Artificial Neural Networks And Deep Learning

Christian Rossi

Academic Year 2024-2025

Abstract

Neural networks have matured into flexible and powerful non-linear data-driven models, effectively tackling complex tasks in both science and engineering. The emergence of deep learning, which utilizes neural networks to learn optimal data representations alongside their corresponding models, has significantly advanced this paradigm.

In the course, we will explore various topics in depth. We will begin with the evolution from the Perceptron to modern neural networks, focusing on the feedforward architecture. The training of neural networks through backpropagation and algorithms like Adagrad and Adam will be covered, along with best practices to prevent overfitting, including cross-validation, stopping criteria, weight decay, dropout, and data resampling techniques.

The course will also delve into specific applications such as image classification using neural networks, and we will examine recurrent neural networks and related architectures like sparse neural autoencoders. Key theoretical concepts will be discussed, including the role of neural networks as universal approximation tools, and challenges like vanishing and exploding gradients.

We will introduce the deep learning paradigm, highlighting its distinctions from traditional machine learning methods. The architecture and breakthroughs of convolutional neural networks (CNNs) will be a focal point, including their training processes and data augmentation strategies.

Furthermore, we will cover structural learning and long-short term memory (LSTM) networks, exploring their applications in text and speech processing. Topics such as autoencoders, data embedding techniques like word2vec, and variational autoencoders will also be addressed.

Finally, we will discuss transfer learning with pre-trained deep models, examine extended models such as fully convolutional CNNs for image segmentation (e.g., U-Net) and object detection methods (e.g., R-CNN, YOLO), and explore generative models like generative adversarial networks (GANs).

Contents

1 Deep learning **1**
- 1.1 Introduction . . . 1
- 1.2 Deep learning . . . 1
- 1.3 Perceptron . . . 2
  - 1.3.1 Human neurons . . . 2
  - 1.3.2 Artificial neuron . . . 2
  - 1.3.3 Hebbian learning . . . 4

2 Feed Forward Neural Networks **7**
- 2.1 Introduction . . . 7
- 2.2 Activation functions . . . 8
  - 2.2.1 Output layer . . . 8
  - 2.2.2 Hidden layers . . . 9
  - 2.2.3 Alternative activation functions . . . 9
- 2.3 Training . . . 11
  - 2.3.1 Nonlinear optimization . . . 11
  - 2.3.2 Gradient descent . . . 12
  - 2.3.3 Gradient descent computation . . . 13
  - 2.3.4 Batch normalization . . . 15
  - 2.3.5 Weight initialization . . . 15
- 2.4 Loss function . . . 17
  - 2.4.1 Loss function selection . . . 20
  - 2.4.2 Perceptron loss function . . . 21
  - 2.4.3 Adaptive learning . . . 22
- 2.5 Overfitting . . . 23
  - 2.5.1 Cross validation . . . 24
  - 2.5.2 Early stopping . . . 24
  - 2.5.3 Weight decay . . . 25
  - 2.5.4 Dropout . . . 26

3 Convolutional Neural Networks **28**
- 3.1 Computer vision . . . 28
  - 3.1.1 Digital images . . . 28
  - 3.1.2 Local transformations . . . 28
- 3.2 Image classification . . . 29
  - 3.2.1 Linear classifier . . . 30
  - 3.2.2 Linear classifier for images . . . 30

3.2.3 Linear classifier training . . . 31
3.2.4 Image classification challenges . . . 31
3.2.5 Feature extraction . . . 32
3.3 Convolutional Neural Network . . . 33
3.3.1 Convolutional layers . . . 34
3.3.2 Activation layer . . . 35
3.3.3 Pooling layer . . . 36
3.3.4 Final architecture . . . 37
3.3.5 Convolutional and dense layers . . . 37
3.3.6 Receptive field . . . 38
3.4 Training . . . 39
3.4.1 Image augmentation . . . 39
3.4.2 Transfer learning . . . 40
3.5 Architectures . . . 40
3.5.1 AlexNet . . . 41
3.5.2 VGG16 . . . 41
3.5.3 Networks in Networks . . . 41
3.5.4 InceptionNet . . . 42
3.5.5 ResNet . . . 43
3.5.6 MobileNet . . . 44
3.5.7 Latest architectures . . . 45

CHAPTER 1

Deep learning

1.1 Introduction

Definition (*Machine learning*)**.** A computer program is considered to learn from experience E with respect to a specific class of tasks T and a performance measure P if its performance at tasks in T, as measured by P, improves with experience E.

Given a dataset $\mathcal{D} = \{x_1, x_2, \ldots, x_N\}$, machine learning can be broadly categorized into three types:

- *Supervised learning*: in this type of learning, the model is provided with desired outputs $\{t_1, t_2, \ldots, t_N\}$ and learns to produce the correct output for new input data. The primary tasks in supervised learning are:
  - *Classification*: the model is trained on a labeled dataset and returns a label for new data.
  - *Regression*: the model is trained on a dataset with numerical values and returns a number as the output.
- *Unsupervised learning*: here, the model identifies patterns and regularities within the dataset $\mathcal{D}$ without being provided with explicit labels. The main task in unsupervised learning is clustering, in which the model groups similar data elements based on inherent similarities within the dataset.
- *Reinforcement learning*: in this approach, the model interacts with the environment by performing actions $\{a_1, a_2, \ldots, a_N\}$ and receives rewards $\{r_1, r_2, \ldots, r_N\}$ in return. The model learns to maximize cumulative rewards over time by adjusting its actions.

1.2 Deep learning

Deep learning, a subset of machine learning, focuses on utilizing large datasets and substantial computational power to automatically learn data representations. In certain cases, traditional classification may fail due to the presence of irrelevant or redundant features in the dataset. Deep learning addresses this issue by learning optimal features directly from the data, which are

then used by machine learning algorithms to perform more accurate classifications. Essentially, deep learning involves learning how to represent data in a way that improves the performance of machine learning models.

1.3 Perceptron

In the 1940s, computers were already proficient at executing tasks exactly as programmed and performing arithmetic operations with impressive speed. However, researchers envisioned machines that could do much more. They wanted computers that could handle noisy data, interact directly with their environment, function in a massively parallel and fault-tolerant way, and adapt to changing circumstances. Their quest was for a new computational model, one that could surpass the constraints of the Von Neumann Machine.

1.3.1 Human neurons

The human brain contains an enormous number of computing units, with approximately 100 billion neurons, each connected to around 7,000 other neurons through synapses. In adults, this results in a total of 100 to 500 trillion synaptic connections, while in a three-year-old child, this number can reach up to 1 quadrillion synapses.

Information in the brain is transmitted through chemical processes. Dendrites gather signals from synapses, which can be either inhibitory or excitatory. When the cumulative charge reaches a certain threshold, the neuron fires, releasing the charge.

The brain's computational model is characterized by its distributed nature among simple, non-linear units, its redundancy which ensures fault tolerance, and its intrinsic parallelism. The perceptron, a computational model inspired by the brain, reflects these principles.

1.3.2 Artificial neuron

The mathematical model of a neuron is represented as follows:

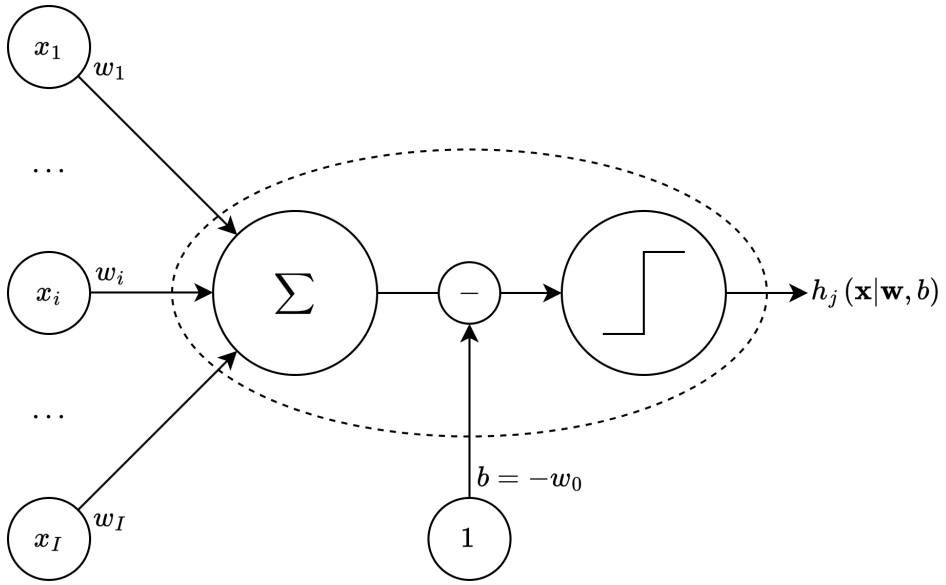


Figure 1.1: Artificial neuron

In this model, the output function $h_j(\mathbf{x}|\mathbf{w}, b)$ is defined as:

$$h_j(\mathbf{x}|\mathbf{w}, b) = h_j\left(\sum_{i=1}^{I} w_i x_i - b\right) = h_j\left(\sum_{i=0}^{I} w_i x_i\right) = h_j\left(\mathbf{w}^T\mathbf{x}\right)$$

The function used in an artificial neuron can either be a step function, with values ranging from 0 to 1, or a sine function, with values ranging from -1 to 1.

History Several researchers were actively investigating models for the brain during the mid-20th century. In 1943, Warren McCulloch and Walter Pitts proposed the threshold logic unit, where the activation function was a threshold unit, equivalent to the Heaviside step function. A few years later, in 1957, Frank Rosenblatt developed the first Perceptron, with weights encoded in potentiometers, and weight adjustments during learning were performed by electric motors. By 1960, Bernard Widrow introduced a significant advancement by representing the threshold value as a bias term in the ADALINE (Adaptive Linear Neuron or later, Adaptive Linear Element).

Example:
Consider a neuron designed to implement the OR operation:

x_0	x_1	x_2	**OR**
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

The corresponding neuron is illustrated below:

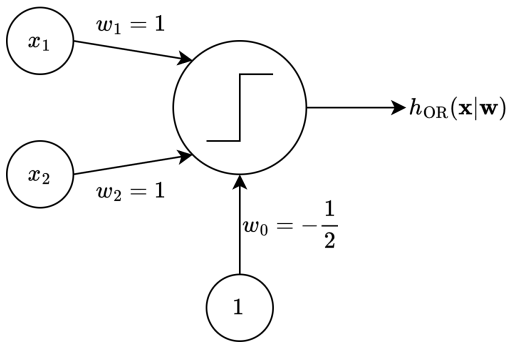


Figure 1.2: OR artificial neuron

The output function for this neuron is defined as:

$$h_{\text{OR}}(w_0 + w_1x_1 + w_2x_2) = h_{\text{OR}}\left(-\frac{1}{2} + x_1 + x_2\right) = \begin{cases} 1 & \text{if } \left(-\frac{1}{2} + x_1 + x_2\right) > 0 \\ 0 & \text{otherwise} \end{cases}$$

Now, consider a neuron designed to implement the AND operation:

x_0	x_1	x_2	**AND**
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

The corresponding neuron is shown below:

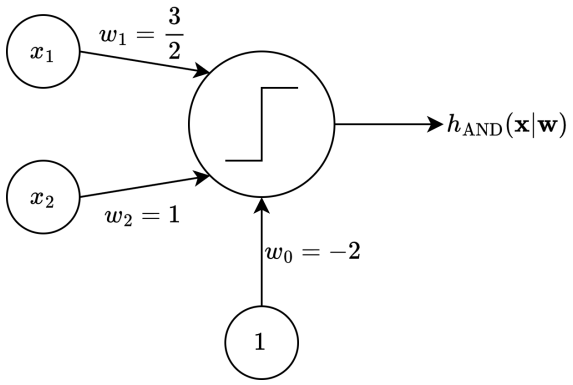


Figure 1.3: AND artificial neuron

The output function for this neuron is given by:

$$h_{\text{AND}}(w_0 + w_1x_1 + w_2x_2) = h_{\text{AND}}\left(-2 + \frac{3}{2}x_1 + x_2\right) = \begin{cases} 1 & \text{if } \left(-2 + \frac{3}{2}x_1 + x_2\right) > 0 \\ 0 & \text{otherwise} \end{cases}$$

1.3.3 Hebbian learning

The strength of a synapse increases based on the simultaneous activation of the corresponding input and the desired target. Hebbian learning can be summarized as follows:

1. Begin with a random initialization of the weights.
2. Adjust the weights for each sample individually (online learning), and only when the sample is not correctly predicted.

Mathematically, this is expressed as:

$$\begin{cases} w_i^{k+1} = w_i^k + \Delta w_i^k \\ \Delta w_i^k = \eta x_i^k t^k \end{cases} \implies w_i^{k+1} = w_i^k + \eta x_i^k t^k$$

Here, η represents the leraning rate, x_i^k is the i-th input to the perceptron at time k, and t^k is the desired output at time k.

Example:
We aim to learn the weights necessary to implement the OR operator with a sinusoidal output. The modified OR truth table is as follows:

x_0	x_1	x_2	**OR**
1	-1	-1	-1
1	-1	1	1
1	1	-1	1
1	1	1	1

We begin with random weights:

$$\mathbf{w} = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}$$

The learning rate is set to $\eta = \frac{1}{2}$. The output function is defined as:

$$h(\mathbf{w}^T\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{w}^T\mathbf{x} > 0 \\ 0 & \text{if } \mathbf{w}^T\mathbf{x} = 0 \\ -1 & \text{if } \mathbf{w}^T\mathbf{x} < 0 \end{cases}$$

The training involves iterating through the data records and adjusting the weights for incorrectly classified samples until all records are correctly predicted.

