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.

We will begin with the evolution from the Perceptron to modern Neural Networks, focusing on the Feed Forward architecture. The training of Neural Networks through backpropagation, along with best practices to prevent overfitting, including cross-validation, stopping criteria, weight decay and dropout.

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. 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 will be a focal point, including their training processes and data augmentation strategies.

Furthermore, we will cover structural learning and Long-Short Term Memory networks, exploring their applications in text and speech processing. Topics such as Autoencoders, data embedding techniques like Word2vec.

Finally, we will discuss transfer learning with pre-trained deep models, examine extended models such as Fully Convolutional Neural Networks for image segmentation and object detection methods, and explore generative models like Generative Adversarial Networks.

Contents

CHAPTER 1

Deep Learning

1.1 Introduction

Definition (*Machine Learning*)**.** A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P if it improves with experience E.

Machine Learning allows systems to automatically improve and adapt by learning from data, enabling them to make predictions or decisions without being explicitly programmed.

1.1.1 Supervised Learning

In Supervised Learning, the model is provided with a dataset $\mathcal{D} = \{x_1, x_2, \ldots, x_N\}$, where each input x_i has a corresponding desired output or label t_i. The goal is for the model to learn the relationship between inputs and outputs so it can correctly predict outputs for new, unseen inputs. Supervised Learning tasks include:

- *Classification*: the model assigns a category or label to input data.
- *Regression*: the model predicts a continuous numerical value.

1.1.2 Unsupervised Learning

In Unsupervised Learning, the model is given a dataset $\mathcal{D} = \{x_1, x_2, \ldots, x_N\}$ without any labels or target outputs. The goal is to identify patterns, structures, or regularities within the data.

A common task in Unsupervised Learning is clustering, in which the model groups similar data points into clusters based on their intrinsic properties. Unsupervised Learning is often used for exploratory data analysis and dimensionality reduction.

1.1.3 Reinforcement Learning

Reinforcement learning focuses on training a model to make decisions by interacting with an environment. At each step, the model takes an action a_t based on the current state s_t, and the environment provides feedback in the form of a reward r_t. The objective is to learn a policy that maximizes cumulative rewards over time.

1.2 Deep Learning

Deep Learning is a specialized branch of Machine Learning that leverages Neural Networks with multiple layers to process and learn from vast amounts of data. Its core strength lies in automatically learning meaningful data representations, enabling it to tackle complex problems that traditional Machine Learning methods struggle with.

Traditional Machine Learning methods often rely on hand-engineered features, which are predefined attributes or patterns extracted from the data. However, these approaches can fail when the dataset contains irrelevant or redundant features that obscure meaningful patterns, or the relationships in the data are too intricate to be captured by manual feature engineering.

Deep Learning overcomes these limitations by automatically learning hierarchical representations of data, where higher layers in the network capture increasingly abstract and complex features. This ability to learn directly from raw data eliminates the need for extensive manual feature engineering, making Deep Learning especially effective for high-dimensional and unstructured data such as images, audio, and text.

1.3 Perceptron

In the 1940s, computers were already excelling at executing tasks with precision and performing arithmetic operations at remarkable speeds. However, researchers aspired for more than mere computational efficiency. They envisioned machines capable of handling noisy data, interacting directly with their environment, operating in a massively parallel and fault-tolerant manner, and adapting to dynamic circumstances. Their ambition led to the quest for a new computational model

Human neurons The human brain consists of an enormous network of computing units, with each neuron connected to many others through synapses. Information transmission in the brain relies on chemical processes: dendrites gather signals from synapses, which can either excite or inhibit the neuron. When the cumulative signal surpasses a certain threshold, the neuron fires, releasing an electrical charge.

This biological computational model is marked by its distributed nature, fault-tolerant redundancy, and inherent parallelism. The Perceptron, inspired by these principles, became one of the first attempts to emulate the brain's computational capabilities.

1.3.1 Perceptron

In the mid-20th century, researchers explored various models of the brain. In 1943, Warren McCulloch and Walter Pitts proposed the threshold logic unit, where the activation function operated as a threshold unit. Later, in 1957, Frank Rosenblatt introduced the first Perceptron, featuring weights encoded in potentiometers and electric motors for weight adjustments during learning. By 1960, Bernard Widrow introduced a critical improvement: the inclusion of a bias term to represent the threshold value.

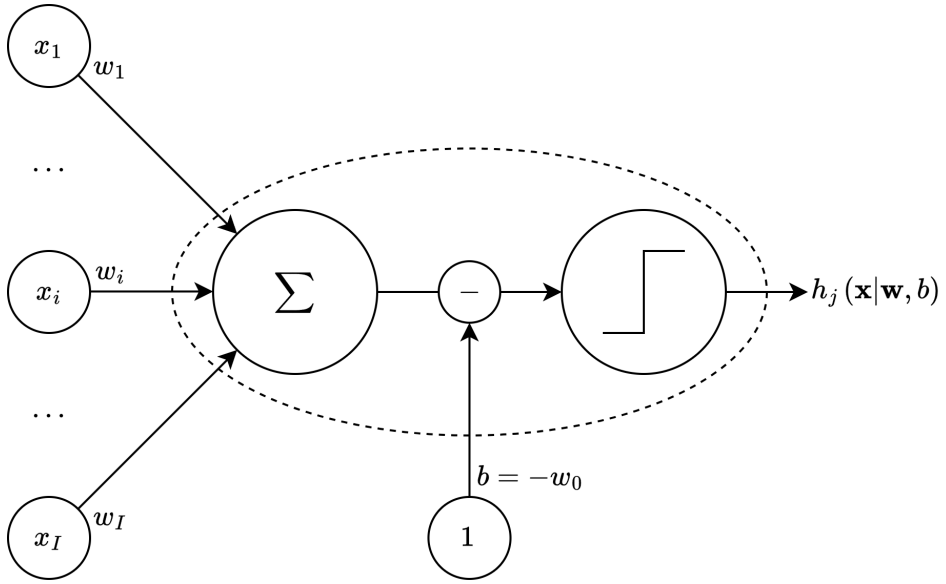

Figure 1.1: Perceptron

The output function $h_j(\mathbf{x} \mid \mathbf{w}, b)$ is given by:

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

The activation function in an artificial neuron can take various forms. The Perceptron computes a weighted sum of its inputs and applies a thresholding function to produce its output:

$$h_j(\mathbf{x} \mid \mathbf{w}) = h_j\left(\mathbf{w}^T \mathbf{x}\right) = \text{sign}(\mathbf{w}^T \mathbf{x})$$

This defines a linear classifier, where the decision boundary is a hyperplane described by:

$$\mathbf{w}^T \mathbf{x} = 0$$

The linear decision boundary enables the Perceptron to implement Boolean logic operators. However, the Perceptron struggles when data cannot be separated by a linear boundary. In such cases, additional strategies are necessary. These challenges paved the way for more advanced models, such as Multi-Layer Perceptrons.

Hebbian learning Hebbian learning, often summarized by the phrase cells that fire together, wire together, follows these principles:

1. Initialize weights randomly.
2. Adjust weights for each sample, but only if the sample is misclassified.

The weight update rule is given by:

$$\mathbf{w}_i^{(k+1)} = \mathbf{w}_i^{(k)} + \eta \mathbf{x}_i^{(k)} t^{(k)}$$

Here, η is the learning rate, $\mathbf{x}_i^{(k)}$ represents the i-th input to the Perceptron at time k, and $t^{(k)}$ is the desired output at the same time.

CHAPTER 2

Feed Forward Neural Networks

2.1 Introduction

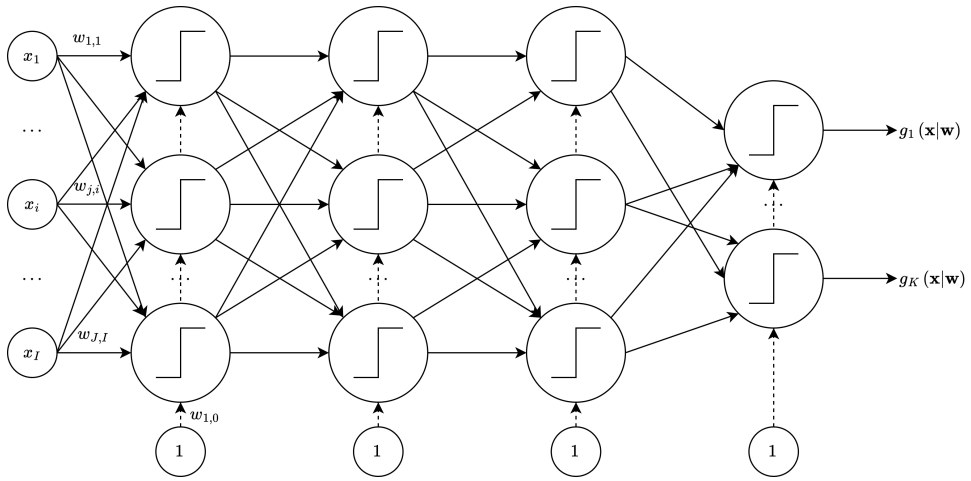

Figure 2.1: Multi-Layer Perceptron architecture

A Multi-Layer Perceptron is a type of Feed Forward Neural Network designed to process input data through multiple layers of interconnected nodes. Each layer is Fully Connected to the next, forming a structure capable of learning complex relationships in data. The Multi-Layer Perceptron architecture is organized into three main components:

- *Input layer*: this layer receives the raw input data and serves as the starting point for the network. The size of the input layer corresponds to the number of input features, which varies depending on the specific problem being addressed.
- *Hidden layers*: these intermediate layers are responsible for transforming the input data into more abstract representations. The number of hidden layers and the neurons within each layer are critical hyperparameters, typically determined through experimentation or optimization techniques.
- *Output layer*: this final layer generates the network's output. The size of the output layer depends on the task at hand.

Multi-Layer Perceptrons are inherently non-linear models, characterized by their use of activation functions, the number of neurons in each layer, and the values of the connection weights. The connections between layers are represented by weight matrices, where the weights determine the strength of the influence between neurons. The output of each neuron in the network depends solely on the outputs from the preceding layer, facilitating a forward propagation of information.

2.2 Activation functions

Activation functions are a fundamental component of Neural Networks, introducing the non-linearity necessary for the model to capture complex patterns in data. They determine the output of each neuron and influence the network's ability to learn.

Name	**Function**
Linear	$g(a) = a$
Sigmoid	$g(a) = \dfrac{1}{1+e^{-a}}$
Hyperbolic tangent	$g(a) = \frac{e^a - e^{-a}}{e^a + e^{-a}}$
Rectified Linear Unit	$g(a) = \begin{cases} 0 & \text{if } a < 0 \\ a & \text{otherwise} \end{cases}$
Leaky Rectified Linear Unit	$g(a) = \begin{cases} 0.01a & \text{if } a < 0 \\ a & \text{otherwise} \end{cases}$
Exponential Linear Unit	$g(a) = \begin{cases} \alpha(e^a - 1) & \text{if } a < 0 \\ a & \text{otherwise} \end{cases}$

Table 2.1: Most used activation functions

The features of each activation function are the following:

- *Sigmoid*: often used in output layers to model probabilities due to its range $(0, 1)$ and simplicity. However, it suffers from saturation, leading to small gradients.
- *Hyperbolic tangent*: preferred in hidden layers as it outputs values in $(-1, 1)$, centering data and often resulting in faster convergence than sigmoid.
- *ReLU*: popular for its simplicity and efficiency, it mitigates the vanishing gradient problem but is susceptible to dying neurons, where certain neurons become inactive.
- *Leaky ReLU* and *ELU*: variants of ReLU designed to address the dying neuron issue by allowing small gradients for negative inputs.

Vanishing gradient Functions like sigmoid and hyperbolic tangent to saturate for extreme input values, producing gradients close to zero. Since backpropagation relies on gradient updates, this can cause gradients to vanish as they propagate through the layers. This issue significantly hinders the training of deep networks.

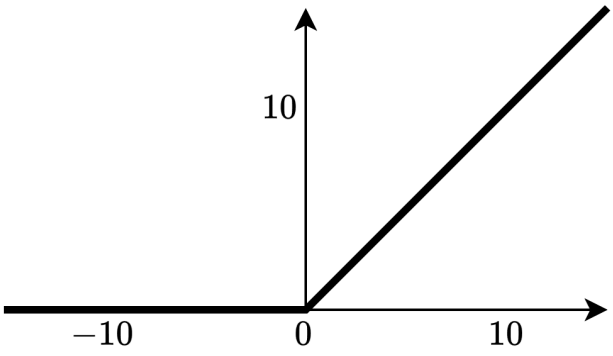

Figure 2.2: Rectified Linear Unit

ReLU has revolutionized Deep Learning by addressing the vanishing gradient problem and enabling faster and sparser activations. Its advantages include also the scale invariance with respect to output scaling. However, it has limitations:

- Non-zero-centered outputs, which can slow optimization.
- Non-differentiability at zero, although this is often addressed by practical implementations.
- Dying neurons, which occur when gradients for negative inputs become zero, effectively disabling these neurons during training.

Output layers The choice of activation function in the output layer depends on the problem type:

- *Regression*: linear activation.
- *Binary classification*: hyperbolic tangent or sigmoid.
- *Multi-class classification*: softmax, defined as:

$$y_k = \frac{e^{z_k}}{\sum_k e^{z_k}}$$

  Here, z_k is the activation value of the k-th output neuron, and the softmax function produces a probability distribution across classes.

Hidden layers Hidden layers benefit from activation functions that introduce non-linearity, enabling the network to learn complex relationships. Popular choices include:

- *Sigmoid* and *hyperbolic tangent*: useful for smaller networks but prone to vanishing gradients in deeper architectures.
- *ReLU* and its variants: often preferred for Deep Learning due to their computational efficiency and ability to handle vanishing gradients.

