Exploring the Power of Generative Architectures such as GANs, Transformers, and VQGAN+CLIP through the Construction of an Illustrated Storybook Generator

Independent Study Thesis

Presented in Partial Fulfillment of the Requirements for
the Degree in Computer Science and Mathematics in the
Mathematical and Computational Sciences Department at
The College of Wooster

by
Ussama Mustafa
The College of Wooster
2023

Advised by:

Khowshik Bhowmik (Computer Science)

and Subhadip Chowdhury (Mathematics)

THE COLLEGE OF
WOOSTER

Abstract

Generative machine learning models have achieved unprecedented feats in recent years and look primed to reach even more impressive heights. By learning data distributions through unsupervised training and by leveraging the power of neural networks, these models are responsible for breakthroughs in various domains. The aim of this paper is to cover some of the prominent generative model architectures through the bottom-up construction of an illustrated storybook generating interface that uses transfer learning on a transformer-based text generator, and the Vector Quantized Generative Adversarial network (VQGAN) coupled with Contrastive Language–Image Pre-training (CLIP) for prompt-driven image generation.

This work is dedicated to the future generations of Wooster students.

Acknowledgments

I would like to sincerely thank my two advisors, Professor Chowdhury and Professor Bhowmik, for their insightful advice and consistent guidance. This paper owes its completion to their invaluable contributions.

Contents

List of Figures

CHAPTER 1

Introduction

Among the various advancements made in the field of artificial intelligence (AI) in recent years, none are more impressive than what generative models have been able to accomplish. Through the power of neural networks [9], these models are capable of generating realistic images, text, and even entire videos.

Architectures such as generative adversarial networks (GAN) [6] and transformers [21] have cemented themselves are building blocks for a series of high performance generative architectures. These architectures have paved the way for popular models such as DALL-E, which can generate images inspired from highly imaginative prompts, or even ChatGPT which has garnered a huge amount of attention through its incredible conversational capacities and seemingly limitless database of knowledge.

In this research, a large amount of focus will be dedicated to the VQGAN+CLIP [3] model which combines two existing models, the Vector Quantized Generative Adversarial Network (VQGAN) [5] and Contrastive Language-Image Pre-Training (CLIP) [14], to generate high resolution images from a given text prompt. Through its unique and ingenuous combination of a variety of cutting-edge architectures, this model is viewed by many as the culmination of text-to-image synthesis.

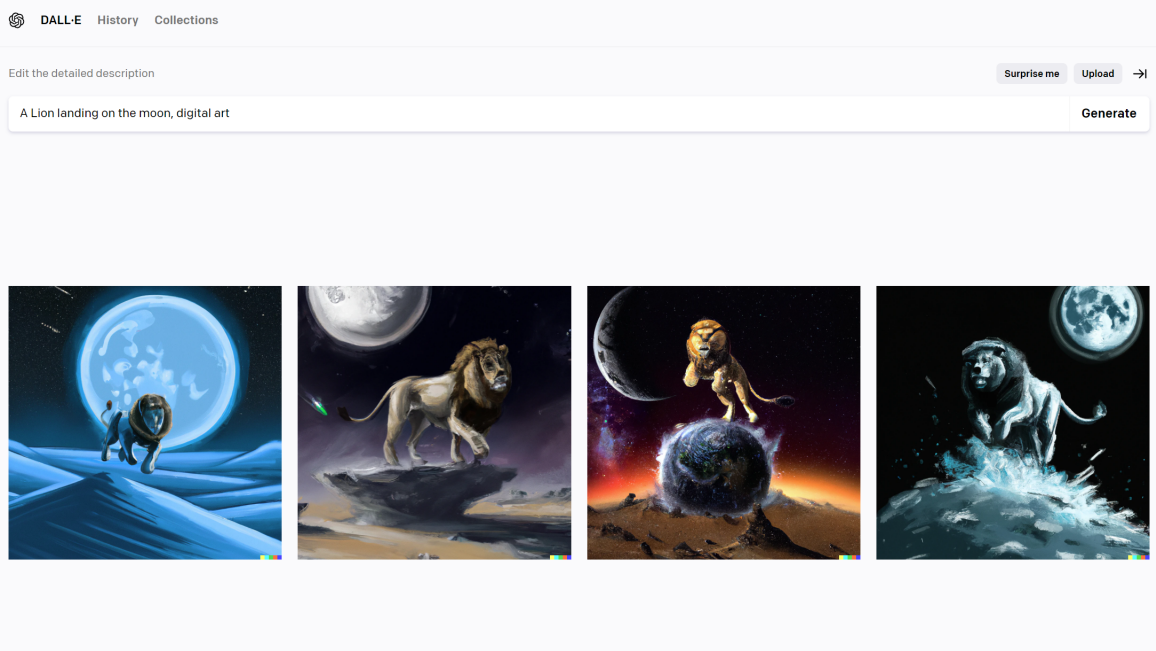

Figure 1.1: Example of an image generated by DALL-E

1.1 History and Background

Before providing a more detailed overview of these generative models, it is important to cover the background of the various concepts which they are based on, starting with machine learning.