Starting from the first row, we have:

$$y_{\text{first row}} = x_0 w_0 + x_1 w_1 + x_2 w_2 = 1 \cdot 0 + (-1) \cdot 0 + (-1) \cdot 0 = 0$$

This does not match the expected output of -1. We adjust the weights:

$$w_0^{\text{new}} = w_0 + \eta x_0 y = 0 + \frac{1}{2} \cdot 1 \cdot (-1) = -\frac{1}{2}$$

$$w_1^{\text{new}} = w_1 + \eta x_1 y = 0 + \frac{1}{2} \cdot (-1) \cdot (-1) = \frac{1}{2}$$

$$w_2^{\text{new}} = w_2 + \eta x_2 y = 0 + \frac{1}{2} \cdot (-1) \cdot (-1) = \frac{1}{2}$$

Now, the weights vector is:

$$\mathbf{w} = \begin{bmatrix} -\frac{1}{2} & \frac{1}{2} & \frac{1}{2} \end{bmatrix}$$

For the second row, we have:

$$y_{\text{second row}} x_0 w_0 + x_1 w_1 + x_2 w_2 = 1 \cdot \left(-\frac{1}{2}\right) + (-1) \cdot \frac{1}{2} + 1 \cdot \frac{1}{2} = -\frac{1}{2}$$

This does not match the expected output of 1. We adjust the weights:

$$w_0^{\text{new}} = w_0 + \eta x_0 y = \left(-\frac{1}{2}\right) + \frac{1}{2} \cdot 1 \cdot 1 = 0$$

$$w_1^{\text{new}} = w_1 + \eta x_1 y = \frac{1}{2} + \frac{1}{2} \cdot (-1) \cdot 1 = 0$$

$$w_2^{\text{new}} = w_2 + \eta x_2 y = \frac{1}{2} + \frac{1}{2} \cdot 1 \cdot 1 = 1$$

Now, the weights vector is:

$$\mathbf{w} = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}$$

For the third row, we have:

$$y_{\text{third row}} x_0 w_0 + x_1 w_1 + x_2 w_2 = 1 \cdot 0 + 1 \cdot 0 + (-1) \cdot 1 = -1$$

This does not match the expected output of 1. We adjust the weights:

$$w_0^{\text{new}} = w_0 + \eta x_0 y = 0 + \frac{1}{2} \cdot 1 \cdot 1 = \frac{1}{2}$$

$$w_1^{\text{new}} = w_1 + \eta x_1 y = 0 + \frac{1}{2} \cdot 1 \cdot 1 = \frac{1}{2}$$

$$w_2^{\text{new}} = w_2 + \eta x_2 y = 1 + \frac{1}{2} \cdot (-1) \cdot 1 = \frac{1}{2}$$

Now, the weights vector is:

$$\mathbf{w} = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \end{bmatrix}$$

For the third row, we have:

$$y_{\text{fourth row}} x_0 w_0 + x_1 w_1 + x_2 w_2 = 1 \cdot \frac{1}{2} + 1 \cdot \frac{1}{2} + 1 \cdot \frac{1}{2} = \frac{3}{2}$$

This matches the expected output of 1.

After verifying the outputs for all rows, we recognize that further iterations (epochs) are needed for full convergence. We repeat the training until all records produce the desired outputs.

After multiple epochs, the final weights vector is:

$$\mathbf{w} = \begin{bmatrix} -\frac{1}{2} & 1 & 1 \end{bmatrix}$$

The number of epochs required depends on both the initialization of the weights and the order in which the data is presented.

A perceptron computes a weighted sum and returns the sign (thresholding) of the result:

$$h_j(\mathbf{x}|\mathbf{w}) = h_j \left(\sum_{i=0}^{I} w_i x_i \right) = \text{Sign}(w_0 + w_1 x_1 + \cdots + w_I x_I)$$

This forms a linear classifier, where the decision boundary is represented by the hyperplane:

$$w_0 + w_1 x_1 + \cdots + w_I x_I = 0$$

The linear boundary explains how the perceptron implements Boolean operators. However, if the dataset does not have a linearly separable boundary, the perceptron fails to work. In such cases, alternative approaches are needed, including non-linear boundaries or different input representations. This concept forms the basis for Multi-Layer Perceptrons (MLPs).

CHAPTER 2

Feed Forward Neural Networks

2.1 Introduction

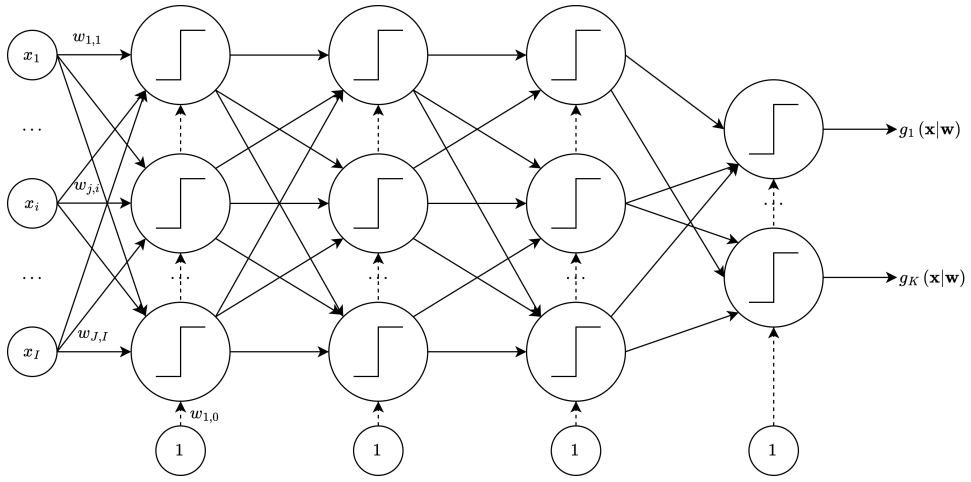


Figure 2.1: Multi-Layer Perceptron architecture

A Multi-Layer Perceptron (MLP) is a type of feedforward neural network (FFNN) that consists of multiple layers of nodes, each layer being fully connected to the next. The MLP architecture is composed of three primary components:

- *Input layer*: this layer receives the input data and passes it to the subsequent layers. The size of this layer depends on the specific problem and the input features.
- *Hidden layers*: these intermediate layers perform the actual computations, transforming the input into more abstract representations. The number of hidden layers and the number of neurons in each hidden layer are determined through hyperparameter tuning, which is often based on a trial-and-error approach.
- *Output layer*: this layer produces the final output of the network, which could be a prediction, classification, or another result. Its size depends on the nature of the problem, such as the number of classes in classification tasks.

The MLP is inherently a non-linear model, characterized by the number of neurons in each layer, the choice of activation functions, and the values of the connection weights. The connections between layers are represented by weight matrices, denoted as $W^{(l)} = \{w_{ij}^{(l)}\}$, where l is the layer index. If a layer has J nodes and receives I inputs, the corresponding weight matrix has dimensions $J \times (I + 1)$ accounting for the bias term.

The output of each neuron depends solely on the outputs from the previous layer, allowing for forward propagation of information. The learning of weights in an FFNN is achieved through a technique known as backpropagation, which iteratively adjusts the weights to minimize the error in the network's predictions.

2.2 Activation functions

Activation functions play a critical role in neural networks by introducing non-linearity into the model. Common activation functions include:

- *Linear*: $g(a) = a$ with a derivative of $g'(a) = 1$.
- *Sigmoid*: $g(a) = \dfrac{1}{1 + e^{-a}}$ with a derivative of $g(a)(1 - g(a))$. This function is widely used due to its simplicity and ability to model probabilities.
- *Hyperbolic tangent* (Tanh): $g(a) = \frac{e^a - e^{-a}}{e^a + e^{-a}}$ with a derivative of $1 - g(a)^2$. This function is often preferred for hidden layers as it outputs values in the range $[-1, 1]$, centering the data.

The choice of the activation function is a design decision, influenced by the nature of the task and the structure of the network.

2.2.1 Output layer

The activation function for the output layer depends on the type of problem being addressed:

- *Regression*: in regression tasks, where the output spans the real number domain $\mathbb{R}$, a linear activation function is typically used for the output neuron.
- *Binary classification*: the choice of activation depends on the coding of the class labels:
  1. For classes coded as $\Omega_0 = -1$, $\Omega_1 = 1$, a Tanh activation function is appropriate.
  2. For classes coded as $\Omega_0 = 0$, $\Omega_1 = 1$, a Sigmoid activation function is commonly used, as it can be interpreted as representing the posterior probability of a class.
- *Multi-class classification*: for problems with K classes, the output layer contains K neurons, one for each class. The classes are typically encoded using one-hot encoding, e.g., $\Omega_0 = \begin{bmatrix}0 & 0 & 1\end{bmatrix}$, $\Omega_1 = \begin{bmatrix}0 & 1 & 0\end{bmatrix}$, and $\Omega_2 = \begin{bmatrix}1 & 0 & 0\end{bmatrix}$. The output neurons utilize a softmax activation function:

$$y_k = \frac{e^{z_k}}{\sum_k e^{z_k}} = \frac{e^{\sum_j w_{kj} h_j (\sum_i^I w_{ji} x_i)}}{\sum_{k=1}^K e^{\sum_j w_{kj} h_j (\sum_i^I w_{ji} x_i)}}$$

  Here, z_k is the activation value of the k-th output neuron. The softmax function normalizes the output vector, providing class probabilities.

2.2.2 Hidden layers

For the hidden layers, activation functions such as the Sigmoid or Tanh are commonly used. These functions introduce non-linearity, allowing the network to model complex patterns in the data.

Theorem 2.2.1. *A single hidden layer feedforward neural network with S-shaped activation functions (such as Sigmoid or Tanh) can approximate any measurable function to any desired degree of accuracy on a compact set.*

This theorem implies that a single hidden layer can theoretically represent any function, though it does not guarantee that the learning algorithm will find the necessary weights. In practice, an excessively large number of hidden units may be required, and the network may struggle to generalize, particularly if overfitting occurs. However, for classification tasks, typically only one additional hidden layer is needed to achieve satisfactory performance.

2.2.3 Alternative activation functions

Activation functions like sigmoid or hyperbolic tangent tend to saturate, meaning their gradients become close to zero, or at least less than one, in many cases. Since backpropagation relies on gradient multiplication, this saturation causes gradients to vanish as they propagate backward through the network. This leads to the well-known vanishing gradient problem, where learning becomes extremely slow or even impossible, especially in deep networks. Although this problem is particularly evident in Recurrent Neural Networks, it also affects deep feed-forward networks, and for a long time, it was a significant obstacle to neural network training.

A popular solution to this issue is the Rectified Linear Unit (ReLU) activation function, defined as:

$$g(a) = \mathrm{ReLU}(a = \max(0, a))$$

Its derivative is:

$$g'(a) = 1_{a>0}$$

ReLU has several notable advantages:

- *Faster convergence*: stochastic Gradient Descent converges approximately six times faster with ReLU compared to sigmoid or tanh activations.
- *Sparse activation*: since only a portion of the hidden units are activated (i.e., those with positive input), this leads to a more efficient representation.
- *Efficient gradient propagation*: ReLU avoids both the vanishing and exploding gradient problems, making it easier to train deep networks.
- *Efficient computation*: ReLU is computationally efficient, as it only involves simple thresholding at zero.
- *Scale-invariance*: the output of ReLU is invariant to the scale of its input, meaning $\max(0, xa) = a \max(0, x)$.

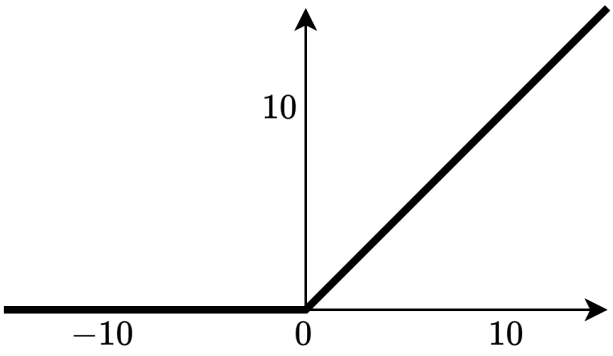


Figure 2.2: Rectified Linear Unit

Despite its advantages, ReLU has some potential downsides:

- *Non-differentiability at zero*: while ReLU is non-differentiable at zero, it is differentiable everywhere else. This non-differentiability rarely causes significant issues in practice.
- *Non-zero centered output*: ReLU outputs are non-zero-centered, which can lead to imbalanced updates when using gradient descent.
- *Unbounded output*: since ReLU does not cap its output, the activations can grow very large, potentially leading to exploding gradients under high learning rates.
- *Dying neurons*: a key issue with ReLU is the dying neuron problem, where neurons can get stuck with negative outputs for all inputs. When this happens, their gradients become zero, effectively killing those neurons, as they stop updating during training.

ReLU variants Several variants of ReLU have been developed to address some of its limitations:

- *Leaky ReLU*: this variant addresses the dying ReLU problem by allowing a small, non-zero gradient for negative input values:

$$f(x) = \begin{cases} x & \text{if } x \geq 0 \\ 0.01x & \text{otherwise} \end{cases}$$

  By using a small slope for negative inputs, Leaky ReLU ensures that neurons are less likely to die.

- *Exponential Linear Unit* (ELU): ELU aims to bring mean activations closer to zero, which can accelerate learning. It introduces an alpha parameter that controls the saturation for negative values:

$$f(x) = \begin{cases} x & \text{if } x \geq 0 \\ \alpha(e^x - 1) & \text{otherwise} \end{cases}$$

  The α parameter is typically tuned by hand, and ELU provides smoother, more balanced outputs.

2.3 Training

Training a neural network involves learning a set of parameters, such as weights $\mathbf{w}$, that allow the model $y(x_n|\mathbf{w})$ approximate the target t_n as closely as possible, given a training set $\mathcal{D} = \{\langle x_1, t_1\rangle, \ldots, \langle x_N, t_N\rangle\}$ we want to find model parameters such that for new data $y_n(x_n|\theta) \sim t_n$. This process can be viewed as finding parameters that generalize well to new data, such that $g(x_n|\mathbf{w}) \sim t_n$.

In regression and classification tasks, this goal is typically achieved by minimizing the error between the predicted outputs and the true labels. For a neural network, the error is often represented as the Sum of Squared Errors (SSE):

$$E(\mathbf{w})_{\text{SSE}} = \sum_{n}^{N} (t_n - g(x_n|\mathbf{w}))^2$$

Here, the SSE represents the error function, and for a feedforward neural network, this error is a non-linear function of the weights, making the optimization process more challenging.

2.3.1 Nonlinear optimization

To minimize a generic error function $J(\mathbf{w})$, we rely on optimization techniques. The goal is to find the weights $\mathbf{w}$ that minimize the error by solving:

$$\frac{\partial J(\mathbf{w})}{\partial \mathbf{w}} = 0$$

However, for neural networks, closed-form solutions are generally not available due to the non-linearity of the model. Instead, we employ iterative methods like gradient descent, which adjusts the weights incrementally in the direction that reduces the error. The steps for gradient descent are as follows:

1. Initialize the weights $\mathbf{w}$ to small random values.
2. Iterate until convergence:
$$w^{k+1} = w^k - \eta \frac{\partial J(\mathbf{w})}{\partial \mathbf{w}}\bigg|_{w^k}$$
Here, η is the learning rate, controlling the step size in each iteration.

