Handbook of Generative AI

Origins, Mathematics, and Applications

First Edition

Handbook of Generative AI

Origins, Mathematics, and Applications

First Edition

Navid Nobani, PhD
University of Milano-Bicocca

Table of Contents

Chapter 1

Introduction to Generative Models

Generative Artificial Intelligence (AI) stands as a pinnacle of innovation and creativity in the realm of machine learning and computational technologies. It encompasses the development of algorithms and models capable of generating data that is remarkably similar to real-world phenomena, spanning from text and images to sounds and videos. This handbook is dedicated to unfolding the intricacies, mathematical details, and profound impact of Generative AI, aiming to provide an extensive and insightful overview of this transformative technology.

Origins and Historical Evolution The inception of Generative AI can be traced back to the mid-20th century, marked by pioneering efforts in computer science and mathematics to endow machines with capabilities akin to human intelligence. Alan Turing's groundbreaking 1950 paper, "Computing Machinery and Intelligence," introduced the Turing Test, setting the stage for the development of machines that could mimic human responses, a concept integral to Generative AI.

The ensuing decades saw advancements in neural networks and machine learning, setting a solid foundation for the field. Yet, it was the advent of deep learning in the 2010s that truly catapulted Generative AI into prominence. Techniques such as Generative Adversarial Networks (GANs) (Chapter 8) and Variational Autoencoders (VAEs) (Chapter 7) showcased unprecedented proficiency in generating realistic and high-fidelity content, captivating both the scientific community and the industry.

The Evolution of Techniques and Applications Generative AI has since undergone a rapid transformation, consistently breaking boundaries with innovative models, enhanced techniques, and broadened applications. From gener-

ating photorealistic imagery and crafting deepfake videos to producing coherent and contextually relevant text with models like GPT (Generative Pre-trained Transformer) (Section **??**), the field has redefined the boundaries of what is deemed achievable.

Today, Generative AI's applications permeate various sectors, including art, healthcare, entertainment, education, and more. In the medical realm, it plays a crucial role in drug discovery and personalized patient care; in content creation, it augments creativity and streamlines production processes; and in countless other areas, it offers novel solutions to complex challenges.

The Imperative to Study Generative AI Delving into Generative AI is not merely an academic endeavor; it is a crucial undertaking to understand, influence, and ethically steer the development of technologies that are swiftly becoming integral components of our digital landscape. The extraordinary capabilities of Generative AI are accompanied by significant responsibilities, highlighting the necessity for a comprehensive and nuanced understanding of its principles, challenges, and societal implications.

Standing at the frontier of AI innovation, equipped with more powerful tools and techniques than ever before, the imperative for a meticulous and critical examination of Generative AI is paramount. This handbook serves as a gateway to this journey, offering clarity, in-depth analysis, and a holistic perspective on the field, encouraging informed discourse and innovative contributions to Generative AI.

Engaging with this guide, readers are invited to explore a transformative domain of technology, one that is reshaping knowledge, creativity, and the potential of human and machine collaboration. Welcome to the captivating world of Generative AI.

Definition Generative Artificial Intelligence (AI) stands as a paramount field within the realm of computer science and machine learning, continuously pushing the boundaries of what machines can achieve in terms of content creation and simulation. Generative models aim to learn the probability distribution of the data, seeking to capture the underlying patterns and structures. Some of the most common generative models include:

- **Hidden Markov Models (HMM):** Used for modeling sequential data, HMMs find applications in speech recognition and natural language processing (See Section 2.2).
- **Gaussian Mixture Models (GMM):** These models represent data as a mixture of several Gaussian distributions and are used for clustering and density estimation (See Section 2.4).
- **Generative Adversarial Networks (GANs):** Comprising a generator and a discriminator, GANs generate realistic data through a competitive training process (See Chapter 8).

- **Variational Autoencoders (VAEs):** A type of generative neural network that encodes data into a lower-dimensional space and decodes it to generate new data points(See Chapter 7).

- **Large Language Models (LLMs):** Large Language Models (LLMs) are powerful AI systems trained on extensive text data to generate human-like responses, understand context, and perform various natural language processing tasks (See Section 6.3).

Discriminative Models Unlike Generative models, Discriminative models focus on learning the conditional probability distribution , aiming to model the decision boundary between different classes. Some of the common discriminative models are listed below (You can find a description for all these models in the Appendix)

- **Logistic Regression:** A simple model used for binary classification tasks.

- **Decision Trees and Random Forests:** Used for classification and regression.

- **Support Vector Machines (SVM):** SVMs find the hyperplane that best separates data into classes.

- **Conditional Random Fields (CRF):** Employed in structured prediction tasks.

- **Multilayer Perceptrons (MLP):** Neural networks used for classification and natural language processing.

Key Differences between Discriminative and Generative models [1]

1. **Purpose:** Generative models aim to learn the joint probability distribution[2] of input features and output labels, enabling them to generate new, unseen data instances. Discriminative models, on the other hand, learn the conditional probability[3] of the output label given the input features, focusing solely on the boundary that separates different classes in the feature space.

[1] A good paper on this topic is Ng, Andrew, and Michael Jordan. "On discriminative vs. generative classifiers: A comparison of logistic regression and naive bayes." Advances in neural information processing systems 14 (2001).

[2] The joint probability distribution P(X,Y) tells us the likelihood of observing a particular combination of input features and output labels

[3] Describing the likelihood of an event A happening given that another event B has already occurred

2. **Use Cases:** Generative models find applications in a wide variety of tasks such as image synthesis, text generation, and speech production. They are particularly useful in scenarios where the goal is to create new samples that resemble the training data. Discriminative models are primarily employed for classification and regression tasks, including image recognition, sentiment analysis, and spam filtering, where the objective is to make accurate predictions based on input data.

3. **Complexity:** Generative models tend to have higher complexity and require more computational resources because they need to capture the underlying data distribution comprehensively. Discriminative models can be less complex as they only need to find the decision boundary between different classes.

4. **Training Data:** Generative models can perform well even with a small amount of training data because they model the distribution of each class. Discriminative models generally require a larger amount of labeled data to achieve high performance.

5. **Performance:** While discriminative models tend to outperform generative models in terms of predictive accuracy when ample labeled data is available, generative models can be more flexible and perform better in the presence of less data or when dealing with novel scenarios.

6. **Interpretability:** Generative models can offer a higher degree of interpretability since they capture the underlying data distribution, providing insights into the structure of the data[4]. Discriminative models may offer less interpretability as they focus on the decision boundary and not the data distribution itself.

By understanding these distinctions and characteristics, practitioners can make more informed choices when selecting between generative and discriminative models for specific tasks and applications.

1.0.1 How Generative Models Work

Generative models are a class of statistical models that are capable of generating new data points. Different from discriminative models, which model the conditional probability $p(y|x)$, generative models seek to learn the joint probability distribution $p(x, y)$. Below we outline key aspects of how generative models work.

[4]Notice that we are talking about the "hypothetical" explainability. Practically speaking, generative models are hard to explain due to their architectural complexities

Learning the Distribution

Generative models aim to understand and capture the entire distribution of data for each class. To illustrate, in a dataset containing images of cats and dogs, the generative model endeavors to learn the distribution that encapsulates all variations of "catness" or "dogness" found within the images.

Sampling and Synthesis

Once the model has effectively learned the data distribution, it possesses the capability to generate new data samples that are statistically representative of this distribution. For example, a generative model that has been trained on a set of handwritten digits is now equipped to synthesize new images of digits that appear as though they could have been part of the original training set.

Density Estimation

Central to the operation of generative models is the task of density estimation, which involves modeling the underlying probability distribution of the dataset. This process is essential for understanding the structure and diversity of the data, and it enables the generative model to produce new instances that faithfully reflect the characteristics of the original dataset.

Handling Missing Data

One of the strengths of generative models is their inherent ability to handle incomplete datasets. By utilizing the probabilistic framework they have constructed, these models can accurately predict missing values, inferring the most likely substitutes for the absent data based on the comprehensive understanding they've developed of the data's distribution.

Applications in Semi-supervised and Unsupervised Learning

Generative models have shown exceptional utility in semi-supervised learning in which a small amount of labelled data is combined with a large amount of unlabelled data during training. This duality allows for more robust models that can make better-informed predictions. They also play a pivotal role in unsupervised learning tasks, such as clustering, where Gaussian Mixture Models (GMMs) and other techniques can identify and group data points into coherent clusters without the need for pre-labelled training data.

Improving Robustness and Flexibility

The generative approach improves the robustness and flexibility of statistical models. By learning the full distribution of data, generative models can better cope with variations and novelties in the data, which might confound more rigid, discriminative approaches. This robustness makes generative models particularly well-suited for complex tasks such as feature generation, anomaly detection, and even in the creative generation of art and music where diversity and novelty are valued.

Chapter 2

Traditional Generative Models

Traditional generative models play a foundational role in machine learning and pattern recognition. These models aim to capture and emulate the data-generating process, enabling them to generate new data samples that share similar statistical properties with the original data.

Overview of Generative Models: Generative models such as the Naïve Bayes Classifier, Hidden Markov Models (HMMs), and Gaussian Mixture Models (GMMs) are foundational in the realm of probabilistic modeling and have applications ranging from text classification to time-series analysis. The Naïve Bayes Classifier, with its simplicity and efficiency, is especially renowned in spam filtering and text categorization, while HMMs are instrumental in sequential data analysis like speech recognition, modeling the data as a series of observable outputs from hidden states. GMMs provide a flexible approach to clustering by assuming that the data originates from multiple Gaussian distributions.

Specialized Generative Models: Alongside these classical models, we have Restricted Boltzmann Machines (RBMs) and Latent Dirichlet Allocation (LDA) which contribute to understanding complex data distributions. RBMs are adept at learning underlying probability distributions for a set of inputs, serving purposes in dimensionality reduction and recommendation systems. LDA, on the other hand, excels in the field of natural language processing by postulating that documents are composed of a mixture of a limited number of topics. These traditional models form the groundwork for advanced generative techniques like Generative Adversarial Networks (GANs) and Variational Autoencoders (VAEs), which are discussed in subsequent chapters, with RBMs and LDA briefly introduced in the Appendix.

2.1 Naïve Bayes Classificer

The Naïve Bayes classifier is a probabilistic machine learning model based on Bayes' Theorem, used for classification tasks. Despite its simplicity, it has shown to be surprisingly effective in various domains, especially in text classification and natural language processing. Let's first take a look at Bayes' Theorem.

2.1.1 Bayes' Theorem

Bayes' Theorem is a fundamental concept in probability theory and statistics, elucidating the relationship between conditional and marginal probabilities of two random events. This theorem is built upon two primary rules of probability: the sum rule and the product rule. The formal definition of Bayes' Theorem is as follows:

$$P(A|B) = \frac{P(B|A) \times P(A)}{P(B)} \tag{2.1}$$

Where the terms are defined as:

- $P(A|B)$ is the **posterior** probability, representing the revised probability of event A after taking into account the occurrence of event B.
- $P(B|A)$ is the **likelihood**, indicating the probability of observing event B given that event A has occurred.
- $P(A)$ is the **prior** probability, which reflects the initial probability of event A before considering any new evidence.
- $P(B)$ is the **marginal likelihood**, representing the total probability of event B across all possible events.

Sum Rule and Product Rule

The sum rule and product rule are foundational concepts in probability that underpin Bayes' Theorem.

Sum Rule: The sum rule, also known as the law of total probability, asserts that the probability of a particular event is the sum of the probabilities of that event occurring under several disjoint scenarios. In the context of Bayes' Theorem, the sum rule is used to calculate $P(B)$, the marginal likelihood. It is expressed as:

$$P(B) = \sum_i P(B|A_i) \times P(A_i) \tag{2.2}$$

where A_i represents each disjoint scenario under which B could occur.

Product Rule: The product rule, also known as the rule of conjunction, states that the probability of two events occurring together (joint probability)

is the product of the probability of one event and the conditional probability of the second event given the first. It is formally stated as:

$$P(A, B) = P(A) \times P(B|A) = P(B) \times P(A|B) \tag{2.3}$$

This rule is essential in deriving the formula for Bayes' Theorem, as it relates the joint probability of A and B with their conditional probabilities.

Example 2.1. *Bayes' Theorem in the Context of Medical Testing*

Scenario Imagine there's a specific disease, Disease X, and there's a test to detect it. We know the following: - Only 1% of the population has Disease X. So, $P(DiseaseX) = 0.01$. - If a person has the disease, the test will correctly identify it 99% of the time. This is the sensitivity of the test, or $P(Positive\,Test|DiseaseX) = 0.99$. - However, the test also has a false positive rate of 5%. This means that 5% of people without the disease will still test positive. This is the specificity of the test, or $P(Positive\,Test|No\,DiseaseX) = 0.05$.

Now, if someone tests positive, what's the probability they actually have Disease X?

Using Bayes' Theorem To find this out, we'll use Bayes' theorem. We want to calculate $P(DiseaseX|Positive\,Test)$:

$$P(DiseaseX|Positive\,Test) = \frac{P(Positive\,Test|DiseaseX) \times P(DiseaseX)}{P(Positive\,Test)}$$

Where: - $P(Positive\,Test|DiseaseX)$ is the probability of a positive test given that one has the disease. As stated, it's 0.99. - $P(DiseaseX)$ is the prior probability of having the disease, which is 0.01. - $P(Positive\,Test)$ is the probability of getting a positive test in general. We can get this using the law of total probability:

$$\begin{aligned} P(\text{PositiveTest}) = {} & P(\text{Disease}) \times P(\text{PositiveTest}|\text{Disease}) \\ & + P(\text{NoDisease}) \times P(\text{PositiveTest}|\text{NoDisease}) \end{aligned}$$

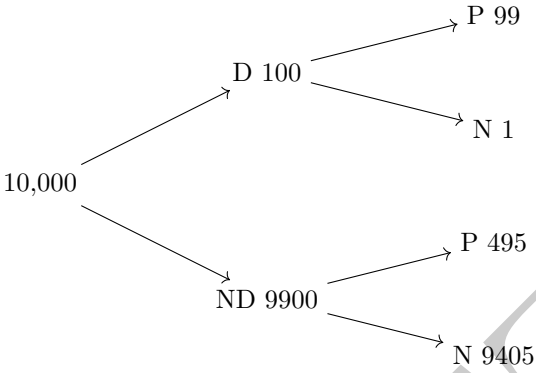
Inserting the known values, we get:

$$P(Positive\,Test) = 0.01 \times 0.99 + 0.99 \times 0.05 = 0.0594$$

Now, plugging into our Bayes' formula:

$$P(DiseaseX|Positive\,Test) = \frac{0.99 \times 0.01}{0.0594} \approx 0.166$$

Alternative calculation To visualize the application of Bayes' theorem, we can use a tree diagram, which provides a breakdown of the problem.



Explanation Starting with a population of 10,000 people:

- **Disease Split:**

D (Disease) : 1% of 10,000 is 100 people.
ND (No Disease) : 99% of 10,000 is 9,900 people.

- **Test Results:**

For those with the disease (D):
P (Positive Test) : 99% of 100 is 99 people.
N (Negative Test) : 1% of 100 is 1 person.
For those without the disease (ND):
P (Positive Test) : 5% of 9,900 is 495 people.
N (Negative Test) : 95% of 9,900 is 9,405 people.

Using the tree, we can calculate the probability that a person with a positive test result actually has the disease:

$$P(Disease|PositiveTest) = \frac{\text{Number of true positives}}{\text{Total positives}}$$
$$P(Disease|PositiveTest) = \frac{99}{99 + 495}$$
$$P(Disease|PositiveTest) \approx 0.166$$

Thus, the probability that someone with a positive test result actually has the disease is approximately 16.6%.

So, given a positive test result, there's only a 16.6% chance of actually having Disease X. This is surprisingly low, but it's because the disease is rare and the test has a relatively high false positive rate.

Conclusion Bayes' theorem helps us take into account both the base rate of an event (how common the disease is) and the accuracy of our test (sensitivity and specificity). Even if a test seems highly accurate, the actual probability can be quite different when the condition being tested for is rare. This understanding is crucial in fields like medicine where the stakes are high.

2.1.2 The Naïve Bayes Classification

The Naïve Bayes classifier works under the "Naïve" assumption that all features are conditionally independent given the target variable. This assumption simplifies the computation of the joint probability distribution, which is decomposed into the product of individual probabilities.

Mathematically, a Naïve Bayes classifier is defined as follows:

Given a set of classes $C = \{c_1, c_2, ..., c_k\}$ and a feature vector $X = (x_1, x_2, ..., x_n)$, the classifier assigns a probability to each class c_i for a given X by using Bayes' theorem:

$$P(c_i|X) = \frac{P(X|c_i)P(c_i)}{P(X)}$$

where:

- $P(c_i|X)$ is the posterior probability of class c_i given predictor X.
- $P(c_i)$ is the prior probability of class c_i.
- $P(X|c_i)$ is the likelihood which is the probability of predictor X given class c_i.
- $P(X)$ is the prior probability of predictor.

Since $P(X)$ is constant for all classes, only the numerator needs to be maximized. Under the Naïve conditional independence assumption, the likelihood is simplified to:

$$P(X|c_i) = \prod_{j=1}^{n} P(x_j|c_i)$$

Thus, the generative aspect comes from the fact that $P(x_j|c_i)$ is estimated from the training data and can be used to generate new feature instances for a given class c_i, hence the model is generative.

In practice, the Naïve Bayes classifier estimates $P(x_j|c_i)$ for all features x_j and classes c_i from the training dataset, and these probabilities are used to make predictions on new data by generating the posterior probabilities of the classes. In summary, the Naïve Bayes Classification is a simple yet effective algorithm for classification tasks, especially when computational efficiency is important or when the dataset isn't very large. While its underlying assumption of feature independence can be a drawback, it still performs remarkably well in many real-world applications.

Extensions and Variants While the standard Naïve Bayes classifier is effective, there have been numerous attempts to improve or modify it:

- **TAN (Tree Augmented Naïve Bayes):** This approach goes beyond the assumption of feature independence within each class of the traditional Naïve Bayes model. In TAN, each feature X_i can have one parent X_j (another feature), which forms a tree structure. The conditional probability is then modeled as $P(X_i|X_j, c)$, instead of $P(X_i|c)$, which captures some of the dependencies between attributes. This can lead to improved performance over standard Naïve Bayes, particularly when the independence assumption is strongly violated.

  Example: Suppose we have attributes A and B that are conditionally dependent given class C. In a TAN model, we might have $P(A|B, C)$ instead of $P(A|C)$ and $P(B|C)$ independently.

- **BNB (Bayesian Network-based Naïve Bayes):** A Bayesian network is a graphical model that represents a set of variables and their conditional dependencies via a directed acyclic graph (DAG). In BNB, the features are not treated as independent, but rather the conditional dependencies between them are modeled explicitly. This can capture the relationships between variables more accurately than the standard Naïve Bayes classifier, potentially leading to more accurate predictions.

  Example: In a Bayesian network for classification, the presence of an edge from X_i to X_j in the DAG represents a direct dependency of X_j on X_i. The joint probability distribution is factored into a product of conditional probabilities: $P(X_1, ..., X_n, C) = P(C) \prod_{i=1}^{n} P(X_i|Parents(X_i))$, where $Parents(X_i)$ are the parents of X_i in the DAG.

