rapid prototyping and experimenting with new ideas, facilitating the advancement of cutting-edge research in the deep learning field.
- Efficient GPU Acceleration: PyTorch leverages the power of GPUs to accelerate computations, significantly reducing training time for large-scale models. It provides a CUDA backend for GPU acceleration, making it suitable for high-performance computing and complex deep learning tasks.
- Deployment Capabilities: PyTorch offers various deployment options, including converting models to optimized formats like ONNX (Open Neural Network Exchange) for efficient deployment on different platforms. Additionally, PyTorch integrates well with other popular frameworks, allowing seamless transfer of models between frameworks for production deployments.

Tensors in PyTorch

In PyTorch, a tensor is a fundamental data structure that represents multi-dimensional arrays of numerical data. It is similar to a matrix or an N-dimensional array. Tensors are like containers that hold numbers. They are used in PyTorch to store and manipulate data.
Think of a tensor as a box with multiple compartments. Each compartment can hold a number. The number can be anything—like the temperature outside, the brightness of a pixel in an image, or the weight of a person.
Tensors can have different shapes, just like boxes can have different sizes. For example, a 1-dimensional tensor could be like a long line of compartments, while a 2-dimensional tensor could be like a grid of compartments.

- Data Storage: Tensors store data of homogeneous type (e.g., floating-point numbers) in contiguous memory. They can be created from Python lists, NumPy arrays, or directly generated within PyTorch.

- Multi-Dimensional: Tensors can have any number of dimensions (rank), such as 0D (scalar), 1D (vector), 2D (matrix), or higher dimensions. For example, a 2D tensor represents a matrix with rows and columns.
- Data Type: Tensors have a specific data type (dtype) that determines the type of values they can hold, such as float, integer, or Boolean. PyTorch supports various data types, including float16, float32, float64, int8, int16, int32, int64, and more.
- Operations: Tensors support a wide range of mathematical operations, such as element-wise arithmetic, matrix operations, and reduction operations (e.g., sum, mean). PyTorch provides a rich library of functions and operations to manipulate tensors efficiently.

Tensors and numpy arrays

Tensors and NumPy arrays share many similarities, as they are both used for handling numerical data in Python. However, there are some differences between them as well. Let's explore the differences and similarities between tensors and NumPy arrays:

Similarities:

- Multi-Dimensional Data: Both tensors and NumPy arrays can represent multi-dimensional data structures, such as vectors, matrices, or higher-dimensional arrays. They allow efficient storage and manipulation of numerical data with different dimensions.
- Mathematical Operations: Both tensors and NumPy arrays support a wide range of mathematical operations, such as element-wise arithmetic, linear algebra operations, statistical calculations, and more. Many mathematical functions and operations available in NumPy can also be applied to tensors.
- Broadcasting: Both tensors and NumPy arrays utilize broadcasting, which allows for implicit element-wise operations between arrays of different shapes. Broadcasting eliminates the need for explicit loops and simplifies the writing of mathematical expressions.

Differences:

- Framework Dependency: Tensors are primarily associated with deep learning frameworks like PyTorch or TensorFlow, whereas NumPy arrays are a fundamental component of the NumPy library. Tensors are specific to deep learning frameworks, while NumPy arrays can be used across a wider range of numerical computing tasks.

- GPU Support: Tensors have built-in support for GPU acceleration, enabling efficient computations on GPUs. Deep learning frameworks provide tools to seamlessly move tensors between CPU and GPU memory, allowing for high-performance training and inference. NumPy arrays, on the other hand, are primarily designed for CPU computations.
- Automatic Differentiation: Tensors in deep learning frameworks often have built-in support for automatic differentiation. This means that gradients can be computed automatically for tensor operations, which is crucial for training neural networks. NumPy arrays do not have native support for automatic differentiation.
- Additional Functionality: Deep learning frameworks often provide additional functionality specific to tensors, such as convolutional operations, pooling operations, and neural network layers. NumPy arrays, being a general-purpose numerical computing tool, do not have these specialized operations.