In cases where the error function has multiple local minima, the final solution depends on the initial starting point. To address this, we can introduce a momentum term that helps the optimization process avoid being trapped in local minima:

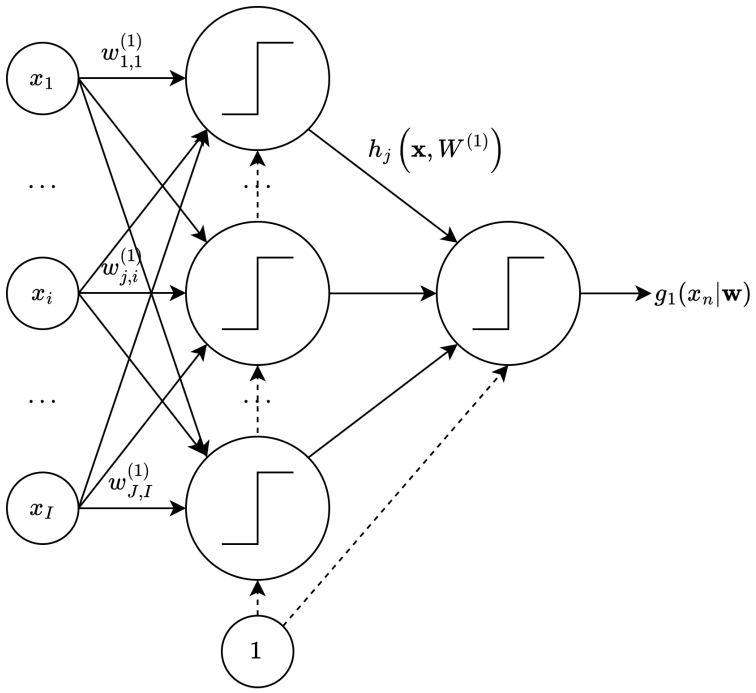
$$w^{k+1} = w^k - \eta \frac{\partial E(\mathbf{w})}{\partial \mathbf{w}}\bigg|_{w^k} - \alpha \frac{\partial E(\mathbf{w})}{\partial \mathbf{w}}\bigg|_{w^k}$$

Here, α represents the momentum coefficient, which encourages the optimization to keep moving in the same direction, effectively smoothing out oscillations and escaping shallow local minima.

Multiple restarts To improve the likelihood of finding the global minimum, especially in complex non-convex error surfaces, multiple restarts of the optimization from different random initializations can be used. This increases the chances of converging to a better solution by exploring various regions of the parameter space.

2.3.2 Gradient descent

Example:
Consider the following feed-forward neural network:



The output of the network is defined as:

$$g_1(x_n|\mathbf{w}) = g_1\left(\sum_{j=0}^{J} w^{(2)}_{1,j} h_j\left(\sum_{i=0}^{I} w^{(1)}_{j,i} x_{i,n}\right)\right)$$

Here, h_j represents the activation function of the hidden neurons, and $w^{(1)}_{i,j}$ and $w^{(2)}_{i,j}$ are the weights of the first and second layers, respectively.

We aim to minimize the sum of squared errors (SSE) between the predicted output and the target values:

$$E(\mathbf{w}) = \sum_{n=1}^{N} \left(t_n - g_1(x_n|\mathbf{w})\right)^2$$

Let's compute the weight update for $w^{(1)}_{3,5}$ using gradient descent. This weight corresponds to the connection between the 5th input and the 3rd hidden neuron. After calculating the derivative of the error function with respect to $w^{(1)}_{3,5}$, we obtain the following update rule:

$$\frac{\partial E(\mathbf{w})}{\partial w^{(1)}_{3,5}} = -2\sum_{n}^{N}(t_n - g_1(x_n, \mathbf{w}))g_1'(x_n, \mathbf{w})w^{(2)}_{1,3}h_3'\left(\sum_{i=0}^{I} w^{(1)}_{3,1}x_{i,n}\right)x_{5,n}$$

This expression includes the derivative of the output function g_1', the weight $w^{(2)}_{1,3}$ and the derivative of the hidden neuron activation function h_3'

In practice, using all data points for weight updates (i.e., batch gradient descent) can be computationally expensive, especially for large datasets. The gradient of the error function for batch gradient descent is given by:

$$\frac{\partial E(\mathbf{w})}{\partial \mathbf{w}} = \frac{1}{N} \sum_{n}^{N} \frac{\partial E(x_n, \mathbf{w})}{\partial \mathbf{w}}$$

However, this can be inefficient, so instead, we can use stochastic gradient descent (SGD), where the gradient is computed using a single sample at each iteration:

$$\frac{\partial E(\mathbf{w})}{\partial \mathbf{w}} \approx \frac{\partial E_{\text{SGD}}(\mathbf{w})}{\partial \mathbf{w}} = \frac{\partial E(x_n, \mathbf{w})}{\partial \mathbf{w}}$$

SGD is faster and unbiased but introduces high variance in the updates, which can cause the optimization process to oscillate.

A middle ground between batch gradient descent and SGD is mini-batch gradient descent, which uses a subset of samples (mini-batch) to compute the gradient:

$$\frac{\partial E(\mathbf{w})}{\partial \mathbf{w}} \approx \frac{\partial E_{\text{MB}}(\mathbf{w})}{\partial \mathbf{w}} = \frac{1}{M} \sum_{n \in \text{minibatch}}^{M<N} \frac{\partial E(x_n, \mathbf{w})}{\partial \mathbf{w}}$$

This approach provides a good balance between the computation cost and the variance of the updates, allowing for faster convergence while maintaining stability.

2.3.3 Gradient descent computation

The gradient descent process can be computed automatically from the structure of the neural network using backpropagation. This method allows for efficient weight updates that can be performed in parallel and locally, requiring just two passes through the network.

Let x be a real number, and consider two functions $f : \mathbb{R} \to \mathbb{R}$ and g: $\mathbb{R} \to \mathbb{R}$. Now, define the composite function $z = f(g(x)) = f(y)$, where $y = g(x)$. Using the chain rule, the derivative of z with respect to x is:

$$\frac{dz}{dx} = \frac{dz}{dy} = \frac{dy}{dx} = f'(y)g'(x) = f'(g(x))g'(x)$$

This concept extends naturally to backpropagation in neural networks. For example, consider the weight update for the weight $w_{j,i}^{(1)}$. Using the chain rule, we can express the partial derivative of the error function E with respect to $w_{j,i}^{(1)}$ as:

$$\begin{aligned}
\frac{\partial E(w_{j,i}^{(1)})}{\partial w_{j,i}^{(1)}} &= \underbrace{-2 \sum_{n}^{N} (t_n - g_1(x_n, \mathbf{w}))}_{\frac{\partial E}{\partial g(x_n, \mathbf{w})}} \underbrace{g_1'(x_n, \mathbf{w})}_{\frac{\partial g(x_n, \mathbf{w})}{\partial w_{1,j}^{(2)} h_j(\cdot)}} \underbrace{w_{1,j}^{(2)}}_{\frac{\partial w_{1,j}^{(2)} h_j(\cdot)}{\partial h_j(\cdot)}} \underbrace{h_j' \left(\sum_{i=0}^{I} w_{j,i}^{(1)} x_{i,n} \right)}_{\frac{\partial h_j(\cdot)}{\partial w_{j,i}^{(1)} x_i}} \underbrace{x_i}_{\frac{\partial w_{j,i}^{(1)} x_i}{\partial w_{j,i}^{(1)}}} \\
&= \frac{\partial E}{\partial g(x_n, \mathbf{w})} \cdot \frac{\partial g(x_n, \mathbf{w})}{\partial w_{1,j}^{(2)} h_j(\cdot)} \cdot \frac{\partial w_{1,j}^{(2)} h_j(\cdot)}{\partial h_j(\cdot)} \cdot \frac{\partial h_j(\cdot)}{\partial w_{j,i}^{(1)} x_i} \cdot \frac{\partial w_{j,i}^{(1)} x_i}{\partial w_{j,i}^{(1)}}
\end{aligned}$$

The gradient descent can be computed efficiently using the forward-backward pass strategy:

1. *Forward pass*: during the forward pass, the input propagates through the network to compute the output of each neuron. The local derivatives for each neuron (dependent only on its immediate inputs) are also computed. These computations do not depend on the other neurons in the network, making it possible to store this information locally.

2. *Backward pass*: in the backward pass, the stored values from the forward pass are used to propagate the gradients back through the network. This involves computing the partial derivatives of the error with respect to each weight and updating them accordingly using the chain rule.

By separating the forward and backward computations, if any part of the network, such as the error function, changes, only the relevant parts need to be recomputed. This flexibility allows for a more efficient calculation of the gradients and weight updates.

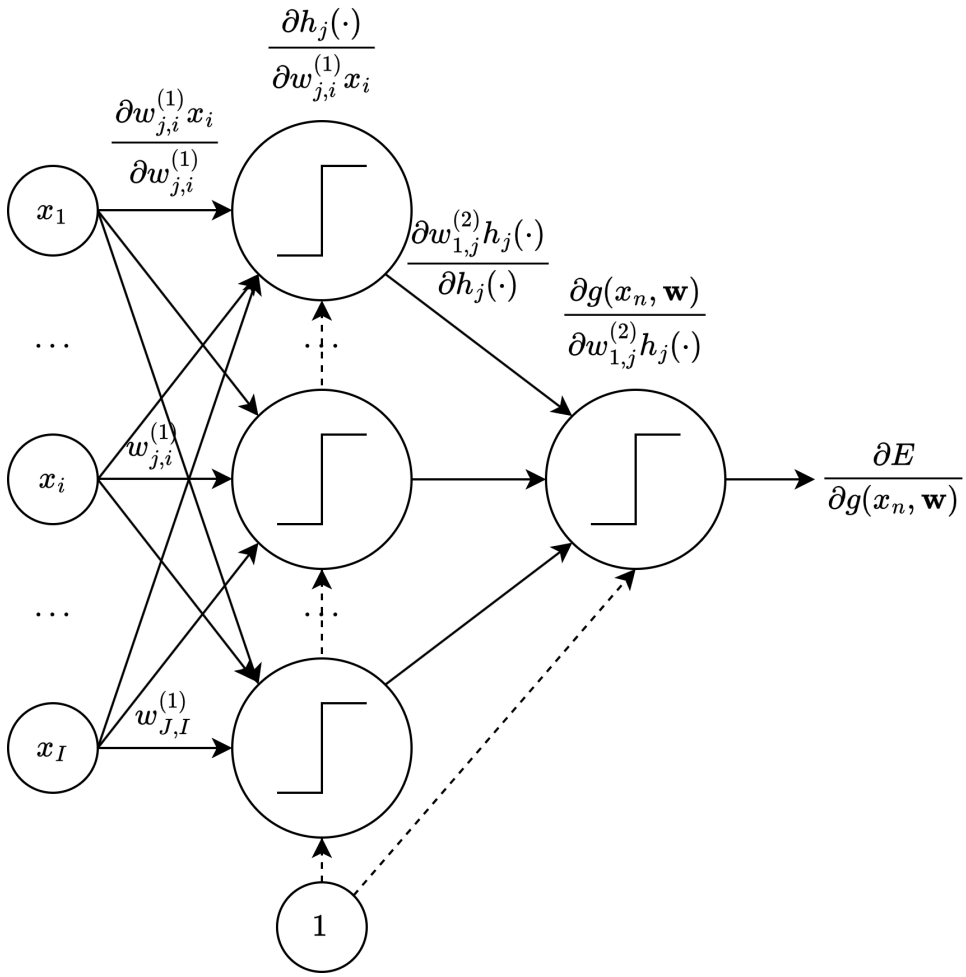


Figure 2.3: Forward and backward passes in Neural Networks

This approach allows for a systematic and parallelizable way of calculating the gradient, minimizing the computation needed for each update and ensuring that the network can be trained efficiently.

2.3.4 Batch normalization

To achieve faster convergence in neural networks, it is beneficial for the inputs to be whitened, meaning they should have a zero mean and unit variance while also being uncorrelated. This practice helps mitigate covariate shift. However, in addition to external covariate shifts, networks can experience internal covariate shifts, where the distribution of each layer's inputs changes during training. Batch normalization is a powerful technique to address this issue.

Batch normalization normalizes the activations of each layer, forcing them to take on values that approximate a unit Gaussian distribution at the start of training. It is typically implemented as follows:

- A BatchNorm layer is added after fully connected or convolutional layers and before the activation functions (nonlinearities).
- This layer effectively acts as a preprocessing step at each level of the network but is integrated in a differentiable manner, allowing for backpropagation.

Algorithm The input comprises values of x over a mini-batch $\mathcal{B} = \{x_1, \ldots, x_m\}$. The outputs are the parameter to be learned γ, β with respect to the given min-batch. Batch normalization

Algorithm 1 Batch normalization

1: Compute mini-batch mean: $\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$
2: Compute mini-batch variance: $\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2$
3: Normalize: $\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}}$
4: Scale and shift: $y_i \leftarrow \gamma \hat{x}_i + \beta$

has several notable benefits:

- *Improved gradient flow*: it enhances the flow of gradients through the network, making it easier to train deep architectures.
- *Higher learning rates*: allows for the use of higher learning rates, leading to faster convergence.
- *Reduced dependency on weight initialization*: the strong dependence on initial weights is mitigated, as the normalization process stabilizes the learning.
- *Regularization effect*: acts as a form of regularization, slightly decreasing the need for techniques like dropout.

By incorporating batch normalization into neural network architectures, practitioners can significantly enhance training efficiency and model performance.

2.3.5 Weight initialization

The effectiveness of gradient descent in neural networks is highly dependent on how weights are initialized at the start. Various approaches to weight initialization can significantly impact the convergence speed and stability of the network:

- *Zero initialization*: setting all weights to zero is ineffective because it results in zero gradients, meaning no learning occurs. Symmetry is never broken, and the model fails to converge.

- *Large weights*: Initializing with large weights is also problematic. In deep networks, large initial weights can cause gradients to explode as they propagate back through the layers, leading to instability and slow convergence.
- *Small weights* (Gaussian distribution): initializing weights with small values from a Gaussian distribution, such as $\mathbf{w} \sim \mathcal{N}(0, \sigma^2) = 0.01$, works for shallow networks but may pose challenges for deeper architectures. Small initial weights in deep networks can cause the gradients to shrink as they pass through each layer, slowing down learning and potentially leading to the vanishing gradient problem.