Both TAN and BNB can be seen as efforts to relax the strong independence assumptions of Naïve Bayes and to capture the underlying dependencies between features, which can be especially important in domains where these dependencies are strong.

Example 2.2. *Spam/Ham Classification*

Consider a scenario where we want to classify emails as either **spam** or **not spam** (ham) based on the frequency of two keywords: "sale" and "bonus". Let's use a small dataset to demonstrate the application of Naïve Bayes.

Training Data:

Email	Frequency of "sale"	Frequency of "bonus"	Label
1	3	0	spam
2	0	2	ham
3	1	2	spam
4	0	1	ham
5	2	1	spam

Given the data, we can calculate the probabilities required for Naïve Bayes.

- $P(\text{spam}) = \frac{3}{5}$
- $P(\text{ham}) = \frac{2}{5}$
- $P(\text{"sale"}|\text{spam}) = \frac{6}{11}$
- $P(\text{"sale"}|\text{ham}) = 0$
- $P(\text{"bonus"}|\text{spam}) = \frac{3}{11}$
- $P(\text{"bonus"}|\text{ham}) = \frac{3}{8}$

Subject: Exclusive Sale Bonus!

Dear Customer,

We're excited to announce that our annual Summer Sale is now even better! For this week only, we're offering an exclusive bonus to all of our loyal customers. Get an additional 10% off on already reduced sale items. This is the perfect opportunity to enjoy the benefits of the sale and get that item you've had your eye on.

Remember, the sale is for a limited time, and stock is limited. Don't miss out on this exclusive chance to save.

Best regards,
The SuperSale Team

Let's see what the predicted class is for the above email in which "sale" appears 2 times and "bonus" appears 1 time:

$$P(\text{spam}|\text{data}) \propto P(\text{spam}) \times P(\text{"sale"} = 2|\text{spam}) \times P(\text{"bonus"} = 1|\text{spam})$$

$$P(\text{ham}|\text{data}) \propto P(\text{ham}) \times P(\text{"sale"} = 2|\text{ham}) \times P(\text{"bonus"} = 1|\text{ham})$$

Inserting the given probabilities:

$$P(\text{spam}|\text{data}) \propto \frac{3}{5} \times \left(\frac{6}{11}\right)^2 \times \frac{3}{11}$$

$$P(\text{ham}|\text{data}) \propto \frac{2}{5} \times 0 \times \frac{3}{8} = 0$$

Given that the probability $P(\text{ham}|\text{data})$ is zero, the email is classified as spam.

As noticed, the lack of the word "sale" automatically results in a zero probability for the email being ham. This is problematic since having no evidence shouldn't conclude a strict zero probability. To overcome this, one common technique is to apply Laplace smoothing or add-α smoothing.

By adding a α count to every word frequency and adjusting the denominators appropriately, we avoid zero probabilities. If we set $\alpha = 1$ (a common choice), the probabilities are updated as:

$$P(\text{"sale"}|\text{spam}) = \frac{6+1}{11+2(1)} = \frac{7}{13}$$

$$P(\text{"sale"}|\text{ham}) = \frac{0+1}{11+2(1)} = \frac{1}{13}$$

$$P(\text{"bonus"}|\text{spam}) = \frac{3+1}{11+2(1)} = \frac{4}{13}$$

$$P(\text{"bonus"}|\text{ham}) = \frac{3+1}{8+2(1)} = \frac{4}{10}$$

Now, for the new email with "sale" frequency of 2 and "bonus" frequency of 1, we can compute:

$$\begin{aligned} P(\text{spam}|\text{data}) &\propto P(\text{spam}) \times [P(\text{"sale"}|\text{spam})]^2 \times P(\text{"bonus"}|\text{spam}) \\ &= \frac{3}{5} \times \left(\frac{7}{13}\right)^2 \times \frac{4}{13} \\ &\approx 0.153 \end{aligned}$$

$$\begin{aligned} P(\text{ham}|\text{data}) &\propto P(\text{ham}) \times [P(\text{"sale"}|\text{ham})]^2 \times P(\text{"bonus"}|\text{ham}) \\ &= \frac{2}{5} \times \left(\frac{1}{13}\right)^2 \times \frac{4}{10} \\ &\approx 0.002 \end{aligned}$$

From the calculated probabilities, $P(\text{spam}|\text{data}) > P(\text{ham}|\text{data})$. Thus, the email would be classified as spam given the higher posterior probability.

2.2 Hidden Markov Models

Hidden Markov Models (HMMs) are a sophisticated extension of Markov models, built upon the foundational principles of Markov processes but enhanced to handle scenarios where the states of the system are not directly observable. To fully grasp the concept and application of HMMs, it is essential to begin with an understanding of Markov models, as they provide the structural underpinning upon which HMMs are constructed.

Markov models are based on the Markov property, where the future state of a process is conditional solely on the present state and not on the sequence of events that preceded it. This property simplifies the complexity of probabilistic modeling by reducing the dependency horizon to one step at a time. By starting with Markov models, we set the stage for appreciating the additional complexities addressed by HMMs, such as dealing with hidden states and indirect observations.

This introductory foundation is crucial because it allows us to first consider the simpler case of fully observable processes before introducing the additional layer of complexity inherent in HMMs. Understanding Markov models facilitates a smoother transition to the more complex, yet more powerful, realm of HMMs, where we not only consider state transitions but also the probabilistic generation of observable data from these hidden states. Thus, we begin our exploration with Markov models to build a solid base from which the principles of HMMs will be more intuitively understood.

2.2.1 Markov Models

Markov models are quintessential in the study of stochastic processes across various disciplines such as statistics, physics, economics, and computer science. Named after the Russian mathematician Andrey Markov, these models are instrumental in modelling the probabilistic transitions of a system through different states in a sequential manner.

A Markov model is defined by a collection of states and the probabilities of transitioning from one state to another, encapsulated within a structure known as the transition matrix.

Consider a sequence of state variables $q_1, q_2, \ldots, q_i$. A Markov model embodies the Markov assumption on the probabilities of this sequence: that when predicting the future, the past doesn't matter, only the present state is relevant. This assumption simplifies the analysis of stochastic processes by asserting that the future state depends only on the current state and not on the sequence of events that preceded it.

$$\text{Markov Assumption: } P(q_i = a | q_1 \ldots q_{i-1}) = P(q_i = a | q_{i-1}) \tag{2.4}$$

This implies that the transition probabilities in a Markov model are memoryless, which allows the model to be characterized entirely by the set of transition probabilities from one state to another. The Markov property greatly reduces

the complexity of the probability calculations and is fundamental to the operation of Markov chains, Markov processes, and many models in time series analysis and signal processing.

2.2.2 Zero$^{\text{th}}$-Order Markov Model

A zero$^{\text{th}}$-order Markov model, also known as a memoryless model, posits that state transitions are entirely independent. The probability of transitioning to any state remains constant, unaffected by the process's history. This model, akin to i.i.d[1] random variables, are not typically employed for sequence modelling but can be applicable to independent state processes.

2.2.3 First-Order Markov Model

The first-order Markov model is the most elementary form, where the next state's probability is contingent only on the current state. For a sequence of states $X_1, X_2, \ldots, X_n$, the first-order Markov property is mathematically expressed as:

$$P(X_n|X_1, X_2, \ldots, X_{n-1}) = P(X_n|X_{n-1})$$

This model's simplicity and reasonable approximation to many real-world processes make it widely utilized.

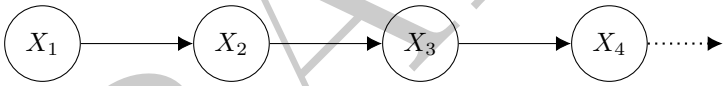


Figure 2.1: Illustration of a first-order Markov chain, where each state X_i transitions to the next state X_{i+1} with a probability that depends only on X_i. This chain demonstrates the Markov property, which asserts that the future state is conditionally dependent only on the current state and not on the sequence of previous states.

2.2.4 Higher-Order Markov Models

For scenarios where the first-order assumption is inadequate, higher-order Markov models are employed. A k-$^{\text{th}}$ order Markov model considers the probability of transitioning to the next state based on the current and the preceding $k-1$ states. A second-order Markov model, for instance, determines the transition probability based on the current state and its predecessor:

$$P(X_{n+1} = x|X_{n-1} = x_{n-1}, X_n = x_n)$$

[1] When variables are i.i.d., each data point you collect or each outcome you observe is assumed to provide information that does not depend on the other data points or outcomes, and all data points follow the same statistical rules. This assumption is foundational in many statistical methods and probabilistic models.

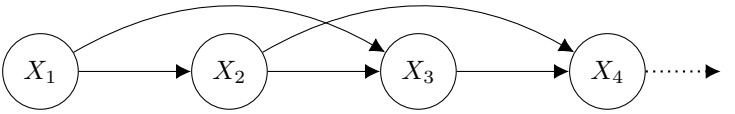


Figure 2.2: Representation of a second-order Markov chain, where the transition from state X_i to state X_{i+2} depends on the states X_i and X_{i+1}. This pattern generalizes to k-th order Markov chains, where the transition to state X_{i+1} depends on the previous k states.

Though higher-order models capture more historical context and can more accurately represent the process, they require more computational resources and extensive data for precise transition probability estimation.

2.3 Hidden Markov Models

Hidden Markov Models (HMMs) extend the concept of Markov models to scenarios where the system being analyzed is not fully observable. While Markov models are predicated on the assumption of fully observable states where transitions between these states are explicitly known, HMMs operate on the principle that the states themselves are hidden and only certain outcomes or emissions linked to these states can be observed. This additional layer of abstraction makes HMMs particularly powerful in modelling complex systems where the state of the system can only be inferred through indirect observations.

In an HMM, the system undergoes transitions between hidden states according to a Markov process, but unlike a standard Markov model, each state produces an observation that is probabilistically dependent on the state. The sequence of observed outputs can provide insights into the sequence of hidden states, but the relationship is not one-to-one; multiple state sequences can result in the same observation sequence. Figure **??** shows different components of a HMM.

The power of HMMs lies in their ability to deal with uncertainty and provide a structured way to model the stochastic processes that generate sequences of observed data. They have found applications in a diverse range of fields, from natural language processing and speech recognition to bioinformatics and financial modeling, where the underlying process needs to be understood from incomplete or indirect measurements.

How HMMs are different from Markov Models A Markov model is a related concept, where the system being modeled is assumed to have the Markov property; that is, the future state depends only on the current state and not on the sequence of events that preceded it. All states in a Markov model are directly visible to the observer, which contrasts with an HMM where the states

are not directly observed. The relationship between the two models can be expressed as follows:

- A Markov model is a special case of an HMM where the observable output is directly related to the state, making the state transition probabilities the sole parameters of the model.
- An HMM extends the Markov model by including the concept of hidden states. The hidden states follow the Markov property, but they generate the observed data according to the observation probability distribution, adding another layer of complexity to the model.

Thus, while a Markov model's structure is simpler and more transparent, HMMs offer a more powerful and flexible framework for modelling complex sequences where the states themselves are not directly observable.

The state-space model for an HMM is formalized as follows:

- Let $S = \{s_1, s_2, ..., s_N\}$ be a finite set of states.
- The state at time t, X_t, is a random variable that takes on values from S.
- The process begins in state X_0 with an initial state distribution π, such that $P(X_0 = s_i) = \pi_i$ for $i = 1, 2, ..., N$.
- The state transition probabilities are given by an $N \times N$ matrix A, where $a_{ij} = P(X_{t+1} = s_j | X_t = s_i)$.
- The observation probabilities are described by a sequence of random variables Y_t that are conditionally dependent on the state X_t. The probability of an observation y_t given state s_j is given by $b_j(y_t)$[2].

Using these components, the HMM can be represented by the following equations:

State Equation: The state transition dynamics in an HMM are governed by the state equation. For a given time step t, the next state X_{t+1} is conditionally dependent on the current state $X_t = s_i$. This dependency is modeled by a categorical distribution, denoted as $X_{t+1}|(X_t = s_i) \sim \text{Categorical}(a_i)$, where a_i is the i-th row of the transition probability matrix A. This matrix contains the probabilities of transitioning from any state to any other state in the model.

Observation Equation: The observation equation models the process by which observations are produced by the HMM. At each time step t, an observation Y_t is generated based on the current state $X_t = s_j$. The observation

[2]In the context of HMMs, the observation probabilities are characterized by a sequence of random variables Y_t that are conditionally dependent on the hidden state X_t. The random variable Y_t represents the observation at time t, while y_t denotes a specific observed value at time t. The probability of observing a specific value y_t given the hidden state s_j is represented by the probability mass function $b_j(y_t)$. Therefore, the notation Y_t is used when referring to the random variable as a whole, and y_t is used when referring to a specific observation or outcome.

likelihood is given by $Y_t|(X_t = s_j) \sim \text{Discrete}(b_j)$, where b_j represents the observation probability distribution associated with the state s_j. The distribution b_j defines the probability of each possible observation emanating from state s_j.

In an HMM, the state sequence is not directly observable, hence the name 'hidden'. These equations jointly characterize the evolution of hidden states and the probabilistic generation of observations, encapsulating the essence of the Hidden Markov Model.

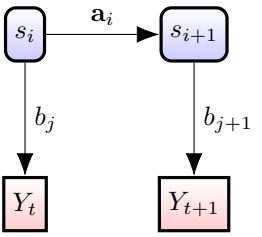


Figure 2.3: Graphical representation of the Hidden Markov Model equations.

Are HMMs generative? HMMs are considered generative models because they define a joint probability distribution over observation sequences and their corresponding hidden state sequences. In other words, HMMs can generate data by first generating a sequence of hidden states according to the transition probabilities and then generating observations from these states based on the observation probabilities.

We can express the reason why HMMs are generative models through the following joint probability distribution for a sequence of observations $\mathbf{Y} = \{Y_1, Y_2, ..., Y_T\}$ and hidden states $\mathbf{X} = \{X_1, X_2, ..., X_T\}$:

$$P(\mathbf{Y}, \mathbf{X}) = P(X_1) \prod_{t=2}^{T} P(X_t|X_{t-1}) \prod_{t=1}^{T} P(Y_t|X_t)$$

Where:

- $P(X_1)$ is the initial state probability, given by the initial state distribution π.
- $P(X_t|X_{t-1})$ is the state transition probability from state X_{t-1} to X_t, given by the transition probability matrix A.
- $P(Y_t|X_t)$ is the observation probability of Y_t given state X_t, described by the observation probability distribution B.

By specifying the complete joint probability distribution of both hidden states and observations, HMMs can be used to generate new sequences of data that are statistically similar to those on which the model was trained. This

generative nature is particularly useful for simulation, bootstrapping datasets[3], and for Bayesian inference where one may want to estimate the hidden states given observed data.

Example 2.3. *Data generation*

Consider a HMM with two weather conditions as hidden states – 'Rainy' and 'Sunny' – and two clothing choices as observations – 'Coat' and 'T-shirt'.

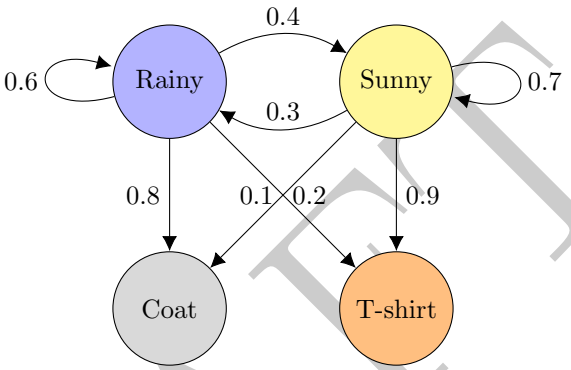


Figure 2.4: A simple example for generating data using an HMM with two weather conditions as hidden states – 'Rainy' and 'Sunny' – and two clothing choices as observations – 'Coat' and 'T-shirt'.

The HMM parameters are as follows:
Transition probabilities:

$$P(\text{Rainy} \to \text{Rainy}) = 0.6$$
$$P(\text{Rainy} \to \text{Sunny}) = 0.4$$
$$P(\text{Sunny} \to \text{Rainy}) = 0.3$$
$$P(\text{Sunny} \to \text{Sunny}) = 0.7$$

Emission probabilities:

$$P(\text{Coat}|\text{Rainy}) = 0.8$$
$$P(\text{T-shirt}|\text{Rainy}) = 0.2$$
$$P(\text{Coat}|\text{Sunny}) = 0.1$$
$$P(\text{T-shirt}|\text{Sunny}) = 0.9$$

[3]Bootstrapping is a statistical resampling technique used to estimate the distribution of a sample statistic by repeatedly sampling with replacement from the original dataset. It allows one to infer about the population from sample data by generating many samples of the same size as the original dataset.

Initial state probabilities:

$$P(\text{Rainy}) = 0.5$$
$$P(\text{Sunny}) = 0.5$$

Here's an example of generating a sequence of observations for 5 days:

1. Start with an initial state, chosen based on the initial state probabilities. Let's assume it's 'Rainy'.
2. Choose the clothing for the day based on the emission probabilities for 'Rainy'. With a higher probability for 'Coat', we'll likely choose 'Coat'.
3. Transition to the next state using the transition probabilities for 'Rainy'. There's a 60% chance the next day will also be 'Rainy'.
4. Repeat the process for the next day. Let's say the second day is 'Sunny', we'll likely choose a 'T-shirt'.
5. Continue for the desired sequence length.

Given these steps, the sequence of clothing choices over five days might be:

- Day 1: Rainy - Coat
- Day 2: Sunny - T-shirt
- Day 3: Sunny - T-shirt
- Day 4: Rainy - Coat
- Day 5: Rainy - Coat

This sequence illustrates the generative capability of an HMM to simulate new data sequences that are statistically similar to the training data. Such simulations are useful for bootstrapping datasets and performing Bayesian inference to estimate hidden states given observed data.

2.3.1 Components of HMMs

Let's recap what we have seen until now by reviewing the fundamental components:

- **State Sequence:** A sequence of hidden states that describes the temporal evolution of the system. It is not directly observable.
- **Observation Sequence:** A sequence of observable symbols that we can measure or observe.
- **State Transition Probabilities (A):** These probabilities define the likelihood of transitioning from one state to another. In a first-order HMM, these depend only on the current state.

- **Emission Probabilities (B):** These describe the likelihood of emitting an observation symbol from a hidden state. They are sometimes called output probabilities.
- **Initial State Probabilities (π):** These represent the probability distribution over initial states when the sequence starts.