Machine learning (ML) is a subfield of AI that involves the construction of algorithms and models that can learn from given data and make predictions on other sets of data by following training algorithms.

Early developments in ML began in the 1950s and focused on the development of primitive algorithms that could make simple predictions on unseen data after learning from training data. These algorithms were optimized using what is referred to as supervised learning, which is the process of learning from labeled training data. Nowadays, unsupervised learning is much more common as most of the data used is unlabeled.

One of the earliest and most well-known ML algorithms is the perceptron.

Developed by Frank Roseblatt, this model is a type of single-layer neural network that can learn to classify data by adjusting the weights of its inputs. The perceptron represented important progress in ML as it was a very generalizable model that laid the foundations for the development of more complex neural network architectures.

In the 1980s and 1990s, there was a resurgence of interest in ML and AI, driven in part by the availability of larger datasets and the development of more powerful computers. This period saw the development of many important ML algorithms, such as decision trees, support vector machines (SVMs), and other more sophisticated neural networks architectures.

Years later, the rise of generative models gave way to GANs [6], a type of ML architecture that can be used to generate new, synthetic data that is similar to a the samples from a training dataset. GANs consist of two networks: a generator and a discriminator. The two networks are trained in an adversarial process, with the generator aiming to produce reconstructed data from scratch that the discriminator cannot distinguish from the training data, and the discriminator trying to accurately distinguish reconstructed data from the original training data.

In 2017, the transformer architecture [21] was introduced as a way to improve the performance of neural networks on natural language processing (NLP) tasks such as language translation. Transformers use the self-attention mechanisms, which allows the model to focus on different parts of the input data at different times, rather than processing the input sequentially. This allows the model to make use of the global context of the input data, which can be particularly useful for tasks such as translation, where the meaning of a word can depend on the context in which it appears.

Since their introduction, there has been active research on improving and extending the transformer architecture, including the development of variant architectures such as the Transformer-XL and the GPT-3.

As image generation became a bigger topic in the ML scene, An attempt to improve the performance of generative models was introduced through a technique called vector quantization. While his technique had been theorized years prior, with early work in the 1990s using k-means clustering to discretize continuous data, its recent application in models such as VQVAE [12] is what really popularized it.

VQGAN [5] further builds on the concept of vector quantization by combining VQVAE with a GAN, before then introducing a transformer to learn codebook sequences to create high-resolution images. This model's combination of various state of the art architectures gained significant attention in the ML community and has led to an even more impressive implementation in text-to-art synthesis by pairing it with CLIP [14]. Covering the VQGAN+CLIP [3] architecture will be one of the main focuses of this paper.

1.2 Motivation

With the emergence of transformers, GANs, or even VQGAN+CLIP, it is clear that the field of ML has seen remarkable progress in recent years. However, despite the successes of these models, there is still much to be understood about the underlying mathematics and theoretical foundations that dictate how they function. By delving deeper into the mathematics behind these architectures, we can gain a more nuanced understanding of their strengths and limitations, as well as explore avenues for future research and applications.

This is precisely one of the driving motivations behind this paper. By undertaking

a mathematical exploration of these generative architectures and proposing an implementation through the construction of an illustrated storybook generator, we seek to gain a more comprehensive understanding of how they are built, what makes them operate effectively, and how they might be improved upon in the future.

Furthermore, mathematics is commonly considered an exact science, but its role in generating content that is often considered abstract highlights its versatility and power. As someone with a vested interest in arts and an academic background in mathematics, this dichotomy between the precise nature of mathematics and the fluid nature of the artistic content it can help create was a very interesting aspect to explore.

1.3 Objectives and Method

In this paper, the methods behind these generative architectures will be explored through the construction of an illustrated storybook generator. The strengths and limitations of the discussed models will be covered and will guide the implementation.

To begin, the mathematics behind foundational ML concepts and the inner workings of the neural network architecture will be explained, including forward and back propagation as well as the universal approximation theorem.

Following that, more advanced neural network variants such as recurrent neural networks (RNNs), long short-term memory networks (LSTMs) and convolutional neural networks (CNNs) will be covered.

Then, we will delve into an overview of the GAN and transformer architectures before providing primers on VQGAN and CLIP, and their union as a text-to=image synthetizer.

Finally, the explored models will be used to provide an implementation of an interface that allows users to generate illustrated storybooks by using transformers, VQGAN+CLIP, and helper libraries such as Hugging Face [22] and Gradio [1].