2.2.1 Function approximation

Theorem 2.2.1. *A single hidden layer Feed Forward Neural Network with S-shaped activation functions can approximate any measurable function to any desired degree of accuracy on a compact set.*

This universal approximation theorem highlights the expressive power of Neural Network. However, practical challenges such as optimization, generalization, and the need for sufficient hidden units often require careful design choices. For classification tasks, a single additional hidden layer is typically sufficient to achieve good performance.

2.3 Training

Training a neural network involves learning a set of parameters, primarily weights $\mathbf{w}$ to approximate the target values t_n as accurately as possible. Given a training dataset $\mathcal{D} = \{\langle \mathbf{x}_1, t_1 \rangle, \ldots, \langle \mathbf{x}_N, t_N \rangle\}$, the goal is to determine parameters such that, for new data, the model's predictions $y_n(\mathbf{x}_n \mid \boldsymbol{\theta})$ are close to the true target values t_n. In essence, the training process seeks parameters that generalize well, ensuring $g(\mathbf{x}_n \mid \mathbf{w}) \approx t_n$ even for unseen inputs. In Neural Networks, the discrepancy between predictions and targets is often quantified using the Sum of Squared Errors:

$$\mathcal{L}(\mathbf{w}) = \sum_{n}^{N} \left(t_n - g(\mathbf{x}_n \mid \mathbf{w})\right)^2$$

This loss function is nonlinear with respect to the weights, making the optimization process more challenging than for simpler linear models.

Nonlinear optimization To minimize the error $\mathcal{L}(\mathbf{w})$, we use optimization techniques to adjust the weights. The aim is to find $\mathbf{w}$ that satisfies:

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

For Neural Networks, solving this equation directly is infeasible due to the complexity and nonlinearity of the error surface. Instead, we employ iterative methods such as gradient descent, which progressively updates the weights in the direction that reduces the loss.

2.3.1 Gradient descent

Gradient descent updates the weights using the formula:

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

Here, η is the learning rate, controlling the step size in each iteration.

Gradient descent may struggle with local minima or oscillations in the error surface. To mitigate this, a momentum term is often added:

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

Here, α is the momentum coefficient, which determines how much past gradients influence the current update.

Nestorov accelerated gradient The Nesterov accelerated gradient method builds upon the momentum technique to improve optimization performance. It achieves this by first making a momentum-based prediction and then refining the update using the gradient evaluated at the predicted position. This two-step process helps anticipate the trajectory of the optimization, resulting in faster convergence.
The update process is as follows:

1. *Momentum-based prediction*: the algorithm predicts the next position using the momentum term:
$$\mathbf{w}^{\left(k+\frac{1}{2}\right)} = \mathbf{w}^{(k)} - \alpha \frac{\partial E(\mathbf{w})}{\partial \mathbf{w}}\bigg|_{\mathbf{w}^{(k-1)}}$$
2. *Gradient-based correction*: the weights are then updated by evaluating the gradient at the predicted position:
$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - \eta \frac{\partial E(\mathbf{w})}{\partial \mathbf{w}}\bigg|_{\mathbf{w}^{\left(k+\frac{1}{2}\right)}}$$

Multiple restarts For non-convex error surfaces, the optimization process can converge to local minima depending on the initial weights. To improve the chances of finding the global minimum, a common strategy is to perform multiple restarts:

1. Initialize the weights $\mathbf{w}$ randomly several times.
2. Train the model independently for each initialization.
3. Select the solution with the lowest training error.

This approach explores diverse regions of the parameter space, increasing the likelihood of finding a better overall solution.

Computation Using all data points to compute weight updates, as in batch gradient descent, can be computationally expensive, especially for large datasets. The gradient of the error function in this approach is given by:

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

While precise, this method becomes inefficient as N grows. Alternative approaches balance computation time and optimization efficiency:

- *Stochastic gradient descent*: the gradient is computed using a single data point at each iteration. This method is computationally cheaper and enables faster iterations. However, the updates can have high variance, leading to oscillations in the optimization process.
- *Minibatch gradient descent*: a compromise between batch gradient descent and stochastic gradient descent is mini-batch gradient descent, where gradients are calculated using a small, randomly selected subset (mini-batch) of the dataset. Mini-batch gradient descent strikes a balance, reducing computational costs while maintaining stable updates. This approach allows for faster convergence and scalability, especially with large datasets.

Backpropagation In Neural Networks, gradients are efficiently computed through backpropagation, a method that leverages the network's structure to update weights in parallel. Backpropagation utilizes the chain rule for derivatives, enabling efficient computation of gradients for complex, layered functions. Backpropagation is executed in two steps:

1. *Forward pass*: the input propagates through the network to compute the output for each neuron. Local derivatives (dependent only on a neuron's immediate inputs) are calculated and stored for later use.

2. *Backward pass*: gradients are propagated backward using the stored values from the forward pass. The error's partial derivatives with respect to each weight are computed and used to update the weights.

Layers learning rate In Deep Neural Networks, layers exhibit varying learning dynamics due to differences in the magnitude of gradients. Conversely, deeper layers might experience exploding gradients, which can destabilize training. These disparities highlight the need for layer-specific learning rates to optimize learning across all layers effectively.

2.3.2 Batch normalization

Batch normalization is a powerful technique to combat internal covariate shift by normalizing the activations of each layer, ensuring they approximate a unit Gaussian distribution during training.
Batch normalization is typically applied after Fully Connected or convolutional layers, and before the activation functions.

Given a mini-batch $\mathcal{B} = \{\mathbf{x}_1, \dots, \mathbf{x}_m\}$, the algorithm normalizes the inputs and applies learnable parameters γ and β.

Algorithm 1 Batch normalization

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

Batch normalization improve the gradient flow, enabling the use of higher learning rates. Moreover, the initialization of the weights is less crucial since it performs implicit regularization.

2.3.3 Weight initialization

The effectiveness of gradient descent in training Neural Networks heavily depends on the initialization of weights at the start. Different strategies for initializing weights are:

- *Zero initialization*: setting all weights to zero leads to symmetric gradients, causing every neuron in the same layer to update identically, rendering the network ineffective.

- *Large weights*: initializing with large random weights can cause gradients to explode as they propagate through the layers, destabilizing training.

- *Small weights*: using small random values can work for shallow networks. However, in deeper architectures, this can cause gradients to diminish, leading to the vanishing gradient problem.

To address these limitations, more sophisticated initialization methods have been developed to balance the gradient flow through the network.

Xavier initialization Proposed by Glorot and Bengio, Xavier initialization ensures that the variance of activations remains consistent across layers. This helps prevent signals from vanishing or exploding as they propagate forward or backward. The weights are drawn from a distribution:

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

Here, n_{in} represents the number of input units to the neuron.

For more stability, the approach can be extended to account for both input and output units. In this case, weights are initialized as:

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

He initialization Designed specifically for networks using ReLU activation functions, He initialization sets weights with a larger variance to accommodate the ReLU's behavior, where only positive values are passed. The weights are initialized as:

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

This ensures that gradients maintain their magnitude during backpropagation, which is especially crucial in deep architectures. By maintaining better gradient flow, He initialization prevents dying neurons.

2.4 Loss function

The loss function measures the discrepancy between the predicted outcomes of a model and the actual observed data, serving as the foundation for parameter estimation. This concept is central to many learning algorithms and optimization problems, regardless of the underlying model or data distribution.

Let $\boldsymbol{\theta} = \begin{pmatrix}\theta_1 & \theta_2 & \dots & \theta_p\end{pmatrix}^T$ represent the vector of parameters of a model. The goal is to estimate the parameters $\boldsymbol{\theta}$ that minimize the loss function or maximize the likelihood of observing the given data. Here's a generalized process:

1. *Define the likelihood function*: the likelihood function, $\mathcal{L}(\boldsymbol{\theta})$, represents the probability of the observed data given the model parameters.

2. *Log-likelihood transformation*: since the likelihood function often involves a product of probabilities, it is computationally convenient to work with the log-likelihood function, $\log \mathcal{L}(\boldsymbol{\theta})$, which transforms the product into a sum. This simplifies both mathematical derivations and numerical stability.

3. *Compute the gradient of the log-likelihood*: to estimate $\boldsymbol{\theta}$ using Maximum Likelihood Estimation, the log-likelihood function is maximized. This requires calculating the gradient and setting it to zero to identify critical points.

4. *Solve for the optimal parameters*: depending on the complexity of the model, the optimization problem can be solved:

   - *Analytically*: when a closed-form solution exists, such as estimating the mean of a Gaussian distribution.
   - *Numerically*: for more complex models, iterative optimization techniques like gradient descent are used to approximate the optimal parameters.

Selection Choosing an appropriate loss function is crucial for defining the task and steering the learning process. A loss function not only quantifies the difference between predicted and actual values but also plays a significant role in shaping the optimization behavior of the model. When designing a loss function, several key factors should be considered:

- *Leverage knowledge of the data distribution*: it's important to integrate any prior knowledge or assumptions about the underlying data distribution when selecting a loss function.
- *Exploit task-specific and model knowledge*: a good loss function should reflect the objectives of the specific task at hand.
- *Creativity in loss function design*: predefined loss functions may not capture all aspects of the problem. In such cases, creativity in designing a custom loss function can help better model the problem and improve performance.

2.4.1 Perceptron

In the case of the Perceptron, it can be shown that the error function minimized by the Hebbian rule is closely related to the distance of misclassified points from the decision boundary. The goal is to minimize the following loss function:

$$\mathcal{L}(\mathbf{w}) = -\sum_{i \in M} t_i(\mathbf{w}^T \mathbf{x})$$

To minimize this error function $\mathcal{L}(\mathbf{w})$ using stochastic gradient descent, we compute the gradients with respect to the model parameters. Stochastic gradient descent is then applied iteratively to each misclassified point:

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} + \alpha t_i \mathbf{x}_i$$

Here, α denotes the learning rate. This iterative process adjusts the weights to progressively reduce misclassification.

2.5 Validation

Underfitting occurs when the model is too simplistic to capture the underlying patterns in the data, leading to poor performance. To address this, we can increase the complexity of the model. However, increasing complexity too much can result in overfitting. In this case, the

model starts to fit the noise in the training data rather than the true underlying patterns, which reduces its ability to generalize to new, unseen data.

The goal during training is to find a model that strikes a balance between complexity and generalization. The optimal model should perform well not just on the training data but also on future data that it has not seen before.

The loss on the training data alone is not a reliable indicator of future performance, as it tends to be overly optimistic. The training data can differ from new, unseen data, so it's crucial to evaluate the model on a separate test set to get a more accurate estimate of its generalization capability. To assess the model's ability to generalize, the dataset is typically divided into three distinct subsets:

- *Training set*: used to train the model and adjust its parameters.
- *Validation set*: used to fine-tune the model's hyperparameters and select the best model configuration.
- *Test set*: used for the final evaluation of the model's performance on unseen data.

2.5.1 Cross validation

Cross validation s a technique that helps estimate a model's performance by using the available training data for both training and error estimation on unseen data. This method allows us to evaluate the model's ability to generalize without the need for a separate test set in the initial stages of model development. Various cross validation techniques include:

- *Hold-out validation*: when a large dataset is available, it is split into distinct subsets for training and validation.
- *Leave-one-out cross validation*: the model is trained on all data points except one, which is then used for validation. This process is repeated for each data point in the dataset, ensuring that each instance is used for both training and validation. While this method provides an unbiased estimate of performance, it can be computationally expensive, especially with large datasets.
- *K-fold cross-validation*: the dataset is randomly split into K equally sized subsets (folds). The model is then trained on $K-1$ folds and validated on the remaining fold. This process is repeated for all folds, with each fold serving as the validation set once. The final model performance is averaged across all K iterations.

2.5.2 Early stopping

Overfitting in Neural Networks often manifests as a consistently decreasing training error as the number of gradient descent iterations increases. While the model continues to improve on the training data, its ability to generalize to unseen data may start to degrade after a certain point. Early stopping is a regularization technique aimed at preventing overfitting by stopping the training process once the model's performance on a validation set begins to decline.
In early stopping we start by holding out a validation set, nad then while we train on the training set we cross validate and if the validation error starts to increase while the training error continues to decrease, halt the training process.

Early stopping can be applied to model selection and evaluation, especially when fine-tuning hyperparameters. These hyperparameters, such as learning rate or network architecture, significantly influence the model's capacity and ability to generalize.

2.5.3 Weight decay

Regularization techniques aim to reduce overfitting by constraining the model's complexity based on prior assumptions about the data. In the context of Neural Networks, weight decay is one of the most widely used regularization methods. It helps control model complexity by penalizing large weights, which are often associated with overfitting.

In traditional training, we typically maximize the likelihood of the data given the model parameters. However, to impose additional structure and reduce complexity, we can take a Bayesian approach by incorporating a prior distribution over the weights $\mathbf{w}$, leading to a Maximum A Posteriori estimation:

$$\mathbf{w}_{\text{MAP}} = \underset{\mathbf{w}}{\operatorname{argmax}} \Pr(\mathcal{D} \mid \mathbf{w}) \Pr(\mathbf{w})$$

In practice, small weights tend to improve the generalization ability of Neural Networks. This can be modeled by assuming a Gaussian prior on the weights, which encourages the model to prefer smaller weights. Incorporating this prior into the optimization problem results in the following objective function:

$$\hat{\mathbf{w}} = \underset{\mathbf{w}}{\operatorname{argmin}} \sum_{n=1}^{N} (t_n - g(\mathbf{x}_n \mid \mathbf{w})^2) + \gamma \|\mathbf{w}\|_2^2$$

Here, γ is the regularization parameter that controls the strength of the weight decay. Larger values of γ penalize large weights more heavily, promoting smaller weights and thereby reducing the model's complexity. By controlling γ, we can find a balance between fitting the data well and maintaining a model that generalizes effectively.