In deep networks, weight initialization plays a critical role in avoiding the vanishing or exploding gradient problems:

- *Small weights*: if the initial weights are too small, gradients shrink during backpropagation, eventually becoming so small that learning effectively halts.
- *Large weights*: if the initial weights are too large, gradients can grow excessively, leading to unstable learning or divergence.

To address these issues, specific initialization methods have been proposed to ensure that the gradients remain balanced as they propagate through the network.

Xavier initialization Xavier initialization, proposed by Glorot and Bengio, aims to maintain the variance of the activations across layers, ensuring that signals neither vanish nor explode. Suppose we have an input vector $\mathbf{x}$ with I components, and a linear neuron with random weights $\mathbf{w}$. The output of the neuron is:

$$h_j = w_{j,1}x_1 + \cdots + w_{j,i}x_i + \cdots + w_{j,I}x_I$$

The variance of each term $w_{j,i}x_i$ is:

$$\text{Var}(w_{j,i}x_i) = \text{Var}(w_{j,i})\text{Var}(x_i)$$

Assuming x_i and w_i are independent and identically distributed , the variance of the output h_j becomes:

$$\text{Var}(h_j) = I \times \text{Var}(w_i)\text{Var}(x_i)$$

To keep the variance of the output the same as the input, Xavier initialization sets the weights as:

$$\mathbf{w} \sim \mathcal{N}\left(0, \frac{1}{n_{\text{in}}}\right)$$

Here, n_{in} is the number of input units to the neuron. For the gradients, Glorot and Bengio derived that the variance of the weights should also consider the number of output units n_{out}, leading to the refined initialization:

$$\mathbf{w} \sim \mathcal{N}\left(0, \frac{2}{n_{\text{in}} + n_{\text{out}}}\right)$$

This approach works well for activation functions like sigmoid or hyperbolic tangent.

He initialization More recently, He initialization was proposed specifically for networks using ReLU activations. Since ReLU activation functions only pass positive values, He initialization sets a slightly larger variance to prevent shrinking gradients. It uses the following distribution:

$$\mathbf{w} \sim \mathcal{N}\left(0, \frac{2}{n_{\text{in}}}\right)$$

This method helps maintain better gradient flow, especially in deep networks with ReLU activations, and is widely adopted in modern architectures.

2.4 Loss function

Consider an independent and identically distributed sample drawn from a Gaussian distribution with a known variance, σ^2. The goal is to estimate the parameter μ using Maximum Likelihood Estimation (MLE), which selects parameters that maximize the probability of the observed data.

Let $\boldsymbol{\theta} = \begin{pmatrix}\theta_1 & \theta_2 & \dots & \theta_p\end{pmatrix}^T$ represent the vector of parameters. The task is to find the MLE of $\boldsymbol{\theta}$. Here is the step-by-step approach:

1. *Construct the likelihood function*: the likelihood function $L(\mu)$ is the probability of the data given μ, assuming a Gaussian distribution. The joint likelihood of the data $x_1, x_2, \dots, x_N$ is:

$$L(\mu) = \Pr(x_1, x_2, \dots, x_N | \mu, \sigma^2) = \prod_{n=1}^{N} \Pr(x_n | \mu, \sigma^2) = \prod_{n=1}^{N} \frac{1}{\sqrt{2\pi}} e^{-\frac{(x_n - \mu)^2}{2\sigma^2}}$$

2. *Log-likelihood function*: since the likelihood is a product, it is convenient to take the logarithm to transform it into a sum, yielding the log-likelihood:

$$l(\mu) = \log\left(\prod_{n=1}^{N} \frac{1}{\sqrt{2\pi}} e^{-\frac{(x_n - \mu)^2}{2\sigma^2}}\right) = \sum_{n=1}^{N} \log \frac{1}{\sqrt{2\pi}} e^{-\frac{(x_n - \mu)^2}{2\sigma^2}}$$

   Simplifying:

$$l(\mu) = N \log \frac{1}{\sqrt{2\pi}} - \frac{1}{2\sigma^2} \sum_{n}^{N} (x_n - \mu)^2$$

3. *Compute the gradient of the log-likelihood*: to find the MLE for μ, we need to maximize the log-likelihood by taking its derivative with respect to μ and setting it to zero:

$$\frac{\partial l(\mu)}{\partial \mu} = \frac{\partial}{\partial \mu}\left(N \log \frac{1}{\sqrt{2\pi}\sigma} - \frac{1}{2\sigma^2} \sum_{n}^{N} (x_n - \mu)^2\right)$$

   Focusing on the second term, the derivative is:

$$\frac{\partial l(\mu)}{\partial \mu} = -\frac{1}{2\sigma^2} \frac{\partial}{\partial \mu} \sum_{n}^{N} (x_n - \mu)^2 = \frac{1}{2\sigma^2} \sum_{n}^{N} 2(x_n - \mu)$$

4. *Solve the equation for MLE*: setting the derivative equal to zero to maximize the likelihood:

$$\frac{1}{2\sigma^2}\sum_{n}^{N} 2(x_n - \mu) = 0$$

This simplifies to:

$$\sum_{n}^{N} x_n = \sum_{n}^{N} \mu$$

Hence, the MLE for μ is:

$$\mu^{\text{MLE}} = \frac{1}{N}\sum_{n}^{N} x_n$$

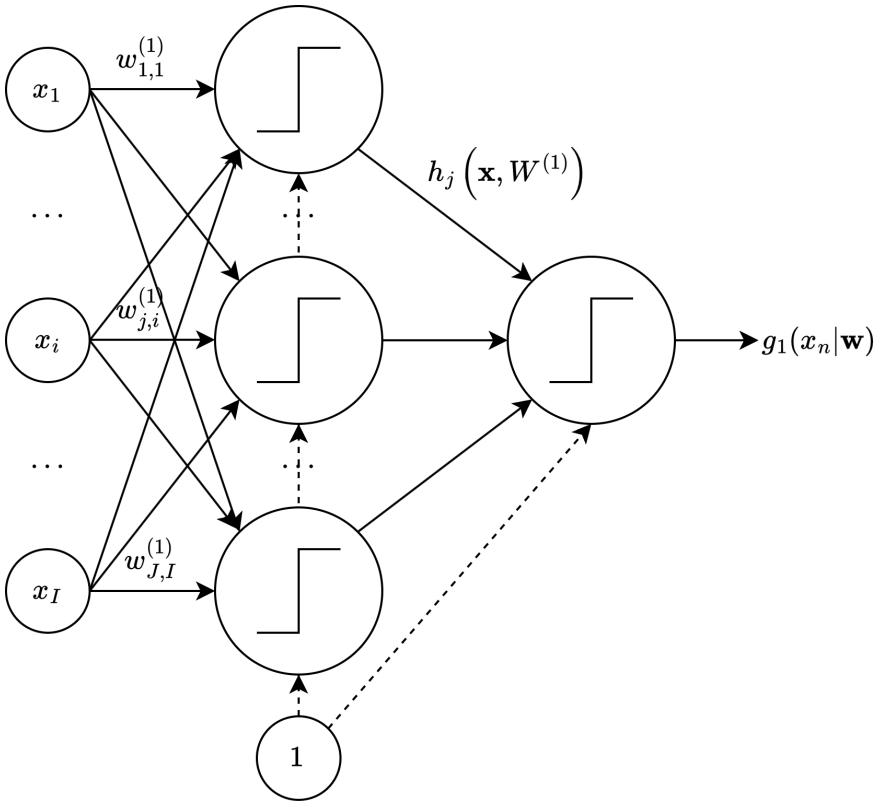
This is the sample mean, which is the optimal estimate for μ under the MLE framework.

To maximize or minimize the log-likelihood, we can apply several techniques:

- *Analytical methods*: directly solve the equations for the MLE, as done above.
- *Optimization techniques*: use methods such as Lagrange multipliers for constrained optimization problems.
- *Numerical methods*: apply iterative approaches like gradient descent when closed-form solutions are intractable.

In our case, we derived the MLE for μ analytically as the sample mean, but for more complex models, numerical optimization techniques may be necessary.

Example:
Consider the following Feed Forward Nwural Network.



The output of the network is defined as:

$$g_1(x_n|\mathbf{w}) = g_1\left(\sum_{j=0}^{J} w_{1,j}^{(2)} h_j\left(\sum_{i=0}^{I} w_{j,i}^{(1)} x_{i,n}\right)\right)$$

Here, h_j represents the activation function of the hidden neurons, and $w_{i,j}^{(1)}$ and $w_{i,j}^{(2)}$ are the weights of the first and second layers, respectively. The goal is to approximate a target function t having N observations:

$$t_n = g(x_n|\mathbf{w}) + \epsilon_n \qquad \epsilon_n \sim N(0, \sigma^2) \rightarrow t_n \sim N(g(x_n|\mathbf{w}), \sigma^2)$$

To estimate the weights $\mathbf{w}$ in a model $g(x_n|\mathbf{w})$ using Maximum Likelihood Estimation (MLE), follow these steps:

1. *Write the likelihood function $L(\mathbf{w}) = P(data|\mathbf{w})$ for the data*: assume that the observed values t_n are drawn from a Gaussian distribution with known variance σ^2, and that the model $g(x_n|\mathbf{w})$ provides the mean of the distribution. The probability of each observation is given by:

$$\Pr(t_n|g(x_n|\mathbf{w}), \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(t_n - g(x_n|\mathbf{w}))^2}{2\sigma^2}}$$

   The likelihood function for the entire dataset is the product of these probabilities:

$$L(\mathbf{w}) = \Pr(t_1, t_2, \ldots, t_N|g(x|\mathbf{w}), \sigma^2) = \prod_{n=1}^{N} \Pr(t_n|g(x_n|\mathbf{w}), \sigma^2)$$

   Substituting the expression for each $\Pr(t_n)$, we get:

$$L(\mathbf{w}) = \prod_{n=1}^{N} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(t_n - g(x_n|\mathbf{w}))^2}{2\sigma^2}}$$

2. *Write the log-likelihood function $l(\mathbf{w}) = \log P(data|\mathbf{w})$* :taking the logarithm of the likelihood function simplifies the product into a sum:

$$l(\mathbf{w}) = \log\left(\prod_{n=1}^{N} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(t_n - g(x_n|\mathbf{w}))^2}{2\sigma^2}}\right)$$

   This simplifies to:

$$l(\mathbf{w}) = \sum_{n=1}^{N} \log \frac{1}{\sqrt{2\pi\sigma^2}} - \frac{(t_n - g(x_n|\mathbf{w}))^2}{2\sigma^2}$$

   Further simplifying:

$$l(\mathbf{w}) = N \log \frac{1}{\sqrt{2\pi\sigma^2}} - \frac{1}{2\sigma^2} \sum_{n=1}^{N} (t_n - g(x_n|\mathbf{w}))^2$$

3. *Optimize the weights* $\mathbf{w}$ *to maximize the log-likelihood*: to find the weights $\mathbf{w}$ that maximize the log-likelihood, we solve the following optimization problem:

$$\mathbf{w}^{\text{MLE}} = \underset{\mathbf{w}}{\text{argmax}}\, l(\mathbf{w})$$

This is equivalent to minimizing the sum of squared errors:

$$\mathbf{w}^{\text{MLE}} = \underset{\mathbf{w}}{\text{argmin}} \sum_{n=1}^{N} (t_n - g(x_n|\mathbf{w}))^2$$

This optimization problem is typically solved using numerical techniques like gradient descent, depending on the complexity of $g(x_n|\mathbf{w})$.

4. *Approximate the posterior probability for* t:f or binary classification where $t_n \in \{0, 1\}$ and $t_n \sim \text{Bernoulli}(g(x_n|\mathbf{w}))$, the likelihood for the data is given by:

$$\Pr(t_n|g(x_n|\mathbf{w})) = g(x_n|\mathbf{w})^{t_n} \cdot (1 - g(x_n|\mathbf{w}))^{1-t_n}$$

The likelihood function for the entire dataset is:

$$L(\mathbf{w}) = \prod_{n=1}^{N} g(x_n|\mathbf{w})^{t_n} \cdot (1 - g(x_n|\mathbf{w}))^{1-t_n}$$

5. *Compute the log-likelihood*: taking the logarithm of the likelihood function gives:

$$l(\mathbf{w}) = \sum_{n=1}^{N} [t_n \log g(x_n|\mathbf{w}) + (1 - t_n) \log(1 - g(x_n|\mathbf{w}))]$$

This expression is known as the cross-entropy loss:

$$-\sum_{n=1}^{N} t_n \log g(x_n|\mathbf{w}) + (1 - t_n) \log(1 - g(x_n|\mathbf{w}))$$

2.4.1 Loss function selection

The choice of an appropriate loss function is crucial for defining the task and guiding the learning process. The loss function not only measures the error between the predicted and actual values but also influences the optimization behavior of the model. Designing a loss function requires careful consideration of various factors:

- *Leverage knowledge of the data distribution*: when selecting a loss function, it is important to incorporate any prior knowledge or assumptions regarding the underlying data distribution. For example, if the data is normally distributed, a squared error loss might be appropriate, while for binary classification tasks, cross-entropy loss is typically used.
- *Exploit task-specific and model knowledge*: a good loss function should align with the goals of the task at hand. For instance, in classification problems, we want to maximize the probability of correct predictions, and in regression, we aim to minimize the difference

between predicted and true values. Tailoring the loss function to the model and its intended use case can significantly improve performance.

- *Creativity in loss function design*: in some cases, predefined loss functions may not fully capture the nuances of the problem, and this is where creativity can play a role. Custom loss functions can be designed by combining multiple loss components or incorporating domain-specific constraints to better align with the task's objectives.

In summary, selecting the right loss function is both a science and an art—it requires a blend of theoretical insights, practical understanding of the problem domain, and sometimes, creative thinking to balance accuracy and interpretability.

2.4.2 Perceptron loss function

Hyperplanes Consider a hyperplane $L : \mathbf{w}^T x + w_0 = 0 \in \mathbb{R}^2$. Any two points x_1 and x_2 lying on the hyperplane L can be characterized by their relationship to this hyperplane. The normal vector $\mathbf{w}^*$ to the hyperplane can be defined as follows:

$$\mathbf{w}^* = \frac{\mathbf{w}}{\|\mathbf{w}\|}$$

For any point x_0 on the hyperplane L, we can express the signed distance of any point x from the hyperplane L as:

$$\mathbf{w}^{*^T}(x - x_0)\frac{1}{\|\mathbf{w}\|} = \frac{1}{\|\mathbf{w}\|}\left(\mathbf{w}^T x + w_0\right)$$

It can be shown that the error function minimized by the Hebbian rule is related to the distance of misclassified points from the decision boundary. When coding the perceptron output as 1 or -1, we can represent the following conditions:

- If an output that should be 1 is misclassified, then $\mathbf{w}^T x + w_0 < 0$.
- Conversely, for an output that should be -1, we have $\mathbf{w}^T x + w_0 > 0$.