Training a Model

- Data Preparation: Preparing the data involves loading and preprocessing the dataset. This includes splitting the data into training, validation, and testing sets, applying transformations (such as normalization or data augmentation), and creating data loaders to efficiently load the data during training.
- Model Definition: Define the architecture of your neural network model. In PyTorch, you typically create a class that inherits from the ***nn.Module*** class. Inside the class, you define the layers and operations of your model in the ***__init__*** method, and the forward pass logic in the forward method.
- Loss Function Selection: Choose an appropriate loss function based on the problem you are trying to solve. Common loss functions include Cross-Entropy Loss for classification tasks and Mean Squared Error for regression tasks. PyTorch provides various loss functions in the ***torch.nn*** module.
- Optimizer Selection: Select an optimizer to update the model's weights during training. Popular optimizers include Stochastic Gradient Descent (SGD), Adam, or RMSProp. PyTorch provides these optimizers in the ***torch.optim*** module.
- Training Loop: Write a loop that iterates over the training dataset in batches. For each batch, perform the following steps:
  - Zero the gradients: Set the gradients of the model parameters to zero before computing the gradients for the current batch.

  - Forward Pass: Pass the input batch through the model to get the predicted outputs.
  - Compute Loss: Calculate the loss between the predicted outputs and the ground truth labels using the chosen loss function.
  - Backward Pass: Perform backpropagation to compute the gradients of the model parameters with respect to the loss.
  - Update Parameters: Use the optimizer to update the model parameters based on the computed gradients. Optionally, track metrics like accuracy or additional custom metrics during training.
- Validation: After each epoch or a certain number of training iterations, evaluate the model's performance on the validation set. Compute the validation loss and any desired evaluation metrics to monitor the model's progress and potentially perform early stopping if the performance stops improving.
- Testing: Once the training is complete, evaluate the final model on the testing set to assess its generalization performance. Compute the testing loss and any desired evaluation metrics to evaluate the model's effectiveness.
- Saving and Loading the Model: Save the trained model's parameters or the entire model to disk, allowing you to reuse or deploy the model later without retraining.

These steps provide a high-level overview of the typical process involved in training a neural network in PyTorch. Customizations and additional steps can be added based on specific requirements and the nature of the problem at hand.

Saving and Loading a Model

- State Dictionary: In PyTorch, the learnable parameters (i.e. weights and biases) of an torch.nn.Module model are contained in the model’s parameters (accessed with model.parameters()). A state_dict is simply a Python dictionary object that maps each layer to its parameter tensor. Note that only layers with learnable parameters (convolutional layers, linear layers, etc.) and registered buffers (batchnorm’s running_mean) have entries in the model’s state_dict. Optimizer objects (torch.optim) also have a state_dict, which contains information about the optimizer’s state, as well as the hyperparameters used. Because state_dict objects are Python dictionaries, they can be easily saved, updated, altered, and restored, adding a great deal of modularity to PyTorch models and optimizers.

- Saving a Model: To save a model, you need to specify the model's state dictionary and the desired file path to save it. The state dictionary contains all the learnable parameters and other necessary information of the model.

  `torch.save(model.state_dict(), 'model.pth')`

  In the example above, model is the instance of your trained model, and 'model.pth' is the file path where you want to save the model. It will create a file with the specified name (e.g., model.pth) and store the model's state dictionary in it.
- Loading a Model: To load a saved model, you need to create an instance of the model and load the saved state dictionary into it.

  `model = YourModelClass()
  model.load_state_dict(torch.load('model.pth'))`

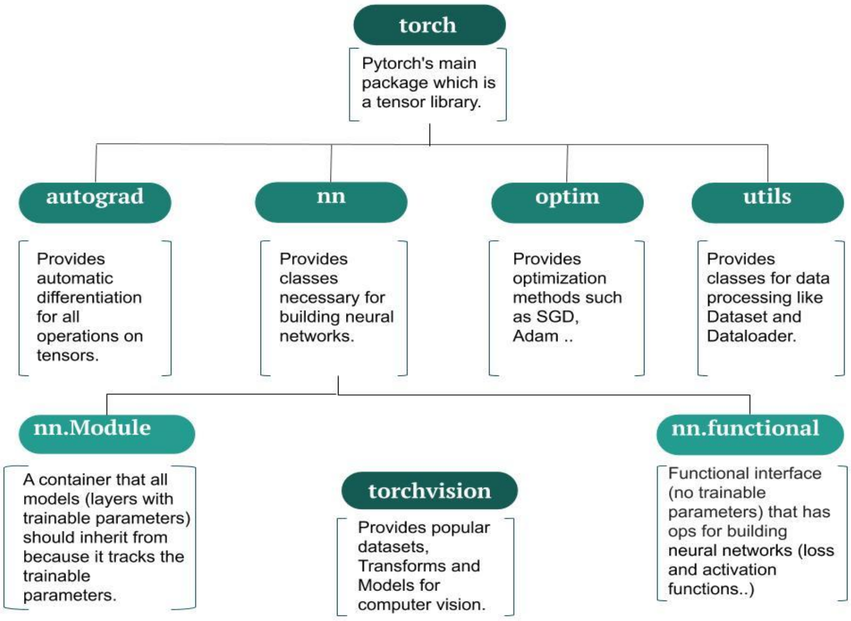
  In the example above, YourModelClass() represents the class or architecture of your model. You need to create an instance of your model before loading the saved state dictionary. The torch.load() function loads the saved state dictionary from the specified file path (e.g., 'model.pth') and assigns it to the model.