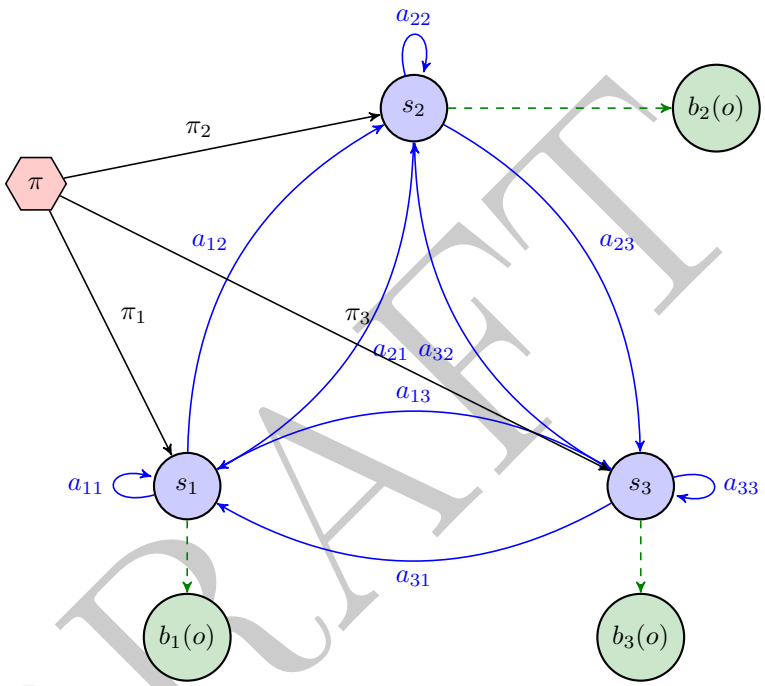


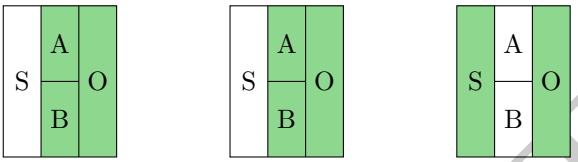
Figure 2.5: Illustration of a Hidden Markov Model (HMM). The states of the model are represented by blue circles (S1, S2, S3), with their self-transition probabilities (e.g., a_{11}, a_{22}, a_{33}) indicated by blue looped arrows. The transitions between different states are shown with blue arrows labelled with their respective probabilities (e.g., a_{12}, a_{23}, a_{31}). The initial state probabilities are denoted by black arrows from the red hexagonal node π to each state (e.g., π_1, π_2, π_3). Observations are represented by green dashed arrows pointing from each state to its corresponding observation symbol (e.g., $b_1(o)$, $b_2(o)$, $b_3(o)$).

Three Fundamental Problems in Hidden Markov Models Following Jurafsky and Martin in *Speech and Language Processing*, we divide hidden Markov models into three fundamental problems as follows:

Problem 1 (Likelihood): Given an HMM $\lambda = (A, B)$ and a sequence of observations O, the objective is to compute the likelihood $P(O|\lambda)$.

Problem 2 (Decoding): For an observation sequence O and an HMM $\lambda = (A, B)$, the task is to find the most probable sequence of hidden states S.

Problem 3 (Learning): With an observation sequence O and the set of states in the HMM, the goal is to determine the optimal HMM parameters A and B.



$P(O|A, B) =?$ S=? $A, B =?$

Figure 2.6: Graphical summarisation of HMM problems

Problem 1 (Likelihood):

- **Purpose:** Determine how likely a given observation sequence is, considering a particular HMM.
- **Details:** Given a specific HMM defined by the parameter set $\lambda = (A, B, \pi)$, where A is the matrix of transition probabilities, B is the matrix of emission probabilities, and π is the vector of initial state probabilities, one must compute the likelihood of an observation sequence O. This likelihood is denoted as $P(O|\lambda)$, which represents the probability of observing the sequence O given the model parameters λ. The task involves utilizing the full parameter set λ in the likelihood computation.
- **Significance:** This helps in assessing how well the model explains the observations. A higher likelihood indicates the observations are more consistent with the model.
- **Solution**: To solve a likelihood problem in HMMs, one typically employs the Forward Algorithm. This algorithm is designed to efficiently compute the probability of observing a given sequence of events, considering the model's parameters: the states, the possible observations, the transition probabilities between states, and the emission probabilities of observations from states. By iteratively applying these probabilities, the Forward Algorithm sums over all possible paths that could lead to the observed sequence, ultimately providing the likelihood of the sequence given the HMM. This approach is fundamental in many applications of HMMs, such as speech recognition and bioinformatics.

Problem 2 (Decoding):

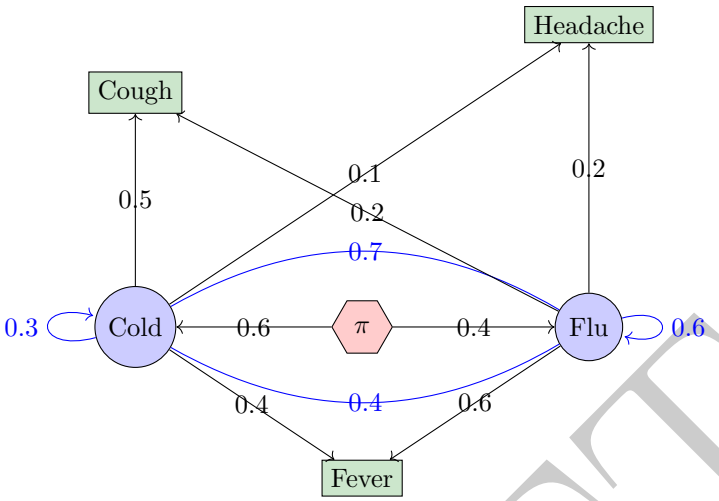
- **Purpose:** Find the most likely sequence of hidden states that could have produced a given observation sequence.
- **Details:** Given an HMM with parameters A and B and an observation sequence O, the goal is to determine the hidden state sequence Q that is most probable given the observations and the model.
- **Significance:** This is particularly useful in applications where the underlying states or sequences leading to the observed data are of interest. Decoding the hidden states can reveal insights into the process that generated the data, which is crucial for understanding the underlying system dynamics and for making predictions about future states or observations.
- **Solution**: The process of finding the most likely sequence of states involves dynamic programming algorithms like the Viterbi algorithm, which efficiently computes the most likely path through the state space that results in the observed sequence.

Problem 3 (Learning):

- **Purpose:** Adjust the model parameters to best fit a given observation sequence.
- **Details:** Given an observation sequence O and the set of possible states in the HMM, this problem pertains to adjusting (or "learning") the model parameters A, B, and π such that they maximize the likelihood of the observations given the model.
- **Significance:** This is vital for scenarios where the HMM isn't predefined. With ample observational data, the objective is to construct an HMM that explains the data best by "learning" the optimal parameters. The learned model can then be used for various tasks like prediction, anomaly detection, and sequence classification. The quality of the learned model directly impacts the performance of these tasks, making the learning phase crucial in the practical application of HMMs.
- **Solution**: This is typically achieved using iterative algorithms such as the Baum-Welch algorithm, a special case of the Expectation-Maximization (EM) algorithm, which iteratively estimates the probabilities until convergence.

Example 2.4. *Decoding problem: Diagnosing a Disease Based on Symptoms*

Imagine a scenario in which a doctor is trying to diagnose whether a patient has a cold or the flu based only on the symptoms observed over three days.



States (Hidden):

$$S = \{\text{Cold}, \text{Flu}\}$$

Observations (Visible):

$$O = \{\text{Cough}, \text{Fever}\}$$

State Transition Probabilities:

$$A = \begin{bmatrix} & \text{Cold} & \text{Flu} \\ \text{Cold} & 0.7 & 0.3 \\ \text{Flu} & 0.4 & 0.6 \end{bmatrix}$$

Observation Likelihoods:

$$B = \begin{bmatrix} & \text{Cough} & \text{Fever} & \text{Headache} \\ \text{Cold} & 0.5 & 0.4 & 0.1 \\ \text{Flu} & 0.2 & 0.6 & 0.2 \end{bmatrix}$$

Initial State Probabilities:

$$\pi = \{\text{Cold: } 0.6, \text{Flu: } 0.4\}$$

Given an observation sequence $O = \{\text{Cough}, \text{Fever}\}$, one could use the **Viterbi algorithm** to determine the most likely sequence of states S that led to these observations. (The appendix provides a description for this algorithm).

Using the Viterbi Algorithm Assuming the initial state distribution $\pi(s)$ is such that $\pi(\text{Cold}) = 0.6$ and $\pi(\text{Flu}) = 0.4$, we apply the Viterbi algorithm to the observation sequence {Cough, Fever}.

Step 1: Initialization

We compute the initial probabilities for each state:
For 'Cold':

$$v_1(\text{Cold}) = \pi(\text{Cold}) \times B(\text{Cold}, \text{Cough}) = 0.6 \times 0.5 = 0.3$$

For 'Flu':

$$v_1(\text{Flu}) = \pi(\text{Flu}) \times B(\text{Flu}, \text{Cough}) = 0.4 \times 0.2 = 0.08$$

Step 2: Recursion

For each subsequent observation, we compute the Viterbi path probability. For $t = 2$ with observation 'Fever':
For 'Cold':

$$v_2(\text{Cold}) = \max\{v_1(\text{Cold}) \times A(\text{Cold}, \text{Cold}), v_1(\text{Flu}) \times A(\text{Flu}, \text{Cold})\} \times B(\text{Cold}, \text{Fever})$$

$$v_2(\text{Cold}) = \max\{0.3 \times 0.7, 0.08 \times 0.4\} \times 0.4 = \max\{0.21, 0.032\} \times 0.4 = 0.084$$

For 'Flu':

$$v_2(\text{Flu}) = \max\{v_1(\text{Cold}) \times A(\text{Cold}, \text{Flu}), v_1(\text{Flu}) \times A(\text{Flu}, \text{Flu})\} \times B(\text{Flu}, \text{Fever})$$

$$v_2(\text{Flu}) = \max\{0.3 \times 0.3, 0.08 \times 0.6\} \times 0.6 = \max\{0.09, 0.048\} \times 0.6 = 0.054$$

Step 3: Termination

To find the most probable sequence, we look at the last step:
For $t = 2$, we have $v_2(\text{Cold}) = 0.084$ and $v_2(\text{Flu}) = 0.054$. Thus, the most probable state for the second observation is 'Cold'.
Given the higher probability for the 'Cold' state at both $t = 1$ and $t = 2$, the most likely sequence of states is {Cold, Cold}.

2.3.2 HMM Applications

Hidden Markov Models (HMMs) are utilized across various domains to model sequences of observable events generated by systems or processes with underlying hidden states. Below are detailed applications in several fields:

- **Speech Recognition:** In speech recognition, HMMs model the sequence of spoken words as a probabilistic function of acoustic signal patterns. Each word or phoneme can be represented by an HMM, and the models are trained using large datasets of spoken language to recognize new speech inputs.

- **Natural Language Processing (NLP):** HMMs are employed for part-of-speech tagging by treating words as observations and grammatical categories as hidden states. They are also used in named entity recognition to identify and classify proper nouns in text, and in machine translation to model the alignment of words in bilingual corpora.

- **Bioinformatics:** In bioinformatics, HMMs are applied for gene prediction by modeling the regions of DNA sequences as hidden states, which helps in identifying coding and non-coding regions. They are also used for sequence alignment to match biological sequences and infer homology, and for protein structure prediction by modeling the sequence of amino acids and their respective conformations.

- **Financial Analysis:** HMMs are used to model financial time series by capturing the hidden factors driving market movements. They can help in identifying regime shifts and anomalies in stock price data, which are useful for risk management and algorithmic trading strategies.

- **Gesture Recognition:** In computer vision, HMMs help in recognizing and interpreting human gestures by modeling the temporal variations in gesture sequences. This application is significant in developing user interfaces that rely on gesture-based control.

- **Robotics:** HMMs are also applied in robotics for tasks such as path planning and the recognition of complex activities, where the sequence of movements can be observed and the underlying intention or goal is the hidden state to be inferred.

- **Weather Forecasting:** Meteorologists use HMMs to model sequences of weather conditions to predict future states. This involves using atmospheric data as observations with weather patterns as hidden states.

Challenges of HMMs

- **Markov Property Assumption:** HMMs assume that future states depend only on the current state (first-order Markov property). This can be overly simplistic for processes where future states depend on multiple previous states.

- **Predefined State Spaces:** Standard HMMs require the number of states to be defined a priori, which may not be practical or optimal for all applications, especially when the true underlying state space is unknown or variable.

- **Homogeneous Transition Probabilities:** HMMs traditionally assume that transition probabilities are constant over time, which may not hold in non-stationary environments where transition dynamics change.

- **Limited Duration Modeling:** In classical HMMs, the state duration distribution is geometric, which may be inappropriate for modeling state durations that do not fit this distribution well.

Variants of HMMs

To mitigate these challenges, researchers have proposed several variants of HMMs:

- **Hidden Semi-Markov Models (HSMMs):** HSMMs extend HMMs by allowing the state duration distribution to be explicitly modeled, rather than being geometrically distributed. This allows for a more flexible representation of state durations, making HSMMs suitable for processes where state persistence is variable and information-rich.

- **Continuous HMMs (CHMMs):** To handle continuous-time processes, CHMMs have been developed. They model the transitions between states in continuous time, which is particularly useful in applications like asset price modeling in finance where observations occur at irregular time intervals.

- **Higher-Order HMMs:** These models relax the first-order Markov property assumption by allowing the prediction of future states to depend on a history of several past states, not just the current state, enhancing the memory of the model.

- **Factorial HMMs:** This variant assumes multiple hidden state sequences that are independent of each other, which can simultaneously influence the observation process. Factorial HMMs are capable of capturing more complex structures in the data.

By expanding the capabilities of the standard HMM, these variants enable a more nuanced approach to modeling real-world processes that exhibit complex temporal dynamics and dependencies.

2.4 Gaussian Mixture Models

Gaussian Mixture Models (GMMs) are probabilistic models that assume all the data points are generated from a mixture of a finite number of Gaussian distributions with unknown parameters. GMMs belong to a category of unsupervised learning algorithms that are used for identifying hidden structures within a dataset. Gaussian Mixture Models (GMM) are considered generative models due to their capacity to generate new sample points that are statistically similar to a given set of data.

2.4.1 Mathematical Formulation

Consider a mixture of Gaussians as a way to describe a more complex probability distribution than what is possible with a single Gaussian distribution. This approach allows for a more detailed analysis of data that might come from multiple underlying sources.

A Gaussian mixture model can be defined as a weighted sum of K Gaussian densities. These densities are combined to provide a more flexible representation of a dataset. The formula for a mixture of K Gaussian distributions is given by:

$$p(\mathbf{x}) = \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$$

Here, $p(\mathbf{x})$ is the probability density function of the mixture, π_k are the mixing coefficients representing the weight of each Gaussian component in the mixture, $\mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$ is the k-th Gaussian density function with mean $\boldsymbol{\mu}_k$ and covariance $\boldsymbol{\Sigma}_k$. The mixing coefficients π_k must sum to 1, ensuring that the total probability distribution is valid.

To understand the model more deeply, we introduce latent variables $\mathbf{z}$, which are not observed directly but indicate which Gaussian component is responsible for generating each data point. These latent variables have a 1-of-K representation, meaning that for each data point, exactly one element of $\mathbf{z}$ is 1, and all others are 0. The probability of $\mathbf{z}$ taking a particular value (which Gaussian is active) is given by the mixing coefficients.

The joint distribution of observed and latent variables is given by:

$$p(\mathbf{x}, \mathbf{z}) = p(\mathbf{z})p(\mathbf{x}|\mathbf{z})$$

The marginal distribution of $\mathbf{x}$ is then obtained by summing over all possible values of $\mathbf{z}$, resulting in the Gaussian mixture formula stated earlier.

In a more practical sense, if you have a set of observations $\mathbf{x}_1, \ldots, \mathbf{x}_N$, and you want to model this data with a Gaussian mixture, you need to estimate the parameters of the mixture (means, covariances, and mixing coefficients) such that the likelihood of the observed data under the model is maximized. This

is typically done using the Expectation-Maximization (EM) algorithm, which iteratively updates the parameters until convergence.

To fully grasp the concept of a Gaussian Mixture Model (GMM), consider that we are not only looking at the observed variables $\mathbf{x}$, but also at latent variables $\mathbf{z}$. The latent variables have a binary representation where only one component is active at a time (it's set to 1), and all others are 0.

The parameters of a GMM, π_k, must satisfy two conditions:

1. They are all between 0 and 1, inclusive:

$$0 \leq \pi_k \leq 1$$

2. They sum up to 1:

$$\sum_{k=1}^{K} \pi_k = 1$$

These conditions ensure that the π_k values form a valid probability distribution.

The probability of a particular latent variable $\mathbf{z}_k$ being active (equal to 1) is given by the mixing coefficient π_k, which can be written as:

$$p(z_k = 1) = \pi_k$$

The joint probability of $\mathbf{x}$ and $\mathbf{z}$ can be expressed as a product of the marginal probability of $\mathbf{z}$ and the conditional probability of $\mathbf{x}$ given $\mathbf{z}$:

$$p(\mathbf{x}, \mathbf{z}) = p(\mathbf{z})p(\mathbf{x}|\mathbf{z})$$

Since $\mathbf{z}$ uses a 1-of-K representation, the marginal distribution $p(\mathbf{z})$ is simply the product of the probabilities of each z_k:

$$p(\mathbf{z}) = \prod_{k=1}^{K} \pi_k^{z_k}$$

Similarly, the conditional probability $p(\mathbf{x}|\mathbf{z})$ for a Gaussian mixture can be written as a product over the K components:

$$p(\mathbf{x}|\mathbf{z}) = \prod_{k=1}^{K} \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)^{z_k}$$

Finally, the marginal distribution of $\mathbf{x}$ is obtained by summing the joint distribution $p(\mathbf{x}, \mathbf{z})$ over all possible states of $\mathbf{z}$, leading to the mixture distribution:

$$p(\mathbf{x}) = \sum_{\mathbf{z}} p(\mathbf{z})p(\mathbf{x}|\mathbf{z}) = \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$$

2.4.2 Parameter Estimation Using the EM Algorithm

The Expectation-Maximization (EM) algorithm is a method used to find the maximum likelihood estimates of parameters in statistical models, particularly when the model depends on unobserved latent variables. The algorithm works by iteratively applying two steps: the Expectation step (E-step) and the Maximization step (M-step).

Expectation Step (E-Step): During the E-step, we calculate the responsibilities, which indicate the probability that a data point was generated by a particular Gaussian component of the mixture. Responsibilities are denoted by $\gamma(z_{nk})$ and are calculated for each data point $\mathbf{x}_n$ and each Gaussian component k. The formula for calculating responsibility is:

$$\gamma(z_{nk}) = \frac{\pi_k \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\sum_{j=1}^{K} \pi_j \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)}$$

where π_k is the prior probability that a data point comes from the k-th Gaussian component, $\mathcal{N}$ represents the Gaussian probability density function, and $\boldsymbol{\mu}_k$ and $\boldsymbol{\Sigma}_k$ are the mean and covariance matrix of the k-th Gaussian component, respectively. This step involves estimating the probability distribution of the latent variables given the observed data and the current estimate of the model parameters.