2.5.4 Dropout

Dropout is a stochastic regularization technique designed to mitigate overfitting in Neural Networks by randomly deactivating certain neurons during training. This forces the model to learn more robust and independent feature representations, preventing neurons from becoming too dependent on one another (coadaptation). By randomly dropping out neurons, the model is encouraged to generalize better by relying on multiple paths for information flow. The process of applying dropout during training involves applying a mask to deactivate neurons in each layer with a certain probability, iteratively. Once training is complete, all neurons are reactivated for the full network during testing. The behavior of the full network can be thought of as the averaged result of all the sub-networks trained during the dropout process. To compensate for the dropouts during training, weight scaling is applied at test time, effectively averaging the outputs of all possible sub-networks.

CHAPTER 3

Convolutional Neural Networks

3.1 Computer vision

Computer vision is an interdisciplinary field that focuses on enabling computers to interpret and understand the visual world using digital images or videos. Initially, many computer vision techniques relied on mathematical models and statistical analysis of images. However, with the rise of Machine Learning modern approaches have shifted toward data-driven methods. These methods have significantly enhanced the effectiveness and adaptability of algorithms for solving complex visual tasks.

3.1.1 Digital images

A digital color image is typically represented using three separate matrices, each corresponding to one of the primary colors: red, green, and blue (RGB). Each matrix element represents the intensity of a pixel and is usually encoded with values ranging from 0 to 255.

Although images are often stored in compressed formats to save disk space, they must be decompressed for processing in memory. This decompression increases the size of the data considerably, which poses challenges in terms of memory management and processing power. The complexity of managing large amounts of image data becomes even more pronounced when working with video files, as they contain a vast number of frames that must be processed sequentially. When using Neural Networks for image processing tasks, raw image data needs to be efficiently handled, which can result in large computational and memory demands.

3.1.2 Local transformations

Local transformations involve modifying each pixel in an image based on the values of its neighboring pixels within a defined neighborhood U. The transformation can be expressed as:

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

Here, $\mathbf{I}$ is the input image, $\mathbf{G}$ is the output image, U defines the neighborhood around the pixel, and T_U is a spatial transformation function. This function can be either linear or non-linear, depending on the transformation.

For a pixel at coordinates (r, c), 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 transformation function T_U is applied repeatedly across all pixels in the image, making the transformation spatially invariant.

Local linear filters In the case of linear spatial transformations, the output at a given pixel (r, c), denoted as $T_U[\mathbf{I}](r, c)$, is a linear combination of the pixel values within the neighborhood U. This can be expressed as:

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

Here, $w(u, v)$ represents the weights associated with each pixel in the neighborhood U. These weights can be interpreted as defining a filter which is applied uniformly across the entire image. This approach allows the same operation to be applied to all pixels, making it a simple and efficient method for image processing.

This approach applies to both grayscale and color (RGB) images, where each color channel is processed separately:

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

The correlation between a filter $\mathbf{w}$ and an image $\mathbf{I}$ can be computed using:

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

3.2 Image classification

Image classification is the task of 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. The goal of the classifier is to map the image to a corresponding class label. This mapping can be represented as:

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

3.2.1 Linear classifier

In a linear classifier, the image $\mathbf{I}$ is first flattened into a vector. The image has three color channels (RGB), and each channel is linearized column by column.

Each element of this flattened vector corresponds to a neuron in the input layer of the neural network, and the network's output layer consists of L neurons, where L is the number of classes. The network architecture connects each input neuron to every output neuron, resulting in a matrix of weights $\mathbf{w} \in \mathbb{R}^{L\times d}$.

For each input image, the classifier computes a score for each class. The score for the i-th class is the inner product between the image vector $\mathbf{x}$ and the corresponding row in the

weight matrix $\mathbf{w}$. Since this is a linear classifier, nonlinearity is not required, and the softmax activation function can be omitted.

The classifier output $\mathcal{K}(\mathbf{I})$, is a vector of class scores, where each component s_i represents the score for class i:

$$\mathcal{K}(\mathbf{x}) = \mathbf{w}^T\mathbf{x}$$

The classifier's goal is to assign the image to the class with the highest score.

3.2.1.1 Training

The training process involves optimizing the classifier's parameters $\mathbf{w}$ to minimize the loss function over the training data. The loss function $\mathcal{L}(\mathbf{w})$ quantifies the error between the predicted class scores and the true labels:

$$\mathbf{w} = \underset{\mathbf{w}}{\operatorname{argmin}} \sum_{(\mathbf{x}_i, y_i)} \mathcal{L}(\mathbf{w})$$

Geometric interpretation From a geometric perspective, the classifier can be viewed as a linear function in the feature space $\mathbb{R}^d$, where each image is represented as a point in this space. The classifier is a hyperplane that separates different class points.

In the case of a two-dimensional feature space, he classifier's decision boundary is represented by a line. This decision boundary is defined by the equation $f(\mathbf{x}) = w_0x_0 + w_1x_1 + w_2x_2$.

3.2.2 K-Nearest-Neighbours

In the k-nearest neighbors algorithm, the class of a test image is predicted by the most frequent label among its k-closest training images in the feature space. Given a test image $\mathbf{I}_j$, the predicted class $\hat{y}_j$ is determined as follows:

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

Here, j^* represents the mode (most frequent class label) of the k-nearest images to the test image $\mathbf{I}_j$. The distance function $d(\cdot)$ typically used in k-NN could be the Euclidean distance or Manhattan distance, which quantifies the similarity between images in the feature space. Although k-NN is easy to implement and requires no training, it is computationally expensive at test time and struggles with high-dimensional data.

3.2.3 Challenges

Image classification faces a range of challenges due to the complexity and variability inherent in image data. These challenges include:

1. *Dimensionality*: images and videos are high-dimensional, with each image consisting of thousands or millions of pixels. This leads to issues with memory and computational resources.

2. *Label ambiguity*: a single label may not adequately represent the full content of an image, as images can have multiple elements, and classification might not capture all relevant details.

3. *Invariance to transformations*: images can change significantly due to various transformations such as changes in illumination, deformations, or viewpoints. However, these transformations often do not alter the core content or label of the image.

4. *Inter-class variability*: within the same class, images can differ significantly in terms of background, lighting, scale, or perspective.

5. *Perceptual similarity*: similarity between images does not always correspond to pixel-level similarity.

3.3 Convolutional Neural Network

Convolutional Neural Networks are a specialized type of Deep Learning architecture designed to automatically and efficiently extract spatial features from images. A typical Convolutional Neural Network architecture consists of convolutional layers, activation layers, and pooling layers, which are stacked in sequence to learn increasingly abstract representations of the input data. As the network depth increases, the number of channels (i.e., the depth of the feature maps) grows, while the spatial dimensions (height and width) of the feature maps decrease.

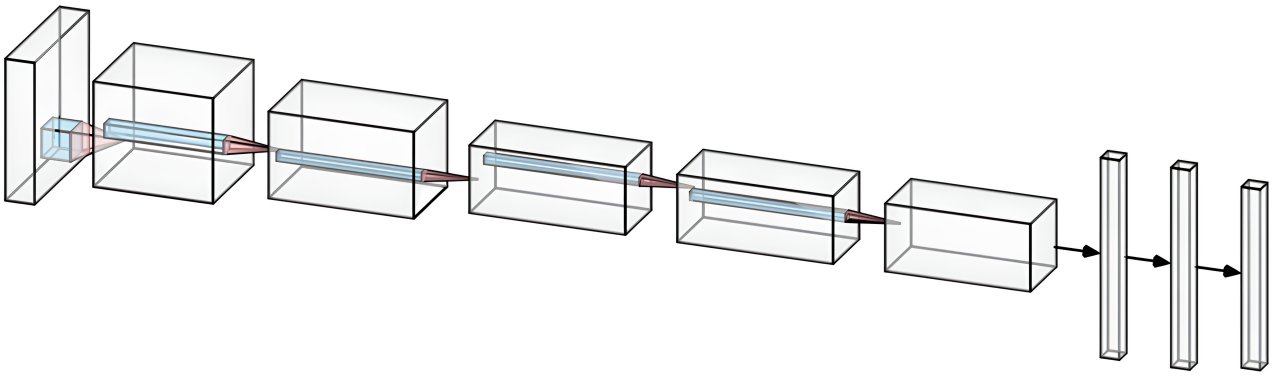

Figure 3.1: Convolutional Neural Network transformations

3.3.1 Convolutional layer

The convolution operation is at the heart of a Convolutional Neural Network. During the convolutional layer, small regions of the input (called receptive fields) are processed using filters, which are learned during training.

The input image is represented as a volume with dimensions: height h, width w, and depth. The filters, which also have a defined height and width, slide over the input image to extract features at different locations. For each region of the input, the filter produces an output known as an activation map.

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

$$a(r, c, l) = \sum_{u,v,k} \mathbf{w}^l(u, v, k)x(r + u, c + v, k) + b^l$$

Here, U represents the small receptive field, C is the number of input channels, and b^l is the bias term. The result of applying each filter to the input is an activation map corresponding to a particular feature.

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

$$\text{parameters} = (h_r \cdot h_c \cdot c)n_f + n_f$$

Here, h_r and h_c are the filter's height and width, c is the input depth (number of channels), and n_f is the number of filters.

Key characteristics of the convolutional layers include:

- *Local processing*: convolutions operate over small spatial regions U, meaning each filter processes localized sections of the image at a time.
- *Channel-wise processing*: filters span the entire depth of the input volume, allowing them to capture information across all channels
- *Output volume*: each filter produces a slice of the output volume, also known as an activation map, where each filter outputs a different slice corresponding to a distinct feature.

As the input passes through successive convolutional layers, its representation becomes increasingly abstract. The early layers may learn simple features like edges and textures, while deeper layers capture higher-level structures like shapes and objects.

3.3.2 Activation layer

Activation layers are crucial for introducing nonlinearities into the network, enabling Convolutional Neural Networks to model complex patterns and relationships in data. Without these nonlinear activation functions, a Convolutional Neural Network would behave similarly to a linear classifier, severely limiting its ability to capture the intricate, hierarchical features typically present in image data.

Activation functions are applied element-wise, meaning they operate independently on each individual value in the feature map (or volume). While they do not alter the spatial dimensions of the feature map, they modify the values within it, enabling the network to learn more complex and abstract features.

The most widely used activation functions in Convolutional Neural Networks are ReLU and Leaky ReLU. These functions are preferred due to their simplicity, computational efficiency, and strong performance in deep networks. Both functions help mitigate issues like vanishing gradients, allowing networks to train more effectively.

3.3.3 Pooling layer

Pooling layers are designed to reduce the spatial dimensions (height and width) of the input volume while retaining the most significant features. This downsampling process enhances the computational efficiency of the network and helps prevent overfitting by reducing the number of parameters and computations. Pooling layers operate independently on each channel of the input volume, meaning they are applied separately to the depth of the feature maps.

Max pooling The most common pooling technique is max pooling, where the maximum value is selected from a defined region (typically a square or rectangular window) of the input. Max pooling effectively reduces the spatial size of the input while preserving the most prominent features.

Strides The stride determines how much the pooling window moves across the input volume during the pooling operation. It defines the step size between each operation as the window slides over the feature map.

In most cases, the stride is equal to the size of the pooling window, meaning the window moves non-overlapping across the input. However, a stride of one (unitary stride) means the pooling window moves by just one pixel at a time, creating an overlap between the regions being pooled. This can help in extracting more detailed features without significantly reducing the spatial dimensions.

Global Average Pooling Global Average Pooling reduces the entire feature map for each channel into one average value. This significantly reduces the number of parameters compared to traditional Fully Connected layers, making the network simpler and less prone to overfitting, while maintaining its ability to capture important features across the image.

Global Average Pooling is then followed by a simple softmax layer for classification. Thus, the number of feature maps before the Global Average Pooling layer should match the number of output classes. However, if there is a mismatch, a hidden layer can adjust the feature dimension.

3.3.4 Architecture

In a Convolutional Neural Network, once the spatial dimensions of the feature maps have been sufficiently reduced through successive convolutional, activation, and pooling layers, the output is flattened into a one-dimensional vector. This vector is then fed into a traditional Fully Connected Neural Network. At this stage, the spatial structure of the image is no longer preserved, and the network shifts focus to combining the high-level, abstract features learned in the earlier layers.

The final Fully Connected layer typically has an output size corresponding to the number of target classes, producing a score (or probability) for each class. The raw output scores are usually passed through a softmax activation function, which converts them into probabilities that sum to one, indicating the likelihood of each class.

Throughout the Convolutional Neural Network, convolutional filters are learned to detect patterns that are relevant to the specific classification task at hand. The architecture of the network generally consists of stacking multiple convolutional, activation, and pooling layers.