The objective then becomes to minimize the following loss function:

$$D(\mathbf{w}, w_0) = -\sum_{i \in M} t_i(\mathbf{w}^T x_i + w_0)$$

Here, $D(\mathbf{w}, w_0)$ is non-negative and proportional to the distance of the misclassified points from the decision boundary defined by $\mathbf{w}^T x + w_0 = 0$.

To minimize the error function $D(\mathbf{w}, w_0)$ using stochastic gradient descent, we take the gradients with respect to the model parameters:

$$\frac{\partial D(\mathbf{w}, w_0)}{\partial \mathbf{w}} = -\sum_{i \in M} t_i \cdot x_i \qquad \frac{\partial D(\mathbf{w}, w_0)}{\partial w_0} = -\sum_{i \in M} t_i$$

Stochastic gradient descent is applied iteratively for each misclassified point:

$$\begin{cases} \mathbf{w}_{k+1} = \mathbf{w}_k + \eta t_i \cdot x_i \\ w_{0,k+1} = w_{0,k} + \eta t_i \end{cases}$$

In this context, η represents the learning rate. This iterative approach is akin to Hebbian learning and effectively implements stochastic gradient descent, allowing the perceptron to adjust its weights and bias based on the misclassified instances.

2.4.3 Adaptive learning

To effectively navigate the challenges of local minima in the optimization landscape, various techniques can be employed, one of which is momentum. This method helps to accelerate gradient descent in the relevant direction and dampens oscillations. The update rule with momentum is given by:

$$\mathbf{w}^{k+1} = \mathbf{w}^k - \eta \frac{\partial E(\mathbf{w})}{\partial \mathbf{w}}\bigg|_{\mathbf{w}^k} - \alpha \frac{\partial E(\mathbf{w})}{\partial \mathbf{w}}\bigg|_{\mathbf{w}^{k-1}}$$

Here, $\mathbf{w}^k$ represents the weights at iteration k, η is the learning rate, and α is the momentum term.

Nestorov accelerated gradient An enhancement to the momentum method is the Nesterov accelerated gradient. This approach first makes a momentum-based update and then adjusts the gradient based on the new position. The two-step update process is defined as follows:

1. First, a momentum-based prediction of the next position:

$$\mathbf{w}^{k+\frac{1}{2}} = \mathbf{w}^k - \alpha \frac{\partial E(\mathbf{w})}{\partial \mathbf{w}}\bigg|_{\mathbf{w}^{k-1}}$$

2. Then, updating the weights using the gradient evaluated at the predicted position:

$$\mathbf{w}^{k+1} = \mathbf{w}^k - \eta \frac{\partial E(\mathbf{w})}{\partial \mathbf{w}}\bigg|_{\mathbf{w}^{k+\frac{1}{2}}}$$

Layer-specific learning rates In deep networks, different neurons in each layer may learn at varying rates due to differences in gradient magnitudes across layers. Early layers are often prone to vanishing gradients, which can hinder learning. To address this, it is beneficial to employ separate adaptive learning rates for different layers. Several algorithms have been proposed to achieve this:

- *Resilient propagation* (Rprop): introduced by Riedmiller and Braun in 1993, this algorithm adjusts the weight updates based on the sign of the gradient, ensuring that the step sizes remain appropriate.
- *Adaptive gradient* (AdaGrad): developed by Duchi et al. in 2010, AdaGrad adapts the learning rate for each parameter based on the historical gradients, allowing for a more tailored approach.
- *RMSprop*: a combination of Stochastic Gradient Descent and Rprop, proposed by Tieleman and Hinton in 2012. RMSprop utilizes an exponentially decaying average of squared gradients to normalize the learning rate.
- *AdaDelta*: introduced by Zeiler et al. in 2012, this method extends AdaGrad by not accumulating past squared gradients, thus overcoming its diminishing learning rates issue.
- *Adam* (Adaptive Moment Estimation): proposed by Kingma and Ba in 2012, Adam combines the benefits of both momentum and RMSprop, using estimates of both the first and second moments of the gradients to adaptively adjust learning rates.

2.5 Overfitting

When learning a function, a single-layer model can theoretically approximate any measurable function to a high degree of accuracy on a compact set. However, this does not guarantee that we can find the necessary weights. In fact, an exponential number of hidden units may be required, making it impractical in practice, especially if the model does not generalize well to unseen data.

Underfitting occurs when the model is too simple to capture the underlying patterns in the data, leading to poor performance. To mitigate this, we can increase the model's complexity. However, if we increase the complexity too much, the model may adhere too closely to the training data, resulting in overfitting. Overfitting causes the model to capture noise rather than the true signal, reducing its ability to generalize to new, unseen data.

The goal of training is to find an approximation that balances complexity and generality, allowing the model to perform well not only on the training set but also on future data. This process involves finding a good balance between underfitting and overfitting, aiming for a model that generalizes well to new, unseen data rather than simply memorizing the training data.

The error or loss on the training data is not a reliable indicator of how well the model will perform on future data. There are several reasons for this:

- The model has been trained on the same data, so any estimate of performance based on the training set will likely be overly optimistic.
- New data will rarely match the training data exactly, meaning the model may encounter patterns it hasn't seen before.
- It's possible to find patterns even in random data, leading to the false belief that the model is performing well.

To accurately assess model performance, it's essential to evaluate it on a separate test set, which represents new, unseen data.

To properly evaluate the model's ability to generalize, we follow these steps:

1. *Test on an independent dataset*: it's critical to evaluate the model on a dataset that was not used during training.
2. *Data splitting*: split the available data into training, validation, and test sets. The test set is hidden and only used for final evaluation.
3. *Cross-validation*: perform random subsampling (with replacement) to split the data. In classification problems, ensure the class distribution is preserved using stratified sampling.

Definition (*Training dataset*)**.** The training dataset is the entire dataset available for building the model.

Definition (*Training set*)**.** The training set is subset of the data used to learn the model parameters.

Definition (*Test set*)**.** The test set is a separate subset of the data used for the final evaluation of the model's performance.

Definition (*Validation set*)**.** The validation set is a subset of the data used to fine-tune model hyperparameters and perform model selection.

Definition (*Training data*)**.** The training data is the data used during the training phase for both fitting and selection of the model.

Definition (*Validation data*)**.** The validation data is the data used to assess the model's performance during training, aiding in hyperparameter tuning and model selection.

2.5.1 Cross validation

Cross-validation is a technique used to estimate the performance of a model by utilizing the available training data for both training (parameter fitting and model selection) and error estimation on unseen data. This helps assess the model-s ability to generalize to new data without the need for a separate test set in the early stages of model development. The possible tecniques are:

- *Hold-out validation*: when a large amount of data is available, it's common to split the dataset into distinct subsets for training and validation. This allows for a straightforward estimation of model performance on unseen data. However, this method may not be suitable when the available data is limited, as it reserves part of the data for validation, reducing the amount used for training.
- *Leave-one-out cross-validation* (LOOCV): in cases where data is scarce, LOOCV can be applied. Here, the model is trained on all but one data point, which is then used for validation. This process is repeated for each data point, ensuring that every instance is used for both training and validation. While this method provides an unbiased estimate of model performance, it can be computationally expensive, particularly for large datasets.
- *K-fold cross-validation*: a more practical and commonly used approach is K-fold cross-validation. In this method, the dataset is randomly split into K equally sized subsets (folds). The model is trained on $K-1$ folds and validated on the remaining fold. This process is repeated K times, with each fold serving as the validation set once. The final model performance is computed as the average error across all K iterations. K-fold cross-validation offers a good balance between computational efficiency and performance estimation, and it is often preferred over LOOCV as it tends to provide more stable results with fewer repetitions.

2.5.2 Early stopping

Overfitting in neural networks often manifests as a monotonically decreasing training error as the number of gradient descent iterations k increases (especially when using stochastic gradient descent). However, while the model continues to perform better on the training set, its ability to generalize to unseen data can degrade beyond a certain point. Early stopping is a regularization technique designed to prevent overfitting by halting the training process when the model's performance on a validation set starts to deteriorate. The steps needed to use early stopping are:

1. *Hold out a validation set*: reserve a portion of the data for validation, separate from the training set.
2. *Train on the training set*: perform gradient descent iterations using the training set.

3. *Cross-Validate on the hold-out set*: continuously evaluate the model on the validation set during training.

4. *Stop training based on validation error*: when the validation error starts to increase (indicating the model is beginning to overfit the training data), halt the training process. This is where the model achieves its best generalization.

Model selection and evaluation occur at multiple levels:

- *Parameter level*: this involves learning the model's parameters, such as the weights $\mathbf{w}$ of a neural network, through optimization during training.

- *Hyperparameter level*: yhis involves choosing the structural components of the model, such as the number of layers L or the number of hidden neurons $J^{(l)}$ in each layer l. These hyperparameters significantly affect the model's capacity and generalization.

2.5.3 Weight decay

Regularization aims to constrain the freedom"of the model based on prior assumptions, in order to reduce overfitting. In the context of neural networks, weight decay is one of the most commonly used regularization techniques. It helps control model complexity by discouraging large weights, which can lead to overfitting.

Typically, we maximize the data likelihood when learning the model parameters:

$$w_{\text{MLE}} = \underset{\mathbf{w}}{\operatorname{argmax}} \Pr(\mathcal{D}|\mathbf{w})$$

However, to impose additional structure and reduce model complexity, we can introduce a Bayesian perspective by incorporating a prior over the weights $\mathbf{w}$. This leads to maximum a posteriori (MAP) estimation:

$$\begin{aligned} w_{\text{MAP}} &= \underset{\mathbf{w}}{\operatorname{argmax}} \Pr(\mathbf{w}|\mathcal{D}) \\ &= \underset{\mathbf{w}}{\operatorname{argmax}} \Pr(\mathcal{D}|\mathbf{w}) \Pr(\mathbf{w}) \end{aligned}$$

In practice, small weights tend to improve the generalization performance of neural networks, which can be reflected by assuming a Gaussian prior on the weights:

$$\Pr(\mathbf{w}) \sim \mathcal{N}(0, \sigma_{\mathbf{w}}^2)$$

By incorporating this prior, we can expand the optimization problem as follows:

$$\begin{aligned} \hat{\mathbf{w}} &= \underset{\mathbf{w}}{\operatorname{argmax}} \Pr(\mathbf{w}|\mathcal{D}) \\ &= \underset{\mathbf{w}}{\operatorname{argmax}} \Pr(\mathcal{D}|\mathbf{w}) \Pr(\mathbf{w}) \\ &= \underset{\mathbf{w}}{\operatorname{argmax}} \prod_{n=1}^{N} \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(t_n - g(x_n|\mathbf{w})^2)}{2\sigma^2}} \prod_{q=1}^{Q} \frac{1}{\sqrt{2\pi}\sigma_{\mathbf{w}}} e^{-\frac{(w_q)^2}{2\sigma_{\mathbf{w}}{}^2}} \end{aligned}$$

Simplifying this, we get the following objective function:

$$\hat{\mathbf{w}} = \underset{\mathbf{w}}{\operatorname{argmin}} \sum_{n=1}^{N} \frac{(t_n - g(x_n|\mathbf{w})^2)}{2\sigma^2} + \sum_{q=1}^{Q} \frac{(w_q)^2}{2\sigma_{\mathbf{w}^2}}$$

This can be rewritten as:

$$\hat{\mathbf{w}} = \underset{\mathbf{w}}{\operatorname{argmin}} \sum_{n=1}^{N} (t_n - g(x_n|\mathbf{w})^2) + \gamma \sum_{q=1}^{Q} (w_q)^2$$

Here, γ is a regularization parameter that controls the strength of weight decay. Larger values of γ will penalize larger weights more strongly, encouraging smaller weights and thus reducing model complexity.

Gamma selection To select the optimal value of γ, we can use cross-validation as follows:

1. *Split the data*: divide the dataset into training and validation sets.
2. *Minimize for different γ values*: for each candidate value of γ, minimize the following training error:
$$\mathrm{E}_{\gamma}^{\text{train}} \sum_{n=1}^{N_{\text{train}}} (t_n - g(x_n|\mathbf{w})^2) + \gamma \sum_{q=1}^{Q} (w_q)^2$$
3. *Evaluate the model*: compute the validation error for each candidate γ:
$$\mathrm{E}_{\gamma}^{\text{validation}} \sum_{n=1}^{N_{\text{validation}}} (t_n - g(x_n|\mathbf{w})^2)$$
4. *Choose the optimal γ^**: select the value of γ that results in the best validation error.
5. *Final model training*: combine all data and minimize the objective with the selected γ^* that results in the best validation error.
$$\mathrm{E}_{\gamma^*} \sum_{n=1}^{N} (t_n - g(x_n|\mathbf{w})^2) + \gamma^* \sum_{q=1}^{Q} (w_q)^2$$

2.5.4 Dropout

Dropout is a stochastic regularization technique used to mitigate overfitting by randomly dropping out or deactivating certain neurons during training. By doing this, the model is forced to learn more robust and independent feature representations, preventing hidden units from relying too heavily on one another (a phenomenon known as co-adaptation). For each training iteration:

1. *Neuron deactivation*: each hidden unit is set to zero with a probability $\Pr_j^{(l)}$, where l denotes the layer and j the unit within that layer. This effectively turns off the neuron for that iteration.
2. *Applying a mask*: a mask is applied to the layer to randomly drop out neurons, removing part of the signal. This generates a sub-network from the original neural network, on which the training continues.
3. *Repeating with new masks*: at each training step, a new mask is applied, creating a different sub-network from the original. Training proceeds as if each sub-network is a distinct model.

4. *Reactivating all neurons at test time*: once training is complete, all neurons are reactivated for the full network during testing. The behavior of the full network is approximately the averaged result of all the sub-networks trained during dropout. To compensate for the dropouts during training, weight scaling is applied at test time, effectively averaging the outputs of all possible sub-networks.

Benefits One of the key advantages of dropout is that it helps prevent co-adaptation among neurons. By randomly deactivating certain neurons during training, the network is forced to distribute the learning process more evenly across all neurons, making it less dependent on specific units. This encourages the network to learn more robust and independent feature representations, improving generalization. Additionally, dropout can be viewed as a form of implicit ensemble learning. Each time a subset of neurons is dropped, the network effectively trains a different sub-network. At inference time, the full network's output can be seen as the average prediction of all these sub-networks, leading to a more generalizable model.

