

$$F = G \frac{m_1 m_2}{d^2}$$

Artificial neural networks

April 2025

$$\frac{df}{dt} = \lim_{h \rightarrow 0} \frac{f(t+h) - f(t)}{h}$$

Topics to be discussed

Part I. Introduction

1. Introduction to Artificial intelligence

Part II. Search and optimisation.

2. Search - basic approaches
3. Search - optimisation
4. Two-player deterministic games
5. Evolutionary and genetic algorithms

Part III. Machine learning and data analysis.

6. Regression, classification and clustering (Part I & II)
8. Artificial neural networks
9. Bayesian models
10. Reinforcement Learning

Part IV. Logic, Inference, Knowledge Representation

11. Propositional logic and predicate logic
12. Knowledge Representation

Part V. AI in Action: Language, Vision

13. AI in Natural language processing
14. AI in Vision and Graphics

Part VI. Summary

15. AI engineering, Explainable AI, Ethics,

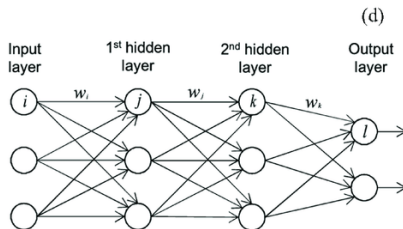
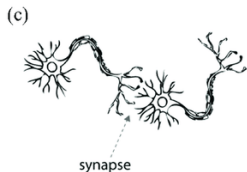
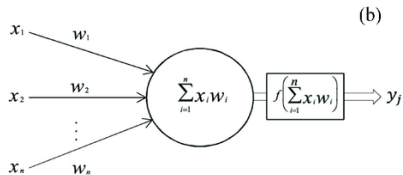
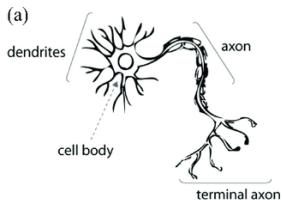
Agenda

- 1 Introduction to Neural Networks
- 2 Perceptron and Bilayer Networks
- 3 Gradient Reverse Propagation
- 4 Network Learning
- 5 Optimization Algorithms for NN
- 6 NN architectures
- 7 Summary

Introduction to Neural Networks

- Motivation: the need for a machine learning method that can learn from complex, high-dimensional data
- Definition: a neural network is a collection of connected units or nodes that can process information through a series of nonlinear transformations

A biological and an artificial neural networks



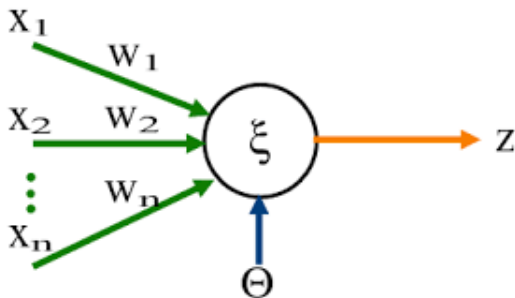
History of Artificial Neural Networks

- 1943: McCulloch-Pitts Neuron
 - Warren McCulloch and Walter Pitts proposed the first mathematical model of a neuron.
 - Their model inspired the development of perceptrons in the 1950s.
- 1957: Rosenblatt's Perceptron
 - Frank Rosenblatt developed the perceptron, the first algorithm capable of learning from its own mistakes.
 - Perceptrons were limited to linearly separable problems and fell out of favor in the 1960s.
- 1969: Backpropagation
 - Bryson and Ho proposed backpropagation for adjusting the weights of a multilayer perceptron.
 - Backpropagation was rediscovered by Rumelhart, Hinton, and Williams in 1986.
- 1982: Hopfield Network
 - John Hopfield proposed a network with feedback connections that could store and retrieve patterns.
 - Hopfield networks were used for optimization problems and as content-addressable memories.
- 1990s: Support Vector Machines
 - Vapnik and colleagues introduced support vector machines (SVMs), a powerful algorithm for classification and regression.
 - SVMs were not neural networks, but they had a similar architecture and inspired research into kernel methods.
- 2010s: Deep Learning
 - Advances in computing power, big data, and new techniques like dropout and rectified linear units (ReLU) enabled the training of deep neural networks.
 - Deep learning has led to breakthroughs in image recognition, natural language processing, and more.
- 2021: GPT-3
 - OpenAI released GPT-3, a language model with 175 billion parameters that can perform a wide range of natural language tasks.
 - GPT-3 represents the current state-of-the-art in language modeling and highlights the continued progress of artificial neural networks.
- 2023: GPT-4
 - ... (fast)
 - ... (fast)