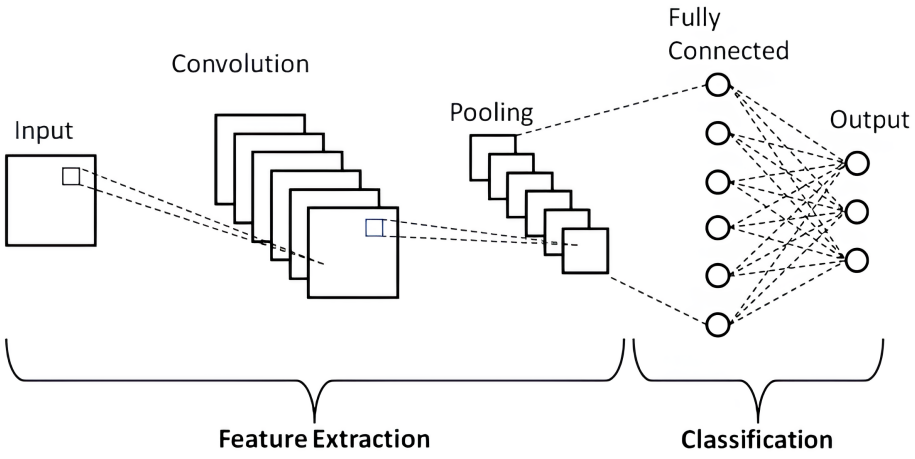

Figure 3.2: Convolutional Neural Network architecture

3.3.5 Convolutional and dense layers

In dense layers, the weight matrix is fully populated, meaning every neuron is connected to every other neuron in the adjacent layers. This leads to comprehensive interactions between neurons, allowing the network to model complex relationships. However, this full connectivity comes with a large number of parameters, which can make training more computationally expensive and prone to overfitting.

In contrast, convolutional layers exhibit sparse connectivity. In these layers, each output neuron is influenced only by a subset of input neurons, which reflects the localized nature of convolutions. As a result, most of the entries in the weight matrix are zero. The convolutional operation spans all channels of the input, leading to a weight matrix with a circular structure.

In convolutional layers, the weights are shared across the entire output channel. The same filter is applied across multiple positions of the input to compute the output. This means that different positions in the input share the same filter, reducing the total number of parameters in the network. The same filter is applied to each local region of the input, which allows the network to recognize the same feature at different spatial locations.

Spatial invariance A defining characteristic of Convolutional Neural Networks is spatial invariance. This means that the network is capable of detecting features regardless of where they appear in the image. All neurons within the same slice of a feature map use the same set of weights and biases. This significantly reduces the number of parameters compared to Fully Connected layers, as many neurons share the same weights and biases across spatial positions. The fundamental assumption behind this approach is that if a feature is useful in one part of the image, it should also be useful in other parts.

3.4 Training

Convolutional Neural Networks are trained using gradient descent to minimize a loss function over a batch of data. The gradient of the loss function is computed using backpropagation. This enables the model to adjust its weights iteratively based on the errors made in prediction.

Deep Learning models are often data-hungry and require large datasets to train effectively. To address the challenge of data scarcity, two common techniques can be used to augment the training process:

- *Data augmentation*: this technique artificially expands the dataset by applying various transformations to the original images, creating variations that are still representative of the same class. These transformations can be geometric (rotations, flips) or photometric (changes in brightness or contrast).

- *Transfer Learning*: this method leverages knowledge gained from training a model on a large dataset. By using pre-trained models as a starting point, the model can benefit from the general features already learned, reducing the need for a large amount of new training data.

3.4.1 Augmentation

To ensure the augmented images are useful, they should retain the original label and promote invariance to transformations. However, care must be taken not to apply transformations that obscure important class-distinguishing features, which could degrade model performance.

Mixup is a domain-agnostic data augmentation technique that does not require prior knowledge of specific transformations. It works by creating new virtual samples through linear interpolation between pairs of examples and their labels Mixup helps expand the training distribution, encouraging the model to generalize better by considering linear interpolations between features. Although Mixup can enhance generalization, it may not fully capture all real-world variations that might occur.

Benefits Data augmentation is a powerful technique to prevent overfitting by effectively expanding the size of the training set. This is especially helpful when there is limited data available. Augmentation can also be integrated directly into the training pipeline as a layer within the network, ensuring that new variations are generated at each epoch. Furthermore, it can address class imbalance by generating additional samples of minority classes, and class-specific transformations can help preserve relevant label information.

Test Time Augmentation Test time augmentation is a technique that improves prediction accuracy by applying augmentations during the testing phase. The process involves:

1. Applying random augmentations to each test image.
2. Classifying all augmented versions of the image.
3. Storing the prediction probabilities for each version.
4. Aggregating these predictions to obtain a final decision.

While test time augmentation can improve performance, it is computationally expensive and requires careful configuration of the transformations to avoid excessive processing time. The effectiveness of test time augmentation depends on the nature of the task and the transformations used during testing.

3.4.2 Transfer learning

Transfer learning offers a powerful way to leverage pre-trained models. Pre-trained models have already learned general visual features from large datasets. Transfer learning allows you to repurpose these models for a new task, minimizing the need for extensive new data. The typical process for applying transfer learning involves:

1. Removing the final Fully Connected layers of the pre-trained model, which are specific to the original task.
2. Adding new Fully Connected layers that are tailored to the new problem, initialized randomly.
3. Freezing the pre-trained layers and only the newly added layers are trained.

Fine-tuning After this step, fine-tuning is needed. With fine-tuning, the entire model is retrained, but starting with the pre-trained weights. The key is to unfreeze some or all of the layers from the pre-trained model and train them on the new dataset. Fine-tuning is typically done at lower learning rates to preserve the knowledge learned from the original task while allowing the model to adapt to the new data gradually.

To maximize the effectiveness of a pre-trained model in the context of transfer learning, the new output layer should have minimal parameters. This reduces the computational load and ensures that only the relevant final weights are learned for the new task. When performing transfer learning, the focus is typically on training only this new output layer, while fine-tuning involves unfreezing some of the later layers and retraining the entire network at a reduced learning rate.

3.4.2.1 Architectures

The main Convolutional Neural Networks that can be used for transfer learning are as follows:

Architecture	**Description**
LeNet	Pioneering convolutional network for handwritten digit recognition
AlexNet	Introduction of ReLU activations, dropout regularization, weight decay, and max-pooling
VGG16	Deeper architecture with smaller 3×3 filters and increased depth for better feature capture
Network in Network	Use of convolutional layers and Global Average Pooling to reduce parameters and improve robustness
InceptionNet	Parallel convolutions of varying filter sizes, inception modules, and bottleneck layers for computational efficiency
ResNet	Introduction of residual learning via identity shortcut connections, enabling very deep networks
MobileNet	Separable convolutions to reduce parameters and computational cost
Wide ResNet	Increased width of residual blocks rather than depth, improving efficiency and parallelization
ResNeXt	Parallel paths within each block, allowing for feature diversity while maintaining efficiency
DenseNet	Dense connectivity between layers, promoting feature reuse and mitigating vanishing gradients
EfficientNet	Uniform scaling of depth, width, and resolution for balanced and efficient model scaling

Inception module The inception module is a key component of the inception architecture, designed to efficiently capture features at different scales. It incorporates a combination of convolution operations with varying filter sizes within a single layer. The module includes 1×1 convolutions as bottleneck layers before applying larger convolutions, such as 3×3 and 5×5. These bottleneck layers reduce the number of input channels, which in turn reduces the computational cost for each module. By strategically reducing dimensionality, the inception module enables deeper networks while maintaining a manageable computational load, optimizing both depth and resource usage.

3.5 Other problems

3.5.1 Semantic segmentation

Semantic segmentation is a computer vision task that involves grouping pixels in an image that share common characteristics or belong to the same object category. The main objective of semantic segmentation is to assign a label from a predefined set of categories Λ to each pixel (r, c) in an image $\mathbf{I}$. Unlike instance segmentation, which distinguishes between different instances of the same object category, semantic segmentation only focuses on classifying pixels based on their category. In other words, all pixels belonging to the same category share the same label, without differentiating between individual objects of that category.

Formally, semantic segmentation can be described as a mapping function:

$$\mathbf{I} \rightarrow \mathbf{S} \in \Lambda^{R \times C}$$

Here, $\mathbf{S}$ represents the segmentation map, where each element $S(x, y)$ is a class label assigned to the pixel at position (x, y). The output $\mathbf{S}$ is a grid of size $R \times C$ (the resolution of the image), with each pixel labeled according to its corresponding class in the set Λ.

3.5.1.1 Fully Convolutional Neural Networks

Fully Convolutional Neural Networks are an extension of traditional Convolutional Neural Networks, designed to overcome the constraint of fixed input sizes. Unlike conventional Convolutional Neural Networks, which rely on Fully Connected layers that require a fixed-size input, Fully Connected Convolutional Neural Networks eliminate these layers and replace them with convolutional and subsampling layers that operate across the entire image.

In Fully Convolutional Neural Networks, the convolutional and pooling layers slide over the entire input image, regardless of its size, producing feature maps whose resolution scales appropriately with the input dimensions. Crucially, the spatial extent of the latent space is preserved throughout the network, meaning the spatial relationships between features are maintained across all layers.

For each output class, Fully Convolutional Networks generate a heatmap, where the resolution of the heatmap is typically lower than the input image, and each position in the heatmap represents the class probabilities for the corresponding region in the input image, based on the receptive field of the convolutional layers.

Classification to segmentation The process of transforming a pre-trained Fully Convolutional Neural Network designed for classification into a semantic segmentation network involves adapting the model to handle input images of arbitrary size while maintaining high-resolution, pixel-level predictions. A common challenge in this transformation is that the heatmaps produced by the classification network are typically low-resolution, which makes it difficult to achieve accurate pixel-wise segmentation. The goal is to improve the precision of the segmentation by refining the coarse predictions generated by these heatmaps. Several approaches can be used to overcome this challenge:

- *Direct heatmap prediction*: this method generates heatmaps directly from the network, but it often results in imprecise predictions due to the use of masks that cover regions larger than a single pixel.

- *Shift and stitch*: leverages the downsampling factor, which is the ratio of the input size to the output heatmap size. It involves computing heatmaps for all possible shifts of the input image. These shifted heatmaps are then stitched back together, with each heatmap's pixel corresponding to the center of the receptive field. While this method improves prediction accuracy, it is computationally expensive due to the need to process multiple shifts of the image.

- *Learned upsampling*: eliminates strides in pooling layers, whether max-pooling or convolutional. By doing so, it supports end-to-end learning and allows the network to perform inference directly on full-sized images. This method enables fine-tuning of pre-trained classification models for segmentation tasks, and it can be trained to learn how to effectively upsample the low-resolution heatmaps into high-resolution pixel predictions.

3.5.1.2 U-net

A common approach to semantic segmentation involves cropping the input image into patches, classifying each patch individually, and assigning the predicted class label to the central pixel of the patch. While this method is straightforward, it has notable limitations, such as a small receptive field and low efficiency, as it does not capture long-range dependencies and global context well.

Semantic segmentation inherently involves a trade-off between semantic understanding and spatial localization. On one hand, fine-grained details are crucial for accurate pixel-level predictions, while on the other hand, a global understanding of the image structure is necessary to provide coherent segmentation. To address this, segmentation models typically combine fine-grained layers (which capture local details) with coarse layers (which provide broader contextual information). This fusion enables the model to make localized predictions that respect the global structure of the image, improving both accuracy and spatial coherence.

A typical semantic segmentation network consists of two main components:

1. *Encoder*: the encoder functions similarly to a standard classification network. It uses a Convolutional Neural Network to progressively extract high-level features from the input image, reducing the spatial resolution while capturing essential semantic information.

2. *Decoder*: the decoder is responsible for reconstructing the spatial resolution by upsampling the encoded features. This step converts the high-level features back into pixel-level predictions, preserving fine details and sharp contours. Effective upsampling is crucial for accurate localization and precise class predictions. Several methods are used to reconstruct spatial resolution during the decoding process:

    - *Nearest neighbor interpolation*: this method simply duplicates values to increase resolution.
    - *Bed of nails*: values are placed at specific positions within an expanded matrix, with the remaining entries set to zero.
    - *Transpose convolution*: this technique learns filters for upsampling, which can be implemented as a combination of standard upsampling followed by a convolutional layer.

While deep networks are necessary for extracting high-level semantic features, excessive downsampling can result in the loss of fine-grained spatial resolution. To mitigate this, many semantic segmentation networks incorporate:

- *Skip connections*: these direct connections pass features from the encoder to the decoder, ensuring that fine-grained details are preserved during the reconstruction process.
- *Layer fusion*: by combining features from both shallow and deep layers, the model can retain spatial precision while benefiting from rich semantic context.

Loss function In semantic segmentation, the categorical cross-entropy loss is commonly used to evaluate model performance at the pixel level. This loss function is computed pixel-wise, with the overall loss being the sum of the individual pixel losses across the entire image or a specified region of interest. Each pixel's contribution ensures that the model optimizes performance across the entire image.

When processing an image or a region within it, the pixel-wise losses provide a mini-batch estimate for gradient computation, which allows for efficient training, even with large images. This approach enables the model to optimize its parameters effectively while ensuring precise segmentation at the pixel level.

The U-Net is a specialized neural network architecture developed for semantic segmentation, particularly suited for biomedical image analysis. Its distinctive U-shaped structure is the basis for its name and is composed of the following components:

- *Contracting path*: this part of the network consists of a series of repeated blocks. Each block typically includes two 3×3 convolutions followed by ReLU activation and a max pooling operation. This downsampling process progressively captures high-level features while reducing the spatial dimensions of the image.
- *Expanding path*: this path mirrors the contracting path, consisting of blocks that progressively upsample the feature maps. Each block includes an upsampling operation followed by two 3×3 convolutions with ReLU activation. The expanding path helps recover spatial resolution, converting the encoded features back into pixel-level predictions.
- *Final layer*: the final layer uses a 1×1 convolution to reduce the feature maps to the desired number of output classes, N, allowing the network to classify each pixel into one of the predefined categories.

The key strength of U-Net lies in its ability to maintain spatial information throughout the process. The architecture is fully convolutional, which means that it can handle input images of varying sizes. Unlike traditional Fully Connected layers, the convolutional approach ensures that spatial dependencies are preserved.

Additionally, U-Net employs full-image training and typically uses a weighted loss function to address the challenge of imbalanced datasets, which is common in tasks like biomedical image segmentation. The weighted loss function ensures that the network performs well even on underrepresented classes, and it helps improve the classification of small or boundary regions in images.