CHAPTER 2

Machine Learning

Machine learning (ML), at its core, is a field of study about understanding and constructing systems that are capable of learning patterns through training in order to accomplish a single or a set of tasks. It is considered a branch of artificial intelligence (AI). ML models are built using algorithms which use sample data, referred to as the training data in order to make predictions with or without the need for supervision.

Whenever we are faced with a task of a level of complexity entailing that we can't use a common algorithm on it, ML can be a very powerful tool for building models that accomplish the task at hand through high performance algorithms.

Deep learning (DL) [2] is a subset of ML which involves training models to accomplish human tasks. It is more specifically concerned with the development of algorithmic structures called neural networks [9] that are inspired by the structure and function of the brain. These neural networks are made up of layers of interconnected nodes, called neurons, which are designed to process and analyze large amounts of complex data.

DL models are much more complex than common ML models such as the preceptron, and are usually trained on extremely large datasets. Common examples

of deep learning networks include chatbots, facial recognition systems or any content generation model.

2.1 Neural Networks

ML is an area of study with incredible depth both in its implementations and applications. The types and numbers of components which can be found in an ML architectures are vast, however, one core component that persists in every model is the neural network.

A neural network is a network composed of an input layer and an output layer, with a series of hidden layers in between. Each layer is composed of a set of neurons which are linked layer to layer by a set of weights that influence the output from one layer to the other.

In high-level terms, neural networks are algorithms which compute, from an input x, an output y. The output usually being in the form of a set of probabilities.

While neural networks can have any positive number of layers and neurons for each layer, in order to explicitly detail the computations which take place in the learning process of a neural network, we will use the simplified example of a neural network with one input layer, one hidden layer, and one output layer, as seen in fig. 2.1.

This neural network takes in 2 inputs x_1 and x_2. The hidden layer has 3 neurons with the set of weights $(w_{11}, w_{12}, w_{13}, w_{21}, w_{22}, w_{23}, w_{31}, w_{41}, w_{51})$. If the network is not pre-trained, the weights are usually initialized randomly. The network also has bias parameters b_1 and b_2 which are constants used to adjust the output of a layer.

Once the network receives 2 inputs, and all weights and bias parameters have been initialized, the forward propagation process begins.

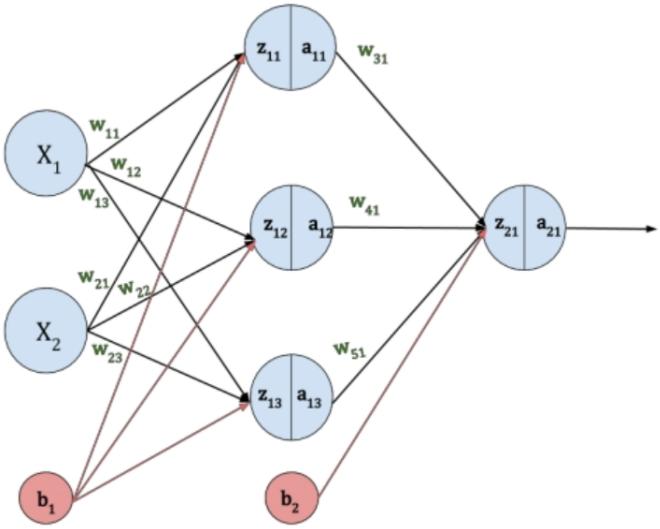

Figure 2.1: Neural Network with 2 Inputs, 3 Hidden Neurons and 1 Output

2.1.1 Forward Propagation

Forward propagation is the process through which the output of a layer is fed to the next layer until the final output layer is reached.

In the neural network used in fig. 2.1, the values z_{11}, z_{12}, z_{13} and z_{21} are intermediate neuron values calculated by multiplying the neuron values of previous layers by their associated weight value and adding the bias value.

$$z_{11} = x_1 w_{11} + x_2 w_{21} + b_1$$

$$z_{12} = x_1 w_{12} + x_2 w_{22} + b_1$$

$$z_{12} = x_1 w_{13} + x_2 w_{23} + b_1$$

This series of computations is often written in a vectorized form:

$$\begin{bmatrix} z_{11} \\ z_{12} \\ z_{13} \end{bmatrix} = \begin{bmatrix} w_{11} & w_{21} \\ w_{12} & w_{22} \\ w_{13} & w_{23} \end{bmatrix} \begin{bmatrix} x_1 \\ X_2 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_1 \\ b_1 \end{bmatrix}$$

$$\iff Z^1 = W^1 X + B^1$$

The calculations done in order to obtain these intermediate neuron values are very simple, which is not ideal when trying to learn non-linear patterns between inputs and outputs. For that reason, the intermediate neuron values are passed into an activation function, usually denoted with σ, to obtain the the values a_{11}, a_{12}, a_{13} and a_{21}. Different activation functions are used for different neural networks, among the most popular are the sigmoid function, the hyperbolic tangent function and the rectified linear unit (ReLU) function.