Maximization Step (M-Step): Once the responsibilities are calculated, we update the parameters of the GMM to maximize the expected log-likelihood. The new estimates for the means, covariance matrices, and mixing coefficients are computed using the following formulas:

$$\boldsymbol{\mu}_k = \frac{1}{N_k} \sum_{n=1}^{N} \gamma(z_{nk}) \mathbf{x}_n, \quad \boldsymbol{\Sigma}_k = \frac{1}{N_k} \sum_{n=1}^{N} \gamma(z_{nk})(\mathbf{x}_n - \boldsymbol{\mu}_k)(\mathbf{x}_n - \boldsymbol{\mu}_k)^T, \quad \pi_k = \frac{N_k}{N}$$

Here, N_k represents the effective number of data points for the k-th component, calculated as $N_k = \sum_{n=1}^{N} \gamma(z_{nk})$. The M-step involves updating the model parameters to maximize the expected complete-data log-likelihood that was computed in the E-step.

Convergence of EM: The EM algorithm alternates between performing the E-step and the M-step until the parameters converge, which means that the estimates of the parameters no longer significantly change with further iterations. This iterative process results in a set of parameters that are likely to be close to the maximum likelihood solution, accounting for the latent variables in the model.

In practice, the EM algorithm is particularly useful in scenarios where direct computation of the maximum likelihood estimate is not feasible due to the complexity of the model. For GMMs, this method allows for the efficient estimation of multiple Gaussian distributions that collectively explain the variability of the data.

Summary The Expectation-Maximization (EM) algorithm starts by setting initial values for the parameters: means, covariances, and mixing coefficients. It operates through a two-step iterative process: the Expectation (E) step and the Maximization (M) step. During the E step, the algorithm utilizes the current parameter values to compute the posterior probabilities, often referred to as responsibilities. These probabilities are then applied in the M step, where the algorithm updates and refines the estimates for the means, covariances, and mixing coefficients. A key feature of the EM algorithm is that each parameter update is designed to incrementally increase the log likelihood function, indicating a more accurate model fit. In practical applications, the algorithm is considered to have reached convergence when the increment in the log likelihood function, or alternatively the change in the parameter values themselves, becomes smaller than a predetermined threshold. This threshold acts as a stopping criterion, signaling that subsequent iterations would yield negligible improvement in the model's accuracy.

2.4.3 Applications of Gaussian Mixture Models

Gaussian Mixture Models (GMMs) are versatile tools with a plethora of applications across various domains. Some of the notable applications include:

- **Density Estimation:** GMMs are adept at estimating the underlying probability distribution of complex datasets. By assuming that the data are generated from a mixture of several Gaussian distributions, GMMs can model a wide range of data distributions, even those that are multimodal in nature.

- **Clustering:** One of the primary uses of GMMs is in clustering. Unlike K-means, which assigns each data point to a single cluster, GMMs assign probabilities of belonging to each cluster, allowing for a more nuanced understanding of data points that might not clearly belong to one specific group. This probabilistic clustering approach is particularly useful in scenarios where the clusters have different sizes and correlation structures.

- **Anomaly Detection:** In anomaly detection, GMMs can identify data points that are statistically rare or unusual under the estimated model. By calculating the likelihood of each data point under the GMM, those with significantly low likelihoods can be flagged as anomalies or outliers, which is crucial in fields like fraud detection, network security, and quality control.

- **Image Analysis:** In the realm of computer vision, GMMs are employed for tasks such as image segmentation, where an image is partitioned into segments based on the distribution of pixel intensities. They are also used in object recognition and background subtraction, especially in scenarios where the background or the objects of interest have a Gaussian-like distribution of colors or intensities.

- **Speech and Audio Processing:** GMMs play a critical role in audio processing applications, notably in speaker verification systems where they are used to model the voice characteristics of individuals. In music analysis, they help in identifying and separating different instruments or sounds in an audio clip. The model's ability to handle variations in speech patterns and musical notes makes it particularly effective in these fields.

Chapter 3

Neural Generative Models

Generative modeling is a fascinating area of machine learning that focuses on the construction of models which can generate new data instances. Traditional generative models, such as Gaussian Mixture Models (GMM) or Hidden Markov Models (HMM) that we saw in Chapter 2, rely heavily on domain knowledge and manual feature selection to capture the underlying data distribution. They often work well on structured data where the relationships between variables are well-understood and can be explicitly programmed.

In contrast, neural generative models, leveraging the power of neural networks, offer a paradigm shift in this domain. With their ability to automatically learn complex, high-dimensional distributions without extensive domain-specific preprocessing, neural generative models have gained significant attention. They excel in handling unstructured data, like images and text, where traditional models struggle.

Neural generative models, such as Variational Autoencoders (VAEs) and Generative Adversarial Networks (GANs), have demonstrated remarkable success in synthesizing realistic and diverse data that is often indistinguishable from real-world data. These models are equipped with components like encoders, decoders, and adversarial networks that operate in a latent space—a conceptual space where the essence of the data is captured in a compressed form.

In this chapter, we delve into the details of the components that comprise neural generative models. We discuss how each component contributes to the model's overall ability to learn and generate new data instances, drawing a clear distinction from their traditional counterparts. The aim is to provide a comprehensive understanding of the inner workings of neural generative models and their place in the broader context of generative machine learning.

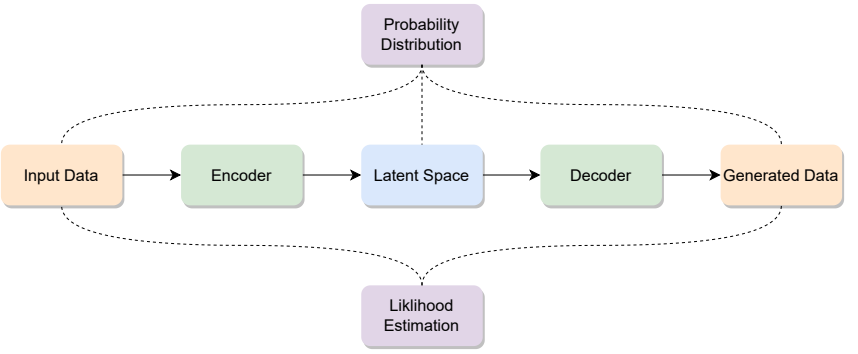


Figure 3.1: Architecture of a Neural Generative Model

3.1 Encoder

The encoder is an essential component in many generative and representational neural architectures. It acts as the mechanism to transform high-dimensional input data into a more compact, often lower-dimensional, representation. Delving deeper:

- **Functionality:** At its core, an encoder's main task is data compression. In generative models like Variational Autoencoders (VAEs), the encoder is responsible for mapping input data to a point in the latent space. This point essentially captures the essential characteristics of the input data but in a much more condensed form.
- **Structure:** Encoders typically comprise a sequence of neural network layers. Depending on the model architecture and data type, these layers could be dense, convolutional, pooling layers, etc. In many architectures, especially convolutional ones, the encoder's design progressively reduces the spatial dimensions of the data, thereby compressing it.
- **Training:** During training, an encoder learns to retain as much relevant information about the input data as possible in its compressed representation. This is typically achieved by jointly training the encoder with a decoder and minimizing the difference (reconstruction loss) between the original data and the decoder's output.
- **Applications:** While primarily known for its role in generative models, encoders are also fundamental in tasks like dimensionality reduction, feature extraction, and anomaly detection. In these applications, the compact representation provided by the encoder is used for visualization, clustering, or other downstream tasks.

3.2 Latent Space and Distribution

Latent space, sometimes referred to as the embedding or feature space, is a conceptual realm where high-dimensional input data is represented in a lower-dimensional, often more interpretable form. Along with the space itself, the probability distribution defined over this space is equally crucial in the context of generative models. Diving deeper:

- **Functionality:** The latent space serves as an intermediate representation where the essential features and patterns of the data are captured in a compressed form. By exploring this space, one can generate new samples or understand the underlying structure of the data.
- **Dimensions and Compactness:** Typically, the dimensionality of the latent space is much smaller than that of the input data. This compact representation forces the encoder to retain only the most salient features of the data, thus ensuring efficiency and often better generalization.
- **Distribution:** In generative models, especially probabilistic ones like VAEs, the latent space is associated with a probability distribution (often Gaussian). Instead of mapping inputs to fixed points, the encoder maps them to parameters of this distribution, such as means and variances. This stochastic nature aids in generating diverse outputs and imparts robustness.
- **Sampling:** The ability to sample from the latent distribution is a cornerstone of generative models. By drawing random samples from this distribution and feeding them to a decoder, one can produce new data instances that are similar, yet not identical, to the training data.
- **Interpolation and Arithmetic:** One of the intriguing properties of a well-defined latent space is the ability to perform arithmetic. For instance, in models trained on images, one can 'average' the embeddings of two faces to produce a hybrid or move along certain dimensions to add or subtract features.
- **Applications:** Beyond generation, a well-structured latent space can be instrumental in tasks like data visualization (e.g., t-SNE plots), clustering, or even semi-supervised learning where the embeddings serve as feature vectors.

3.3 Likelihood Estimation

Given a statistical model with parameters θ, the likelihood $L(\theta|X)$ represents how well the model, with those parameters, explains the observed data X.

Mathematically, if $P(X|\theta)$ is the probability of observing data X given the parameters θ, then the likelihood is given by:

$$L(\theta|X) = P(X|\theta)$$

It's important to note the distinction between probability and likelihood:

- **Probability:** Assesses the chance of observing data X given known parameters θ.
- **Likelihood:** Evaluates how plausible specific parameter values θ are, given observed data X.

Maximizing Likelihood in Generative Models Training generative models often involve adjusting the model parameters to maximize the likelihood of the observed data. This process, known as **Maximum Likelihood Estimation (MLE)**, can be denoted as:

$$\hat{\theta} = \arg\max_{\theta} L(\theta|X)$$

Where $\hat{\theta}$ is the parameter value that maximizes the likelihood.

In the context of generative models, MLE aims to find the parameters θ such that the model's distribution is as close as possible to the true data distribution. This often corresponds to minimizing the divergence or difference between the model's distribution and the actual distribution of the data.

Implications and Considerations

- **Overfitting:** While MLE provides a method to optimize model parameters, it's prone to overfitting, especially with complex models and limited data. Regularization techniques might be needed to prevent this.
- **Computational Challenges:** For some generative models, the likelihood function can be computationally intractable. Variational methods or sampling techniques, such as Markov Chain Monte Carlo (see Appendix), might be employed in such scenarios.
- **Model Comparison:** The likelihood can be used for comparing different models. However, models with more parameters might inherently fit the data better. To account for this, model comparison methods like Akaike Information Criterion (AIC) or Bayesian Information Criterion (BIC) (see Appendix) can be used, which introduce a penalty for model complexity.

3.4 Decoder

In generative models, the decoder plays a pivotal role in translating compressed or encoded representations back into the original or a similar data space. The functionality of the decoder can be understood better by breaking it down:

- **Functionality:** The primary function of a decoder is to reconstruct data. In the context of generative models like Variational Autoencoders (VAEs), the decoder maps points from the latent space back to the data space. For instance, if the input data was an image, the decoder will take a point from the latent space and produce an image as the output.

- **Structure:** Decoders typically consist of a series of neural network layers. Depending on the architecture and the type of data, these might be dense layers, convolutional layers, upsampling layers, etc. The structure is usually the reverse of the encoder (see Figure **??**). For example, in a convolutional autoencoder, while the encoder might use pooling layers to downsample the data, the decoder will use upsampling layers to increase the data's spatial dimensions.

- **Training:** The decoder is trained to minimize the difference between its output and the original data. This difference, often termed the "reconstruction loss", ensures that the encoder and decoder work in tandem to retain as much information about the original data as possible, even after compression into the latent space.

- **Generative Capability:** In generative settings, once the model (like a VAE) is trained, the decoder can be used in isolation to generate new data samples. By feeding random points from the latent distribution into the decoder, new samples that resemble the training data can be produced.

- **Applications:** Beyond generative modeling, decoders are essential in various other domains such as image denoising, inpainting, and super-resolution, where the objective is not just to reproduce the input but to enhance or modify it in specific ways.

The decoder, in essence, serves as the bridge between the compact latent representations and the rich, high-dimensional data space, enabling both data reconstruction and generation.

3.5 An Architectural Comparison

In the final section of this chapter, we present an architectural comparison between traditional generative models and their neural counterparts.

While traditional models often rely on **explicit probabilistic formulations** and **assumptions about the data distribution**, neural generative models harness the flexibility of neural networks to **implicitly learn these distributions**. This section delineates the fundamental architectural differences that give rise to the distinct characteristics and capabilities of these two paradigms.

Through this comparison, we aim to crystallize the understanding of how the evolution from traditional to neural approaches has expanded the horizons

of generative modeling, leading to advancements in fields such as unsupervised learning, data augmentation, and beyond.

- **Representation and Latent Space:**
  - **Neural Models:** Use encoders to compress data into a latent space, capturing the most salient features of the data.
  - **Traditional Models:** Instead of a distinct latent space, models like Gaussian Mixture Models (GMMs) or Factor Analysis capture latent structures by defining underlying probabilistic components or latent factors.
- **Modeling Approach:**
  - **Neural Models:** Rely on interconnected neurons, with the decoder reconstructing data from the latent space.
  - **Traditional Models:** Directly model data using statistical distributions. For instance, a Bayesian network captures joint distributions over variables using a directed acyclic graph, effectively encoding dependencies without a neural encoder's need.
- **Generative Process:**
  - **Neural Models:** A decoder generates new samples based on the latent representation.
  - **Traditional Models:** New samples are directly generated from the learned probabilistic distributions. In a GMM, samples are drawn based on mixture components; in a Bayesian network, samples can be generated by ancestral sampling following the graph structure.
- **Training:**
  - **Neural Models:** Use gradient descent and backpropagation.
  - **Traditional Models:** Rely on statistical principles. For instance, GMMs optimize component means, variances, and weights via the EM algorithm, while Bayesian networks are trained using methods like maximum likelihood or Bayesian inference to learn the structure and parameters of the graph.
- **Data Transformation:**
  - **Neural Models:** Neural encoders transform input data into a compact latent representation, and decoders reverse this transformation.
  - **Traditional Models:** Models like Principal Component Analysis (PCA) or Factor Analysis transform data into a lower-dimensional space using linear combinations of original features, akin to an encoding process. The inverse transformation can be used to reconstruct the original data, similar to decoding.

While the architectural methods differ, both neural and traditional generative models implement mechanisms to capture, represent, and generate data based on its inherent structures and patterns. The parallels highlighted underscore that many concepts are consistently applied, albeit through different methodologies.

3.6 Chapter Summary

This chapter delved into the intricate world of Neural Generative Models, presenting an in-depth examination of their key components and the theoretical underpinnings that make them a powerful tool in modern machine learning. We began by exploring the **encoder**, a fundamental piece of the architecture that compresses input data into a condensed latent representation. Following this, we investigated the **latent space** and the role of probability distributions within it, which are critical for the generation of new, synthetic data points.

Moving forward, we addressed the concept of **likelihood estimation**, discussing how maximizing the likelihood function is pivotal for training generative models effectively. The chapter highlighted the delicate balance that must be maintained during this process, emphasizing the **implications and considerations** to avoid common pitfalls such as overfitting.

The **decoder**, acting in concert with the encoder, was then examined for its role in reconstructing input data from the latent space, or in the case of generative tasks, synthesizing new data from learned distributions. We underscored the symmetry yet distinct functionality that decoders offer in comparison to encoders.

Finally, the chapter concluded with **An Architectural Comparison** that juxtaposed the traditional generative models with neural-based approaches. This comparison underscored the evolution from explicitly defined mathematical models to data-driven, neural network models that learn to approximate the data generating distribution directly from the data itself.

Through this exploration, the chapter aimed to provide a clear and comprehensive overview of Neural Generative Models, equipping the reader with a robust understanding of their architecture, applications, and the theoretical considerations that inform their design and implementation.

Chapter 4

Transformers

Transformers represent a groundbreaking innovation in the field of natural language processing (NLP) and artificial intelligence (AI). They are a type of neural network architecture that has revolutionized various machine learning tasks, particularly in NLP. Transformers were introduced by [**?**] in the paper titled "Attention Is All You Need" in 2017, and since then, they have become the foundation for state-of-the-art language models.

Basic Concepts At the heart of Transformers is the concept of self-attention mechanisms. Unlike traditional sequential models like Recurrent Neural Networks (RNNs) and Convolutional Neural Networks (CNNs), Transformers process input data in parallel, making them highly efficient. The key innovation is the attention mechanism, which allows the model to weigh the importance of different parts of the input sequence when making predictions.

In developing Transformers architecture, several key motivations emerged from the limitations observed in recurrent neural networks (RNNs) and their variants:

- **Performance Concerns with RNNs**: Traditional recurrent neural network structures, especially when extended to more sophisticated variants such as Long Short-Term Memory (LSTM) networks, tend to be computationally slower. As the complexity of these networks increases, so too does the computational overhead, making them less efficient for real-time processing or large-scale deployments.

- **Inherent Sequential Nature of RNNs**: One of the inherent characteristics of RNNs is their sequential data processing nature. This design choice prohibits the parallel processing of input sequences, requiring data to be fed into the model in a serial fashion. This not only limits the throughput of the model but also poses challenges for scalability, especially in applications where rapid sequence processing is essential.

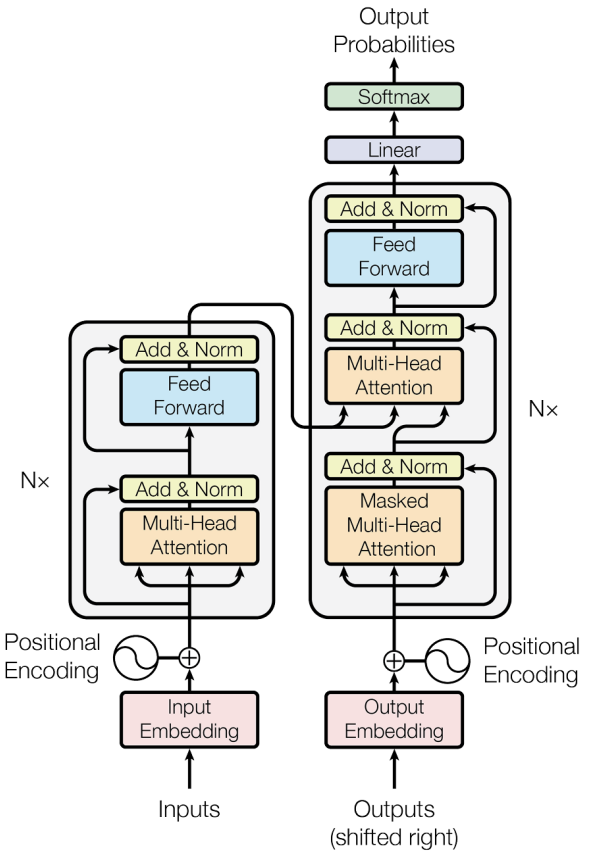


Figure 4.1: The architecture of Transformers, taken from [**?**]

- **Challenges with Gradient Propagation**: RNNs, and to a lesser extent LSTMs, have been observed to suffer from the vanishing[1] and exploding gradient problems. These phenomena can hinder the network's ability to effectively learn from long sequences of data, leading to suboptimal training outcomes and limited model efficacy. In scenarios where long-term dependencies are crucial, this limitation becomes particularly pronounced.