- Definition: the perceptron is a type of neural network that can learn to classify input data into different categories
- Learning algorithm: perceptron learning rule, which updates the weights of the network based on the error between the predicted and actual outputs
- Limitations: the perceptron can only learn linearly separable patterns
- Solution: the bilayer network, which consists of an input layer, a hidden layer, and an output layer, and can learn nonlinear patterns through the use of activation functions

Perceptron

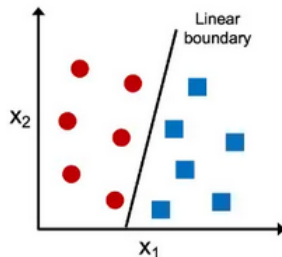
- Definition: the perceptron is a type of neural network that can learn to classify input data into different categories



Limitations of Perceptron: Linear and non-linear

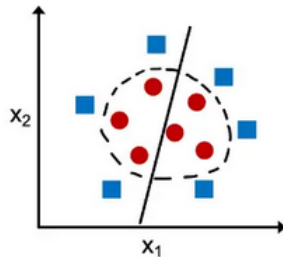
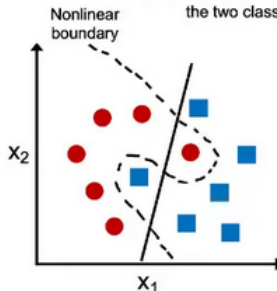
Linearly separable

A linear decision boundary that separates the two classes exists



Not linearly separable

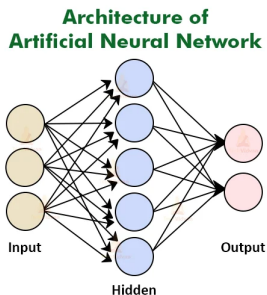
No linear decision boundary that separates the two classes perfectly exists



<https://vitalflux.com/how-know-data-linear-non-linear/>

Bilayer Neural Network Architecture

- Input Layer: Receives input data, which could be features extracted from a dataset or raw input data.
- Hidden Layer(s): Applies non-linear transformations to the input data.
- Output Layer: Produces the final output or prediction of the neural network.



<https://vitalflux.com/how-know-data-linear-non-linear/>

Perceptron Learning Rule

Perceptron Model

- The perceptron is a simple binary classification model.
- It takes a vector of input features $\mathbf{x} = (x_1, x_2, \dots, x_n)$ and produces an output y .
- The output is computed as:

$$y = f(\mathbf{w} \cdot \mathbf{x} + b)$$

where:

- $\mathbf{w} = (w_1, w_2, \dots, w_n)$ are the weights for the input features.
- b is the bias.
- $f(\cdot)$ is the activation function (e.g., step function).

Perceptron Learning Rule

- The perceptron learning rule is a supervised learning algorithm for updating the weights and bias.
- For a misclassified input \mathbf{x} with true label y_t and predicted label y , the update is:

$$\Delta w_i = \eta \cdot (y_t - y) \cdot x_i$$

$$\Delta b = \eta \cdot (y_t - y)$$

where:

- Δw_i is the update for weight w_i .
- Δb is the update for the bias b .
- η is the learning rate, a hyperparameter.
- The weights and bias are updated as:

$$w_i \leftarrow w_i + \Delta w_i$$

$$b \leftarrow b + \Delta b$$

Two-layer Perceptron as a Nonlinear Function Approximator

A two-layer perceptron, also known as a feedforward neural network, is a good choice for approximating nonlinear functions because:

- It can model complex nonlinear relationships between input and output variables.
- It can handle high-dimensional input data and extract meaningful features automatically.
- It can generalize well to unseen data if properly trained.
- It can be trained using backpropagation algorithm to optimize the weights and biases of the network.
- It can be regularized using techniques such as weight decay or dropout to prevent overfitting.

The two-layer perceptron consists of an input layer, one hidden layer, and an output layer. It is used as a function approximator and can approximate any continuous function.