3.5.1.3 Training

The training process for semantic segmentation can be approached in two primary ways:

- *Patch-based training*: in this approach, a classification network is trained using patches cropped from annotated images. Each patch is assigned the label corresponding to the pixel at its center. During training, the network minimizes the classification loss over a mini-batch of patches. Batches are randomly assembled, and resampling is often used

to address class imbalance, ensuring that underrepresented classes are included more frequently in the training data. However, patch-based training can be computationally inefficient, as convolution operations are redundantly applied to overlapping regions of the input image, leading to wasted resources and slower training times.

- *Full-image training*: in this method, the entire image region effectively serves as a mini-batch, allowing the network to compute gradients more efficiently. For Fully Convolutional Networks, full-image training is conceptually similar to patch-wise training, but the batch consists of all receptive fields from the network's units within the image. This eliminates the redundant convolution computations associated with overlapping patches, making full-image training significantly more efficient. Furthermore, it supports end-to-end learning, as the network processes the whole image in one go, improving both performance and computational efficiency.

While patch-based training offers flexibility, particularly in addressing class imbalance through patch resampling, it suffers from inefficiencies due to redundant convolution calculations for overlapping regions. Full-image training, on the other hand, takes advantage of the spatially efficient operations of Fully Convolutional Networks, avoiding redundant computations. However, full-image training introduces its own challenges. Full-image training, which processes entire image regions, tends to reduce this randomness. To reintroduce stochasticity, techniques like random masking of image regions can be applied.

Additionally, full-image training does not have the flexibility to resample patches for balancing class frequencies. To address class imbalance, it is common to weight the loss function to account for differences in label frequency across the image. This ensures that the network gives more emphasis to underrepresented classes, leading to more balanced performance across all segments of the image.

3.5.2 Localization

Localization involves identifying the position of an object within an image and classifying it. The task is to assign the object class to the image from a fixed set of categories, and then locate the object within the image by predicting the bounding box around it. To train a model for localization, an annotated training set is required, where each image has a label and a bounding box around the object.

Formally, we want to map an input image $\mathbf{I} \in \mathbb{R}^{R \times C \times 3}$ to the bounding box coordinates (x, y, h, w):

$$\mathbf{I} \rightarrow (x, y, h, w)$$

Here, (x, y) are the coordinates of the top-left corner of the bounding box, and (h, w) are the height and width of the bounding box, respectively.

One straightforward solution is to train a regression network to directly predict the bounding box coordinates. This approach involves training a network to predict the class label and the coordinates simultaneously in a multi-task learning setting. This means that the network must predict both categorical and continuous outputs. Thus, the network must handle both a classification task and a regression task simultaneously, making it a multi-task learning problem. Since the two outputs (class label and bounding box) have different natures (discrete for class and continuous for bounding box), the training process needs to combine these two tasks.

Loss function To optimize both the classification and regression tasks, we use a combined loss function. The multi-task loss function is a weighted sum of two separate losses:

$$\mathcal{L}(\mathbf{x}) = \alpha\mathcal{S}(\mathbf{x}) + (1-\alpha)\mathcal{R}(\mathbf{x})$$

Here, $\mathcal{S}(\mathbf{x})$ is the classification loss, $\mathcal{R}(\mathbf{x})$ is the regression loss, and $\alpha \in [0, 1]$ is a hyperparameter that controls the trade-off between classification and regression losses.

Choosing an optimal value for α can be challenging, and cross-validation is recommended to tune this parameter effectively. It's important to note that simply adjusting α might not yield meaningful results; a better approach is to monitor the performance of the combined loss function and adjust based on empirical results.

To implement the multi-task loss function, the training loop must be adjusted to compute both losses for each image, combine them, and backpropagate the total loss. This allows the model to optimize for both tasks simultaneously, adjusting the parameters based on the combined objective.

3.5.2.1 Weakly Supervised Learning

In Supervised Learning, a model $\mathcal{M}$ performs inference from input data X to output labels Y, as shown by the following mapping:

$$\mathcal{M} : X \to Y$$

This requires a training set, where the training pairs consist of labeled data that are of the same type as the classifier's input and output. However, for certain tasks, such as segmentation, gathering such annotated data can be expensive and time-consuming.

Weakly Supervised Learning aims to overcome the challenge of expensive annotations by utilizing labels that are easier to gather. The model is trained on a different domain K and is expected to perform inference in the target domain Y. In other words, weak supervision allows the model to solve a task using labels that are not as detailed or expensive to collect as the ideal annotations.

Class Activation Mapping Class Activation Mapping is a technique used to visualize which regions of an image a model focuses on when making classification decisions. It works by highlighting the areas of the image that contribute the most to the prediction of a specific class.

To apply Class Activation Mapping, a Convolutional Neural Network must meet the following criteria: a Global Average Pooling layer and a single Fully Connected layer follows the Global Average Pooling layer.

To obtain the saliency maps the following steps are needed:

1. *Feature maps and Global Average Pooling*: after the convolutional block of the Convolutional Neural Network, there are n feature maps $f_k(x, y)$, each having a resolution close to that of the input image. The Global Average Pooling layer computes the average of each feature map, resulting in n scalar values.

2. *Fully Connected layer and class scores*: the Fully Connected layer computes the class scores for a specific class S_c as a weighted sum of the Global Average Pooling outputs:

$$S_c \sum_k w_k^c F_k$$

   Here, w_k^c represents the importance of feature map k for class c.

3. *Reinterpreting class scores*: the class scores S_c can be rewritten as:

$$S_c = \sum_{x,y} \sum_{k} w_k^c f_k(x, y)$$

4. *Defining the Class Activation Mapping*:

$$M_c(x, y) = \sum_{k} w_k^c f_k(x, y)$$

Here, $M_c(x, y)$ indicates the importance of the activations at position (x, y) for predicting class c.

Global Average Pooling layer encourages the identification of the entire object because all activation values contribute to the classification. Global Max Pooling focuses on specific discriminative features, as only the highest activation value influences the classification. This makes Global Max Pooling more suitable for tasks that rely on identifying distinctive features rather than the entire object.

Weakly Supervised localization In the context of localization, the goal is to predict the coordinates of a bounding box around an object in an image. However, obtaining annotated bounding boxes for every image can be expensive. Instead, we can train a model to perform localization using a dataset annotated only with image-class labels, without bounding box information. By training a classifier using image-class labels, the model can still learn to estimate the location of the object in the image. The model essentially learns to perform localization by leveraging the class information and applying weak supervision to make localization predictions, even when the full set of annotations is unavailable. For Weakly Supervised localization we can use thresholding Class Activation Mapping values.

3.5.2.2 Convolutional Neural Networks visualization

Visualization techniques provide insight into the internal functioning of Convolutional Neural Networks by highlighting the features and regions that contribute to a network's decision. These techniques can be particularly useful for understanding, debugging, and trusting the model's predictions.

The filters in the first convolutional layer are relatively straightforward to interpret. Deeper layers represent higher-level abstractions, combining lower-level features into complex patterns and objects.

Neuron activation To understand what a specific neuron in a deep layer responds to:

1. Select a neuron from a deep layer of a pretrained Convolutional Neural Network.
2. Perform inference on multiple input images and record the activations of the selected neuron.
3. Identify the image that maximally activates the neuron.
4. Extract the receptive field (the image patch influencing the activation).
5. Repeat the process for multiple neurons to understand the broader feature representations.

Generate images Another approach is to generate synthetic images that maximize the activation of a specific neuron or class score:

1. Define an objective to maximize the class score $S_c(\mathbf{I})$ for a given class c, regularized for smoothness:
$$\hat{\mathbf{I}} = \underset{\mathbf{I}}{\operatorname{argmax}}\, S_c(\mathbf{I}) + \lambda \|\mathbf{I}\|_2^2$$
2. Optimize $\hat{\mathbf{I}}$ via gradient ascent over the input $\mathbf{I}$.
3. Repeat for neurons or classes to visualize different aspects of the network's learned representations.

Explanation Visualization techniques like saliency maps, Grad-Class Activation Mapping, and Class Activation Mapping highlight the regions of the image influencing the network's predictions. Saliency maps can help uncover systematic errors in the network, such as: misfocusing on irrelevant regions and failing to capture key features. Visualization fosters interpretability, especially for critical applications. Understanding model behavior builds confidence in its predictions and reveals potential biases.

3.5.3 Human pose estimation

Human pose estimation is a task that involves predicting the locations of body joints in an image, which can be framed as a Convolutional Neural Network regression problem. This task is fundamentally a localization problem, where the goal is to detect key points corresponding to the human body.

The network receives the entire input image, allowing it to capture the full context of each body joint. This holistic approach helps the network learn spatial relationships between joints more effectively. The design and training of the network are relatively straightforward, with transfer learning from pre-trained classification networks offering a significant advantage in addressing common training challenges.

Pose is represented as a vector containing the coordinates of k joints, where each joint is described by its two-dimensional coordinates. In some cases, the pose vector may be normalized with respect to the bounding box that encloses the human body to maintain scale invariance.

The network is trained to predict a vector of size $2k$, where each pair of consecutive values represents the coordinates of each joint. This approach allows for the representation of human pose in terms of joint positions.

One challenge in pose estimation is that not all joints may be visible in the input image, due to occlusion or out-of-frame parts of the body. In such cases, the network is designed to always produce a fixed or default value for the invisible joints, allowing it to handle occlusions gracefully.

To prevent overfitting and improve generalization, augmentation techniques such as translation and flipping are applied to the training data. These augmentations help the network become more robust to variations in the position and orientation of the human body within the image.

3.5.4 Object detection

Given a predefined set of categories Λ and an input image $\mathbf{I}$, the taks involves detecting and drawing bounding boxes around each object instance adn asssign a category label l to each

bounding box.

Assigning multiple labels to an input image $\mathbf{I} \in \mathbb{R}^{R \times C \times 5}$ involves associating each instance of an object with a bounding box and a corresponding category label. The goal is to identify the coordinates $\{(x, y, h, w)_i\}$ of the bounding box enclosing each object instance, along with its label $l_i \in \Lambda$, where Λ is a predefined set of categories. This can be expressed as:

$$\mathbf{I} \rightarrow \{(x, y, h, w, l)_1, \ldots, (x, y, h, w, l)_n\}$$

Here, n is the number of object instances in the image. A training set annotated with labels and bounding boxes for each object.

3.5.4.1 Sliding window

This approach is similar to the sliding window method used in semantic segmentation but adapted for object detection.:

1. *Pretrained model*: use a model trained to process a fixed input size.
2. *Sliding window*: slide a window of the model's input size across the image and classify each region within the window.
3. *Background class*: ensure the model includes a background class to differentiate between objects and non-object areas.

The problems with this technique is that it is inefficient and it is difficult to detect scale variability.

3.5.4.2 Region proposal

Region proposal algorithms (and networks) aim to identify bounding boxes that correspond to candidate objects in an image. These algorithms prioritize high recall (detecting most objects) over precision (reducing false positives). With Deep Learning, region proposal methods have become more sophisticated and integrated into modern object detection pipelines.

The approach involves:

1. Generate candidate bounding boxes that likely contain objects.
2. Use a Convolutional Neural Network to classify the image content within each proposed region.

Region proposals are refined using regression to improve bounding box accuracy. A pretrained Convolutional Neural Netwokr is fine-tuned for the specific object classes. A Fully Connected layer is added after feature extraction for classification.

We need to include a background class to filter out regions that do not correspond to objects.

Loss function Quantifying the network's performance requires assessing how well the detections match the ground truth annotations. A key metric is the Intersection over Union:

$$\text{IoU} = \frac{\text{area of overlap}}{\text{area of union}}$$

The loss function evaluates the predictions $\hat{y}$ against the annotations y. This loss includes the undetected objects, the missclassified objects and the correct dimension of the boxes.

Limitations The main limitations of region proposal methods include their reliance on separate, unoptimized algorithms for generating proposals, which are not jointly trained with the detection pipeline. Training is slow and resource-intensive, requiring significant storage for features and long processing times. Inference is also inefficient, as the Convolutional Neural Network must process each proposal independently, leading to redundant computations. Additionally, the training process involves ad-hoc objectives preventing true end-to-end optimization. These inefficiencies can hinder both accuracy and speed, making them less practical for real-time applications.

3.5.4.3 Fast Region Based Convolutional Neural Network

Fast Region Based Convolutional Neural Network improves upon traditional Region Based Convolutional Neural Network by introducing significant optimizations in both training and inference, enabling faster and more efficient object detection. The steps are as follows:

1. *Feature map extraction*: the entire image is passed through a Convolutional Neural Network to extract feature maps.

2. *Region proposal projection*: region proposals are identified directly from the image and mapped onto the extracted feature maps. This allows regions to be cropped from the feature maps instead of the original image.

3. *Fixed-size region of interestpooling*: since Fully Connected layers require fixed-size inputs, each Region of Interest is divided into smaller grid, and region of interestpooling applies max-pooling over the grid to produce a fixed-size activation. This output is vectorized and passed to subsequent layers.

4. *Multitask learning*: Fully Connected layers predict both object classes and bounding box locations. A multitask loss function, combining classification and regression losses, is optimized during training.

5. *End-to-end training*: the entire network is trained in an end-to-end manner, with gradients back-propagated through the Convolutional Neural Network, region of interestpooling, and Fully Connected layers. The convolutional computations are executed only once per image, regardless of the number of regions.

Fast Region Based Convolutional Neural Network significantly improves inference speed compared to Region Based Convolutional Neural Network, as convolutional operations are no longer repeated for overlapping regions. Most of the computational time is now spent on region proposal extraction.