Some Useful PyTorch Modules

- ***torch.Tensor***: torch.Tensor is the primary data structure in PyTorch that represents multi-dimensional arrays. It supports various operations for numerical computations and serves as the foundation for building neural networks.
- ***torch.nn.Module***: torch.nn.Module is a base class for all neural network modules in PyTorch. It provides functionality for defining and organising network layers, managing parameters, and performing forward and backward computations. You can create custom neural network models by subclassing torch.nn.Module and implementing the forward method.
- ***torch.nn.Linear***: torch.nn.Linear implements a fully connected layer in a neural network. It applies a linear transformation to the input data by multiplying it with a weight matrix and adding a bias term. This module is commonly used in the construction of neural network architectures.
- ***torch.nn.Conv2d***: torch.nn.Conv2d is used for 2D convolutional operations in convolutional neural networks. It applies a specified number of filters (kernels) to the input data, enabling the network to extract spatial features.
- ***torch.nn.ReLU***: torch.nn.ReLU is an activation function that introduces non-linearity in neural networks. It applies the rectified linear unit (ReLU) function element-wise, setting negative values to zero and preserving positive values.

- ***torch.nn.Dropout***: torch.nn.Dropout is a regularization technique used to prevent overfitting in neural networks. It randomly sets a fraction of the input elements to zero during training, forcing the network to learn robust representations.
- ***torch.optim***: The torch.optim module provides various optimization algorithms to update the parameters of neural networks during training. It includes popular optimizers like Stochastic Gradient Descent (SGD), Adam, RMSProp, and more.
- ***torchvision***: The torchvision package provides datasets, data transformations, and model architectures specifically designed for computer vision tasks. It includes pre-trained models, image transformations, and commonly used datasets like MNIST, CIFAR-10, and ImageNet.

These are just a few examples of the PyTorch modules commonly used in deep learning. PyTorch offers a wide range of modules and functions that cater to different aspects of building and training neural networks, providing flexibility and ease of use in deep learning workflows.


torch
Pytorch's main package which is a tensor library.
autograd
Provides automatic differentiation for all operations on tensors.
nn
Provides classes necessary for building neural networks.
optim
Provides optimization methods such as SGD, Adam ..
utils
Provides classes for data processing like Dataset and Dataloader.
nn.Module
A container that all models (layers with trainable parameters) should inherit from because it tracks the trainable parameters.
torchvision
Provides popular datasets, Transforms and Models for computer vision.
nn.functional
Functional interface (no trainable parameters) that has ops for building neural networks (loss and activation functions..)

Useful Links

1. Deep Learning Fundamentals Playlist:
   https://youtube.com/playlist?list=PLZbbT5o_s2xq7LwI2y8_QtvuXZedL6tQU
2. Neural Networks Basic Concepts (includes some math as well):
   https://youtube.com/playlist?list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi
3. Introduction to Tensors: https://youtu.be/v43SlgBcZ5Y
4. Basic Networks: https://youtu.be/w9U57o6wto0
5. Basics of Sequential Networks: https://youtu.be/NaptjtDyvuY
6. Image Classification (MNIST): https://youtu.be/gBw0u_5u0qU
7. Pytorch Official Documentation Introduction:
   https://pytorch.org/tutorials/beginner/basics/intro.html#learn-the-basics