- Input layer: x_1, x_2, \dots, x_n
- Hidden layer: h_1, h_2, \dots, h_m with activation function ϕ
- Output layer: y_1, y_2, \dots, y_k with activation function σ

Two-layer Perceptron and Universal Approximation Property

The two-layer perceptron has the **universal approximation property**:

- Let X be a closed set $X \subseteq \mathbb{R}^d$, a continuous function $f : X \rightarrow \mathbb{R}$,
- For any $\epsilon > 0$, there exists a two-layer perceptron with sigmoid activation functions and enough hidden neurons that approximates f to within ϵ over X .

The two-layer perceptron model is defined as follows:

$$h_i = \sigma(\sum_{j=1}^n w_{ij}^{(1)} x_j + b_i^{(1)}), y_k = \sigma(\sum_{i=1}^m w_{ki}^{(2)} h_i + b_k^{(2)})$$

where $\sigma(z) = \frac{1}{1+e^{-z}}$ is the sigmoid activation function, x_j are the input features, $w_{ij}^{(1)}$ are the weights connecting the input layer to the hidden layer, $b_i^{(1)}$ are the biases of the hidden layer, h_i are the hidden units, $w_{ki}^{(2)}$ are the weights connecting the hidden layer to the output layer, $b_k^{(2)}$ are the biases of the output layer, and y_k are the output predictions

Two-layer Perceptron: Universal Approximation Property

Given a closed set $X \subseteq \mathbb{R}^d$, a continuous function $f : X \rightarrow \mathbb{R}$, and $\epsilon > 0$, there exist a positive integer n and parameters θ such that the two-layer perceptron $g(x; \theta)$ satisfies:

$$|g(x; \theta) - f(x)| < \epsilon \quad \text{for all } x \in X.$$

where $g(x; \theta) = \sum_{i=1}^n \alpha_i \sigma(w_i^T x + b_i)$ is a weighted sum of n activation functions σ , parameterized by the weights w_i , biases b_i , and coefficients α_i .

Cost Function Definition - Reminder

Classical Square Measure

- The cost function, also known as the loss function or objective function, quantifies the difference between predicted and true labels.
- In regression problems, the classical square measure is commonly used as the cost function.
- The classical square measure is defined as:

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)})^2$$

where:

- $J(\theta)$ is the cost function.
- θ is the vector of model parameters.
- m is the number of training examples.
- $h_{\theta}(x^{(i)})$ is the predicted label for the i -th training example.
- $y^{(i)}$ is the true label for the i -th training example.
- The goal of training is to minimize the cost function $J(\theta)$ by finding the optimal values of the model parameters θ .

Optimization of Cost Function - Reminder

Methods for Optimization

- Optimizing the cost function is a key step in training machine learning models.
- Some common methods for optimization include:
 - **Gradient Descent (GD)**: A basic optimization algorithm that adjusts model parameters in the direction of the negative gradient of the cost function.
 - **Stochastic Gradient Descent (SGD)**: A variant of GD that updates model parameters based on a single training example at a time, making it more computationally efficient for large datasets.
 - **Mini-batch Gradient Descent**: A compromise between GD and SGD, where model parameters are updated based on a small batch of training examples.

Gradient Reverse Propagation

- Definition: the backpropagation algorithm, also known as reverse propagation of errors, is a method for training neural networks by minimizing a cost function
- Learning algorithm: the gradient descent algorithm, which iteratively adjusts the weights of the network to minimize the cost function
- Types of activation functions: sigmoid, ReLU, softmax
- Limitations: the gradient descent algorithm can get stuck in local minima or saddle points, and can be slow to converge

Backpropagation Algorithm

The backpropagation algorithm, also known as reverse propagation of errors, is a widely used method for training neural networks. It involves iteratively adjusting the weights of the network in order to minimize a cost function.

Main steps of the backpropagation algorithm:

- ➊ Forward pass: the input data is fed through the network, and the output is computed.
- ➋ Compute error: the difference between the predicted output and the actual output is computed, and the error is propagated backwards through the network.
- ➌ Update weights: the weights of the network are adjusted in the direction that minimizes the error, using an optimization algorithm such as gradient descent.
- ➍ Repeat: these steps are repeated until the network's performance on the training data is satisfactory.