Dropout can be applied selectively to specific layers, typically to the more densely connected ones, rather than across every layer in the network. This targeted regularization can be more effective in reducing overfitting while maintaining computational efficiency. A good practice when training neural networks is to first train the model to the point of overfitting, which ensures that the problem is solvable and that the model has sufficient capacity to learn the task. Once overfitting is observed, techniques such as early stopping, dropout, or weight decay can be introduced to reduce overfitting and improve generalization.

CHAPTER 3

Convolutional Neural Networks

3.1 Computer vision

Computer vision is an interdisciplinary field focused on enabling computers to interpret and understand the visual world from digital images or videos. Initially, most computer vision techniques were based on mathematical models and statistical analysis of images. However, with the rise of Machine Learning, particularly Deep Learning, modern approaches have shifted towards data-driven methods, making these algorithms far more effective and adaptable to complex visual tasks.

3.1.1 Digital images

A digital color image is stored using three separate matrices, each corresponding to one of the primary colors: red, green, and blue (RGB). Each matrix element represents a pixel's intensity and is typically encoded using 8 bits, meaning values range from 0 to 255. When visualized in three-dimensional space, the diagonal of the RGB cube represents different shades of gray, where all three color intensities are equal.

Although images on disk are often stored in compressed formats to save space, such as JPEG or PNG, they need to be decompressed to be processed in memory. This decompression significantly increases the size of the data, making images, especially in large quantities, a challenge to manage in terms of memory and processing power. The problem grows even more pronounced when working with video, as each second of video consists of multiple image frames.

When using neural networks for image processing tasks, the raw image data. This results in large amounts of data that need to be handled efficiently. Fortunately, visual data tends to be highly redundant and compressible, which can be leveraged to reduce memory and computational demands.

3.1.2 Local transformations

Local transformations play a crucial role in image processing, particularly in tasks such as classification. These transformations involve modifying each pixel based on the values of its neighboring pixels within a specified region, or neighborhood, U. Mathematically, a local

transformation can be expressed as:

$$\mathbf{G}(r, c) = T_U[\mathbf{I}](r, c)$$

Here, I is the input image, G is the output image, U defines the neighborhood around the pixel, and T_U is a spatial transformation function, which can be either linear or non-linear.

For a pixel at coordinates (r, c) in the image, the neighborhood U is typically a square region centered at the pixel, and is defined as:

$$\{\mathbf{I}(u, v) \mid (u - r, v - c) \in U\}$$

Here, (u, v) represents the displacement relative to the center of the neighborhood (r, c). The same transformation function T_U is applied repeatedly across all pixels in the image, making it a spatially invariant transformation.

Local linear filters In the case of a linear spatial transformation, the output $T_U[I](r, c)$ is a linear combination of the pixel values within the neighborhood U. This can be described as:

$$T_U[\mathbf{I}](r, c) = \sum_{(u,v)\in U} w_i(u, v) I(r + u, c + v)$$

Here, $w(u, v)$ are the weights associated with the pixels in U. These weights can be thought of as defining a filter, or kernel, that is applied uniformly across the entire image. In this way, the same operation is repeated for all pixels, making it a simple and efficient method for image processing.

The correlation between a filter $\mathbf{w}$ (with elements w_{ij}) and an image $\mathbf{I}$ can be computed using:

$$(\mathbf{I} \otimes \mathbf{w})(r, c) = \sum_{u=-L}^{L} \sum_{v=-L}^{L} w(u, v) \mathbf{I}(r + c, c + v)$$

Here, the filter $\mathbf{w}$ has dimensions $(2L+1) \times (2L+1)$, and acts as a kernel that defines the transformation. The correlation operation essentially slides the filter across the image, computing a weighted sum of pixel values in each neighborhood.

This formula applies both to grayscale images:

$$T_U[\mathbf{I}](r, c) = \sum_{(u,v)\in U} w_i(u, v) \mathbf{I}(r + u, c + v)$$

And to color (RGB) images, where each channel is processed separately:

$$T_U[\mathbf{I}](r, c) = \sum_{i} \sum_{(u,v)\in U} w_i(u, v, i) \mathbf{I}(r + u, c + v, i)$$

In RGB images, the filter $\mathbf{w}(u, v, i)$ applies weights not only across spatial dimensions but also across the three color channels (red, green, blue).

3.2 Image classification

Image classification involves assigning an input image $\mathbf{I} \in \mathbb{R}^{R \times C \times 3}$ to a label y from a predefined set of categories λ. A classifier is a function f_θ that maps the image to a label, expressed as:

$$f_\theta : \mathbf{I} \to f_\theta(\mathbf{I}) \in \lambda$$

3.2.1 Linear classifier

To input an image into a neural network, the image is first flattened into a vector, with the RGB color channels linearized column by column. Each element in this vector corresponds to a neuron in a single-layer neural network, where the number of output neurons matches the number of classification labels. Each input neuron is fully connected to all output neurons, leading to an impractically large number of connections as image dimensions increase. This problem, known as the curse of dimensionality, prevents the straightforward application of deep neural networks to large datasets.

For instance, consider an RGB dataset with 32×32 images and 6000 examples distributed across ten classes (e.g., CIFAR-10). The neural network for this dataset can be visualized as:

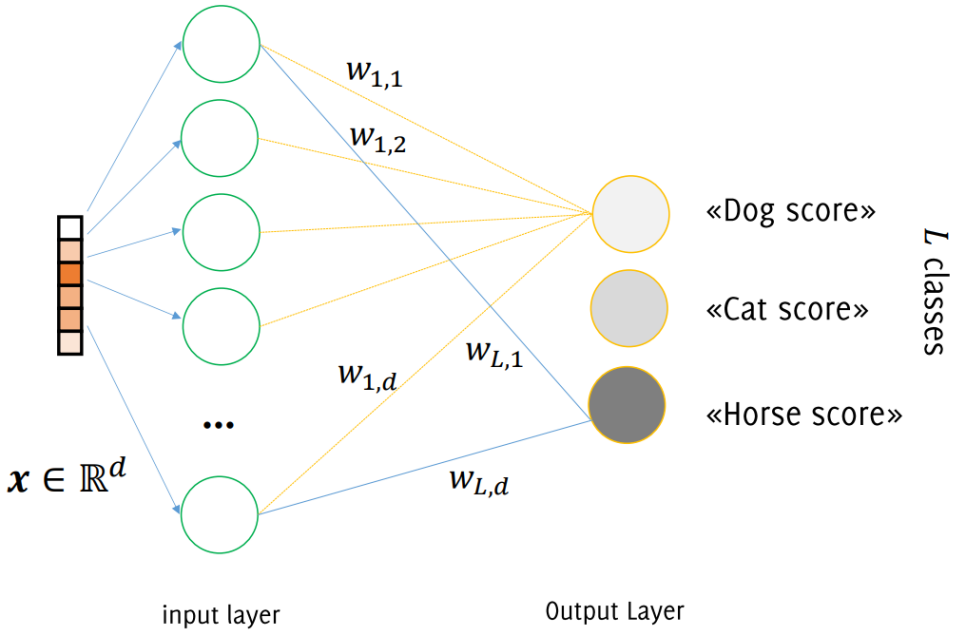


Figure 3.1: Neural Network for CIFAR-10 dataset

The classifier's weights can be organized in a matrix$\mathbf{W} \in \mathbb{R}^{L \times d}$, where L is the number of classes and d is the image's flattened dimension. The score for the i-th class is computed as the inner product of the image vector $\mathbf{x}$ and the corresponding row in $\mathbf{W}$:

$$s_i = \mathbf{W}[i, :]\mathbf{x} + b_i$$

Since this is a single-layer network, nonlinearity is not necessary, and the softmax activation can be omitted, as it does not affect the ranking of class scores.

3.2.2 Linear classifier for images

An image classifier can be understood as a function that maps an image $\mathbf{I}$ to a confidence score vector $\mathcal{K}(\mathbf{I}) \in \mathbb{R}^L$, where the i-th component s_i represents the score indicating the likelihood that the image belongs to class i. A good classifier assigns the correct class to the highest score. In the case of linear classification, $\mathcal{K}$ is a linear function mapping the image vector $\mathbf{x} \in \mathbb{R}^d$ to class scores:

$$\mathcal{K}(\mathbf{x}) = \mathbf{W}\mathbf{x} + b$$

Here, $\mathbf{W} \in \mathbb{R}^{L \times d}$ is the weight matrix, and $b \in \mathbb{R}^L$ is the bias vector. The predicted label for an input image is the class with the highest score:

$$\hat{y}_j = \underset{i=1,\ldots,L}{\operatorname{argmax}} [\mathbf{s}_j]_\mathbf{i}$$

Here, $[\mathbf{s}_j]_\mathbf{i}$ is the i-th component of the score vector $\mathbf{s} = \mathbf{W}\mathbf{x} + \mathbf{b}$ The score for each class is the weighted sum of pixel values, with the weights being the learned parameters of the classifier.

3.2.3 Linear classifier training

The goal of training is to find the parameters $\mathbf{W}$ and $\mathbf{b}$ that minimize the overall loss on the training set. For a linear classifier, this can be formulated as:

$$[\mathbf{W}, \mathbf{b}] = \underset{\mathbf{W} \in \mathbb{R}^{L \times d}, \mathbf{b} \in \mathbb{R}^L}{\operatorname{argmin}} \sum_{(\mathbf{x}_i, \mathbf{y}_i) \in \text{training set}} \mathcal{L}_{\mathbf{W},\mathbf{b}}(\mathbf{x}, y_i)$$

Here, $\mathcal{L}_{\mathbf{W},\mathbf{b}}$ is the loss function that quantifies the error between the predicted and true labels.

Loss function The loss function $\mathcal{L}$ measures how well the classifier performs on the training images, assigning high values to misclassified examples and low values to correct ones. It is typically minimized using gradient descent and its variants. Regularization is often applied to the loss function to avoid overfitting, ensuring a well-behaved solution:

$$[\mathbf{W}, \mathbf{b}] = \underset{\mathbf{W} \in \mathbb{R}^{L \times d}, \mathbf{b} \in \mathbb{R}^L}{\operatorname{argmin}} \sum_{(\mathbf{x}_i, \mathbf{y}_i) \in \text{training set}} \mathcal{L}_{\mathbf{W},\mathbf{b}}(\mathbf{x}, y_i) + \lambda \mathcal{R}(\mathbf{W}, \mathbf{b})$$

Here, $\lambda > 0$ is a regularization parameter that controls the trade-off between fitting the training data and maintaining model simplicity.

Geometric interpretation Each image is represented as a point in $\mathbb{R}^d$, and the classifier corresponds to a linear function in this space. In the 2D case, this function would be:

$$f([x_1, x_2]) = w_1 x_1 + w_2 x_2 + b$$

The decision boundary where $f([x_1, x_2]) = 0$ is a line, separating positive and negative class scores. In higher dimensions, this decision boundary generalizes to a hyperplane in $\mathbb{R}^d$.

3.2.4 Image classification challenges

The main challenges in image classification are:

1. *Dimensionality*: images and videos are high-dimensional data, making it challenging to manage memory and computational resources. Entire batches of images and their activations must be stored during training.

2. *Label ambiguity*: a single label may not fully capture the content of an image, making the classification task inherently ambiguous.

3. *Invariance to transformations*: many transformations, such as changes in illumination, deformations, or viewpoint, can alter an image significantly while preserving its label. Robust classifiers must account for these variations.

4. *Inter-class variability*: images within the same class may differ significantly, making it hard to find common patterns for classification.

5. *Perceptual similarity*: similarity between images does not always correspond to pixel-level similarity. For instance, assigning the closest training image's label to a test image using nearest-neighbor methods like k-NN can be problematic in high-dimensional spaces.

In k-nearest neighbors (k-NN), the class of a test image is predicted by the most frequent label among its k-closest images in the training set:

$$\hat{y}_j = \underset{i=1,\ldots,L}{\operatorname{argmax}} y_{j^*}$$

Here, j^* represents the mode of the k-nearest images, and the distance function $d(\cdot)$ could be Euclidean or Manhattan distance. Setting the parameters k and the distance measure is an issue. Although k-NN is easy to implement and requires no training, it is computationally expensive at test time and struggles with high-dimensional data like images.

3.2.5 Feature extraction

Images cannot be directly fed into a classifier, as raw pixel data is often too complex and high-dimensional to yield meaningful results. Therefore, an intermediate step is required to extract relevant information and reduce the dimensionality of the data, making it easier for the classifier to work effectively.

Effective feature extraction plays a crucial role: the quality of the features directly impacts the performance of the classifier. The goal is to capture the most important aspects of the image while minimizing irrelevant information.

Handcrafted features The advantages of handcraft features are:

- *Leverage prior domain expertise*: features are often designed by experts who understand the problem and the data.
- *Interpretability*: handcrafted features tend to be easier to understand, which helps in diagnosing why certain features work or fail.
- *Tunability*: features can be adjusted and fine-tuned based on performance feedback, allowing for more control over the classification process.
- *Efficient with limited data*: handcrafted features often require less data to train compared to deep learning methods.
- *Feature weighting*: it is possible to give more importance to certain features that are known to be relevant to the task at hand.

While the drawbacks are:

- *Labor-intensive*: designing and programming handcrafted features requires significant time and effort.
- *Limited applicability*: in many visual recognition tasks, handcrafted features are less effective than automated feature extraction methods, as human-designed features may miss important patterns.

- *Risk of overfitting*: since the features are tailored to a specific training set, there is a risk of overfitting, which leads to poor generalization on new data.
- *Lack of generalization*: handcrafted features are often problem-specific and may not transfer well to other tasks or datasets.

Rather than relying on handcrafted features, which require manual design and domain expertise, we can leverage data-driven features. These features are automatically learned from the data itself, offering a more scalable and effective approach. This is achieved by using a Convolutional Neural Network, which is specifically designed to extract hierarchical features from images.

CNNs learn to identify patterns such as edges, textures, and more complex structures through multiple layers of convolution and pooling, allowing them to capture increasingly abstract representations of the image. This eliminates the need for manual feature engineering and often results in superior performance, particularly in tasks like image classification, object detection, and visual recognition. By letting the model discover the most relevant features directly from the data, CNNs reduce human bias and can generalize better to new, unseen data.

3.3 Convolutional Neural Network

Convolutional Neural Networks (CNNs) are a type of deep learning architecture designed to automatically and efficiently extract spatial features from images. CNNs are composed of several core building blocks, each serving a specific function in the feature extraction process. The main components of a CNN include:

- Convolutional layers.
- Nonlinearities (activation functions, such as ReLU).
- Pooling layers (e.g., max pooling for subsampling).