Transformer Architecture The Transformer architecture consists of two main components: the encoder and the decoder. These components are stacked on top of each other to form deep models.

[1]The vanishing gradient problem occurs in deep neural networks when the gradients, necessary for learning, become extremely small and diminish as they are propagated back through the layers, leading to ineffective training, especially in the early layers.

- **Encoder:** The encoder takes input data and processes it in parallel through multiple layers of self-attention and feedforward neural networks. It encodes the input data into a series of hidden representations.

- **Decoder:** The decoder takes the encoded representations and generates output sequences step by step. It also employs self-attention mechanisms but with additional masked attention to ensure that each position only depends on earlier positions in the output sequence.

4.0.1 Building blocks

1. **Multi-Head Self Attention Mechanism:** The self-attention mechanism allows the model to focus on different parts of the input sequence when producing an output sequence. The multi-head aspect lets the model focus on different positional aspects simultaneously.

   (a) **Query, Key, Value (QKV) Vectors:** For each token in the input, three vectors are derived: a Query vector, a Key vector, and a Value vector. These vectors help compute attention scores.

$$\text{Attention}(Q, K, V) = \text{Softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{4.1}$$

   Q**:** Query Matrix.
   - The query matrix, denoted as Q, represents the input element for which we want to calculate the attention weights. It is a matrix where each row corresponds to a query vector, often associated with a specific element or position in the input sequence.

   K**:** Key Matrix.
   - The key matrix, denoted as K, represents the elements that are used to compute compatibility with the query. Like the query matrix, it is also a matrix where each row corresponds to a key vector.
   - The key vectors are used to compare how well they match or are related to the query vectors, and this comparison is an essential part of the attention mechanism.

   V**:** Value Matrix.
   - The value matrix, denoted as V, represents the values that are associated with the elements in the input sequence. Similar to the query and key matrices, it is also organized as a matrix with value vectors in its rows.
   - The values in the matrix represent the information or content associated with the corresponding elements in the input sequence. They are the information that will be selected or

weighted by the attention mechanism based on the compatibility between the query and key vectors.

d_k**:** Dimension of Key Vectors.

- The variable d_k represents the dimension of the key vectors, which is typically a constant and is often related to the architecture of the neural network model.
- The dimension of the key vectors plays a role in the calculation of the compatibility between the query and key vectors and is used to scale the dot product of the query and key vectors.

(b) **Scaled Dot-Product Attention:** Attention scores between tokens are calculated using the dot product of the Query and Key vectors, scaled by the square root of the model's depth, and passed through a softmax function.

(c) **Weighted Sum of Value Vectors:** These scores produce a weighted average of the Value vectors, giving more importance to certain tokens when constructing the output sequence.

Figure 4.2 shows how these calculations are made.

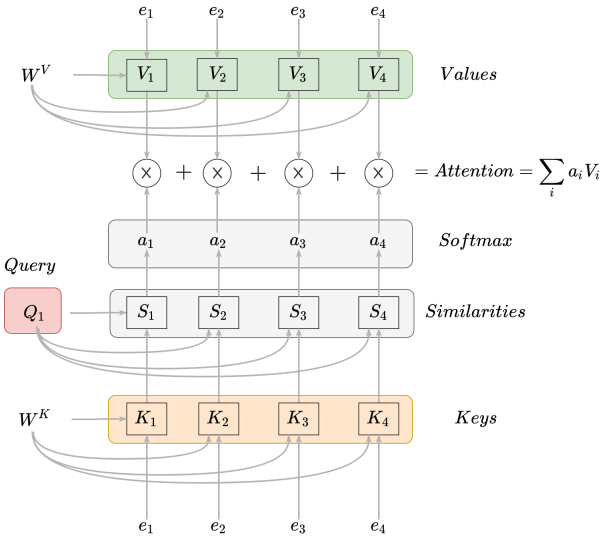


Figure 4.2: Calculating Attention

2. **Positional Encoding:** Since the Transformer lacks inherent sequential processing, positional encodings are added to the input embeddings at the bottom layers, providing the model with some information about the relative or absolute position of tokens.

Example 4.1.

Consider a sequence of tokens, each represented by a 4-dimensional embedding vector:

$$\text{Token 1 Embedding } (\mathbf{E}_1) = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \quad \text{Token 2 Embedding } (\mathbf{E}_2) = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix},$$

$$\text{Token 3 Embedding } (\mathbf{E}_3) = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

For positional encoding, we use sine and cosine functions for each dimension:

$$PE(pos, 2i) = \sin\left(\frac{pos}{10000^{2i/d}}\right), \quad PE(pos, 2i+1) = \cos\left(\frac{pos}{10000^{2i/d}}\right)$$

where pos is the position, i is the dimension, and $d = 4$.

The positional encodings for positions 1, 2, and 3 are calculated as follows:

$$\text{Positional Encoding for Position 1 } (\mathbf{PE}_1) = \begin{bmatrix} \sin\left(\frac{1}{10000^0}\right) \\ \cos\left(\frac{1}{10000^0}\right) \\ \sin\left(\frac{1}{10000^{1/2}}\right) \\ \cos\left(\frac{1}{10000^{1/2}}\right) \end{bmatrix} = \begin{bmatrix} 0.8415 \\ 0.5403 \\ 0.0998 \\ 0.9950 \end{bmatrix},$$

$$\text{Positional Encoding for Position 2 } (\mathbf{PE}_2) = \begin{bmatrix} \sin\left(\frac{2}{10000^0}\right) \\ \cos\left(\frac{2}{10000^0}\right) \\ \sin\left(\frac{2}{10000^{1/2}}\right) \\ \cos\left(\frac{2}{10000^{1/2}}\right) \end{bmatrix} = \begin{bmatrix} -0.9093 \\ -0.4161 \\ 0.1987 \\ 0.9801 \end{bmatrix},$$

$$\text{Positional Encoding for Position 3 } (\mathbf{PE}_3) = \begin{bmatrix} \sin\left(\frac{3}{10000^0}\right) \\ \cos\left(\frac{3}{10000^0}\right) \\ \sin\left(\frac{3}{10000^{1/2}}\right) \\ \cos\left(\frac{3}{10000^{1/2}}\right) \end{bmatrix} = \begin{bmatrix} 0.1411 \\ -0.9899 \\ 1.2955 \\ 0.9553 \end{bmatrix}$$

The final inputs to the Transformer, summing the token embeddings and their corresponding positional encodings, are:

$$\mathbf{E}_1 + \mathbf{PE}_1 = \begin{bmatrix}1\\0\\0\\0\end{bmatrix} + \begin{bmatrix}0.8415\\0.5403\\0.0998\\0.9950\end{bmatrix} = \begin{bmatrix}1.8415\\0.5403\\0.0998\\0.9950\end{bmatrix},$$

$$\mathbf{E}_2 + \mathbf{PE}_2 = \begin{bmatrix}0\\1\\0\\0\end{bmatrix} + \begin{bmatrix}-0.9093\\-0.4161\\0.1987\\0.9801\end{bmatrix} = \begin{bmatrix}-0.9093\\0.5839\\0.1987\\0.9801\end{bmatrix},$$

$$\mathbf{E}_3 + \mathbf{PE}_3 = \begin{bmatrix}0\\0\\1\\0\end{bmatrix} + \begin{bmatrix}0.1411\\-0.9899\\1.2955\\0.9553\end{bmatrix} = \begin{bmatrix}0.1411\\-0.9899\\2.2955\\0.9553\end{bmatrix}$$

3. **Feed-Forward Neural Networks:** Each attention output is passed through a position-wise feed-forward network (the same one applied independently to each position), followed by layer normalisation.

4. **Residual Connections:** Residual connections, also known as skip connections, are a key component in deep neural networks that help to **mitigate the vanishing** gradient problem. These connections enable the flow of gradients directly through the network by adding the output of a layer to the input of a layer a few positions ahead in the network. If we denote the output of a given sub-layer as $\text{SubLayer}(x)$, where x is the input to the sub-layer, then the output y of the residual block is given by:

$$y = \text{LayerNorm}(x + \text{SubLayer}(x))$$

   This addition of x ensures that the gradients can propagate through the network **without diminishing in strength**, thus making it possible to train deeper models more effectively. Residual connections are especially crucial in transformers as they consist of many such stacked sub-layers.

5. **Layer Normalisation:** Layer normalization is a technique used to stabilize the training of deep neural networks. It works by normalizing the activations of a layer across the features instead of the batch. For a given layer's output x, the layer normalization can be described as:

$$\hat{x} = \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}}$$

   where μ and σ^2 are the mean and variance of the features, and ϵ is a small constant for numerical stability. The normalized output $\hat{x}$ is then scaled and shifted by learnable parameters, which allows the model to learn the appropriate scale and bias for each layer.

Example 4.2.

Consider an input vector x and a sub-layer SubLayer(x) in a neural network:

$$x = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}, \quad \text{SubLayer}(x) = \begin{bmatrix} 0.5 \\ 1.5 \\ -1.0 \end{bmatrix}$$

The output y of the residual block is calculated as:

$$y = \text{LayerNorm}(x + \text{SubLayer}(x))$$

First, we compute the addition of x and SubLayer(x):

$$x + \text{SubLayer}(x) = \begin{bmatrix} 1.5 \\ 3.5 \\ 2.0 \end{bmatrix}$$

The mean μ and variance σ^2 of this addition, assuming $\epsilon = 10^{-5}$ for numerical stability, are:

$$\mu = \frac{1.5 + 3.5 + 2.0}{3} = 2.3333$$

$$\sigma^2 = \frac{(1.5 - 2.3333)^2 + (3.5 - 2.3333)^2 + (2.0 - 2.3333)^2}{3} = 0.7222$$

The normalized output y is then:

$$y = \frac{\begin{bmatrix} 1.5 \\ 3.5 \\ 2.0 \end{bmatrix} - 2.3333}{\sqrt{0.7222 + 10^{-5}}} = \begin{bmatrix} -0.9806 \\ 1.3728 \\ -0.3922 \end{bmatrix}$$

This example illustrates how a residual connection operates by adding the output of the sub-layer to the input and then normalizing it, which aids in alleviating the vanishing gradient problem in deep neural networks, particularly in transformers.

6. **Dropout:** Dropout is a regularization technique that involves randomly setting a fraction of input units to 0 at each update during training time, which helps to prevent overfitting. In transformers, dropout is applied to the output of each sub-layer before it is added to the sub-layer input and normalized. This prevents the model from relying too much on any one feature and promotes the learning of more robust features that are useful in combination with many different random subsets of the other features.

7. **Stacked Layers:** Both the encoder and decoder consist of a specified number of identical layers (usually 6 in base models) stacked on top of each other.

8. **Output Linear Layer & Softmax:** In the decoder, after the stacked layers, a linear layer projects the representations onto a set of vocabulary size scores, which are then transformed into probabilities using a softmax function.

Example 4.3.

Assume the decoder outputs a 2-dimensional vector $\mathbf{d}$ and we have a vocabulary size of 3. Let:

$$\mathbf{d} = \begin{bmatrix} 0.5 \\ -1 \end{bmatrix}, \quad \mathbf{W} = \begin{bmatrix} 1 & 2 \\ 0 & -1 \\ -1 & 1 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 0 \\ 1 \\ -1 \end{bmatrix}$$

The output of the linear layer $\mathbf{z}$ is calculated as:

$$\mathbf{z} = \mathbf{W} \cdot \mathbf{d} + \mathbf{b} = \begin{bmatrix} 1 & 2 \\ 0 & -1 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} 0.5 \\ -1 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \\ -1 \end{bmatrix} = \begin{bmatrix} -1.5 \\ 2.0 \\ -2.5 \end{bmatrix}$$

The softmax function converts these scores into probabilities. For a vector $\mathbf{z}$, the softmax function is defined as:

$$\text{Softmax}(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{3} e^{z_j}}$$

Applying the softmax function to $\mathbf{z}$ yields:

$$\text{Softmax}(\mathbf{z}) = \frac{1}{e^{-1.5} + e^{2.0} + e^{-2.5}} \begin{bmatrix} e^{-1.5} \\ e^{2.0} \\ e^{-2.5} \end{bmatrix} = \begin{bmatrix} 0.0290 \\ 0.9603 \\ 0.0107 \end{bmatrix}$$

This example demonstrates how the output from the decoder's stacked layers is projected onto a set of vocabulary size scores using a linear layer, and then transformed into probabilities using the softmax function in a Transformer model.

In practice, sentences that exceed the model's maximum length are either truncated or divided into smaller segments that fit within this limit. For applications where longer contexts are essential, models might employ strategies like sliding window attention or utilize architectures specifically designed to handle longer sequences, such as Longformer or Transformer-XL.

Thus, while a short sentence like "I love cats," with a length of three tokens, is easily within the capacity of standard Transformer models, longer sentences might require special handling depending on the model's design and the computational resources at hand.

4.1 Concept of Attention Mechanism

The attention mechanism is a breakthrough concept in natural language processing that allows models to selectively focus on certain parts of the input data, much like how human attention works. It has become particularly important in the context of sequence-to-sequence models, such as those used in machine translation, speech recognition, and text summarization.

Basic Idea The fundamental idea behind the attention mechanism is to enable the model to assign different weights to different parts of the input, depending on how relevant they are for the task at hand. This is akin to a human reader who, while reading a text, focuses more on the parts that are more informative or relevant to the context.

Example 4.4. *Attention Calculation*

Consider the sentence: "He loves cats". Let's compute the attention scores for the word "loves" when it acts as a query.

- **Input Sentence:** "He loves cats"

Given the simplified embeddings for each word:

$$e_{\text{He}} = [2, 4]$$
$$e_{\text{loves}} = [3, 2]$$
$$e_{\text{cats}} = [1, 3]$$

And let's assume that we learned these transformation matrices for Queries (Q), Keys (K), and Values (V) from the training phase:

$$Q = \begin{bmatrix} 1 & 2 \\ 0 & 1 \end{bmatrix}, K = \begin{bmatrix} 1 & 0 \\ 2 & 1 \end{bmatrix}, V = \begin{bmatrix} 2 & 1 \\ 1 & 0 \end{bmatrix}$$

Computing Q, K, and V for "loves":

$$Q_{\text{loves}} = e_{\text{loves}} Q = [3, 2] \begin{bmatrix} 1 & 2 \\ 0 & 1 \end{bmatrix} = [3, 8]$$

$$K_{\text{loves}} = e_{\text{loves}} K = [3, 2] \begin{bmatrix} 1 & 0 \\ 2 & 1 \end{bmatrix} = [7, 2]$$

$$V_{\text{loves}} = e_{\text{loves}} V = [3, 2] \begin{bmatrix} 2 & 1 \\ 1 & 0 \end{bmatrix} = [8, 3]$$

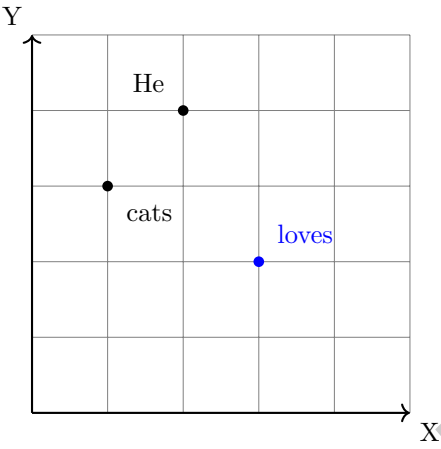


Figure 4.3: Original position of words

Calculating the similarity scores between "loves" and all words:

$$
\begin{aligned}
S_{\text{loves, He}} &= Q_{\text{loves}} \cdot K_{\text{He}} = [3, 8] \cdot [6, 8] = 62 \\
S_{\text{loves, loves}} &= Q_{\text{loves}} \cdot K_{\text{loves}} = [3, 8] \cdot [7, 2] = 37 \\
S_{\text{loves, cats}} &= Q_{\text{loves}} \cdot K_{\text{cats}} = [3, 8] \cdot [5, 4] = 45
\end{aligned}
$$

Applying the softmax function to the scores, we obtain the attention weights:

$$
\begin{aligned}
a_{\text{loves, He}} &= \frac{e^{62}}{e^{62} + e^{37} + e^{45}} \approx 1.0 \\
a_{\text{loves, loves}} &= \frac{e^{37}}{e^{62} + e^{37} + e^{45}} \approx 0.0 \\
a_{\text{loves, cats}} &= \frac{e^{45}}{e^{62} + e^{37} + e^{45}} \approx 0.0
\end{aligned}
$$

The final attention output for "loves" would be a weighted sum of the value vectors:

$$
\begin{aligned}
\text{Attention}_{\text{loves}} &= a_{\text{loves, He}} V_{\text{He}} + a_{\text{loves, loves}} V_{\text{loves}} + a_{\text{loves, cats}} V_{\text{cats}} \\
&= 1.0 \times [8, 4] + 0.0 \times [8, 3] + 0.0 \times [4, 5] \\
&= [8.0, 4.0]
\end{aligned}
$$

Figure 4.4 shows the final position of the word *loves*.

Input length In Transformer models, the maximum length of a sentence that can be processed for attention calculations depends on several factors:

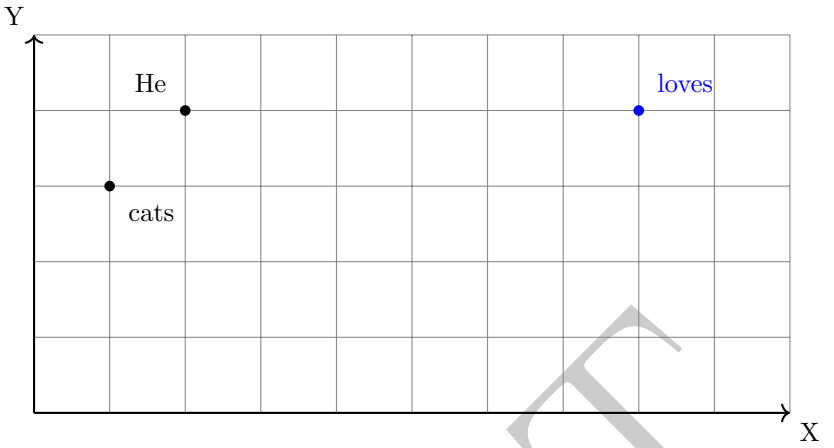


Figure 4.4: Position of the word *loves* has been changed based on the calculated attention

1. **Model Design:** Transformer architectures typically have a predefined maximum sequence length, set during the model's construction. This limit is often influenced by memory and computational constraints. For instance, the original Transformer model, as proposed by Vaswani et al., was designed with a maximum sequence length of 512 tokens.

2. **Computational Resources:** The sequence length is also limited by available computational resources. The attention mechanism in Transformers computes interactions between each pair of tokens in the sequence. For a sequence of length N, the self-attention mechanism generates an $N \times N$ matrix, which scales quadratically with the sequence length. Therefore, the practical limit on sentence length may be dictated by the available computational power and memory, such as GPU memory.