Gradient Descent Algorithm

Gradient descent is a widely used optimization algorithm that is used to adjust the weights of the network during the backpropagation process. The basic idea is to iteratively adjust the weights in the direction that minimizes the cost function. Here are the main steps involved in the gradient descent algorithm:

- 1 Compute gradient: compute the gradient of the cost function with respect to the weights.
- 2 Update weights: adjust the weights in the direction of the negative gradient.
- 3 Repeat: these steps are repeated until the cost function is minimized.

There are several variations of gradient descent, including batch gradient descent, stochastic gradient descent, and mini-batch gradient descent. Each of these methods has its own advantages and disadvantages, and the choice of which one to use depends on the specific problem being addressed.

Gradient Reversal Propagation (GRP)

- Forward pass:
 - Input: x
 - Output: $y(x, \theta)$
 - Loss: $L(y(x, \theta), y_{true})$
- Backward pass:
 - Compute gradient of loss with respect to model parameters: $\nabla_{\theta} L(y(x, \theta), y_{true})$
- Reverse the gradient:
 - Reversed gradient: $-r \cdot \nabla_{\theta} L(y(x, \theta), y_{true})$
- Update the model parameters:
 - New parameters: $\theta \leftarrow \theta - \alpha \cdot (-r \cdot \nabla_{\theta} L(y(x, \theta), y_{true}))$

where:

- x : input to the network
- $y(x, \theta)$: output of the network given input x and model parameters θ
- L : loss function
- y_{true} : true output
- θ : model parameters
- $\nabla_{\theta} L$: gradient of the loss function with respect to model parameters θ
- r : reversal scalar
- α : learning rate.

Gradient Reversal Propagation (GRP) with Partial Differentiation

Forward pass: $y = f(x, \theta), \mathcal{L}(y, y_{true})$

Backward pass: $\frac{\partial \mathcal{L}}{\partial \theta} = \frac{\partial \mathcal{L}}{\partial y} \cdot \frac{\partial y}{\partial \theta}$

Reverse the gradient: $-r \cdot \frac{\partial \mathcal{L}}{\partial \theta}$

Update the model parameters: $\theta \leftarrow \theta - \alpha \cdot (-r \cdot \frac{\partial \mathcal{L}}{\partial \theta})$

where:

- x : input to the network
- y : output of the network given input x and model parameters θ
- \mathcal{L} : loss function
- y_{true} : true output
- θ : model parameters
- $\frac{\partial \mathcal{L}}{\partial \theta}$: partial derivative of the loss function with respect to model parameters θ
- $\frac{\partial \mathcal{L}}{\partial y}$: partial derivative of the loss function with respect to output y
- $\frac{\partial y}{\partial \theta}$: partial derivative of output y with respect to model parameters θ
- r : reversal scalar
- α : learning rate.

Backpropagation Algorithm - pseudocode

Algorithm Backpropagation Algorithm

Input: Training data $(x_i, y_i)_{i=1}^n$, learning rate η

Output: Trained model parameters W and b

Initialize model parameters W and b ; **repeat**

for $i = 1$ to n **do**

 Compute $z = W^T x_i + b$ and $a = \sigma(z)$

 Compute the error term $\delta^L = \nabla_a J \odot \sigma'(z)$

for $l = L - 1$ to 1 **do**

 Compute $\delta^l = ((W^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$

end

 Compute the gradient w.r.t. weights: $\nabla_W J = \delta^L a^{L-1T}$

 Compute the gradient w.r.t. biases: $\nabla_b J = \delta^L$

 Update parameters: $W \leftarrow W - \eta \nabla_W J$ and $b \leftarrow b - \eta \nabla_b J$

end

until convergence;

Note: In this algorithm, $\sigma(z)$ represents the activation function applied element-wise to the input z , \odot represents element-wise multiplication, and J represents the loss function. The superscript l denotes the layer index, and L denotes the output layer index. The gradient w.r.t. weights and biases are denoted by $\nabla_W J$ and $\nabla_b J$, respectively.

Activation Functions

Activation functions are a key component of neural networks, and they are used to introduce nonlinearity into the network. There are several commonly used activation functions, including:

- Sigmoid: $\sigma(z) = \frac{1}{1+e^{-z}}$
- ReLU (rectified linear unit): $f(z) = \max(0, z)$
- Softmax: $\sigma_j(z) = \frac{e^{z_j}}{\sum_{k=1}^K e^{z_k}}$

Each of these activation functions has its own advantages and disadvantages, and the choice of which one to use depends on the specific problem being addressed. For example, sigmoid and softmax are often used in multi-class classification problems, while ReLU is commonly used in image recognition tasks.