These layers work together to progressively transform an input image into more abstract and useful representations for the task at hand, such as classification or object detection.

Architecture The architecture of a CNN typically involves stacking multiple blocks of these layers. As the input image passes through the network, it is transformed through a sequence of increasingly complex feature maps or volumes:

- As the depth of the network increases, the height and width of the image representation shrink.
- Each layer takes a 3D input volume (height, width, and channels) and outputs a transformed volume.

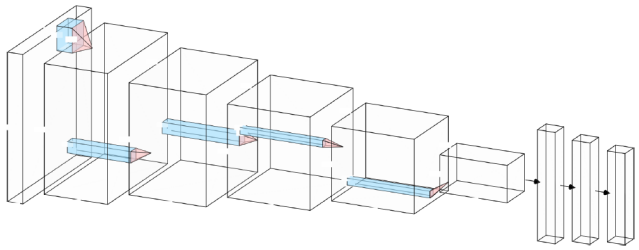


Figure 3.2: Convolutional Neural Network architecture

3.3.1 Convolutional layers

The core operation of a CNN is the convolution. In a convolutional layer, small regions of the input (known as receptive fields) are processed using filters (or kernels), which are learned during training. The input image, represented as a volume with dimensions height h, width w, and three channels for RGB, is transformed by these filters.

Each convolutional layer typically outputs a set of activation maps or "feature maps," each corresponding to a different filter. As the network grows deeper, the number of channels (i.e., the depth of the feature maps) increases, while the spatial dimensions (height and width) decrease.

For a filter w applied over the input image, the output at a given location (r, c) and channel l is computed as:

$$a(r, c, l) = \sum_{(u,v)\in U, k=1,\ldots,C} w^l(u, v, k)x(r + u, c + v, k) + b^l \qquad \forall (r, c), l = 1, \ldots, N_F$$

Here:

- (r, c) refers to a spatial location in the output activation map a.
- l denotes the output channel index (or filter).
- U represents the spatial neighborhood (receptive field) of the convolution filter.
- C is the number of input channels (e.g., 3 for an RGB image).

A filter is characterized by:

- *Spatial size*: a small neighborhood U (e.g., 3x3 or 5x5) over which the convolution is applied.
- *Depth*: the same as the number of channels in the input volume, ensuring that the filter processes the full input depth.

The total number of parameters in a convolutional layer is determined by the size of the filters and the number of filters. Specifically:

$$\text{Total parameters } = (h_r \cdot h_c \cdot C)\dot{N}_F + N_F$$

Here, h_r and h_c are the height and width of the filter, C is the input depth, and N_F is the number of filters.

Key characteristics The key characteristics of the convolutional layers in a Convolutional Neural Network are:

- *Local processing*: convolutions operate over a small spatial neighborhood U, meaning the filter looks at localized regions of the image at a time.
- *Channel-wise* processing: Filters span the entire depth of the input volume to capture information across all channels.
- *Output volume*: the result of applying a filter is a slice of the output volume, often referred to as an activation map. Each filter produces a different slice of this volume.

Summary Convolutional layers are defined by a set of learned filters. Filters perform linear combinations of the input over localized spatial regions. Filters are small in spatial extent, but span the full depth of the input volume. The depth of each output activation map corresponds to the number of filters used.

As the input passes through successive convolutional layers, its representation becomes increasingly abstract, capturing high-level features such as shapes, textures, and patterns. This allows CNNs to efficiently and automatically learn features directly from raw image data, significantly improving performance on tasks like image classification.

3.3.2 Activation layer

ctivation layers introduce nonlinearities into the network, which is essential for the CNN to model complex patterns. Without nonlinear activation functions, a CNN would be equivalent to a linear classifier, limiting its capacity to capture intricate relationships in the data.

Activation functions are applied element-wise, meaning they operate on each individual value in the feature map (or volume) independently. Importantly, they do not change the size or dimensions of the volume but modify the values within it.

The most used activation functions are:

- *ReLU* (Rectified Linear Unit): the ReLU activation function is defined as:

$$f(x) = \max(0, x)$$

  It introduces nonlinearity by thresholding the input values—any negative values are set to zero, while positive values remain unchanged. ReLU has become the most widely used activation function in deep neural networks, particularly since its adoption in the AlexNet architecture. Its simplicity and effectiveness have made it the default choice for most CNNs. ReLU accelerates convergence during training by mitigating the vanishing gradient problem that often affects sigmoid and tanh functions.

  One issue with ReLU is the dying neuron problem, where some neurons can become permanently inactive if they consistently output zero. This occurs when their weights are updated in such a way that they never activate again.

- *Leaky ReLU*: to address the dying neuron problem, Leaky ReLU introduces a small slope for negative values, allowing a small gradient to flow even when the input is negative:

$$T(x) = \begin{cases} x & \text{if } x \geq 0 \\ 0.01x & \text{otherwise} \end{cases}$$

This modification helps prevent neurons from becoming inactive, ensuring they can still contribute to learning even when their inputs are negative.

- *Tanh* (Hyperbolic Tangent): the tanh activation function has a range of $(-1, 1)$ and is continuous and differentiable:
$$f(x) = \tanh(x)$$
Tanh is similar to the sigmoid function but is symmetric around zero, which can make optimization easier as it centers the data. However, like sigmoid, tanh can still suffer from the vanishing gradient problem when used in deep networks.

- *Sigmoid*: the sigmoid activation function is given by:
$$f(x) = \frac{1}{1 + e^{-x}}$$
It outputs values between 0 and 1, making it useful for binary classification tasks. However, sigmoid has largely fallen out of favor in deep CNN architectures due to the vanishing gradient problem, where gradients become very small during backpropagation, slowing down learning in deeper layers.

ReLU and Leaky ReLU are by far the most popular activation functions in CNNs due to their simplicity, computational efficiency, and performance in deep networks. Functions like sigmoid and tanh are more commonly used in traditional multi-layer perceptron (MLP) architectures and certain specialized layers like output layers, where specific output ranges are needed. By introducing nonlinearities, activation functions allow CNNs to learn complex, nonlinear mappings from input to output, which is crucial for tasks like image classification and object detection.

3.3.3 Pooling layer

Pooling layers are used to reduce the spatial dimensions (height and width) of the input volume, while preserving the most important information. This downsampling helps in making the network more computationally efficient and less prone to overfitting, as it reduces the number of parameters in the network.

Max pooling ooling layers operate independently on each depth slice (or channel) of the input volume. The most common type of pooling is Max Pooling, which selects the maximum value within a defined region of the input. For example, in a 2x2 pooling window, Max Pooling considers each 2x2 region of the input, and only retains the maximum value from that region. This results in a significant reduction of the spatial size—typically discarding 75% of the input data (since the 2x2 window compresses 4 values into 1). Max Pooling effectively downscales the image while keeping the most prominent features. This helps retain important characteristics like edges or corners, which are crucial for tasks like image classification.

Strides The stride determines how much the pooling window moves across the input volume. Typically, the stride is equal to the size of the pooling window (e.g., for 2x2 pooling, a stride of 2 is commonly used). This halves the height and width of the input volume.

If not explicitly specified, Max Pooling usually has a stride equal to the pooling size (e.g., stride = 2 for 2x2 pooling), reducing the spatial dimensions of the image by a factor of 2. If strides is unitary, is possible to use a stride of 1, where the pooling operation moves by one pixel

at a time. This doesn't reduce the spatial dimensions but instead applies the pooling operation over overlapping regions. This is typically combined with nonlinear pooling (like max pooling) and is useful when downsampling isn't needed but feature extraction is still required.

3.3.4 Final architecture

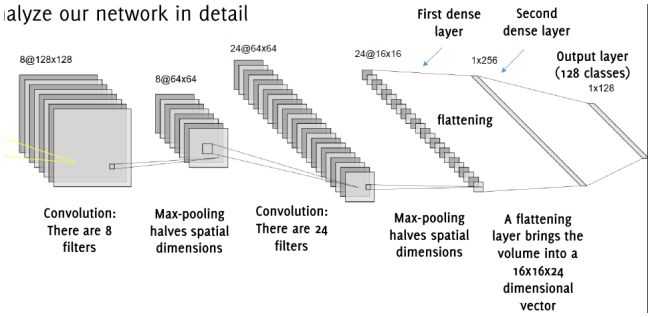


Figure 3.3: Convolutional Neural Network complete architecture

As we progress through the layers of a Convolutional Neural Network (CNN), the number of feature maps (or channels) increases, while the spatial dimensions of the image (height and width) decrease. This allows the network to capture more abstract and high-level features, while reducing the computational complexity.

Once the spatial dimensions of the feature maps are sufficiently reduced, the output is flattened into a 1D vector. This vector is then fed into a traditional fully connected (Dense) neural network, where each neuron is connected to every neuron in the previous layer. At this point, the spatial structure is lost, and the network focuses on combining the high-level features learned during the convolutional and pooling stages for the final classification.

Flattening converts the 3D feature maps into a single vector, enabling their input into fully connected layers. This step is essential as it transforms the hierarchical features extracted by the convolutional layers into a format suitable for classification.

The fully connected layers follow the flattened vector and are responsible for classification. The final fully connected layer typically has an output size equal to the number of target classes, providing a score (or probability) for each class. Dense layers are called fully connected because each output neuron is connected to every input neuron. The final layer often applies a softmax activation function to convert the raw output scores into probabilities for each class.

Learning representation Throughout the CNN, convolutional filters are learned to detect patterns relevant to the classification task. This includes applying operations such as ReLU (thresholding) and max pooling (downsampling) to progressively simplify the feature maps.

The architecture typically involves stacking many convolutional, activation, and pooling layers to learn meaningful representations from the data. This depth allows the network to capture low-level features in the initial layers, and more complex patterns (e.g., shapes, objects) in the deeper layers.

3.3.5 Convolutional and dense layers

Convolution is fundamentally a linear operation. When an input image is unrolled into a vector, the convolutional weights can be conceptualized as the weights of a dense layer. Both

convolutional and dense layers can be represented as linear operators, expressed mathematically as:

$$\mathbf{a} = \mathbf{W}\mathbf{x} + \mathbf{b}$$

Dense layers In dense layers, the weight matrix is fully populated, meaning each input neuron is connected to every output neuron through distinct weights. This results in a comprehensive interaction among neurons.

Convolutional layers Conversely, convolutional layers exhibit sparse connectivity. Here, each output neuron is influenced only by a subset of input neurons, reflecting the localized nature of convolution. As a result, most entries in the weight matrix are zero. The convolutional operation spans all channels of the input, leading to a circular structure within the weight matrix.

In convolutional layers, weights are shared across the entire output channel. The same filter is applied to compute the output for multiple positions, effectively shifting the filter across the input. This mechanism results in several output neurons sharing the same weights and bias values, which are constant across blocks corresponding to the filters.

Spatial invariance A defining characteristic of Convolutional Neural Networks (CNNs) is spatial invariance. All neurons within the same slice of a feature map utilize the same weights and biases, significantly reducing the total number of parameters in the network. The underlying assumption here is that if a feature is useful for computation at one spatial position, it should also be useful at different positions.

Every convolutional layer can be mathematically implemented as a fully connected layer that performs equivalent computations. The weight matrix for a convolutional layer would be large, with the number of rows equal to the number of output neurons and the number of columns equal to the number of input neurons. This matrix is predominantly sparse, filled mostly with zeros, except for specific blocks where local connectivity is established. Moreover, many weights within these blocks are identical due to parameter sharing. Interestingly, the converse interpretation is also valid and offers insightful perspectives on their functional equivalence.

3.3.6 Receptive field

The concept of the receptive field is fundamental in deep Convolutional Neural Networks. Unlike fully connected networks, where each output is influenced by the entire input, each output in a CNN is determined by a specific localized region of the input. This localized region is referred to as the receptive field for that output.

As the network depth increases, the receptive field expands. This expansion occurs through various mechanisms, including convolutions, strides, and max pooling, which collectively contribute to the widening of the receptive field. Typically, the receptive field is defined concerning the final output unit of the network relative to the input; however, this definition is equally applicable to intermediate feature maps.

In deeper layers, neurons are influenced by larger patches of the input. Notably, convolutions alone are sufficient to increase the receptive field, eliminating the necessity for max pooling.

As we progress deeper into the network:

- The spatial resolution of the feature maps diminishes.
- The number of feature maps generally increases.

At these deeper layers, the focus shifts to detecting higher-level patterns, allowing for a degree of insensitivity to their exact spatial location. This is essential, as higher-level patterns are often more abundant and informative than low-level details.

3.4 Training

Each Convolutional Neural Network can be conceptualized as a multi-layer perceptron characterized by sparse and shared connectivity. In principle, CNNs can be trained using gradient descent to minimize a loss function over a batch of data.

The gradient can be computed using backpropagation, which leverages the chain rule, provided that we can derive the operations for each layer of the CNN. It's important to consider weight sharing during the derivative computation, as this leads to a reduction in the number of parameters that need to be accounted for.

Deep learning models are often data-hungry, requiring large datasets for effective training. To cope with data scarcity we can employ two main tecniques.

- *Data augmentation*: each annotated image represents a broader set of images that are likely to belong to the same class. Data augmentation expands this set by applying transformations to the images, which can be:
  - *Geometric transformations*: such as shifts, rotations, affine/perspective distortions, shearing, scaling, and flipping.
  - *Photometric transformations*: including adding noise, modifying average intensity, altering contrast, or superimposing other images.
- *Transfer Learning*: using pre-trained models can leverage existing knowledge from larger datasets, effectively reducing the need for extensive new training data.

3.4.1 Image augmentation

To make the model more robust, the augmented images should retain the original label and promote invariance to transformations. However, augmentation should be carefully chosen to avoid transformations that obscure class-distinguishing features.

Mixup augmentation Mixup is a domain-agnostic data augmentation method that doesn't require knowledge of specific transformations. It creates virtual samples by interpolating pairs of examples and their labels. Given two training samples (I_i, y_i) and (I_j, y_j) (possibly from different classes), virtual samples can be created as follows:

$$\begin{cases} \tilde{I} = \lambda I_i + (1-\lambda) I_j \\ \tilde{y} = \lambda y_i + (1-\lambda) y_j \end{cases}$$

Here, $\lambda \in [0, 1]$ and y_i, and y_j are one-hot encoded labels. Mixup effectively extends the training distribution by encouraging the model to generalize through linear interpolations of features.