In this example, the activation function used is the sigmoid function, defined as:

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

The activation is then applied to $Z^1 = W^1 X + B^1$:

$$a_{11} = \sigma(z_{11})$$

$$a_{12} = \sigma(z_{12})$$

$$a_{13} = \sigma(z_{13})$$

$$A^1 = \sigma(Z^1)$$

Finally, the output of each hidden layer is used to compute the final output.

$$\begin{aligned} & z_{21} = a_{11}w_{31} + a_{12}w_{41} + a_{13}w_{51} + b_2 \\ \iff & Z^2 = W^2A^1 + B^2 \\ & a_{21} = \sigma(z_{21}) \\ \iff & A^2 = \sigma(Z^2) \end{aligned}$$

Following the forward propagation step, the output obtained from the neural network is compared to an expected output to gauge how accurate the algorithm was. The error between the output and the expected result is computed using a loss function.

2.1.2 Loss function

The loss function is a core component of the training process of a neural network. Suppose we have an output A^2 and an expected value y. To see how far off the obtained output was from the expected output, we can compute the squared Euclidean distance between the two.

$$\mathcal{L} = (y' - A^2)^2$$

When computing the loss value, the use of a quadratic function is common. A reason for that is that it is often more mathematically tractable than other loss functions because of the properties of variances, as well as being symmetric given that an error above y' causes the same loss as the same magnitude of error below y'.

Once the loss is computed, the weights and bias values in the neural network must be updated so as to minimize the error. This is the back propagation step.

2.1.3 Back Propagation

Back propagation is the process of propagating the computed loss back across the neural network in order to update the weights and biases of each neuron, so as to minimize the loss in the next output. This process is a key step in the training of neural networks and entails a series of mathematical operations aimed at learning the patterns between input and output variables.

Since the output of the neural network depends on the weight and bias values, the goal of this process is to compute the direction through which modifying the bias and weight parameters will reduce the error the most. To do so, we compute the rate of change in error by computing the partial derivatives of the loss function with respect to the weights and bias values. This derivation process determines how sensitive the loss function is to modifications in each weight and bias and is refereed to as the gradient descent optimization method.

Since we are taking partial derivatives of the loss, which depends on the output that is obtained through giving the weights and bias values to the sigmoid function, we need to compute the derivative of the sigmoid function.

$$
\begin{aligned}
\sigma(x) &= \frac{1}{1+e^{-x}} \\
\sigma' &= (1+e^{-x})^{-2}e^{-x} \\
&= \frac{1}{1+e^{-x}} s \frac{1+e^{-x}-1}{1+e^{-x}} \\
&= \frac{1}{1+e^{-x}}\left(1-\frac{1}{1+e^{-x}}\right) \\
&= \sigma(x)(1-\sigma(x)
\end{aligned}
$$

The ability to express the derivative of σ as a function of itself is very useful when it comes to the gradient descent optimization.

In figure fig. 2.2, we can see the back propagation steps for the neural network in fig. 2.1 colored in red.

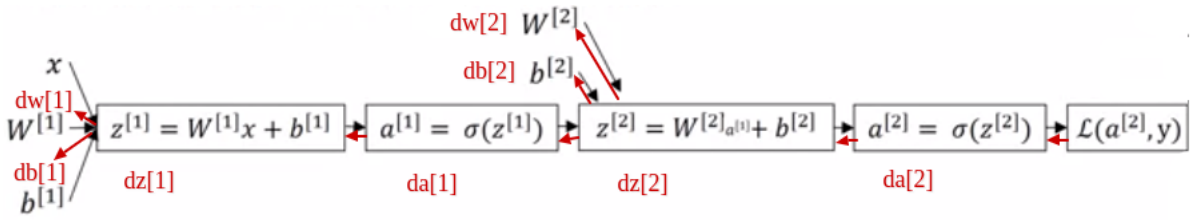

Figure 2.2: Back Propagation Process

We now begin the back propagation process by taking the partial derivatives starting from the $da[2]$ step.