Limitations of Gradient Descent

While gradient descent is a powerful optimization algorithm, it has some limitations that can make it difficult to use in practice. Here are some of the main limitations:

- Local minima: gradient descent can get stuck in local minima, which can prevent it from finding the global minimum of the cost function.
- Saddle points: gradient descent can also get stuck in saddle points, which can slow down convergence and make it difficult to find the minimum of the cost function.
- Slow convergence: gradient descent can be slow to converge, especially if the cost function has many local minima or saddle points.

Techniques to mitigate limitations

There are several techniques that can be used to mitigate these limitations, including using different optimization algorithms (such as Adam or RMSprop), using different activation functions, and using techniques such as early stopping or regularization.

- **Momentum:** momentum is a technique that can be used to accelerate convergence and overcome local minima. The idea is to add a momentum term to the weight update rule, which allows the optimizer to continue moving in the same direction if the gradient is consistent across multiple iterations.
- **Learning rate scheduling:** learning rate scheduling is a technique that adjusts the learning rate of the optimizer during training. The idea is to start with a high learning rate and gradually decrease it over time, which can help the optimizer converge more quickly and avoid getting stuck in local minima.
- **Batch normalization:** batch normalization is a technique that can be used to speed up convergence and improve the stability of the training process. The idea is to normalize the inputs to each layer of the network, which can reduce the amount of covariate shift and help the optimizer converge more quickly.
- **Dropout:** dropout is a regularization technique that can be used to prevent overfitting. The idea is to randomly drop out some of the neurons in each layer during training, which can help prevent the network from relying too heavily on any one feature.

The choice of which techniques to use depends on the specific problem being addressed and the characteristics of the data.

- Definition: the process of training a neural network to perform a specific task, such as classification or regression
- Types of learning: supervised learning, unsupervised learning, reinforcement learning
- Applications: image recognition, natural language processing, speech recognition, robotics

Online vs. Offline Learning in Neural Networks

Online Learning

- Also known as *online gradient descent*.
- Update model parameters after each training sample.
- Formula for weight update:

$$w_{i+1} = w_i - \alpha \nabla L(y_i, f(x_i))$$

where:

- w_i and w_{i+1} are the weights at iteration i and $i + 1$, respectively.
- α is the learning rate.
- $\nabla L(y_i, f(x_i))$ is the gradient of the loss function L with respect to the predicted output $f(x_i)$.

Offline Learning

- Also known as *batch gradient descent*.
- Update model parameters after each epoch (complete pass through the training data).
- Formula for weight update:

$$w_{i+1} = w_i - \alpha \frac{1}{N} \sum_{j=1}^N \nabla L(y_j, f(x_j))$$

where:

- N is the number of training samples.
- $\frac{1}{N} \sum_{j=1}^N \nabla L(y_j, f(x_j))$ is the average gradient of the loss function L with respect to the predicted output $f(x_j)$ over all training samples.

Neural Network Weights Initialization

Importance of Initialization

- Initialization of weights and biases in a neural network is crucial for training success.
- Proper initialization can help avoid issues such as vanishing or exploding gradients.

Common Initialization Techniques

- Zero Initialization

$$w_{ij} = 0, \quad b_i = 0$$

- Random Initialization

$$w_{ij} \sim \text{Uniform}(-r, r), \quad b_i \sim \text{Uniform}(-r, r)$$

where:

- w_{ij} is the weight from neuron j in layer $l - 1$ to neuron i in layer l .
- b_i is the bias of neuron i in layer l .
- $\text{Uniform}(-r, r)$ represents random values drawn from a uniform distribution between $-r$ and r .

- Xavier Initialization

$$w_{ij} \sim \text{Gaussian}\left(0, \frac{1}{\sqrt{n_{l-1}}}\right), \quad b_i = 0$$

where:

- n_{l-1} is the number of neurons in layer $l - 1$.
- $\text{Gaussian}\left(0, \frac{1}{\sqrt{n_{l-1}}}\right)$ represents random values drawn from a Gaussian distribution with mean 0 and standard deviation $\frac{1}{\sqrt{n_{l-1}}}$.