3.5.4.4 Region Proposal Network

The Region Proposal Neural Network replaces traditional region of interest and extraction algorithms with a trainable network that operates on the same feature maps used for classification, improving efficiency and integration. Region Proposal Neural Network is implemented as a small Convolutional Neural Network that outputs potential bounding box proposals and objectness scores for each region.

At each spatial location in the feature map, Region Proposal Neural Network generates k anchor boxes with predefined scales and aspect ratios. A convolutional layer reduces the feature map to a lower-dimensional vector. Anchor classification predicts the probability that

each anchor contains an object (objectness score). Bounding box regression refines anchor box coordinates to better match ground truth bounding boxes. The final outputs are $2k$ objectness scores and $4k$ bounding box corrections.

The Region Proposal Neural Network proposals are passed to the detection head, which performs:

- *Region of interest*: crops and resizes features from the latent space for each proposal into a fixed-size representation.
- *Classification*: Fully Connected layers predict object classes, including a background class.
- *Bounding box refinement*: further adjusts bounding box locations.

The detection head produces L classification scores (one per class, including background) and $4L - 4$ bounding box corrections.

Training involves optimizing a multi-task loss that combines: objectness loss, regression loss, final classification loss, and final bounding box regression loss,

Advantages Fully trainable in an end-to-end manner, with Region Proposal Neural Network integrated directly into the pipeline. Faster inference compared to Fast Fast Region Based Convolutional Neural Network, as redundant computations are eliminated. High accuracy and efficiency by leveraging a shared feature map for both region proposal and detection.

3.5.4.5 You Only Look Once

The Neural Network works as follows:

1. *Image division into a grid*: the image is divided into a coarse grid.
2. *Anchor boxes for each cell*: each grid cell contains B anchor boxes (base bounding boxes). For each anchor, the network predicts: he adjustments to the base bounding box to better match the object: and the classification scores for the object within the bounding box, across C categories (including the background).
3. *Single forward pass*: the entire prediction is completed in a single forward pass through the image using a single Convolutional Neural Network.

Training You Only Look Onceinvolves challenges, particularly in defining the loss function to balance matched and unmatched predictions effectively.

You Only Look Onceshare similarities with the Region Proposal Neural Network used in Faster Region Based Convolutional Neural Network. Region Proposal Neural Network are typically more accurate. Single-shot detectors are faster but trade off some accuracy for speed.

3.5.5 Instance segmentation

Instance segmentation involves assigning multiple labels and associated information to an input image $\mathbf{I}$. Each object in the image is identified by a unique label l_i, drawn from a fixed set of categories Λ. For each object, the network predicts the coordinates of the bounding box that encloses it, given by (x_i, y_i, h_i, w_i), where x and y are the center position of the box, and h and w are its height and width.

Additionally, for each object, the network identifies the set of pixels S_i that lie within the bounding box, corresponding to the pixels that belong to that object instance. The final output is a set of predictions for all detected objects, expressed as:

$$\mathbf{I} \rightarrow \{(x_i, y_i, h_i, w_i, l_i, S_i)\}_{i=1}^{N}$$

Here, N is the number of objects detected.

This task combines the challenges of object detection, which involves handling multiple object instances in an image, and semantic segmentation, where each pixel is assigned a label. The complexity of instance segmentation arises from the need to not only detect multiple objects but also to precisely segment each instance of the object and assign the correct label to each pixel in the scene.

3.5.5.1 Mask Region Based Convolutional Neural Network

Similar to Faster Region Based Convolutional Neural Network, Mask Region Based Convolutional Neural Network performs object detection by classifying each region of interest and estimating the corresponding bounding boxes. However, it extends Faster Region Based Convolutional Neural Network by adding an additional branch for predicting object masks in parallel with the existing branch for bounding box recognition.

In Mask Region Based Convolutional Neural Network, semantic segmentation is performed within each region of interest, where the mask for each object instance is predicted separately for each class. The network estimates a binary mask for each region of interestcorresponding to each class label, capturing the precise object boundaries. This addition makes Mask Region Based Convolutional Neural Network capable of performing both object detection (via bounding boxes) and instance segmentation (via pixel-level object masks).

Mask Region Based Convolutional Neural Network builds upon the Faster Region Based Convolutional Neural Network backbone, which is responsible for extracting features, and incorporates the mask prediction branch. The loss function for training includes both the bounding box loss and the mask estimation loss, ensuring that the network learns to both detect objects and generate accurate masks.

3.5.5.2 Feature Pyramid Network

The goal of a Feature Pyramid Network is to leverage the inherent pyramidal structure of a Convolutional Neural Network feature hierarchy while creating a feature pyramid that retains strong semantic meaning at all scales. Feature Pyramid Network combines semantically strong features (low resolution) with semantically weak features (high resolution) through a top-down pathway and lateral connections.

Feature Pyramid Network addresses the need to recognize objects at vastly different scales. Traditional approaches used image pyramids, where the image is resized at multiple scales to capture features at various resolutions. However, using these image pyramids during inference and training led to significant increases in memory usage and inference time. Additionally, training in an end-to-end manner with image pyramids was infeasible. As a result, many methods only used image pyramids during inference, leading to inconsistencies between training and inference stages.

Components Feature Pyramid Network produces predictions independently at each level of the feature pyramid. Instead of using multi-scale image pyramids, Feature Pyramid Network

builds its feature pyramid directly from the input image and generates proportionally sized feature maps at different scales. The input is a single-scale image of arbitrary size. The output is a proportionally sized feature maps at multiple levels of the pyramid. The main components are:

1. *Bottom-up pathway*: this pathway creates feature maps at progressively smaller spatial dimensions as it moves upward through the network. The output of each convolutional layer in the bottom-up pathway is used as input to the top-down pathway.

2. *Top-down pathway*: the top-down pathway derives higher resolution features by upsampling the feature maps produced by the bottom-up pathway. These features are enhanced with information from the bottom-up pathway via lateral connections.

3. *Lateral connections*: these connections combine features from different layers in the bottom-up pathway with features from the top-down pathway. The lateral connections reduce the channel dimension and help refine the features as they are passed upward through the network.

Feature Pyramid Network is not an object detector by itself but serves as a feature extractor that can be combined with an object detector. Since the predictor head slides over all locations in all pyramid levels, it eliminates the need for multi-scale anchors at specific levels. This allows Feature Pyramid Network to efficiently recognize objects at different scales while maintaining consistency in training and inference.

3.5.6 Metric learning

In a face identification system, a simple approach using a Convolutional Neural Network for classification can be a good starting point. However, this setup has limitations when it comes to handling new individuals. In traditional Convolutional Neural Network classification, once a network is trained, it would need to be retrained entirely if a new person is added. This approach becomes inefficient as the number of individuals grows.

A more practical solution involves a metric learning approach, where we store a picture (or a template) for each individual, and then identify an individual by comparing the input image to the stored templates. The identification process becomes:

$$\hat{i} = \underset{j=1,\ldots,3}{\operatorname{argmin}} = \|\mathbf{T} - \mathbf{T}_j\|_2$$

Here $\mathbf{T}$ represents the input image and $\mathbf{T}_j$ are the templates stored in the database. This method makes enrollment straightforward, as adding a new person simply involves adding their template to the database. However, the challenge lies in how to perform the identification efficiently by learning a better distance measure for comparing images.

In a typical Convolutional Neural Networks setup, the feature extraction part is generally more versatile than the classification part. The feature extractor learns a latent representation of the input image which captures useful patterns that help in classification. This latent representation can also be used for image retrieval, where we compare images based on their latent representations. However, the original network was not specifically trained for comparison.

Metric learning In metric learning, the goal is to train the Convolutional Neural Network to measure the distance between images in a way that preserves semantic meaning. Specifically, we want the network to learn the distance such that:

$$\|f_{\mathbf{w}}(\mathbf{I}) - f_{\mathbf{w}}(\mathbf{T}_i)\|_2 < \|f_{\mathbf{w}}(\mathbf{I}) - f_{\mathbf{w}}(\mathbf{T}_j)\|_2$$

This approach uses Siamese Networks, where two identical networks (sharing the same weights $\mathbf{w}$) process two images simultaneously. The goal is to learn the weights such that the network can compare image pairs and measure their similarity.

During training, the Siamese network is fed with pairs of images, which may or may not refer to the same person. A contrastive loss function is typically used to train the network:

$$\mathbf{w} = \underset{\omega}{\operatorname{argmin}} \sum_{i,j} \mathcal{L}(\mathbf{I}_i, \mathbf{I}_j, y_{i,j})$$

Triplet loss Another commonly used loss function in metric learning is triplet loss. This loss function compares three images: a positive image $\mathbf{P}$, a negative image $\mathbf{N}$, and a query image $\mathbf{I}$. The triplet loss function is designed to minimize the distance between $\mathbf{I}$ and $\mathbf{P}$ (both images of the same person) while maximizing the distance between $\mathbf{I}$ and $\mathbf{N}$ (images of different people):

$$\mathcal{L}_q(\mathbf{I}, \mathbf{P}, \mathbf{N}) = \max\left(0, m + \|f_{\mathbf{w}}(\mathbf{I}) - f_{\mathbf{w}}(\mathbf{P})\|_2^2 - \|f_{\mathbf{w}}(\mathbf{I}) - f_{\mathbf{w}}(\mathbf{N})\|_2^2\right)$$

Here, m is a margin that enforces a minimum separation between positive and negative pairs. The triplet loss ensures that the network learns to distinguish between individuals by embedding similar individuals closer together in the feature space while keeping different individuals farther apart. The selection of triplets during training is crucial, and many methods have been developed to intelligently sample triplets that will improve the training efficiency.

3.6 Autoencoder

Autoencoders are Neural Networks used primarily for Unsupervised Learning tasks, particularly data reconstruction. Their structure consists of an input layer, an encoder $\mathcal{E}$, a decoder $\mathcal{D}$, and an output layer. The objective is to minimize the reconstruction error, typically by optimizing the Mean Squared Error between the input and the output.

Autoencoders are trained to reconstruct all the data in a given training set. The reconstruction loss over a batch S is defined as:

$$\mathcal{L} = \sum_{s \in S} \|s - \mathcal{D}(\mathcal{E}(s))\|_2$$

The training process uses standard backpropagation algorithms to optimize the parameters of both the encoder and decoder.

In essence, the Autoencoder learns an identity mapping between the input and output, with the encoder producing a latent representation of the input data. These features $z = \mathcal{E}(s)$ are referred to as the latent representation of the data.

Regularization Autoencoders do not always provide exact reconstruction because the dimensionality of the latent space is typically much smaller than that of the input. However, this dimensionality reduction is beneficial, as it forces the latent representation to be a compact

and meaningful description of the data. To guide the latent representation to exhibit certain desired properties, a regularization term can be added to the loss function:

$$\mathcal{L} = \sum_{s \in S} \|s - \mathcal{D}(\mathcal{E}(s))\|_2 + \lambda \mathcal{R}(s)$$

Here, $\mathcal{R}$ could enforce properties like sparsity, or smoothness and sharp edges in the case of image data. The use of deep Autoencoders enables the network to learn more powerful, nonlinear representations. Moreover, convolutional layers and transpose convolution can be employed to create deep convolutional Autoencoders.

The quality of the reconstruction improves as the latent space dimension increases. In the extreme case, if the latent space dimension is as large as the input, the network can perfectly reconstruct the input data, effectively learning an identity mapping. However, reducing the latent space dimension forces the Autoencoder to learn a more compact, meaningful representation of the input.

3.6.1 Initialization

Autoencoders can serve as an effective way to initialize a model, especially when dealing with limited labeled data and a large pool of unlabeled data. The process involves the following steps:

1. Train the Autoencoder in a fully unsupervised manner using the unlabeled data S.
2. Remove the decoder and retain only the encoder's weights.
3. Add a Fully Connected layer for classification, using the latent representation produced by the encoder.
4. Fine-tune the model using the labeled data L. If L is sufficiently large, the encoder weights can also be fine-tuned.

This approach helps prevent overfitting because the latent vector of the Autoencoder already provides a good representation of the inputs, which can be leveraged for downstream tasks.

3.6.2 Generative Autoencoder

Autoencoders can also be used as generative models, but with some modifications. Here's how it works:

1. Train the Autoencoder on a dataset S.
2. Discard the encoder.
3. Sample random vectors $z \sim \phi_z$ from a latent distribution to generate a new latent representation, which is then fed into the decoder.

However, this approach faces challenges, particularly in estimating the correct distribution of the latent space. As the dimensionality of the latent space increases, the likelihood of sampling from under-populated regions of the latent space rises. This makes it harder to generate meaningful samples from the decoder. In low-dimensional latent spaces, this can

work reasonably well, but as the dimension grows, it becomes difficult to estimate the latent distribution effectively.

Variational Autoencoders address this issue by forcing the latent space to follow a Gaussian distribution. This constraint not only improves reconstruction accuracy but also enables the Autoencoder to generate new data samples in a controlled manner, making Variational Autoencoders true generative models.

3.7 Generative models

The goal of generative models is to generate new data samples that resemble those in a given training set. These models can be applied to various tasks such as data augmentation, simulation, and planning. They are particularly useful for solving inverse problems like super-resolution, inpainting, and colorization. Generative models can also be used to produce realistic samples for artwork creation.

In addition to their direct applications, training generative models often helps infer latent representations that can serve as valuable general features. These models aim to capture the distribution of natural images. By learning such distributions, generative models can act as powerful regularizers in other tasks.

Beyond specific applications like image generation, the advent of models like Generative Adversarial Networks introduced a new training paradigm. Although this was a major breakthrough, the advent of more advanced foundational models has since revolutionized the use of generative models.

3.7.1 Generative Adversarial Networks

Generative Adversarial Networks offer a unique approach to generative modeling by sidestepping the need for an explicit density model ϕ_s describing the manifold of natural that describes the manifold of natural images. Instead, Generative Adversarial Networks aim to find a model that can generate samples resembling the training data.