$$
\begin{aligned}
da[2] &= \frac{\partial \mathcal{L}}{\partial A^2} = \frac{\partial (y - A^2)^2}{\partial A^2} \\
dz[2] &= \frac{\partial \mathcal{L}}{\partial Z^2} = \frac{\partial \mathcal{L}}{\partial A^2}\frac{\partial A^2}{\partial Z^2} = \frac{\partial (y - A^2)^2}{\partial A^2}\frac{\partial \sigma(Z^2)}{\partial Z^2} \\
dw[2] &= \frac{\partial \mathcal{L}}{\partial W^2} = \frac{\partial \mathcal{L}}{\partial Z^2}\frac{\partial Z^2}{\partial W^2} = dz[2]\frac{\partial}{\partial W^2}W^2A^1 + B^2 = dz[2]A^1 \\
db[2] &= \frac{\partial \mathcal{L}}{\partial B^2} = \frac{\partial \mathcal{L}}{\partial Z^2}\frac{\partial Z^2}{\partial B^2} = dz[2]\frac{\partial}{\partial B^2}W^2A^1 + B^2 = dz[2] \\
da[1] &= \frac{\partial \mathcal{L}}{\partial A^1} = \frac{\partial \mathcal{L}}{\partial Z^2}\frac{\partial Z^2}{\partial A^1} = dz[2]\frac{\partial}{\partial A^1}W^2A^1 + B^2 = dz[2]W^2 \\
.dz[1] &= \frac{\partial \mathcal{L}}{\partial Z^1} = \frac{\partial \mathcal{L}}{\partial A^1}\frac{\partial A^1}{\partial Z^1} = da[1]\frac{\partial \sigma(Z^1}{\partial Z^1} = da[1]\sigma(Z^1)(1 - \sigma(Z^1)) \\
dw[1] &= \frac{\partial \mathcal{L}}{\partial W^1} = \frac{\partial \mathcal{L}}{\partial Z^1}\frac{\partial Z^1}{\partial W^1} = dz[1]\frac{\partial}{\partial W^2}W^1X + B^1 = dz[1]X \\
db[1] &= \frac{\partial \mathcal{L}}{\partial B^1} = \frac{\partial \mathcal{L}}{\partial Z^1}\frac{\partial Z^1}{\partial B^1} = dz[1]\frac{\partial}{\partial W^2}W^1X + B^1 = dz[1]
\end{aligned}
$$

Now that the gradient descent has been computed, the next step is to update the weights. Before updating weights, a new parameter must be introduced: the

learning rate α, which is a value between 0 and 1 and controls how drastically we want to change the previous weight value. This gives us:

$$W^1_{new} = W^1 - \alpha \frac{\partial \mathcal{L}}{\partial W^1} = W^1 - \alpha dz[1]X$$
$$W^2_{new} = W^2 - \alpha \frac{\partial \mathcal{L}}{\partial W^2} = W^2 - \alpha dz[2]A^1$$
$$B^1_{new} = B^1 - \alpha \frac{\partial \mathcal{L}}{\partial B^1} = B^1 - \alpha dz[1]$$
$$B^2_{new} = B^2 - \alpha \frac{\partial \mathcal{L}}{\partial B^2} = B^2 - \alpha dz[2]$$

Once the weights and biases are updated, the forward propagation begins again and the loss is once again computed, leading to another back propagation step. This process is repeated until the error between the model's output and the expected output is deemed small enough. There are cases, which will be covered more extensively in the following sections, where the bias and weight parameters converge, prohibiting the model from updating itself.

2.1.3.1 Generalization

This section will propose a generalization of the backpropagation algorithm that can be applied to any neural network rather than the one in fig. 2.1.

Suppose there is a neural network with a loss function $\mathcal{L}$ and weights w_{ij} connecting the i-th and j-th neurons in adjacent layers. The goal being to compute the gradient of the loss function with respect to each weight in the network so that the weights are updated in a way that minimizes the loss.

In more generic neural networks, the used loss function is the following:

$$\mathcal{L} = \frac{1}{2}(y - \hat{y})^2$$

where y is the desired output and $\hat{y}$ is the output generated by the network.

To compute the gradient of the loss function with respect to a weight w_{ij}, we can use the chain rule:

$$\frac{\partial \mathcal{L}}{\partial w_{ij}} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial w_{ij}}$$

To compute the first term, $\frac{\partial \mathcal{L}}{\partial \hat{y}}$, we can simply take the derivative of the loss function with respect to the predicted output:

$$\frac{\partial \mathcal{L}}{\partial \hat{y}} = (y - \hat{y})$$

The term $\frac{\partial \hat{y}}{\partial w_{ij}}$, represents the sensitivity of the predicted output to the weight w_{ij}. In order to compute this term, we apply the chain rule once more as we need to take the derivative of the predicted output with respect to z, the sum of the inputs to the neuron :

$$\frac{\partial \hat{y}}{\partial w_{ij}} = \frac{\partial \hat{y}}{\partial z} \cdot \frac{\partial z}{\partial w_{ij}}$$

The first term, $\frac{\partial \hat{y}}{\partial z}$, is simply the derivative of the activation function applied to the sum of the inputs:

$$\frac{\partial \hat{y}}{\partial z} = f'(z) = f(z)(1 - f(z))$$

The second term, $\frac{\partial z}{\partial w_{ij}}$, is the input to the neuron j coming from neuron i. Plugging these terms back into the equation for the gradient of the loss function, we get:

$$\frac{\partial L}{\partial w_{ij}} = (y - \hat{y}) f(z)(1 - f(z)) \cdot x_i$$

where x_i is the output of neuron i.