Pretrained Models One of the most significant advancements in NLP is the use of pre-trained Transformer-based models. Models like BERT (Bidirectional Encoder Representations from Transformers), GPT (Generative Pre-trained Transformer), and T5 (Text-to-Text Transfer Transformer) are pre-trained on massive text corpora and fine-tuned for various downstream tasks. These models have achieved remarkable results on tasks such as language understanding, generation, translation, and more.

Applications Transformers have a wide range of applications beyond NLP, including computer vision, speech recognition, and reinforcement learning. In NLP, they are used for tasks like:

- **Text Classification:** Transformers excel at classifying text into predefined categories.
- **Machine Translation:** They have improved the quality of machine translation systems significantly.
- **Question Answering:** Transformers are employed in question-answering systems, where they understand the context and generate relevant answers.
- **Summarization:** They can generate concise summaries of long texts, aiding in content summarization.

In computer vision, transformers contribute to tasks such as:

- **Image Classification:** They have shown promising results in classifying images into various categories.
- **Object Detection:** Transformers are used for detecting and identifying objects within an image, often with high accuracy.
- **Image Generation:** They can generate realistic images and artworks, showcasing their creativity.

In the field of speech recognition, transformers are instrumental in:

- **Speech-to-Text Conversion:** They transcribe spoken language into text with high accuracy.
- **Voice Synthesis:** Transformers are used in synthesizing human-like speech, contributing to more natural sounding voice assistants.

Moreover, in reinforcement learning, transformers aid in:

- **Decision Making:** They help in building models that can make decisions based on complex inputs.
- **Game Playing:** Transformers are used in developing AI that can play complex games like Go and chess, exhibiting advanced strategy understanding.

Chapter 5

Generative Pre-trained Transformers (GPT)

The Generative Pre-trained Transformer (GPT) family represents a series of innovative language models developed by OpenAI, distinguished by their deep transformer architectures and extensive pre-training on large text corpora. The GPT models have set new standards in natural language processing, demonstrating unparalleled proficiency in text generation, comprehension, and various other NLP tasks.

BERT (Bidirectional Encoder Representations from Transformers) and GPT (Generative Pre-trained Transformer) are two prominent examples of models that leverage the attention mechanism to create contextual word embeddings.

BERT utilizes the attention mechanism to process text bidirectionally. This means that for each word in a sentence, BERT looks at all the other words around it (both preceding and following words) to understand its context. The model employs what is known as 'self-attention' in the Transformer architecture, allowing each word to attend to all words in the sentence. Consequently, the word embedding generated for each word is contextually informed by its surrounding words. For instance, BERT would generate different embeddings for the word "bank" in "river bank" and "financial bank" based on the different contexts provided by surrounding words.

GPT, on the other hand, uses a unidirectional or auto-regressive approach. This means it only attends to previous words in the sequence for each word it processes. The attention mechanism in GPT assigns weights to preceding words to generate a contextually relevant embedding for the current word. Although GPT's context is limited to words that come before the current word, it still effectively captures the context within its scope. This model has shown remarkable performance in generating coherent and contextually relevant text, as each word it generates takes into account the context provided by all the previous words in the sequence.

In both BERT and GPT, the attention mechanism is the driving force behind their ability to produce embeddings that are deeply contextual. This contextuality is a significant leap from earlier non-contextual embeddings, enabling these models to achieve state-of-the-art performance in a wide range of natural language processing tasks.

Architecture:

- **Embedding Layer:**
  - **Token Embeddings:** In GPT models, the initial step involves converting the input text into a meaningful numerical representation. This is achieved using token embeddings. Each unique word or subword in the text is mapped to a high-dimensional vector. These embeddings are learned during training and are capable of capturing semantic and syntactic properties of the language.
  - **Position Embeddings:** Since GPT and similar Transformer models do not inherently process sequential data in order, position embeddings are added to the token embeddings to encode the sequence information. These embeddings provide the model with information about the position or order of each word in a sentence. They are crucial for the model to understand the sequence and structure of the language, enabling it to make sense of word order and syntactic relationships.
- **Transformer Layers:**
  - **Masked Multi-Head Self-Attention:** The core of the GPT model is its multiple layers of Transformer blocks. Each block contains a masked multi-head self-attention mechanism. This mechanism allows the model to weigh the importance of different words in a sentence relative to each other. The 'masked' aspect in GPT ensures that the prediction for a given word does not include information from future words in the sequence, making it suitable for tasks like text generation.
  - **Feedforward Neural Networks:** Alongside the self-attention mechanism, each Transformer block in GPT includes a feedforward neural network. This network consists of fully connected layers that process the output of the attention mechanism. The feedforward network applies further transformations to the data, allowing the model to better capture complex patterns and relationships in the text. The combination of self-attention and feedforward networks within each Transformer block enables the model to effectively process and generate language with a high degree of fluency and coherence.
- **Normalization and Activation:**

- **Layer Normalization:** GPT models employ layer normalization as a crucial component after each sub-block in the Transformer architecture (i.e., after the self-attention and feed-forward layers). Layer normalization is applied across the features for each individual data sample. This process normalizes the output of each layer, ensuring that the mean and variance are consistent across different layers. This normalization is crucial for stabilizing the learning process, as it mitigates the problem of internal covariate shift—where the distribution of each layer's inputs changes during training, making it difficult for the model to converge.
- **Activation Functions:** Following layer normalization, GPT models utilize activation functions to introduce non-linearity into the network. This is essential because, without non-linearities, the entire network, irrespective of its depth, could collapse into a linear model, thus severely limiting its representational power. GPT typically uses the GELU (Gaussian Error Linear Unit)[1] activation function. The GELU activation allows the model to make more complex decisions and capture more intricate patterns in the data, compared to purely linear transformations. It helps the model in learning and representing complex and abstract relationships within the data, which is essential for tasks such as language understanding and generation.

• **Output Layer:**

- The Output Layer in GPT models plays a crucial role in translating the complex representations and patterns learned by the model into tangible results. This layer is typically a fully connected neural network layer that maps the high-dimensional data from the final Transformer layer to a space that has the same dimensionality as the model's vocabulary.
- In text generation tasks, the Output Layer's primary function is to produce a probability distribution over the entire vocabulary for each position in the input sequence. This means that for each word or token that the model generates, it assigns probabilities to all possible subsequent words or tokens in the vocabulary, indicating how likely they are to follow the given sequence of words.
- The probabilities are usually obtained using a softmax function, which converts the raw output scores from the fully connected layer into normalized probabilities. The softmax function ensures that the

[1] GELU (Gaussian Error Linear Unit) is a type of activation function used in neural networks, particularly effective in Transformer models like BERT and GPT. It combines properties of both linear and non-linear functions. GELU allows some values to pass through (like a linear function) while it also gates other values based on their magnitude (similar to a non-linear function). This behavior makes it useful for complex tasks like language understanding, where it helps the model in making more nuanced decisions based on the input data.

sum of the probabilities of all possible next words equals one, making it a proper probability distribution.

- The word with the highest probability is often selected as the next word in the sequence for generation tasks. However, techniques like beam search or sampling methods may be used to introduce variety and creativity in the generation process.
- This final Output Layer is essential for the model to perform various natural language processing tasks, including text generation, language translation, and even more complex tasks that require an understanding of language context and structure.

Differences between Transformer Decoder and GPT Architecture The primary difference between the decoder part of a Transformer architecture and the GPT (Generative Pre-trained Transformer) architecture lies in their design and intended use:

- **Transformer Decoder:**
  - The Transformer model typically features an encoder-decoder structure. The decoder is tailored for output sequence generation, such as in machine translation.
  - It includes a masked self-attention mechanism to prevent positions from attending to subsequent positions, ensuring that the prediction for a token does not depend on future tokens.
  - The decoder also features an encoder-decoder attention mechanism, allowing each position in the decoder to attend over all positions in the input sequence, thus integrating input information into the generation process.
- **GPT Architecture:**
  - GPT is based on the Transformer's decoder architecture but operates as a standalone model, without the encoder part. It's primarily designed for language modeling and generation.
  - It utilizes a similar masked self-attention mechanism, ideal for generative tasks where each prediction should only depend on preceding tokens.
  - Unlike the Transformer decoder, GPT lacks an encoder-decoder attention mechanism, as it does not pair with an encoder. It is pre-trained on large text corpora for broad language understanding, which can be fine-tuned for specific tasks.

In summary, while both the Transformer decoder and GPT are based on similar principles, the Transformer decoder is typically part of a larger encoder-decoder

structure and includes an additional attention mechanism to integrate encoder information, whereas GPT functions independently as a generative model focused on language modeling and generation.

Pre-training:

- **Extensive Training on Large Corpora:** GPT models are pre-trained on a diverse and voluminous collection of text data. This pre-training stage involves processing a vast amount of text, which helps the model capture a wide array of linguistic structures and real-world knowledge.

- **Unsupervised Learning Approach:** During pre-training, GPT models use an unsupervised learning technique, typically language modeling. The primary task is to predict the next word in a sentence given the previous words. This task requires the model to understand and generate coherent and contextually relevant language, thus learning a broad range of language patterns.

- **Developing Contextual Understanding:** The pre-training process enables GPT models to develop an understanding of syntax, grammar, and even some aspects of world knowledge. The models learn to produce text that is not only grammatically correct but also contextually coherent, laying a strong foundation for the generation of human-like text.

Fine-tuning:

- **Adaptation to Specific Tasks:** After the extensive pre-training phase, GPT models are fine-tuned on more specialized tasks. This stage involves training the model on smaller, task-specific datasets, allowing it to adapt to the nuances and specific requirements of the task.

- **Adjustment of Model Weights:** During fine-tuning, the pre-trained weights of the model are adjusted. Although the bulk of the learning has already been done in the pre-training stage, fine-tuning allows the model to align its knowledge and capabilities with the specific objectives and data characteristics of the target task.

- **Achieving Task-Specific Expertise:** Fine-tuning can involve a range of tasks, from text classification and sentiment analysis to question-answering and language translation. This stage is critical for the model to achieve high performance on specialized tasks, leveraging its pre-trained knowledge base and adapting it to specific domain requirements.

Versions: There have been several iterations of GPT, each larger and more complex than the last:

- **GPT-1:**

  GPT-1 stands as the inaugural model in the Generative Pre-trained Transformer series. It set the stage for the subsequent advancements in transformer-based language models, demonstrating the efficacy of transformers for natural language processing tasks.

  Architecture: GPT-1's architecture is a transformer-based model, specifically utilizing a decoder-only structure. It comprises 12 layers of transformer blocks, encompassing a total of 117 million parameters.

  - **Transformer Blocks:** Each block contains masked self-attention layers and position-wise feedforward layers, with layer normalization applied before each sub-layer and a residual connection applied after each sub-layer.
  - **Masked Self-Attention:** This mechanism enables the model to weigh the importance of different words in a sequence, considering only the preceding words in the sequence for each position.
  - **Positional Encoding:** GPT-1 utilizes learned positional embeddings added to the word embeddings to maintain the order of the words in a sequence.

  Training: GPT-1 undergoes a two-stage training process:

  - **Unsupervised Pre-training:** The model is pre-trained on a large corpus of text data, learning to predict the next word in a sentence. This stage enables GPT-1 to capture a wide range of language patterns and general knowledge.
  - **Supervised Fine-tuning:** Following pre-training, GPT-1 can be fine-tuned on specific tasks using labeled data. During this stage, the model adapts to the peculiarities of the particular task.

  Capabilities and Impact: GPT-1 showcased impressive performance across various NLP benchmarks and tasks, including text generation, question answering, and text summarization.

  - The model demonstrated that transformers could generate coherent and contextually relevant text over extended passages.
  - GPT-1 underscored the importance of scale in transformer models, setting a precedent for larger models in the GPT series.
  - Despite its capabilities, GPT-1 also exhibited limitations, particularly in generating text that deviates from the training data or in handling more complex language nuances.

- **GPT-2:**

GPT-2 is the second model in the Generative Pre-trained Transformer series and is a significant advancement over GPT-1, particularly in terms of size, capabilities, and performance. While GPT-2 shares many architectural features with GPT-1, it introduces several key differences:

- **Larger Scale:** GPT-2 is substantially larger than GPT-1, boasting 1.5 billion parameters, which makes it one of the largest language models at the time of its release. This increase in scale contributes to its enhanced performance on a wide range of NLP tasks.
- **Multi-Head Attention:** GPT-2 utilizes multi-head self-attention mechanisms, allowing the model to capture more complex relationships between words and enhancing its ability to model context.
- **More Layers:** GPT-2 comprises 48 transformer layers, effectively doubling the depth of GPT-1. The additional layers enable the model to capture deeper and more intricate patterns in the data.
- **Improved Text Generation:** GPT-2 is known for its proficiency in generating coherent and contextually relevant text over longer passages. It excels in various tasks, including text completion, text generation, creative writing, and content generation. It has been employed in chatbot applications and more.
- **Zero-shot and Few-shot Learning:** GPT-2 exhibits the capability for zero-shot and few-shot learning, meaning it can perform tasks without explicit training on them. This showcases its remarkable ability to generalize from the pre-trained knowledge it has acquired.

Like GPT-1, GPT-2 undergoes a two-stage training process. It begins with unsupervised pre-training on a vast text corpus to learn language patterns and general knowledge. Subsequently, it can be fine-tuned on specific tasks with labeled data to adapt to the requirements of those tasks.

GPT-2's larger scale and increased capacity contribute to its superior performance and ability to generalize to a wider array of NLP tasks. It demonstrates the potential of large-scale transformer models in natural language processing and continues to influence the development of even larger models in the GPT series. GPT-2 builds upon the foundations of GPT-1, emphasizing the significance of model size and capacity in achieving remarkable performance in various NLP applications. It represents a milestone in the development of transformer-based language models and has played a crucial role in advancing the field of natural language processing.

- **GPT-3:** GPT-3, the third iteration of the Generative Pre-trained Transformer series, represents a substantial leap in terms of model size, capabilities, and versatility. While it retains the fundamental transformer architecture, GPT-3 introduces several groundbreaking features:

- **Unprecedented Scale:** GPT-3 is one of the largest language models in existence, with a staggering 175 billion parameters. This massive scale significantly enhances its performance on a wide range of natural language processing tasks.
- **Zero-shot and Few-shot Learning:** GPT-3 is renowned for its remarkable zero-shot and few-shot learning abilities. It can perform tasks without specific training on them, often with only a few examples or instructions. This demonstrates its extraordinary capacity to generalize from its pre-trained knowledge.
- **Multimodal Capabilities:** GPT-3 extends beyond text and demonstrates the capability to process and generate content across various modalities, including text, images, and even code. It excels in tasks like text completion, translation, summarization, image captioning, and more.
- **Few-shot Translation:** GPT-3 can perform translation tasks in multiple languages, showcasing its multilingual capabilities. It can translate languages with high accuracy even without prior training for a specific language pair.
- **Natural Language Understanding:** GPT-3 exhibits strong natural language understanding, making it adept at tasks such as question answering, sentiment analysis, and text classification.
- **Creative Text Generation:** GPT-3 is capable of creative text generation, generating coherent and contextually relevant text for creative writing, content generation, and storytelling.
- **Ethical and Controversial Use Cases:** GPT-3's capabilities have raised ethical concerns, particularly regarding the potential for misuse in generating deceptive or harmful content.

GPT-3, like its predecessors, follows a two-stage training process: unsupervised pre-training on a massive text corpus followed by supervised fine-tuning on specific tasks. However, its enormous scale and capacity result in unparalleled performance and flexibility. It has become a significant milestone in the development of transformer-based language models, pushing the boundaries of what is achievable in natural language processing.

- **GPT-4:** GPT-4, the fourth iteration of the Generative Pre-trained Transformer series, represents a further advancement in the realm of large-scale language models. While it retains the fundamental transformer architecture, GPT-4 introduces several notable features and improvements:

  - **Unprecedented Scale:** GPT-4 continues the trend of scaling up, boasting an even larger model size compared to its predecessors, often exceeding 1 trillion parameters. This colossal scale is a defining characteristic of GPT-4, contributing to its enhanced performance.

- **Enhanced Multimodal Capabilities:** Building on the multimodal capabilities of GPT-3, GPT-4 excels in processing and generating content across various modalities, including text, images, audio, and video. It can undertake tasks such as text-to-speech synthesis, image generation, and more.
- **Zero-shot and Few-shot Learning Excellence:** GPT-4 retains and refines its zero-shot and few-shot learning abilities, demonstrating impressive performance in tasks with minimal examples or instructions.
- **Improved Natural Language Understanding:** GPT-4 further enhances its natural language understanding capabilities, making it proficient in tasks like question answering, sentiment analysis, and text classification.
- **Code Generation and Understanding:** GPT-4 is proficient in generating and understanding code in various programming languages. It can assist in coding tasks, perform code translation, and provide explanations for code snippets.
- **Ethical and Responsible Use:** The ethical considerations around GPT-4 are even more pronounced, as its capabilities raise concerns about the potential for misuse, including generating deceptive or harmful content.

GPT-4's training process follows the same two-stage approach as its predecessors: unsupervised pre-training on a massive text corpus and supervised fine-tuning on specific tasks. However, the model's massive scale and capacity result in even more remarkable performance and adaptability.

GPT-4 builds upon the foundation laid by GPT-1, GPT-2, and GPT-3, emphasizing the pivotal role of model size and versatility in achieving groundbreaking results in natural language processing and beyond. Its extensive capabilities across modalities, natural language understanding, and code generation make it a prominent model in the AI landscape, while its ethical implications highlight the need for responsible and ethical deployment.

Feature	GPT-1	GPT-2	GPT-3	GPT-4
Release Year	2018	2019	2020	2023
Model Size	117M parameters	1.5B parameters	175B parameters	Multi-trillion parameters
Architecture	Transformer	Transformer	Transformer	Advanced Transformer
Number of Layers	12	48	96	More than 96
Type of Attention	Standard self-attention	Modified self-attention	Sparse attention	Advanced attention mechanisms
Training Data	BooksCorpus	WebText	Common Crawl, WebText, Books	Diverse and larger dataset
Fine-Tuning Capability	No	No	Yes	Yes
Language Understanding	Basic	Intermediate	Advanced	Highly advanced
Language Generation	Basic	Good	Excellent	State-of-the-art
Task Flexibility	Limited	Moderate	High	Extremely high
Context Length	Short	Longer than GPT-1	Up to 2048 tokens	Up to 40965 tokens
Tokenization Method	BPE	BPE	BPE	Advanced tokenization
Energy Efficiency	Standard	Less efficient	Less efficient	More efficient

5.1 Hallucination in Language Models and GPT Models

5.1.1 Definition and Overview

Hallucination in language models refers to the phenomenon where a model generates text that is either factually incorrect or not grounded in the input data. This issue is particularly prevalent in large-scale language models like GPT (Generative Pre-trained Transformer) models.

5.1.2 Causes in GPT Models

- **Overfitting:** When GPT models are trained on a specific dataset, they may overfit to patterns in the training data, leading to generation of text that aligns more with these patterns rather than with the real-world facts or the input context.
- **Lack of World Knowledge:** Despite their vast training datasets, GPT models might lack comprehensive world knowledge, leading to generation of plausible but incorrect information.
- **Bias in Training Data:** If the training data contains biases or inaccuracies, the model may replicate these in its outputs.