Augmented training pairs $\{(A_l(I), y)\}_l$ guide the model to learn invariances to selected transformations. However, synthetic augmentation might not fully capture real-world variations, such as background variations, exposure issues, and other factors.

Overfitting Data augmentation helps counteract overfitting by expanding the effective size of the training set. This technique can even be implemented as a layer within the network to ensure varied augmentation at each epoch. Additionally, data augmentation can mitigate class imbalance by generating more samples of minority classes, and in some cases, class-specific transformations can preserve relevant label information. If augmentation introduces hidden class-discriminative features, the model may inadvertently rely on these for classification.

Test time augmentation TTA enhances prediction accuracy by applying augmentations during testing. The process involves:

- Applying random augmentations to each test image.
- Classifying all augmented versions and storing the prediction probabilities.
- Aggregating the predictions to obtain a final decision.

While effective, TTA is computationally demanding and requires careful configuration of transformations to avoid excessive processing.

3.4.2 Transfer learning

In cases with limited data, transfer learning provides a powerful approach to leverage pre-trained models, such as ResNet, EfficientNet, or MobileNet, which have already learned general visual features. Transfer learning typically involves:

1. Removing the final fully connected (FC) layers of the pre-trained model.
2. Adding new FC layers tailored to the new problem, initialized randomly.
3. Freezing the pre-trained layers and training only the new layers.

There are two primary strategies in transfer learning:

- *Transfer learning*: only the newly added FC layers are trained, ideal when the pre-trained model closely matches the new problem but has limited new data.
- *Fine-tuning*: the entire model is retrained, but starting from the pre-trained weights. Fine-tuning works best when a substantial amount of new data is available, or when the pre-trained model doesn't directly match the new task. Lower learning rates are generally used for fine-tuning to preserve learned weights while gradually adapting them.

To maximize the effectiveness of a pre-trained model we can add a new output layer with minimal parameters. In transfer learning, train only this output layer. In fine-tuning, unfreeze some of the last layers and train the entire network at a reduced learning rate.

3.5 Architectures

The first Convolutional Neural Network (CNN), known as LeNet, was introduced in 1998. LeNet was pioneering for its ability to recognize handwritten characters, marking a significant step forward in image recognition.

3.5.1 AlexNet

AlexNet, developed by Krizhevsky, Sutskever, and Hinton in 2012, expanded upon LeNet's architecture. While similar in its use of convolutional layers, AlexNet features five convolutional layers and three fully connected layers. Its input size is $224 \times 224 \times 3$, and it trains a substantial 60 million parameters.

AlexNet introduced key innovations to counteract overfitting and improve performance. Notably, it was the first network to incorporate ReLU activations, dropout regularization, weight decay, and max-pooling. These techniques made AlexNet far more efficient and robust, setting a new standard in Deep Learning for image classification.

3.5.2 VGG16

VGG16, introduced in 2014, is a deeper evolution of AlexNet's convolutional structure. It uses smaller 3×3 filters and increases the network's depth, scaling up the parameter count to 138 million. VGG16 achieved remarkable success, securing first place in the localization track and second place in the classification track of the 2014 ImageNet Challenge. The input size for VGG16 remains the same as AlexNet at $224 \times 224 \times 3$.

The VGG16 paper offers an in-depth exploration of the impact of network depth on performance. The authors systematically increase the network's depth by adding more convolutional layers, made possible through the exclusive use of small 3x3 filters in all layers. This approach allows for larger receptive fields through a sequence of 3×3 convolutions, yielding several advantages:

- Fewer parameters compared to using larger filters in a single layer.
- Increased non-linearity due to additional activation layers, enhancing the network's representational power.

This strategy enables VGG16 to capture complex image features efficiently, while maintaining manageable computational complexity.

3.5.3 Networks in Networks

The Network in Network (NIN) architecture introduces a novel approach to convolutional neural networks by replacing standard convolutional layers with mlpconv layers. In this approach, rather than using a single convolutional layer followed by an activation function, each convolutional layer is substituted with a small Multi-Layer Perceptron, comprising a stack of fully connected layers followed by ReLU activations. This design allows for a more powerful functional approximation than traditional convolutional layers, as it captures complex feature interactions across channels in a sliding manner over the entire image.

NIN also incorporates Global Average Pooling (GAP) layers as a structural regularization method, replacing fully connected layers at the network's end. Instead of flattening feature maps and feeding them into dense layers, GAP computes the average of each feature map, producing a single value per map. This is equivalent to multiplying the feature map by a non-trainable, block-diagonal constant matrix, in contrast to a trainable dense matrix in traditional FC layers.

Global Averaging Pooling GAP has several advantages:

- *Reduction in parameters and overfitting*: fully connected layers contain a high number of parameters and are prone to overfitting. GAP reduces parameter count, making the model lighter and more robust.
- *Direct connection to output classes*: GAP enables a straightforward connection between feature maps and output classes, enhancing model interpretability, particularly useful for tasks such as localization.
- *Robustness to spatial transformations*: by averaging over spatial dimensions, GAP enhances the network's robustness to image shifts and transformations, unlike fully connected layers, which are sensitive to the spatial arrangement of features.
- *Compatibility with variable input sizes*: GAP can accommodate images of different input sizes, making NIN more flexible than conventional architectures that require a fixed input size.

Finally, GAP is followed by a simple softmax layer for classification. Importantly, the number of feature maps before the GAP layer should match the number of output classes. However, if there is a mismatch, a hidden layer can adjust the feature dimension. This structure reduces overfitting, improves interpretability, and makes the network resilient to spatial transformations, as convolutional feature extraction is shift-invariant, but fully connected layers are not.

NIN's use of GAP layers aligns feature maps directly with classes, acting as a structural regularizer and making the network lighter, more interpretable, and robust to spatial changes in input images.

3.5.4 InceptionNet

The InceptionNet architecture, introduced by Google in 2014, marked a significant step forward in Deep Learning, achieving high performance with relatively low computational costs. Unlike traditional networks that simply increase depth or width to improve accuracy—at the cost of a significant rise in parameters and risk of overfitting—InceptionNet introduced inception modules as a more efficient and scalable approach. InceptionNet was highly successful, winning the 2014 ILSVRC (ImageNet Large Scale Visual Recognition Challenge) with an impressive 6.7% top-5 error rate.

InceptionNet uses parallel convolutional layers of varying filter sizes (1×1, 3×3, 5×5) within each module. These filters capture features at multiple scales, allowing the network to detect small and large image features within a single layer. The outputs of these different filters are concatenated along the channel dimension, preserving the spatial dimensions while significantly expanding the depth of the activation map.

Inception module To manage computational costs, each inception module includes 1×1 convolutions as bottleneck layers before the larger 3×3 and 5×5 convolutions. This bottleneck layer reduces the number of input channels, significantly lowering the total computation required for each module. By reducing dimensionality at strategic points, the inception module effectively balances depth expansion with manageable resource requirements. This setup results in a network that is both deep and computationally efficient, capable of leveraging non-linearities while minimizing redundant calculations.

Some of the key elements of the inception module include:

- Multiple filter sizes (1×1, 3×3, 5×5) at the same layer level, enabling detection of different feature scales.
- 1×1 bottleneck layers before larger convolutions, which reduce input channels and thus the number of operations.
- Zero-padding to preserve spatial dimensions across layers, allowing smooth concatenation of outputs from each branch in depth.

GoogleLeNet The original version of InceptionNet, called GoogleLeNet, contains 22 layers, including multiple inception modules and pooling layers. It starts with two initial layers of convolution and pooling, followed by a stack of nine inception modules. Instead of using fully connected layers, GoogleLeNet applies Global Average Pooling at the end, followed by a linear classifier and softmax for classification, which helps reduce overfitting and parameter count. Overall, GoogleLeNet contains only 5 million parameters—a small fraction of other deep networks of its time.

To further enhance training stability and address the dying neuron problem, GoogleLeNet adds two auxiliary classifiers at intermediate layers. These auxiliary classifiers compute intermediate losses during training, promoting meaningful feature extraction at earlier layers. During inference, these classifiers are omitted, leaving only the final output layer for predictions.

The key features of InceptionNet are:

1. *Parallel connections and multiple filter sizes*: each inception module includes multiple filter sizes (1×1, 3×3, 5×5), enabling feature extraction across multiple scales in parallel.
2. *Bottleneck layers* (1×1 convolutions): by reducing the number of input channels before applying larger filters, bottleneck layers optimize computational efficiency and parameter usage.
3. *Auxiliary losses for training stability*: auxiliary classifiers enhance training convergence by providing additional loss signals, although they are removed for inference.

InceptionNet's modular, scalable design and computational efficiency made it a transformative architecture in Deep Learning, setting a standard for designing high-performing networks with limited resources.

3.5.5 ResNet

ResNet (Residual Network) introduced a revolutionary approach to building very deep neural networks, with 152 layers trained on ImageNet and even deeper models, such as 1202-layer networks, trained on CIFAR datasets. ResNet was the winner of the 2015 ILSVRC competition, achieving a 3.6% top-5 error rate, surpassing human-level performance in classification and localization tasks.

The key insight behind ResNet is that simply increasing the depth of a network does not guarantee improved accuracy. Empirical results showed that as depth increases, performance often worsens due to optimization difficulties, rather than overfitting, as both training and test errors increase. This indicates that deeper networks are inherently harder to optimize than shallower ones, even when overfitting isn't an issue.

ResNet addresses this problem by introducing identity shortcut connections (or skip connections), which create residual learning blocks. Instead of learning the full output function

$F(x)$, the network learns a residual mapping $F(x) = H(x) - x$, where $H(x)$ is the ideal output function. This formulation implies that the network only needs to learn a small delta or adjustment over the input, which makes it easier to optimize.

The benefits of identity shortcut connections are:

- *Mitigates the vanishing gradient problem*: the shortcut connection allows gradients to flow more freely through the network, making it possible to train very deep networks.
- *No additional parameters*: the identity mapping does not add any new parameters or significant computational cost.
- *Enables identity propagation*: if layers are optimal, the residual weight can converge to zero, allowing information to pass unaltered via the identity mapping.

In essence, the residual block encourages each block to focus on learning a small update over the previous layer's output, which is easier to optimize than a direct transformation. This approach allows the network to stack more layers without degradation.

Architecture ResNet is constructed using residual blocks, where each block consists of two convolutional layers followed by ReLU activations, with an identity shortcut connection that bypasses these layers, directly adding the input to the output. This design enables the network to learn residual mappings, which significantly eases the optimization of deeper networks.

For very deep models, such as those with over 50 layers, ResNet incorporates bottleneck layers. These layers use 1×1 convolutions to reduce the dimensionality of the input before applying larger convolutions, which helps mitigate the computational load, similar to the approach used in the inception module. Additionally, downsampling occurs through convolutions with a stride of 2, which reduces the spatial dimensions of the feature maps while doubling the number of filters at each downsampling stage.

The architecture concludes with a Global Average Pooling layer, which reduces the spatial dimensions to a single value per feature map, followed by a softmax classifier for final classification. This design eliminates the need for fully connected layers, making the model more efficient and less prone to overfitting. Overall, ResNet's use of residual blocks and bottleneck layers allows for the construction of very deep networks while maintaining computational efficiency and optimizing performance.

The innovations of ResNet are:

1. *Residual learning*: each block learns a residual mapping, simplifying optimization and allowing much deeper networks.
2. *Identity shortcuts*: skip connections help mitigate gradient vanishing, enabling stable training of networks with hundreds of layers.
3. *Bottleneck layers*: 1×1 convolutions reduce the depth within each block, keeping computational complexity manageable.

3.5.6 MobileNet

MobileNet was specifically designed to optimize deep learning models for mobile and embedded applications, where resource constraints such as computational power, memory, and battery

life are critical. Traditional convolutional networks, particularly those with standard 2D convolutional layers, are computationally demanding and require a large number of parameters, making them impractical for mobile devices.

In standard 2D convolutional layers, each filter mixes information across all input channels, which results in a high number of parameters and operations. These layers can be quite computationally expensive, making them unsuitable for devices with limited resources.

MobileNet addresses these limitations by introducing a more efficient approach known as separable convolutions, which break the convolution operation into two simpler steps:

1. *Depth-wise convolution*: this step applies a 2D convolution independently on each input channel. Instead of mixing channels together, it performs a convolution on each individual channel separately, significantly reducing the number of computations.

2. *Point-wise convolution*: this step is a 1×1 convolution that combines the output of the depth-wise convolution. Since it uses 1×1 filters, it does not perform spatial convolution, but rather mixes the information across channels.

By splitting the convolution into these two steps, MobileNet significantly reduces the computational load and the number of parameters. The depth-wise convolution minimizes operations across input channels, and the point-wise convolution efficiently mixes the information across channels without the spatial complexity of a full convolution.

The benefits of these network are:

- *Fewer parameters*: since each convolution step is more efficient, MobileNet models require far fewer parameters than traditional networks.

- *Lower computational cost*: the separation of convolution into depth-wise and point-wise operations reduces the number of floating-point operations (FLOPs), making the network less computationally demanding.

- *Suitable for mobile and embedded devices*: with fewer parameters and operations, MobileNet is well-suited for real-time inference on devices with limited processing power and memory.

3.5.7 Latest architectures

Recent advancements in deep learning have led to the development of several novel architectures, each addressing specific challenges such as network efficiency, depth, and feature reuse. Some of the most notable architectures include:

- *Wide ResNet*: this variation of ResNet increases the number of filters in each layer, making the residual blocks wider instead of deeper. By increasing the width rather than the depth, Wide ResNet is more computationally efficient and can be parallelized more effectively, improving performance without a significant increase in complexity.

- *ResNeXt*: ResNeXt expands upon the ResNet framework by introducing multiple parallel paths within each block. While similar to the inception module, where activation maps are processed in parallel, ResNeXt differs in that all paths share the same topology. This increases the network's capacity while maintaining computational efficiency, as the parallel paths allow for greater diversity of learned features.

- *DenseNet*: DenseNet introduces short connections between convolutional layers within each block, where each layer takes the output of all preceding layers as its input. This dense connectivity pattern leads to better feature reuse, as every feature is propagated through the entire network. It helps alleviate the vanishing gradient problem by maintaining strong gradients throughout the network, which improves training efficiency and allows the network to learn more compact representations.

- *EfficientNet*: EfficientNet proposes a new scaling method that uniformly scales all dimensions of the network—depth, width, and resolution—using a compound coefficient. This simple yet effective method achieves state-of-the-art performance with fewer parameters and operations by optimizing the scaling of these dimensions in a balanced way. EfficientNet sets a new standard for efficient deep learning models by improving accuracy while reducing computational overhead.