Sigmoid Function

- The sigmoid function maps any real-valued number to a value between 0 and 1.
- It is often used as an activation function in neural networks for binary classification problems.

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

- Properties:
 - $\sigma(x)$ is continuous and differentiable
 - $\sigma(x)$ is monotonically increasing
 - $\sigma(x)$ saturates at large positive and negative values

ReLU Function

- The Rectified Linear Unit (ReLU) function maps any negative input to 0 and any positive input to itself.
- It is often used as an activation function in neural networks for image classification problems.

$$f(x) = \begin{cases} 0 & x < 0 \\ x & x \geq 0 \end{cases}$$

- Properties:
 - $f(x)$ is piecewise linear and non-differentiable at $x = 0$
 - $f(x)$ is computationally efficient
 - $f(x)$ can suffer from "dying ReLU" problem if too many units are "dead" (i.e., outputting 0)









Softmax Function

- The softmax function maps a vector of real-valued numbers to a vector of probabilities that sum to 1.
- It is often used as an activation function in neural networks for multi-class classification problems.

$$\sigma_j(z) = \frac{e^{z_j}}{\sum_{k=1}^K e^{z_k}}$$

- Properties:
 - $\sigma_j(z)$ is continuous and differentiable
 - $\sigma_j(z) \in [0, 1]$
 - $\sum_{j=1}^K \sigma_j(z) = 1$
 - $\sigma_j(z)$ represents the probability of the input vector belonging to the j th class in a multi-class classification problem.

Activation functions

Activation function	Equation	Example	1D Graph
Unit step (Heaviside)	$\phi(z) = \begin{cases} 0, & z < 0, \\ 0.5, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Sign (Signum)	$\phi(z) = \begin{cases} -1, & z < 0, \\ 0, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Linear	$\phi(z) = z$	Adaline, linear regression	
Piece-wise linear	$\phi(z) = \begin{cases} 1, & z \geq \frac{1}{2}, \\ z + \frac{1}{2}, & -\frac{1}{2} < z < \frac{1}{2}, \\ 0, & z \leq -\frac{1}{2}, \end{cases}$	Support vector machine	
Logistic (sigmoid)	$\phi(z) = \frac{1}{1 + e^{-z}}$	Logistic regression, Multi-layer NN	
Hyperbolic tangent	$\phi(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$	Multi-layer Neural Networks	
Rectifier, ReLU (Rectified Linear Unit)	$\phi(z) = \max(0, z)$	Multi-layer Neural Networks	
Rectifier, softplus	$\phi(z) = \ln(1 + e^z)$	Multi-layer Neural Networks	

Copyright © Sebastian Raschka 2016
(<http://sebastianraschka.com>)

Common Optimization Algorithms in NN

- Optimizers are algorithms or methods used to change the attributes of your neural network to reduce losses.
- Gradient Descent, SGD, Momentum, Adam, AdaGrad, and RMSProp are commonly used optimization algorithms.

Gradient Descent

- Most basic but commonly used.
- Adjusts model parameters in the direction of steepest descent.

SGD

- Variant of Gradient Descent.
- Updates model parameters more frequently.

Momentum

- Reduces high variance in SGD.
- Accelerates SGD in relevant direction.

Adam

- Combination of SGD with momentum and AdaGrad.
- Adapts learning rate based on first and second moments of gradients.

AdaGrad

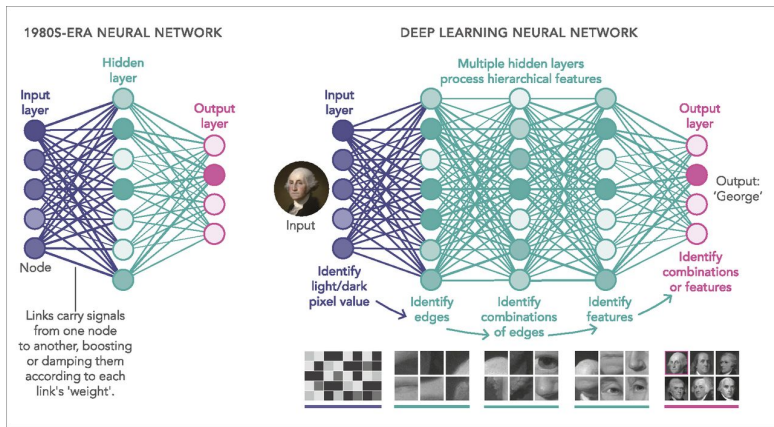
- Adaptive method for setting learning rate.
- Adapts learning rate based on historical gradient information.