Rather than sampling directly from a complex distribution ϕ_s Generative Adversarial Networks work by sampling a seed from a known distribution ϕ_z (often referred to as noise) and passing it through a learned transformation. This transformation generates realistic samples that appear to be drawn from the original data distribution ϕ_s. The Neural Network learns this transformation in an unsupervised manner.

Loss function The most significant challenge in Generative Adversarial Networks is designing a suitable loss function to evaluate the realism of the generated images. The solution is to train two Neural Networks that address different tasks, competing in a two-player (adversarial) game:

- *Generator* $\mathcal{G}$: this network produces realistic samples by taking random noise as input. The generator's goal is to fool the discriminator into thinking the generated samples are real.
- *Discriminator* $\mathcal{D}$: this network takes an image as input and assesses whether it is real (from the training set) or generated by the generator. The discriminator's task is to distinguish between real and fake images.

During training, both networks are optimized simultaneously, and at the end of the process, only the generator is kept. The discriminator becomes redundant because, after training, it can

no longer distinguish between real and fake images. Notably, the generator has never directly seen an image from the training set S.

Discriminator A good discriminator should exhibit the following properties:

1. $\mathcal{D}(s, \theta_d)$ should be maximized when $s \in S$ (true image from the training set).
2. $1 - \mathcal{D}(s, \theta_d)$ should be maximized when s is generated by $\mathcal{G}$.
3. $1 - \mathcal{D}(\mathcal{G}(z, \theta_g), \theta_d)$ should be maximized when $z \sim \phi_z$ (random noise).

To train the discriminator, the binary cross-entropy loss is maximized:

$$\max_{\theta_d} \left(\mathbb{E}_{s\sim\theta_s}[\log \mathcal{D}(s, \theta_d)] + \mathbb{E}_{z\sim\theta_s}[\log\left(1 - \mathcal{D}(\mathcal{G}(z, \theta_g), \theta_d)\right)]\right)$$

The first term ensures that the discriminator classifies real images correctly, while the second term ensures it classifies generated images as fake.

Generator The generator's goal is to minimize the discriminator's ability to distinguish real from fake. Hence, the objective for the generator is:

$$\min_{\theta_g} \max_{\theta_d} \left(\mathbb{E}_{s\sim\theta_s}[\log \mathcal{D}(s, \theta_d)] + \mathbb{E}_{z\sim\theta_s}[\log\left(1 - \mathcal{D}(\mathcal{G}(z, \theta_g), \theta_d)\right)]\right)$$

This approach is solved using an iterative numerical method, typically stochastic gradient descent.

Conclusions The training of Generative Adversarial Networks is notoriously unstable and requires careful synchronization between the generator and discriminator steps. Later advancements, such as the Wasserstein Generative Adversarial Network, have addressed some of these instability issues. Generative Adversarial Networks are trained using standard backpropagation and dropout, with no direct evaluation of the generator during training.

One challenge in Generative Adversarial Network training is the difficulty in quantitatively assessing the generator's performance. Since the generator's output is implicitly defined, evaluating its effectiveness remains a complex task.

CHAPTER 4

Advanced topics

4.1 Recurrent Neural Network

Many real-world problems involve dynamic data, where patterns unfold over time. To address this, various approaches can be employed depending on the need for temporal modeling:

- *Memoryless models*: these models operate without retaining a true memory of past states. Instead, they rely on a fixed number of past observations, often referred to as delay taps, to predict future values.
- *Feed Forward Neural Networks*: can extend the capabilities of traditional memoryless models by incorporating non-linear transformations.

While these methods can handle certain types of sequential data, they are inherently limited because they lack a true memory mechanism to capture long-term dependencies.

4.1.1 Models with memory

Dynamic data often requires models capable of retaining information over time to capture long-term dependencies. These models rely on a hidden state that evolves dynamically, often influenced by noise and external inputs, which drive state transitions. This hidden state, while not directly observable, must be inferred to compute outputs.

4.1.1.1 Recurren Neural Networks

Recurrent Neural Network offer a flexible and powerful approach by leveraging distributed hidden states that evolve through non-linear dynamics. Recurrent Neural Networks are deterministic systems designed to efficiently store and process sequential information. The hidden state in a Recurrent Neural Network is updated at each time step based on the current input and the prior state, allowing the network to learn complex temporal patterns through its non-linear transformation capabilities.

Backpropagation The key strength of Recurrent Neural Networks lies in their recurrent connections, which allow information from prior time steps to persist and influence future

computations. This architecture enables Recurrent Neural Networks to model both short-term and long-term dependencies, although training them presents unique challenges. One such challenge is the vanishing gradient problem, which arises during backpropagation through time.

In backpropagation through time, the network is unrolled for a fixed number of time steps, and gradients are computed for each step before being averaged to update the weights. However, as the network propagates gradients backward through many time steps, they often shrink exponentially due to the nature of activation functions like Sigmoid and hyperbolic tangent. This makes it difficult for the network to learn dependencies that span a long temporal range.

Vanishing gradient Addressing the vanishing gradient problem is crucial for enabling Recurrent Neural Networks to capture long-term dependencies effectively. One approach involves designing architectures like Long Short-Term Memory and Gated Recurrent Units. These models incorporate specialized mechanisms, such as gates and memory cells, that allow them to retain information over extended time spans while avoiding the issues of vanishing or exploding gradients. Another strategy is gradient clipping, which constrains gradients within a specific range to maintain stability during training.

4.1.1.2 Long Short-Term Memory

The vanishing gradient problem, which hampers the training of traditional Recurrent Neural Networks over long time sequences, was effectively addressed by Hochreiter and Schmidhuber in 1997. They introduced the Long Short-Term Memory architecture, which uses memory cells designed with logistic and linear units interacting multiplicatively.

A Long Short-Term Memory cell controls the flow of information through three key mechanisms:

1. Information enters the cell when its input gate is activated.
2. The information is retained in the cell as long as its forget gate remains active.
3. Information is extracted from the cell by activating its output gate.

These gates allow the Long Short-Term Memory to selectively store and retrieve information, providing a mechanism for maintaining long-term dependencies in a stable manner. Importantly, the loop within the cell operates with fixed weights, making it possible to backpropagate gradients effectively through time.

The Long Short-Term Memory architecture inspired simpler variants, such as the Gated Recurrent Unit. Gated Recurrent Units streamline the memory cell design by merging the forget and input gates into a single update gate. Additionally, they combine the cell state and hidden state into one, reducing complexity while retaining much of the Long Short-Term Memory's effectiveness.

In recurrent neural network architectures, computation graphs can be built with continuous transformations using recurrent layers. When working with full input sequences, bidirectional Recurrent Neural Networks take advantage of information in both temporal directions. This involves:

- Having one Recurrent Neural Network process the sequence from left to right, and another process it from right to left.

- Concatenating the hidden layer outputs from both directions to create a richer feature representation.

Initialization plays a crucial role in training Recurrent Neural Networks effectively. The initial state of the Recurrent Neural Networks must be defined before processing sequences. There are several strategies for this:

- A common approach is to initialize the states to a fixed value, such as zero.
- A more effective method is to treat the initial state as a set of learnable parameters. In this case, the initial state values are randomly initialized and refined through training.

The training process involves backpropagating the prediction error through time, all the way to the initial state values, and computing the gradient of the error with respect to these parameters. Gradient descent is then used to update the initial state values, enabling the network to learn optimal starting conditions for its sequences.

4.2 Sequence to sequence learning

Sequence-to-sequence learning addresses tasks where inputs and outputs are sequences of varying lengths. This framework is highly versatile and has proven effective in solving a range of problems, including:

- *Image captioning*: the input is a single fixed-size image, and the output is a sequence of words describing the image. The challenge lies in producing a sequence of varying length that captures the content of the image.
- *Sentiment analysis*: the input is a sequence of characters or words and the goal is to classify the sequence into predefined categories. The input sequences can have varying lengths, while the output remains fixed in type and size.
- *Language translation*: given a text in one language, the objective is to generate its equivalent in another language. This task is particularly complex because each language has its own semantics, and the lengths of corresponding sentences often differ.

In these tasks, sequence modeling involves capturing the probability of a target sequence. A language model predicts the probability of a sequence, such as a sentence, by analyzing patterns and dependencies between elements. When conditioned on an input sequence, this becomes a conditional language model.

The goal in sequence-to-sequence learning is to find the target sequence $\mathbf{y}$ hat maximizes the conditional probability $\Pr(\mathbf{y} \mid \mathbf{x})$, where $\mathbf{x}$ is the output sequence. Formally, this can be expressed as:

$$\mathbf{y}^* = \underset{\mathbf{y}}{\operatorname{argmax}} \Pr(\mathbf{y} \mid \mathbf{x})$$

For tasks like human translation, this conditional probability is guided by intuition and expertise, as translators rely on their knowledge of grammar, semantics, and context to map the input sequence to the output.

In Machine Learning, however, this mapping is learned from data. The model approximates $\Pr(\mathbf{y} \mid \mathbf{x}, \boldsymbol{\theta})$, where $\boldsymbol{\theta}$ represents the parameters of the model. The prediction task is then reformulated as:

$$\mathbf{y}' = \underset{\mathbf{y}}{\operatorname{argmax}} \Pr(\mathbf{y} \mid \mathbf{x}, \boldsymbol{\theta})$$

Sequence-to-sequence models are commonly implemented as encoder-decoder architectures, where the encoder processes the input sequence and generates a compact representation, often in the form of a vector. This representation encapsulates the source sequence's essential information and serves as input to the decoder. The decoder then generates the target sequence by iteratively predicting the most likely subsequent token based on the encoded representation and previously generated tokens.

A straightforward implementation of this approach uses Recurrent Neural Networks. The Recurrent Neural Network encoder updates its hidden state as it processes each word in the input sequence, producing a final state that summarizes the input. The Recurrent Neural Network decoder then takes this final state and reconstructs the output sequence, predicting one word at a time. Importantly, both encoding and decoding processes require explicit start signals to initialize their respective operations. Once the model is trained, it predicts the output sequence $\mathbf{y}$ given an input sequence $\mathbf{x}$ by selecting the sequence $\mathbf{y}'$ that maximizes the conditional probability:

$$\mathbf{y}' = \underset{\mathbf{y}}{\operatorname{argmax}} \prod_{t=1}^{n} \Pr(\mathbf{y}_t \mid \mathbf{y}_{<t}, \mathbf{x}, \boldsymbol{\theta})$$

Here, $\Pr(\mathbf{y}_t \mid \mathbf{y}_{<t}, \mathbf{x}, \boldsymbol{\theta})$ represents the probability of predicting the token $\mathbf{y}_t$ at time step t, given all previously generated tokens $\mathbf{y}_{<t}$.

To compute this argmax efficiently, we can use different decoding strategies:

- *Greedy decoding*: at each step, the decoder selects the most probable token. While simple and fast, this approach is suboptimal because it cannot revise earlier decisions, often leading to subpar sequences.
- *Beam search*: this method maintains a set of the most probable sequence hypotheses at each step. By exploring multiple paths simultaneously, it balances computational cost with the likelihood of finding a better overall sequence.

These decoding strategies provide different trade-offs between computational efficiency and the quality of the generated output, with beam search being particularly useful for tasks requiring high-quality sequences.

4.2.1 Training

Given a training sample $\langle \mathbf{x}, \mathbf{y} \rangle$, where $\mathbf{x}$ is the input sequence and $\mathbf{y}$ is the corresponding target sequence, the model computes a probability distribution over possible words at each time step t. Using a one-hot vector representation for y_t, the loss at time step t can be calculated using cross-entropy:

$$\mathcal{L}(\underset{t}{\Pr}, y_t) = -y_t^T \log(\underset{t}{\Pr})$$

For the entire sequence, the total cross-entropy loss becomes:

$$\mathcal{L}(\underset{t}{\Pr}, y_t) = -\sum_{t=0}^{n} y_t^T \log(\underset{t}{\Pr})$$

During training, sequence-to-sequence models follow the classical encoder-decoder architecture. The key steps differ between training and inference:

- *Training time*: the decoder does not use its own output as input for the next time step. Instead, the ground truth from the target sequence is provided as input to each time step of the decoder. This technique is called teacher forcing and helps stabilize training by reducing error propagation.
- *Inference time*: during inference, the decoder feeds its own output from the previous time step as input to the next time step. This allows the model to generate sequences autonomously.

4.2.2 Dataset preparation

Sequence-to-sequence models rely on specialized tokens to standardize and preprocess data for effective training.

Token	Description
⟨PAD⟩	Used during training to pad shorter input sequences in a batch to match the length of the longest sequence, ensuring uniform input width for batch processing.
⟨EOS⟩	Indicates the end of a sequence, helping the decoder identify where a sentence ends. Also used during batching to standardize sequence lengths.
⟨UNK⟩	Replaces rare or unknown words (out-of-vocabulary tokens) in the input, improving resource efficiency and reducing the vocabulary size.
⟨SOS⟩ and ⟨GO⟩	Marks the start of decoding. Preceded to the target input sequence to signal the decoder to begin generating output.

In natural language processing, words are often treated as discrete atomic symbols. While this avoids sparsity and high memory dimensionality, effective input encoding is critical for improving the performance of real-world applications. Proper preprocessing and encoding ensure that the model captures the essential semantic and syntactic features of the input data.