To compute the gradient of the loss function with respect to a bias b_i, we can use a similar process:

$$\frac{\partial L}{\partial b_i} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial b_i}$$

The first term, $\frac{\partial L}{\partial \hat{y}}$, is the same as before. The second term, $\frac{\partial \hat{y}}{\partial b_i}$, is simply the derivative of the predicted output with respect to the bias:

$$\frac{\partial \hat{y}}{\partial b_i} = f'(z)$$

Plugging these terms back into the equation for the gradient of the loss

This process can then be repeated for each weight in the network as well as each bias term, starting at the output layer and working backwards through the hidden layers, to compute the gradients of the loss function with respect to all the parameters of interest in the network. Once the gradients are computed, the algorithm needs to set the learning rate and update the weights accordingly.

2.2 Universal Approximation Theorem

Now that the overarching architecture of neural networks has been explained, it is clear that their structure and training process are relatively simple. This leads to the following question: Why are neural networks so powerful and how can they accurately compute solutions to complex problems?

Neural networks try to model a function $f(x)$ which maps an input value x to a desired output y. The accuracy of $f(x)$ in predicting y differs depending on the distribution of the training data and the chosen choice of architecture, varying with the number of inputs and outputs and complexity of the hidden layers. Therefore, $f(x)$ can be arbitrarily complex and thus arbitrarily accurate.

Essentially, neural networks attempt to answer the following: given a function $f(x)$, can we find weights w_j, learning rates α_j and biases θ_j, with $j = 1, ..., N$, such that $f(x) \approx \sum_{j=1}^{N} \alpha_j \sigma(w_j x + \theta_j)$ to an arbitrary tolerance ϵ?

This is where the universal approximation theorem comes in.

Definition 2.1 *A function $\sigma(x)$ is sigmoidal if:*

$$\lim_{x \to \infty} \sigma(x) = 1$$

$$\lim_{x \to -\infty} \sigma(x) = 0$$

Theorem 2.1 (Universal approximation theorem)**.**
Let σ be any sigmoidal continuous function, then finite sums of the form:

$$G(x) = \sum_{j=1}^{N} \alpha_j \sigma(w_j x + \theta_j)$$

are dense in the set of continuous functions C in an interval I_n. In other words, given any $f \in C(I_n)$ and $\epsilon > 0$, there is a sum $G(x)$ of the above form such that:

$$|G(x) - f(x)| < \epsilon \text{ for all } x \in I_n$$

Proof. Suppose we let $w_j \to \infty$ for $j = 1, 2, ..., N$. Then we get:

$$\sigma(w_j x) = 1 \text{ for } x \geq 0$$

$$\sigma(w_j x) = 0 \text{ for } x \leq 0$$

Since we are maximizing the value of w_j, $\sigma(w_j x)$ we get what is referred to as a Heaviside step function: $H(x) = \lim_{w \to \infty} \sigma(wx)$, as in fig. 2.3.

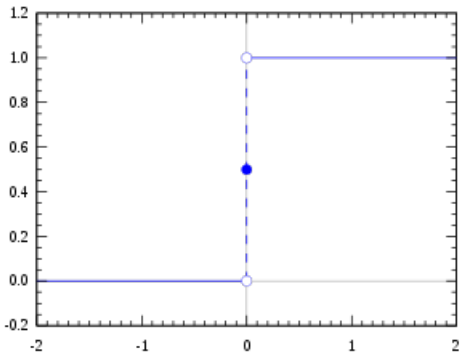

Figure 2.3: Heaviside Step Function

We now define a shifted version of the step function with a shifting constant is b_j such that:

$$\sigma(w_j(x - b_j))1 \text{ for } x \geq b_j$$

$$\sigma(w_j(x - b_j))0 \text{ for } x \leq b_j$$

Which gives us the shifted Heaviside step function: $H(x, b) = \lim_{w \to \infty} \sigma(w(x - b))$.

We can then use two such functions to create a piece of a function, as in fig. 2.4:

$$P(x, b, \delta) = H(x, b) - H(x, b + \delta)$$

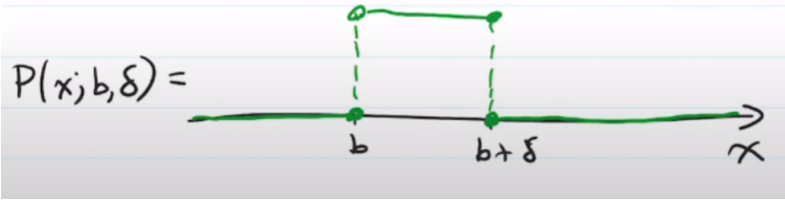

Figure 2.4: Subtraction of 2 Heaviside Step Functions

Since $f(x)$, is continuous, $\lim_{x \to a} f(x) = f(a)$. Then, $\forall a \in I_n$, $\exists$ an interval $(a_j, a_j + \Delta x)$

such that:

$$|f(x) - f(a_j)| < \epsilon, \forall x \in (a_j, a_j + \Delta x)$$

if we choose $b_j = a_j$, $\delta_j = \Delta x$, and $\alpha_j = f(a_j)$, therefore:

$$|f(x) - f(a_j)| < \epsilon, \text{for } a_j \leq x \leq a_j + \delta_j$$
$$|f(x) - a_j P(x, b_j, \delta_j)| < \epsilon$$