RMSProp

- Modification of AdaGrad.
- Uses exponentially weighted moving average for gradient accumulation.

Deep learning, multilayer neural networks

While "multilayer neural network" typically refers to neural networks with multiple hidden layers, "deep learning neural network" generally refers to neural networks with many hidden layers, typically more than three, that have been shown to be capable of learning complex patterns and representations from large amounts of data.

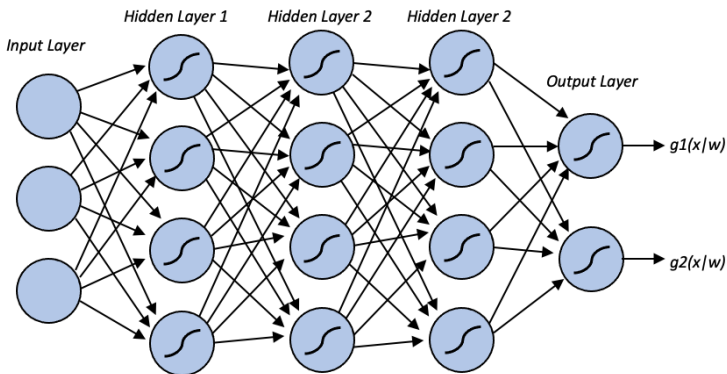


Neural Networks Architectures

- Feedforward Neural Networks (FFNN)
- Convolutional Neural Networks (CNN)
- Recurrent Neural Networks (RNN)
 - Long Short-Term Memory Networks (LSTM)
 - Gated Recurrent Unit Networks (GRU)
- Autoencoder Neural Networks
- Generative Adversarial Networks (GAN)

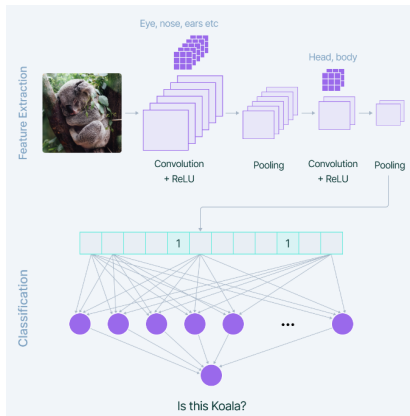
Feedforward Neural Networks (FFNNs)

- FFNNs are called "feedforward" because information flows in one direction, from input to output, without loops or cycles.
- They consist of multiple layers, including input layer, hidden layers, and output layer. Each layer contains multiple neurons that process input data and pass forward the output to the next layer. Activation functions are used in neurons to introduce non-linearity.



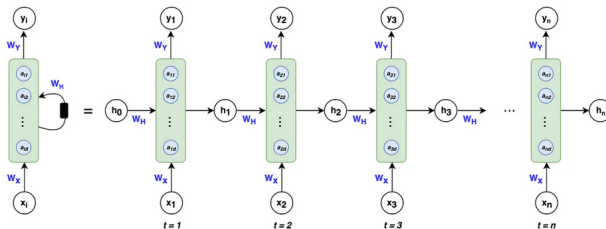
Convolutional Neural Networks (CNNs)

- CNNs are a type of deep learning model commonly used for image recognition and computer vision tasks. They are designed to automatically learn features from images.
- CNNs consist of multiple layers, including convolutional layers, pooling layers, and fully connected layers. **Convolutional layers** apply convolution operations to input images, capturing local patterns and features. **Pooling layers** downsample feature maps, reducing spatial dimensions and keeping important features. Fully connected layers are **traditional layers**, which make predictions based on the learned features.