5.1.3 Detection and Mitigation

- **Detection:**
  - *Automated Fact-Checking:* Implementing systems that use algorithms to cross-reference generated text with trusted data sources, verifying factual content and detecting inconsistencies or inaccuracies.
  - *Output Analysis:* Utilizing advanced analytical tools and natural language processing techniques to assess the coherence, relevance, and factual accuracy of the generated text. This includes checking for logical fallacies or improbable statements that are out of context.
  - *Anomaly Detection:* Employing statistical methods to identify outliers or unusual patterns in the generated text that may indicate hallucinations or errors.
- **Mitigation:**
  - *Improved Training Data:* Curating training datasets to be more comprehensive, accurate, and free from biases. This may involve removing or correcting misleading information in the training data and ensuring a wide representation of factual knowledge.

- *Model Adjustments:* Modifying the model architecture or training algorithms to better handle ambiguities and uncertainties in the text. This could include techniques like regularization, which helps prevent overfitting to the training data.
- *Incorporating External Knowledge Bases:* Linking the model with external, up-to-date databases or knowledge graphs during the generation process. This allows the model to cross-check information in real-time and reduce the likelihood of generating false information.
- *Human Oversight:* Establishing a system where human experts review and correct the model's outputs, especially in critical applications. This not only helps in rectifying errors but also provides feedback for further improving the model.
- *User Feedback Loops:* Implementing mechanisms for users to report errors or hallucinations, which can be used to continuously train and refine the model.

Hallucinations in language models, particularly in GPT models, pose significant challenges for reliability and accuracy. Through robust detection mechanisms and targeted mitigation strategies, the integrity and utility of these models in various applications can be enhanced.

Chapter 6

Language Models

Language models are a cornerstone of natural language processing (NLP) and artificial intelligence (AI). They are computational models designed to understand, generate, and manipulate human language. Language models have played a pivotal role in various NLP applications, enabling machines to process and generate text-based content with remarkable accuracy.

Basic Concepts At its core, a language model's objective is to predict the next word or sequence of words in a given context. It operates based on conditional probability, estimating the likelihood of different words or phrases following a specific input. The more probable a word or phrase, the higher its predicted likelihood. This predictive ability is harnessed for a multitude of language-related tasks.

6.1 Statistical Language Models

Statistical Language Models (SLMs) : These models use statistical techniques to capture and predict the likelihood of word sequences based on observed frequencies in large text corpora.

SLMs operate on the principles of probability and statistics to model the structure of language. The fundamental concept is the conditional probability of a word or sequence of words given the preceding context. One of the most crucial concepts in language models is N-Grams:

6.1.1 N-gram Models

N-gram models are a popular class of SLMs. They estimate the probability of a word based on the previous N-1 words in the sequence. For example, in a bigram model, the probability of a word depends only on the preceding word. These models are computationally efficient but may have limitations in capturing long-range dependencies in language.

Applications SLMs have been used in a wide range of NLP applications, including:

- **Language Modeling:** They are used to model and generate text in tasks like text completion and speech recognition.
- **Machine Translation:** Early machine translation systems relied on SLMs to estimate translation probabilities.
- **Information Retrieval:** SLMs have been used in information retrieval systems to rank documents based on their relevance to a query.

Challenges While SLMs have been fundamental in NLP, they come with certain limitations. These challenges include:

- **Limited Context:** N-gram models have a limited context window and struggle to capture long-distance dependencies in language.
- **Data Sparsity:** SLMs require extensive training data to estimate probabilities accurately, and they may struggle with rare or unseen words and phrases.
- **Out-of-Vocabulary Words:** SLMs may encounter words that were not present in the training corpus, making it challenging to estimate their probabilities.

Example 6.1. *A bi-gram from scratch*

Given the text: "The cat sat on the mat. The cat chased the rat. The cat slept."

Tokenization and Bigram Counts Tokens and bigram counts based on the text:

$\langle s \rangle$ The : 3
The cat : 3
cat sat : 1
sat on : 1
on the : 1
the mat : 1
mat$\langle / s \rangle$: 1
cat chased : 1
chased the : 1
the rat : 1
rat$\langle / s \rangle$: 1
cat slept : 1
slept$\langle / s \rangle$: 1

Word Counts Counts of each word in the text:

$\langle s \rangle$: 3
The : 3
cat : 3
sat : 1
on : 1
the : 2
mat : 1
chased : 1
rat : 1
slept : 1

Bigram Probabilities Calculated probabilities for each bigram:

$$
\begin{aligned}
P(\text{The}|\langle s\rangle) &= \frac{3}{3} \\
P(\text{cat}|\text{The}) &= \frac{3}{3} \\
P(\text{sat}|\text{cat}) &= \frac{1}{3} \\
P(\text{on}|\text{sat}) &= \frac{1}{1} \\
P(\text{the}|\text{on}) &= \frac{1}{1} \\
P(\text{mat}|\text{the}) &= \frac{1}{2} \\
P(\langle/s\rangle|\text{mat}) &= \frac{1}{1} \\
P(\text{chased}|\text{cat}) &= \frac{1}{3} \\
P(\text{the}|\text{chased}) &= \frac{1}{1} \\
P(\text{rat}|\text{the}) &= \frac{1}{2} \\
P(\langle/s\rangle|\text{rat}) &= \frac{1}{1} \\
P(\text{slept}|\text{cat}) &= \frac{1}{3} \\
P(\langle/s\rangle|\text{slept}) &= \frac{1}{1}
\end{aligned}
$$

6.2 Neural Language Models

Neural Language Models (NLMs) represent a significant advancement in the field of natural language processing (NLP) and artificial intelligence (AI). These models are designed to understand, generate, and manipulate human language by leveraging the power of neural networks. NLMs have transformed the landscape of language-related tasks and applications.

At the core of NLMs lies the idea of using neural networks to process and generate text. Unlike traditional statistical language models that rely on counting and probabilities, NLMs employ deep learning techniques to capture intricate patterns and semantics in language. Key concepts include:

- Recurrent Neural Networks (RNNs): RNNs are a type of neural network architecture that can process sequences of data. They have been used in early NLMs to capture sequential dependencies in language.

- Long Short-Term Memory (LSTM) Networks: LSTMs are an improvement over RNNs and are capable of capturing long-range dependencies in text. They have been instrumental in enhancing the performance of NLMs.
- Transformers: Transformers, introduced in 2017, have revolutionized NLMs. They use self-attention mechanisms to capture context and dependencies efficiently, making them the foundation for state-of-the-art language models.

Architectures NLMs come in various architectures, with some of the prominent ones being:

- Encoder-Decoder Models: These models use separate encoder and decoder networks to process input and generate output. They are commonly used for tasks like machine translation and summarization.
- Pretrained Models: Pretrained models like BERT (Bidirectional Encoder Representations from Transformers), GPT (Generative Pre-trained Transformer), and RoBERTa are pretrained on large text corpora and fine-tuned for specific NLP tasks. They have achieved impressive results across a range of applications.
- Domain-Specific Models: Some NLMs are tailored for specific domains, such as medical or legal. These models are trained on domain-specific data and excel in domain-specific NLP tasks.

Training Language Models The training of language models typically involves large-scale datasets. In the case of neural language models, they are trained on massive text corpora, learning to predict the next word in a sentence or fill in missing words. This training phase equips them with a broad understanding of language patterns and semantics, allowing them to generalize across diverse tasks.

Challenges and Ethical Considerations Despite their capabilities, language models face challenges and ethical concerns. These include biases present in training data, responsible AI use, and concerns about the generation of harmful or inappropriate content. Addressing these challenges is vital for the responsible development and deployment of language models.

6.3 Large Language Models (LLMs)

Characterized by their substantial size, Large Language Models (LLMs) have been trained on extensive datasets, equipping them with a profound understanding of human language. This capability enables them to produce coherent and contextually relevant text across various tasks.

Scale and Complexity The architecture of LLMs involves millions or billions of parameters, making them some of the most extensive neural network models in AI. The size of these models allows them to capture complex language patterns, understand semantics, and store a vast amount of world knowledge, which is critical for performing a wide range of language understanding tasks.

$$\text{Model Size} \propto \text{Capability to Capture Language Nuances}$$

Training LLMs Training LLMs is a compute-intensive process that requires large-scale datasets and significant resources. Models such as BERT (Section **??**) and GPT (Section 5) utilize massive text corpora, learning to predict the probability of a word given its context or to identify the most appropriate word to complete a sentence. This pretraining phase enables them to generalize and adapt to a variety of NLP tasks effectively.

The formula for the pretraining loss function $\mathcal{L}_{\text{pretrain}}$ used in training LLMs such as BERT and GPT can be expressed as:

$$\mathcal{L}_{\text{pretrain}} = -\sum_{t=1}^{T} \log P(w_t | w_1, \ldots, w_{t-1}; \Theta)$$

where:

- $\mathcal{L}_{\text{pretrain}}$ is the pretraining loss that the model aims to minimize during the training process.
- T is the length of the text sequence that the model is being trained on.
- t is the position of the current word in the sequence for which the model is predicting the probability.
- w_t is the actual word at position t in the sequence.
- $w_1, \ldots, w_{t-1}$ represents the context, which is the sequence of words before the current word w_t.
- $P(w_t | w_1, \ldots, w_{t-1}; \Theta)$ is the probability model which estimates the likelihood of word w_t given its context and is parameterized by Θ, representing the model parameters.

The goal of this pretraining loss function is to train the model so that it can predict the probability of a word given its preceding context. This training process helps the model understand language patterns and structures, enabling it to perform a variety of NLP tasks more effectively after pretraining.

6.3.1 Applications of LLMs

LLMs have a broad spectrum of applications in NLP:

- **Text Generation:** LLMs are highly skilled at generating text that is not only coherent and contextually relevant but also diverse in style and content. They are particularly useful for creative writing, generating chatbot responses, and even for drafting simple reports or code. The ability to maintain a coherent thread in generated text makes them invaluable in these areas.

- **Language Translation:** LLMs excel in translating text between languages with high levels of fluency and accuracy, rivaling human translators in certain contexts. They do this by learning complex mappings between languages from large bilingual or multilingual corpora. These systems are now integral to global communication and information sharing.

- **Question Answering:** These models parse through large volumes of text to provide precise and contextually relevant answers to queries. They enhance search engines, power virtual assistants, and are increasingly used in customer service to provide quick and accurate responses to user inquiries.

- **Sentiment Analysis:** By understanding the nuances of language, LLMs can accurately determine the sentiment conveyed in text. This plays a crucial role in analyzing customer feedback, monitoring social media, and even in financial markets where sentiment analysis can predict market trends based on news articles or social media posts.

Chapter 7

Variational Autoencoders (VAEs)

Variational Autoencoders (VAEs) are a type of machine learning model used for generating complex data distributions. They consist of two parts: an encoder that compresses data into a lower-dimensional representation and a decoder that reconstructs the data from this compressed form. VAEs are unique in that they create a probabilistic representation of the data, learning to encode data as distributions rather than fixed points. This approach allows them to generate new, diverse data samples similar to the original dataset, making them useful for tasks like image generation and anomaly detection.

Before delving into the intricate details of VAEs, it is imperative first to comprehend the foundational concepts and components upon which they are built, namely, the variational lower bound and Autoencoders.

7.1 Preliminary concepts

7.1.1 Variational Lower Bound

This statistical construct also referred to as the evidence lower bound (ELBO), is pivotal in enabling the VAE framework to efficiently learn and approximate complex data distributions. The variational lower bound is a method employed in variational Bayesian statistics[1] That facilitates the approximation of complex models for which the true likelihood is computationally intractable. In essence, it provides a bound on the log-likelihood of observed data, which can be maximized in a more computationally feasible manner.

[1] Variational Bayesian statistics is a technique in machine learning that approximates complex distributions with simpler ones. It optimizes the parameters of these simpler distributions to closely match the true distribution, providing a practical way to perform inference on complex models. The "variational" aspect comes from the use of optimization techniques to 'vary' the parameters of the simpler distribution until the best approximation is found.

The concept can be expressed formally as follows:

$$\mathcal{L}(q) = \mathbb{E}_{Z\sim q(Z|X)}[\log p(Z, X)] + H(q(Z|X)) \tag{7.1}$$

Here, $\mathcal{L}(q)$ represents the variational lower bound for some observed data X, and $q(Z|X)$ denotes the variational distribution—a tractable approximation to the true posterior distribution $p(Z|X)$. The first term, $\mathbb{E}_{Z\sim q(Z|X)}[\log p(X|Z)]$ or the joint log-likelihood of the visible and hidden variables under the approximate posterior over the latent variables, is the expected log likelihood[2] of the data, indicating how well the model explains the observed data under the variational distribution[3]. The second term, $H(q(Z|X))$, is the entropy of the variational distribution, which quantifies the distribution's uncertainty or disorder[4].

The importance of the variational lower bound lies in its relationship to the marginal likelihood of the data:

$$\log p(X|Z) = \mathcal{L}(q) + D_{KL}(q(Z|X)\|p(Z)) \tag{7.2}$$

This equation reveals that maximizing the variational lower bound is equivalent to minimizing the Kullback-Leibler divergence between the variational distribution and the true posterior, thus enhancing the approximation q to more closely resemble $p(Z|X)$. This optimization forms the core of the learning algorithm in VAEs and is crucial for their ability to generate new data points that are coherent with the original dataset.

7.1.2 Autoencoders

Autoencoders are a class of neural network models aimed at learning representations (encodings) of input data. They are used primarily for tasks such as dimensionality reduction, feature learning, and unsupervised pretraining.

Architecture The architecture of an autoencoder is composed of two main parts:

1. **Encoder (Recognition network):**

   The encoder in an autoencoder plays a crucial role in transforming and compressing the input data into a latent-space representation. This process is fundamental to the autoencoder's ability to learn efficient codings

[2]Log likelihood is a measure used in statistics to quantify the fit of a statistical model to a sample of data. Specifically, it is the natural logarithm of the likelihood function, representing the joint probability of the observed data given a set of model parameters. By taking the logarithm, we transform this joint probability into a sum, which is advantageous for mathematical and computational reasons, particularly when dealing with maximum likelihood estimation.

[3]$\mathbb{E}_{Z\sim q(Z|X)}$ means that Z is a random variable whose distribution is given by $q(Z|X)$, where q is the approximate posterior distribution.

[4]This entropy component promotes a situation where the variational posterior assigns a high probability to a wide range of z values as potential generators of x, instead of converging on a singular point estimate as the most probable value.

of data. Below is a detailed breakdown of the encoder components and the encoding function:

- **Input** x: The input to the encoder, denoted as x, represents the raw data that needs to be encoded. This data can vary in form, ranging from images to numerical vectors.
- **Weights** W **and Biases** b: These are the learnable parameters of the neural network. The weights W influence the impact of each input feature on the encoded representation, while the biases b ensure that the neurons can output non-zero values even when all inputs are zero. The combination of W and b transforms the input data into a new representation.
- **Activation Function** σ: This non-linear function is applied to the linear transformation $Wx + b$. Common choices for the activation function σ include sigmoid, tanh, and ReLU functions. The introduction of non-linearity by σ is essential for the network to capture complex patterns in the data.
- **Encoding Function** $f(x)$: The encoding process is encapsulated in the function $f(x)$, which is defined as:

  $$f(x) = \sigma(Wx + b)$$

  Here, the input x undergoes a linear transformation followed by the application of the non-linear activation function σ, resulting in a compressed, latent representation of the input data. This encoding is designed to retain as much relevant information as possible while reducing the dimensionality of the data.

The encoder's objective is to distill the essence of the input data into a more compact form, shedding any redundant or irrelevant information. The efficiency of this encoding is vital, as it determines the effectiveness of the subsequent reconstruction of the original data by the decoder. The learning process involves fine-tuning the weights W and biases b to minimize the reconstruction error, leading to an optimal encoding of the data within the constraints of the network's architecture and capacity.

2. **Decoder (Generative network):** The decoder part of the network reconstructs the input data from the encoded representation. The decoding function is often the mirror image of the encoding function and can be represented as:

   $$g(f(x)) = \sigma'(W'f(x) + b')$$

   where σ', W', and b' are the activation function, weights, and biases for the decoder, respectively.

Variations

There are several variations of autoencoders, each with specific characteristics and use cases. These include:

- **Sparse Autoencoders:** Incorporate sparsity constraints on the hidden layers to learn more robust features.
- **Denoising Autoencoders:** Aim to reconstruct the input from a corrupted version, thus forcing the network to learn more robust features.
- **Variational Autoencoders:** Introduce a probabilistic approach, where the encoding represents the parameters of a probability distribution modeling the data.

7.2 Variational Autoencoders (VAEs)

Unlike Autoencoders, as we saw in the previous section, VAEs instead of learning a deterministic latent representation, model the latent space as a probability distribution. This probabilistic interpretation enables VAEs to generate new data points by sampling from the latent space.

In a VAE:

- The encoder maps input data to a probability distribution in the latent space, characterized by mean (μ) and standard deviation (σ).
- A sample is drawn from this distribution, which serves as a point in the latent space.
- The decoder reconstructs the data from this sample.

7.3 VAE Training and Loss Functions

Variational Autoencoders are trained using a combination of two loss terms: a reconstruction loss and a KL divergence. These terms ensure that the VAE not only accurately reconstructs the input data but also that the learned latent representations follow a desired distribution. The loss function $\mathcal{L}$ for a VAE is given by:

$$\mathcal{L}(\theta, \phi; x) = \mathbb{E}_{z \sim q_\phi(z|x)}[\log p_\theta(x|z)] - D_{KL}(q_\phi(z|x) \| p_\theta(z)) \tag{7.3}$$

where:

- θ are the parameters of the generative model (decoder), $p_\theta(x|z)$, which defines the probability of the data given the latent variables.
- ϕ are the parameters of the variational approximation (encoder), $q_\phi(z|x)$, which approximates the true posterior distribution $p(z|x)$.

- The first term, $\mathbb{E}_{z\sim q_\phi(z|x)}[\log p_\theta(x|z)]$, is the expected log likelihood of the data given the latent variables, representing the reconstruction loss. It measures how well the generative model can reconstruct the data from the latent variables.

- The second term, $D_{KL}(q_\phi(z|x)\|p_\theta(z))$, is the Kullback-Leibler divergence between the variational distribution and the prior distribution of the latent variables. It acts as a regularizer by enforcing the latent variables to adhere to a prior distribution, typically a standard Gaussian distribution.

7.3.1 Reconstruction Loss

The reconstruction loss measures the difference between the input data and its reconstructed version, produced by the VAE. This term is crucial to ensure that the VAE can accurately reconstruct the input data from its latent representation. For data like images, the Mean Squared Error (MSE) is often used:

$$\mathcal{L}_{\text{recon}} = \frac{1}{N}\sum_{i=1}^{N} ||x_i - \hat{x}_i||^2$$

Where:

- x_i is the original input data.

- $\hat{x}_i$ is the reconstructed data.

- N is the number of data points.

7.3.2 KL Divergence

The Kullback-Leibler (KL) divergence in Variational Autoencoders (VAEs) plays a crucial role in ensuring that the learned latent variable distributions are regularized towards a predefined prior distribution (typically a standard Gaussian distribution). This section provides an extended explanation of its role and the associated formula.