We then repeat this process for $x = a_{j+1} = b_j + \delta_j$, and construct:

$$G(x) = \sum_{j=1}^{N} a_j P(x, b_j, \delta_j)$$

Therefore, there exists a sum $G(x)$ such that $|G(x) - f(x)| < \epsilon$ for all $x \in I_n$ □

One very important thing to note is that theorem 2.1 guarantees existence, but makes no claims about N, the number of neurons in the hidden layers. It is possible that N grows exponentially as ϵ decreases. Nonetheless, this theorem is extremely powerful since it essentially states that any task that can be thought of as a function computation, can be approximated by a neural networks to a certain degree of accuracy.

2.3 Limitations

The architecture of neural networks we have covered so far are referred to as feed forward neural networks (FFN). These networks are often used to describe the core concepts of machine learning given that they are the most simplistic example of a working neural network. However, with this simplicity comes limitations. In fact,

despite the advantages of using FFN in certain applications, there are also certain drawbacks that are associated with such an approach:

- Size of data and computational power needed: these networks require large amounts of data in order to function properly, as well as a high level of computational power that is required to facilitate such functionality.

- Independence between layers: FFNs assume that inputs and outputs are independent and do not retain the information provided by past inputs and outputs. This can be a detriment to performance when FFNs are used in tasks which require context to be kept within the network, such tasks can include language translation or any auxiliary natural language processing task.

- Black box architecture: As FFNs increase in complexity, they become a sort of black box framework where the learning process is completely opaque to the user. An FFN with thousands of layers made by thousands of neurons is almost incomprehensible and does not correlate to the core principle of what was actually meant to be built. Most of the times, engineers and scientists want to understand the underlying process rather than just receiving a prediction or a result to solve the problem at hand.

- Instability of learning: FFNs are also susceptible to a problem that can arise when using any form of a neural network: exploding and vanishing gradients, This refers to instances where the weights or parameters of a neural network become unstable, limiting the ability of the model to continue learning and make accurate predictions in relation to any given dataset. Essentially, the gradients computed during back propagation become too big or too small, thus failing to correctly update the parameters of the network and halting the learning process.

2.4 Recurrent Neural Networks

Now that FFNs and the classic neural network architecture have been introduced, we can turn our interest to other variants of neural networks which improve on this iteration? Recurrent neural networks [18] are a type of artificial neural network which works with sequential input data, such as sentences or time series data. These networks use algorithms which are prominent in the deep learning scene and are commonly used for ordinal or temporal problems, such as language translation, or even stock prediction.

Like FFNs, RNNs utilize training data to guide the learning process but what distinguishes them is their "memory" component which retains information from prior inputs to influence current inputs and subsequent outputs. This is a direct way of compensating for one of the main limitations of FFNs stated in the previous section.

Through their sequential nature, the output of RNNs depend on the prior elements within the sequence. As seen in fig. 2.5, the output of the RNN on the left is not only fed forward but also retained, allowing the network to keep the information of previous iterations in order to provide more context for future outputs. It is also important to note that RNNs can have as many inputs and outputs, which is why some RNNs are categorized as one-to-one, one-to-many, many-to-one and many-to-many.

Another characteristic of these networks that separates them from FFNs is that their parameters are shared across each layer of the network. In fact, in the case of FFNs, it is common that each 2 neurons are connected by a different weight value across layers, while RNNs share the same weight parameter within each layer of the network. These weights are still adjusted through the back propagation process and gradient descent to guide learning. One difference in the learning of RNNs is that they use a variant of back propagation known as back propagation throught

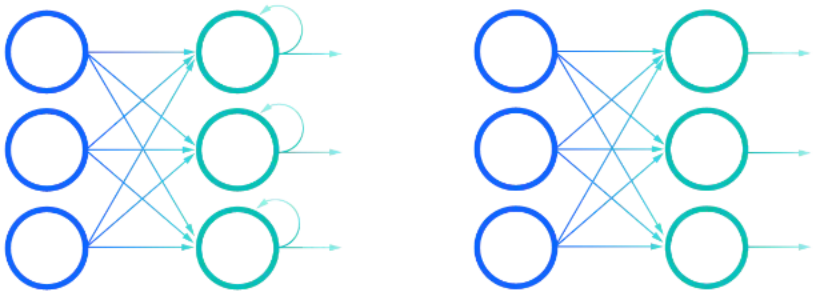

Figure 2.5: Comparison between an RNN (left) and an FFN (right)

time (BPTT) [18], which retains the core ideas of standard back propagation but is adapted to sequential data.