4.2.3 Words

N-gram language model An N-gram language model estimates the probability of a word sequence $S = \mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_k$ within a specific domain. This probability is expressed as:

$$\Pr(s_k) = \prod_{i=1}^{k} \Pr(\mathbf{w}_i \mid \mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_{i-1})$$

However, to manage computational complexity, N-gram models introduce a simplifying assumption: the probability of a word depends only on the preceding $n-1$ words. Thus, the probability becomes:

$$\hat{\Pr}(s_k) = \prod_{i=1}^{k} \Pr(\mathbf{w}_i \mid \mathbf{w}_{i-n+1}, \ldots, \mathbf{w}_{i-1})$$

To compute these probabilities, frequency-based estimates are typically used, incorporating smoothing techniques to handle unseen word sequences.

The main issue of N-gram includes data sparsity and vocabulary size. These challenges are addressed by:

1. *Limiting context size*: use shorter context windows to reduce complexity. However, this approach sacrifices the ability to model long-range dependencies.

2. *Trade-off*: shorter contexts simplify computation but fail to capture broader linguistic patterns, leaving valuable information untapped.

Embeddings Word embeddings are techniques that represent words or phrases as continuous vectors in a lower-dimensional numerical space. This approach captures semantic and contextual similarities between words, where words with similar meanings are mapped closer together in the embedding space.

Given a vocabulary $\mathbf{V}$, each unique word $\mathbf{w}$ is mapped to an m-dimensional vector, significantly reducing the high-dimensional space of discrete word representations. This approach helps address the curse of dimensionality by:

- *Compression*: dimensionality reduction makes models computationally efficient.

- *Smoothing*: transitioning from discrete word representations to continuous spaces improves generalization.

- *Densification*: sparse, high-dimensional vectors are replaced with dense, compact representations.

The primary objective of word embeddings is to ensure that similar words are positioned closer in the feature space, enabling the model to learn and leverage semantic and contextual relationships.

4.2.4 Neural Network Language Model

In a Neural Network Language Model, the input consists of past words represented as one-hot encoded vectors. These high-dimensional sparse representations are projected into a continuous vector space using a shared matrix $\mathbf{C}$, referred to as the projection layer. Each word is thus transformed into a dense, low-dimensional embedding.

After projection, the embeddings of the past words are concatenated and passed through non-linear transformations to learn contextual relationships. Finally, the model predicts the next word by outputting a probability distribution over the vocabulary using a softmax function.

Training Neural Network Language Models with stochastic gradient descent is computationally expensive, especially for large vocabularies. Due to their complexity, Neural Network Language Models are not well-suited for large datasets. Neural Network Language Models often exhibit poor performance on rare or unseen words due to insufficient representation in the training data.

Bengio et al. (2003) introduced Neural Network Language Models with the aim of improving language model accuracy. Although their work demonstrated significant improvements in LM performance, the exploration of word embeddings was regarded as an area for future research.

4.2.5 Word2Vec

Mikolov et al. (2013) focused on improving word vector representations by simplifying the model while still achieving better performance. Their approach aimed to enable training on much larger datasets with a simpler architecture. Unlike previous models, Word2Vec eliminates the need for a hidden layer, with the projection layer being shared across the network (rather than just the weight matrix). This shared projection layer helps create dense word embeddings by leveraging both past and future context in a word's definition.

The key idea is that the meaning of a word can be inferred from the surrounding context. Word2Vec offers two main architectures for learning word embeddings:

- *Skip-gram model*: the model projects a target word into multiple surrounding words as output. This model is effective for learning representations of rare words by predicting the context words based on the central target word.

- *Continuous Bag Of Words*: takes a context (usually a fixed-size window around the target word) and averages the representations of the surrounding words to predict the target word. This architecture aims to learn a word's meaning by using the surrounding context to predict it. In the Continuous Bag Of Words architecture, the context consists of $\frac{n}{2}$ previous words and $\frac{n}{2}$ following words, where n is the size of the context window. These context words are projected into the projection layer using a shared matrix $\mathbf{C}$, and their embeddings are averaged to create a compact, unified representation. This averaged vector is then used to predict the central word in the context. The goal is to maximize the probability of the target word given the surrounding context.

Complexity Word2Vec significantly outperforms traditional Neural Network Language Models in terms of both speed and accuracy. Its efficiency allows it to be trained on large datasets, and the quality of the learned word vectors is typically four times better than the embeddings produced by Neural Network Language Models. This makes Word2Vec a highly scalable and effective approach for learning word representations.

4.3 Attention

In basic sequence-to-sequence models, the fixed source representation can become a bottleneck, as it may not capture all relevant information effectively. This issue arises for both the encoder and decoder:

- *Encoder*: it may be challenging to compress the entire sentence into a single representation.

- *Decoder*: different parts of the input may be more relevant at various stages of the generation process.

Attention mechanisms address this problem by allowing the model to focus on different parts of the input sequence as needed. The decoder uses attention to decide which parts of the source sequence are more relevant for generating each output token, alleviating the need for a single compressed representation. Instead, the decoder checks the relevant input portions at each step, avoiding the bottleneck caused by fixed source encoding. Attention scores are computed at each decoding step and then passed through a softmax function to assign a probability distribution over the input tokens.

Several methods have been proposed to compute attention scores, typically by combining input and output tokens before applying the softmax function:

- *Dot product attention*: the simplest attention mechanism involves computing the product of the input vector and the decoded vector:

  $$s(\mathbf{h}_t, \mathbf{s}_k) = \mathbf{h}_t \mathbf{s}_k$$

  This is a non-trainable function and provides a basic measure of similarity between tokens in the input and output sequences.

- *Bilinear attention*: a weight matrix $\mathbf{W}$ is introduced to allow for more flexible interactions between the input and decoded vectors:

  $$s(\mathbf{h}_t, \mathbf{s}_k) = \mathbf{h}_t \mathbf{W} \mathbf{s}_k$$

  The matrix $\mathbf{W}$ ontains trainable parameters, and its size corresponds to the number of neurons in the model. The combination of the input and attention tokens is then processed through a nonlinear function:

  $$\tanh \mathbf{W}_C [\mathbf{h}_t, \mathbf{s}_k]$$

  This attention mechanism is commonly known as Luong attention and is more expressive than the simple dot product.

- *Multi-Layer Perceptron attention*: Bahdanau attention mechanism uses a Multi-Layer Perceptron to calculate attention scores. The input and decoded vectors are combined and passed through a set of learned weights:

  $$s(\mathbf{h}_t, \mathbf{s}_k) = \mathbf{w}_2^T \tanh \mathbf{W}_1 [\mathbf{h}_t, \mathbf{s}_k]$$

  In this model, a bidirectional Long Short-Term Memory is used instead of a unidirectional one. The combination of these tokens is referred to as the attention vector, which highlights the most relevant parts of the input for generating the output.

4.3.1 Generative chatbots

Generative chatbots can be directly modeled using sequence-to-sequence models, where the conversation between two agents is treated as a sequence of input-output pairs.

In this setup, a Recurrent Neural Network is trained to map the input sequence to the output sequence. This approach is borrowed from machine translation and is simple and general, where the chatbot learns to generate responses by conditioning on the input context. Attention mechanisms can also be applied to improve performance, allowing the model to focus on different parts of the input sequence when generating responses.

Chatbots can be categorized based on the context handling:

- *Single-turn*: the input vector is built solely from the incoming question. These chatbots may struggle to maintain conversation context and can sometimes generate irrelevant responses due to the lack of historical context.

- *Multi-turn*: these chatbots consider the context of multiple turns of the conversation when building the input vector, which helps to maintain the flow of the conversation and improve the relevance of the response.

And also based on core algorithm:

- *Generative chatbots*: these chatbots encode the input (the question) into a context vector, then generate the response word-by-word using a probability distribution over the vocabulary of the answer. A popular architecture for this approach is the encoder-decoder model.
- *Retrieval-based chatbots*: these chatbots rely on a knowledge base of question-answer pairs. When a new question is received, the model encodes it into a context vector and then uses a similarity measure to retrieve the most relevant question-answer pairs from the knowledge base.

Challenges Concatenating multiple conversation turns into a single long input sequence is a potential approach, but it often results in poor performance due to several limitations. Long Short-Term Memory cells often fail to capture long-term dependencies in sequences longer than 100 tokens. There is no explicit representation of individual turns, which prevents the chatbot from leveraging context-specific information effectively.

Hierarchical attention In 2017, Xing et al. extended attention mechanisms from single-turn response generation to multi-turn responses by introducing a hierarchical attention mechanism. These networks process sequences at multiple levels. The hierarchical model generates a hidden representation of the sequence by first encoding the words, then the sentences, allowing the model to capture long-term dependencies and better understand context across multiple turns.

4.4 Transformer

The Natural Language Processingcommunity once believed that Long Short-Term Memorys with attention could yield state-of-the-art performance on a variety of tasks.

Long Short-Term Memorys and other Recurrent Neural Networks perform inference (and training) sequentially. This sequential nature prevents parallelization at the sample level, limiting efficiency. While parallelization can occur at the batch level, memory constraints restrict batching across too many examples. These issues become particularly critical for long sequence lengths.

To overcome these bottlenecks, Google introduced the Transformer model, which replaces Recurrent Neural Networks with an attention mechanism. The key advantage of the Transformer is that it can perform operations in parallel, speeding up training.

In a Transformer, the encoder looks at all source tokens simultaneously. It processes each token iteratively in parallel, updating its representations over multiple layers. The decoder, similarly, attends to both previous target tokens and source representations, updating its own state over multiple layers.

The Transformer architecture significantly reduces training time and allows for efficient parallelization across sequences.

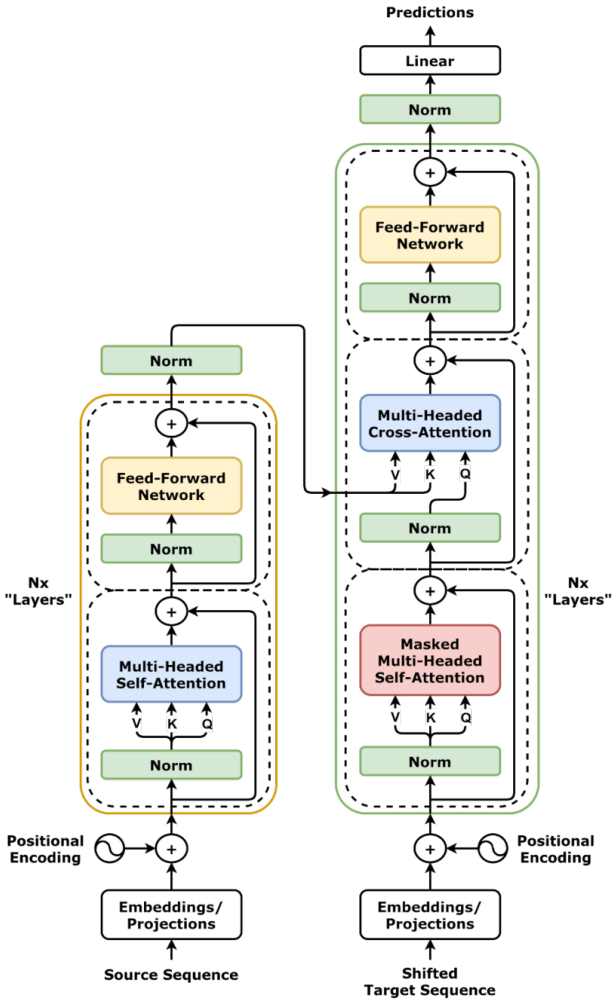

Figure 4.1: Transformer

4.4.1 Encoder attention

The encoder uses self-attention, where each token attends to all other tokens within the same input sequence. This is implemented using the following components:

- *Query*: a vector from which the attention mechanism queries information.
- *Key*: a vector that the query looks at to compute weights for attention.
- *Value*: the weighted sum of values is the attention output.

Self-attention allows for parallel execution and, consequently, parallel training, which helps improve training speed.

Multi-head attention In some cases, attention needs to capture different aspects of a word's role in a sentence. Multi-head attention enables the model to focus on multiple aspects simultaneously, both during encoding and decoding. This is achieved by concatenating several attention heads. The use of multiple attention heads doesn't increase the model size, as the heads share the same model capacity.

4.4.2 Decoder attention

In the decoder, attention mechanisms differ during training and inference:

- *At inference time*: the decoder generates one token at a time since the sequence length is not known in advance, preventing any look-ahead.
- *At training time*: the entire target sequence is known, allowing parallel processing of tokens but requiring the use of look-ahead masks to prevent the decoder from seeing future tokens during training.

While the training time complexity of Recurrent Neural Networks models is $\mathcal{O}(\text{len(source)} + \text{len(target)})$, the Transformer achieves training in $\mathcal{O}(1)$ with respect to the length of fixed sequences, making it highly efficient.

4.4.3 Other elements

The other elements of the Transformer are:

- *Decoder-encoder attention*: the decoder attends to the encoder states by using queries from the decoder and keys-values from the encoder states.
- *Feed Forward Neural Networks*: after receiving attention-based information from other tokens, the model processes the information in feed-forward layers. These are typically composed of two linear layers with ReLU activations. Residual connections are used to prevent issues like vanishing gradients and improve the flow of information.
- *Layer normalization*: normalizes the representation of each vector independently within a batch, ensuring stable training. A scale and bias are applied globally across the layer and are trainable parameters.

4.4.4 Positional encoding

Since the Transformer model processes tokens in parallel, the order of tokens doesn't naturally affect the model's behavior, unlike Recurrent Neural Networks that process tokens sequentially. The self-attention mechanism is permutation-invariant, meaning changing the order of the tokens doesn't impact attention values.

To address this, positional encodings are added to the input embeddings to give the model information about the position of each token within the sequence. The input representation is the sum of:

- *Token embeddings*: standard word embeddings.
- *Positional embeddings*: specific to each token's position in the sequence.

Positional embeddings can be either learned or fixed. The original Transformer used fixed positional encodings, but modern state-of-the-art Transformers often learn these encodings to adapt more flexibly to varying sequence lengths