Regularization Effect

Minimizing the KL divergence in the VAE loss function ensures that the latent space has the desired properties, such as continuity and the ability to generate realistic samples.

Formula Breakdown

The KL divergence component of the VAE loss function is given by:

$$\mathcal{L}_{\text{KL}} = \frac{1}{2}\sum_{j=1}^{J}(1 + \log(\sigma_j^2) - \mu_j^2 - \sigma_j^2)$$

where μ_j and σ_j^2 are the mean and variance of the Gaussian distribution modeled by the encoder for each dimension j of the latent space. This expression can be dissected as follows:

- The term $\log(\sigma_j^2)$ encourages the variance to be neither too small nor too large.
- The terms μ_j^2 and σ_j^2 encourage the encoder distributions to center around zero and have a variance of one, aligning them with the standard Gaussian prior.

Intuition and Visualization

The KL divergence measures the overlap between two Gaussian distributions: the one defined by the VAE's encoder and the standard Gaussian distribution. A smaller KL divergence indicates a higher overlap, showing that the distribution learned by the VAE is closer to the standard Gaussian.

In conclusion, the KL divergence in VAEs serves as a regulatory mechanism, balancing the learned latent space distributions with the standard Gaussian prior, aiding in learning a generalized and well-structured latent space.

7.3.3 Total Loss

The total loss for VAEs is a combination of the reconstruction loss and the KL divergence:

$$\mathcal{L}_{\text{VAE}} = \mathcal{L}_{\text{recon}} + \mathcal{L}_{\text{KL}}$$

7.4 Applications of VAEs

Variational Autoencoders (VAEs) have found a wide range of applications in machine learning and generative modelling. Below are some of the key areas where VAEs are applied.

7.4.1 Image Generation

Variational Autoencoders (VAEs) are particularly adept at generating realistic images. This process involves sampling points from the latent space, which the VAE has structured in a way that captures the essence of the training data.

The ability to generate new images that share characteristics with the training set makes VAEs valuable in various applications.

Applications in Art and Media

VAEs are frequently used in creative fields to generate art, realistic faces, or other visual content. They can create diverse and complex images that are often indistinguishable from real-world photographs or artistic styles.

Mechanism of Image Generation

The image generation process in VAEs involves two main steps:

1. **Sampling from Latent Space:** Random points are sampled from the latent space, which contains compressed, abstract representations of data.
2. **Decoding to Generate Images:** The sampled points are fed into the decoder, which reconstructs them into images.

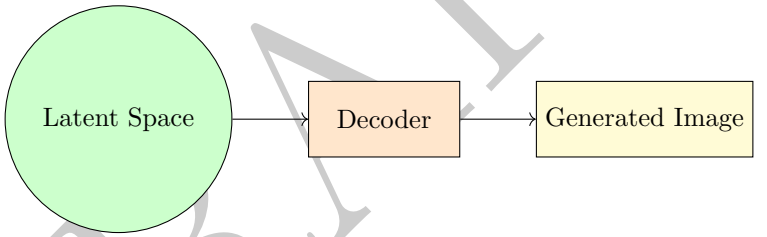


Figure 7.1: Image Generation using VAEs

7.4.2 Anomaly Detection

VAEs have found significant applications in anomaly detection. By learning the distribution of normal data, VAEs can identify data points that significantly deviate from this distribution as anomalies.

Use Cases

Anomaly detection using VAEs is particularly useful in:

- **Fraud Detection:** Identifying unusual patterns that might indicate fraudulent activities.
- **Industrial Quality Control:** Detecting defects or irregularities in manufactured products.

Mechanism of Anomaly Detection

The VAE is trained on a dataset representing 'normal' conditions. During inference, data points that the VAE struggles to reconstruct accurately are flagged as anomalies, as they do not conform to the learned normal data distribution.

7.4.3 Data Denoising

VAEs are effective for denoising data. They are trained on noisy datasets to reconstruct clean versions of that data.

Process of Data Denoising

The training involves presenting the VAE with noisy input data while setting the target output as clean or noise-free data. Over time, the VAE learns to filter out the noise and enhance the quality of the data.

7.4.4 Semi-supervised Learning

VAEs are valuable in semi-supervised learning scenarios, where the available dataset has a mix of labeled and unlabeled data.

Enhancing Classification with Limited Labels

In such settings, VAEs leverage the unlabeled data to better understand the data distribution, aiding in making more accurate predictions even with a limited amount of labeled data.

7.4.5 Interpolation and Morphing

One of the intriguing capabilities of VAEs is their ability to perform smooth interpolation in the latent space, leading to applications in morphing.

Creating Morphing Effects

By interpolating between latent space representations of different data points (such as images), VAEs can create a seamless transition or morphing effect from one image to another. This technique is often used in graphics and animation to generate intermediate frames or transform one image into another smoothly.

Chapter 8

Generative Adversarial Networks (GANs)

Generative Adversarial Networks (GANs) were introduced by Ian Goodfellow and his colleagues in 2014[1]. GANs are used for generating synthetic data that is nearly indistinguishable from real data.

A GAN consists of two main components:

- **Generator (G):** This network takes a random noise vector as input and generates data (like images).
- **Discriminator (D):** This network takes real and fake data as input and attempts to distinguish between them.

8.0.1 Adversarial Training

The training of a GAN involves a min-max game between the Generator and the Discriminator:

$$\min_G \max_D V(D,G) = \mathbb{E}_{\mathbf{x}\sim p_{\text{data}}}[\log D(\mathbf{x})] + \mathbb{E}_{\mathbf{x}\sim p_{model}}[\log(1 - D(X)]$$

The Discriminator's goal is to maximize the probability of correctly classifying both real and fake data. Conversely, the Generator's goal is to produce data that is indistinguishable from real data, thereby minimizing the probability that the Discriminator makes a correct classification.

- $\mathbb{E}_{\mathbf{x}\sim p_{\text{data}}}[\log D(\mathbf{x})]$: The expected log probability that the discriminator correctly identifies real data as real. The expectation is taken over the true data distribution p_{data}. The discriminator aims to maximize this term, to be as accurate as possible on real data.

[1] Goodfellow, Ian, et al. "Generative adversarial nets." Advances in neural information processing systems 27 (2014).

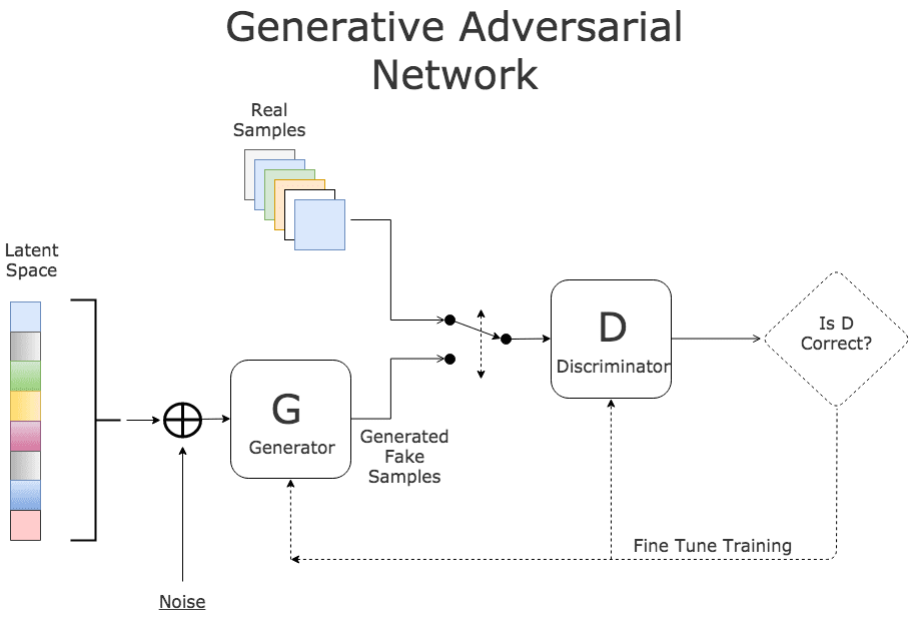


Figure 8.1: An example of a GAN

- $\mathbb{E}_{\mathbf{x}\sim p_{model}}[\log(1 - D(X)]$: The expected log probability that the discriminator correctly identifies fake data (produced by the generator) as fake. This is where z is a noise vector sampled from some probability distribution p_z, which the generator uses to produce data. The generator's aim is to minimize this term by fooling the discriminator into thinking the fake data is real.

In a Generative Adversarial Network (GAN), the feedback from the discriminator to the generator is provided indirectly through the process of backpropagation. Here is an overview of how this mechanism works:

1. **Discriminator's Output:** The discriminator evaluates the data generated by the generator and outputs a probability score, indicating the likelihood of the data being real (from the training dataset) or fake (generated).

2. **Loss Calculation:** Based on the discriminator's output, a loss function is calculated. This loss reflects the performance of the generator – specifically, how convincing its generated data is.

3. **Backpropagation:** The loss is backpropagated through the network, and gradients are computed. These gradients, which are partial derivatives of the loss with respect to the generator's parameters, provide information on how to adjust these parameters to improve performance.

4. **Generator's Update:** The generator updates its parameters (weights and biases) using these gradients, typically employing an optimization algorithm like Stochastic Gradient Descent (SGD) or Adam. The updates aim to reduce the loss, meaning making the generated data more convincingly real.

5. **Iterative Process:** This process is repeated iteratively. Over many iterations, the generator gradually becomes better at producing data that closely resembles the real data, as judged by its increasing ability to 'fool' the discriminator.

It is important to note that the generator does not directly see the real data or the discriminator's exact outputs. It only receives gradients, which act as a statistical signal on how to adjust its outputs to be more realistic.

Example 8.1. *A Good and a Bad Discriminator!*

Case 1: Good Discriminator A good discriminator correctly identifies real and fake samples.

- For a real sample $\mathbf{x_{real}}$: $D(\mathbf{x_{real}}) = 0.95$
- For a fake sample $\mathbf{x_{fake}}$: $D(\mathbf{x_{fake}}) = 0.05$
- The losses are:

$$\log D(\mathbf{x_{real}}) = \log 0.95 \approx -0.05$$
$$\log(1 - D(\mathbf{x_{fake}})) = \log(1 - 0.05) \approx -0.05$$

$$\begin{aligned} \text{Total Loss} &= \log D(\mathbf{x_{real}}) + \log(1 - D(\mathbf{x_{fake}})) \\ &= \log 0.95 + \log(1 - 0.05) \\ &\approx -0.05 - 0.05 \\ &\approx -0.10 \end{aligned}$$

Case 2: Poor Discriminator A poor discriminator struggles to differentiate between real and fake samples.

- For both real and fake samples ($\mathbf{x_{real}}, \mathbf{x_{fake}}$): $D(\mathbf{x}) = 0.5$
- The losses are:

$$\log D(\mathbf{x_{real}}) = \log 0.5 = -\ln(2) \approx -0.69$$
$$\log(1 - D(\mathbf{x_{fake}})) = \log(1 - 0.5) = -\ln(2) \approx -0.69$$

$$
\begin{aligned}
\text{Total Loss} &= \log D(\mathbf{x}_{\mathbf{real}}) + \log(1 - D(\mathbf{x}_{\mathbf{fake}})) \\
&= \log 0.5 + \log(1 - 0.5) \\
&= -\ln(2) - \ln(2) \\
&\approx -0.69 - 0.69 \\
&\approx -1.38
\end{aligned}
$$

8.0.2 Training GANs

Training GANs is notoriously challenging. The following strategies are often used to stabilize training:

- **Feature Matching:** This technique aims to address the instability in training by having the Generator create data that matches the statistics of the real data at some intermediate layer of the Discriminator. This can be achieved by minimizing the difference between the expected value of features from the real data and the generated data. The feature matching objective for the Generator can be written as:

$$\min_G \|\mathbb{E}_{\mathbf{x}\sim p_{\text{data}}} f(\mathbf{x}) - \mathbb{E}_{\mathbf{z}\sim p_{\mathbf{z}}} f(G(\mathbf{z}))\|^2$$

  where $f(\mathbf{x})$ represents the features of the real data $\mathbf{x}$ on an intermediate layer of the Discriminator, and $G(\mathbf{z})$ are the features of the generated data.

- **Mini-Batch Discrimination**: This technique allows the Discriminator to look at multiple data points simultaneously to make a decision, which helps in identifying mode collapse, where the Generator produces limited varieties of samples. A batch might consist of a mix of real and fake data samples. The ratio of real to fake data in each batch can vary depending on the specific implementation and training strategy.

- **Experience Replay**: Also known as replay buffer, this involves storing generated samples from previous iterations and then randomly inserting these past generated samples into current mini-batches. This helps in maintaining diversity in the generated samples and prevents the Generator from overfitting to the current Discriminator.

8.0.3 Applications of GANs

Image Generation: GANs can be trained to produce images that mimic a given dataset, generating new data points sampled from the learned data distribution.

Super-Resolution: Super-resolution GANs aim to transform low-resolution inputs into high-resolution outputs by learning a mapping function that captures the high-frequency details of the training images.

Style Transfer: Style transfer with GANs involves modifying the style of a source image to match the style of a target image, while preserving the source content.

Image-to-Image Translation: This task requires translating images from one domain to another (e.g., day to night, sketch to photograph) by learning a domain transformation function.

Data Augmentation: GANs can augment a dataset by generating new, synthetic examples that expand the variability and size of the training set.

In these applications, the GAN loss function is often extended or combined with other loss functions to achieve the desired results. For instance, in super-resolution and style transfer, additional loss terms that measure content fidelity or style adherence may be added to the GAN loss. The generalized GAN loss function is given by:

$$\min_G \max_D V(D, G) = \mathbb{E}_{\mathbf{x} \sim p_{\text{data}}}[\log D(\mathbf{x})] + \mathbb{E}_{\mathbf{z} \sim p_{\mathbf{z}}}[\log(1 - D(G(\mathbf{z})))]$$

For super-resolution, an example of a combined loss function might be:

$$\mathcal{L}_{\text{SR}}(G) = \mathcal{L}_{\text{GAN}}(G, D) + \lambda \cdot \mathcal{L}_{\text{MSE}}(G(\mathbf{z}), \mathbf{x}_{\text{HR}})$$

Where $\mathcal{L}_{\text{MSE}}$ is the mean squared error loss comparing the high-resolution output of the generator with the true high-resolution image, and λ is a parameter that controls the relative importance of the MSE loss to the adversarial loss.

For style transfer, the loss function may look like:

$$\mathcal{L}_{\text{style}}(G) = \alpha \cdot \mathcal{L}_{\text{content}}(G(\mathbf{z}), \mathbf{x}_{\text{content}}) + \beta \cdot \mathcal{L}_{\text{style}}(G(\mathbf{z}), \mathbf{x}_{\text{style}})$$

Here, $\mathcal{L}_{\text{content}}$ and $\mathcal{L}_{\text{style}}$ represent content and style losses, respectively, and α and β are the weights that balance the two types of loss.

8.1 Variants of GANs

Generative Adversarial Networks (GANs) have undergone significant evolution and diversification since their inception. In this section we focus on Deep Convolutional GANs (DCGAN) and Conditional GANs (CGAN), providing a brief intoduction into their architectures and applications.

8.1.1 Deep Convolutional GANs (DCGAN)

Deep Convolutional GANs (DCGAN) are a variant of GANs that leverage deep convolutional neural networks for both the Generator and Discriminator. DCGANs introduce architectural guidelines to improve the stability of GAN training and the quality of generated images.

DCGANs follow several architectural guidelines:

- Use Convolutional Layers: Replace fully connected layers with convolutional layers to better capture spatial hierarchies in images.
- Use Batch Normalization: Apply batch normalization to stabilize training and help in faster convergence.
- Eliminate Fully Connected Layers: Fully connected layers are removed to encourage feature learning in a spatial hierarchy, increasing input variability.
- Use Specific Activation Functions: Leaky ReLU is preferred for its ability to address the 'dying ReLU' problem and maintain gradients during backpropagation.
- Generator Output: Use Tanh activation in the Generator's output layer to model the normalized range of image data.
- Discriminator Output: Use Sigmoid activation in the Discriminator's output layer for binary classification of real vs. fake images.

Improvements Over Traditional GANs: DCGANs offer several improvements over the traditional GAN architecture:

- **Improved Stability:** The architectural choices in DCGANs, like batch normalization and Leaky ReLU, contribute to more stable training of GANs, reducing issues like mode collapse.
- **Higher Quality Images:** The use of convolutional layers enables DCGANs to generate higher quality images with more detailed features, suitable for tasks requiring finer visual fidelity.
- **Feature Learning:** DCGANs have shown an ability to learn a hierarchy of representations from object parts to scenes, making them useful for unsupervised feature learning.

– **Applicability:** DCGANs have been successfully applied in various domains, including image synthesis, super-resolution, and style transfer, showcasing their versatility.

Overall, DCGANs represent a significant step forward in the field of generative models, particularly in their ability to generate high-quality images and learn useful feature representations in an unsupervised manner.

8.1.2 Conditional GANs (CGAN)

Conditional Generative Adversarial Networks (CGANs) extend the traditional Generative Adversarial Network (GAN) framework by allowing the generation of data conditioned on some auxiliary information. This can be formulated as follows:

– The Generator and Discriminator in CGANs are conditioned on additional information y, such as class labels or other relevant data.

The objective function for CGANs becomes:

$$\min_G \max_D V(D,G) = \mathbb{E}_{x\sim p_{\text{data}}(x)}[\log D(x|y)] + \mathbb{E}_{z\sim p_z(z)}[\log(1-D(G(z|y)))] \tag{8.1}$$

Here, $G(z|y)$ represents the generator producing data conditioned on y, and $D(x|y)$ represents the discriminator evaluating data conditioned on y.

Some of the main applications of CGANs include:

Image-to-Image Translation: CGANs can be utilized for tasks like converting satellite images to maps, or black and white photos to color. The conditioning information in this case is the input image in one style, and the task is to generate an image in another style.

Super-Resolution: In super-resolution, CGANs are used to generate high-resolution images from low-resolution inputs. The conditioning information is the low-resolution image, and the network learns to add details and textures to upscale the image effectively.

Text-to-Image Synthesis: CGANs can generate realistic images from textual descriptions. Here, the text provides the conditioning, and the network learns to interpret the description to generate a corresponding image.

Photo Editing: CGANs can be applied to interactive photo editing tasks, such as modifying specific attributes of a face in a portrait (e.g., changing hair color, adding glasses). The conditioning input in this scenario would be the editing instructions or attributes.

Data Augmentation: Similar to traditional GANs, CGANs can also be used for data augmentation. However, the conditioning allows for more controlled augmentation, generating data samples with desired characteristics or features.

Medical Image Analysis: In medical imaging, CGANs can be used to generate synthetic medical images for training machine learning models, conditioned on different disease markers or patient characteristics.

In CGANs, the loss function typically incorporates the conditioning information to ensure that the generated data not only appears realistic but also aligns with the conditional input. This characteristic makes CGANs particularly useful in applications where the output needs to be tailored or specific to certain input parameters or criteria.