Recurrent Neural Networks (RNNs)

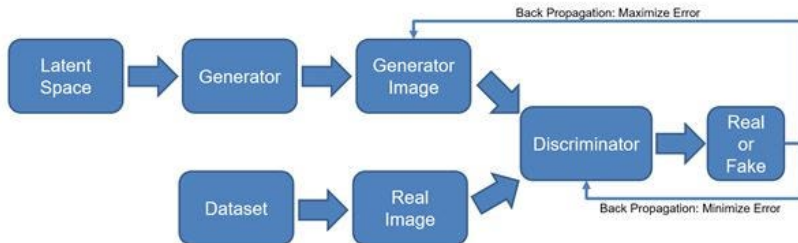
- RNNs are a type of artificial neural network designed for processing sequential data, such as time series or sequences of text. Unlike feedforward neural networks, RNNs have connections that form cycles, allowing them to capture temporal dependencies in data. RNNs maintain a hidden state that is updated at each time step and serves as a memory of past information.
- RNNs can have different types of cells, such as vanilla RNN cells, Long Short-Term Memory (LSTM) cells, and Gated Recurrent Units (GRUs). LSTM and GRU cells are designed to mitigate the vanishing gradient problem, which can occur in vanilla RNNs when training on long sequences.
- RNNs are widely used in natural language processing, speech recognition, and other tasks where sequential data is prevalent.



Left: Shorthand notation often used for RNNs, Right: Unfolded notation for RNNs

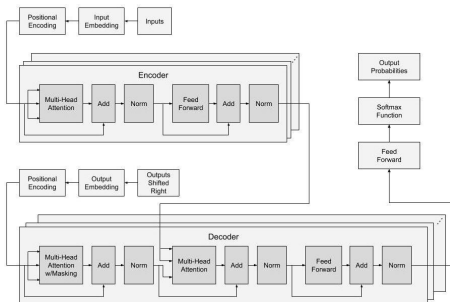
Generative Adversarial Networks (GANs)

- GANs are a type of generative model that can generate new data samples that are similar to the training data. They consist of two neural networks: a generator and a discriminator, which are trained together in a process called adversarial training. The generator learns to generate fake data samples, while the discriminator learns to distinguish between fake and real data samples.
- During training, the generator and discriminator are trained in opposition, with the goal of improving each other's performance. The generator tries to generate realistic samples to deceive the discriminator, while the discriminator tries to correctly classify fake and real samples. This adversarial process continues until the generated samples become indistinguishable from real samples, resulting in a highly realistic generative model.

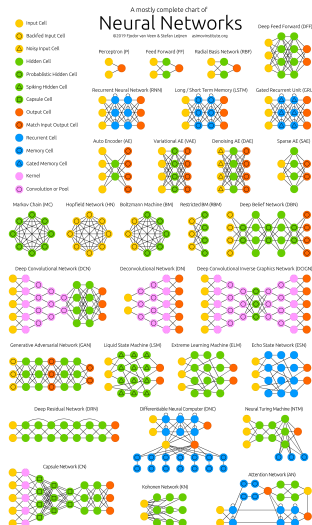


Transformer Neural Networks

- Transformers are a type of neural network architecture that revolutionized natural language processing (NLP) tasks. Unlike traditional recurrent neural networks (RNNs), Transformers rely on self-attention mechanisms, which allow them to process words in parallel, rather than sequentially. They use the self-attention mechanism to compute attention scores for each word in a sequence, which reflect the importance of that word in relation to other words in the sequence.
- The attention scores are used to weight the contributions of different words when computing the output of each word in the sequence. Transformers are highly parallelizable and can efficiently process long sequences, making them well-suited for tasks like machine translation, text generation, and sentiment analysis.



Overview of NN model architectures



Summary - Artificial Neural Networks

- 1 Introduction to Neural Networks
- 2 Perceptron and Bilayer Networks
- 3 Gradient Reverse Propagation
- 4 Network Learning
- 5 Optimization Algorithms for NN
- 6 NN architectures
- 7 Summary

- 1 S. J. Russell, P. Norvig, "Artificial Intelligence: A Modern Approach", Financial Times Prentice Hall, 2019.
- 2 T. Nield, "Essential Math for Data Science: Take Control of Your Data with Fundamental Linear Algebra, Probability, and Statistics", O'Reilly Media, 2022
- 3 A. Geron, "Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow", O'Reilly Media, 3rd ed., 2023
- 4 M. Flasiński, "Introduction to Artificial Intelligence", Springer Verlag, 2016
- 5 M. Muraszewicz, R. Nowak (ed.), "Sztuczna Inteligencja dla inżynierów", Oficyna Wydawnicza PW, 2022
- 6 J. Prateek, "Artificial Intelligence with Python", Packt 2017