Just as for FFNs, when an RNN outputs a prediction vector, the loss $\mathcal{L}$ is computed and then the gradients are obtained by taking the partial derivative with respect to the weights, except this time, the total loss is the sum of losses at a given time t. Since we are back propagating through time and there is only one weight per layer, we thus have the following expression for a given loss $\mathcal{L}$, weight W and learning rate α.

$$\frac{\partial \mathcal{L}_{total}}{\partial W} = \sum_{t=1}^{T} \frac{\partial \mathcal{L}_t}{\partial W}$$

$$W_new = W - \alpha \frac{\partial \mathcal{L}_{total}}{\partial W}$$

It is important to note that this obviously entails that RNNs run into the same problem of unstable gradients just as all neural networks which apply back propagation. So for all the improvements that are proposed by the RNN architecture, the issue of instability of learning remains for networks with increasing complexities.

While RNNs are extremely useful in how they keep context within sequential data by storing previous inputs within the network, they suffer from a short term memory problem. Indeed, the memory component in RNNs only works for short

sequences and it is seen that with longer sequences, the performance of RNNs decreases drastically. In order to remedy this limitation, a variant of RNNs called Long Short-Term Memory (LSTM) networks was created.

2.5 LSTM Neural Networks

LSTM models work around the short-term memory problem by deciding whether to retain or discard the information kept in short-term memory. This process is shown in fig. 2.6 with three main components: the Forget Gate, the Input Gate and the Output Gate. However, prior to defining the roles of each gate, it is important to define the concepts of hidden and cell states.

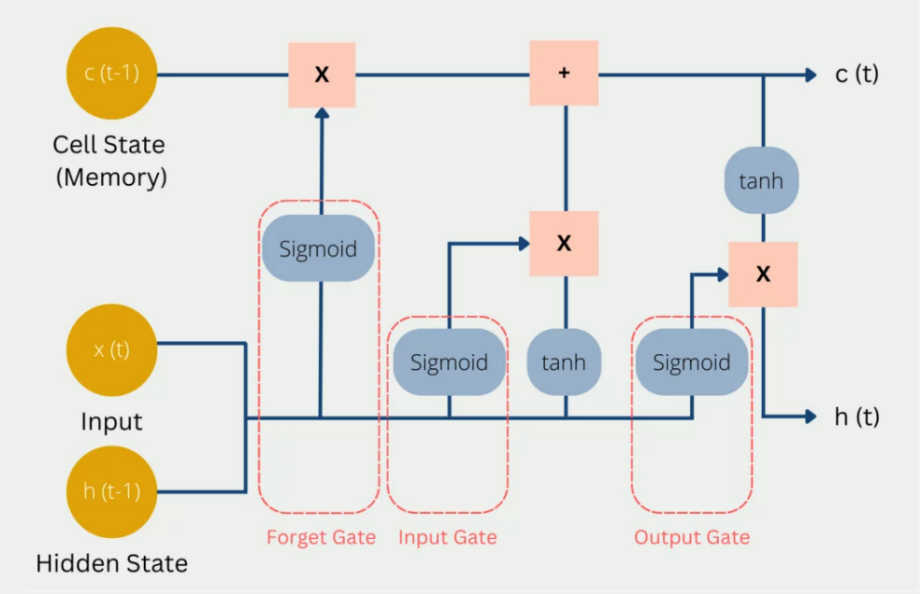

Figure 2.6: Architecture of LSTM network

The cell state is one of the main differentiating aspects of LSTM and RNN architectures. This state refers to the long term memory of the network at a time t, which is updated throughout iterations and does not store information from immediately previous iterations. The cell state essentially contains encoding for the long range of input sequences. On the other hand, the hidden state refers to

the output of the most recent run of the network. It is the same information that is retained in RNNs after every iteration.

More specifically, in LSTM models, the output at every iteration depends on the new input value x_t, as well as the current cell state c_t and the previous hidden state h_{t-1}. All this information goes through a series of computations proctored by the 3 gates which work together to update the hidden and cell states after every run.

- **Forget Gate:** This is where the LSTM network decides if previous and current information is retained or forgotten. In order to do this, the new input value and the previous hidden state are passed through a sigmoidal activation function, and if the output value is closer to 0, then the values are not retained in short-term memory, and vice-versa if the value is closer to 1. More mathematically, the Forget Gate's output is given by:

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t])$$

  Where W_f is a matrix of weights. This output is then sent up in the network and pointwise multiplication occurs with the previous cell state as shown in fig. 2.6. This pointwise multiplication means that components of the cell state which have been deemed irrelevant by the Forget Gate network will be multiplied by a number close to 0 and thus will have less influence on the following steps, and those who have been deemed worth retaining will be multiplied by a number closer to 1 and thus keeping their value in the network.

- **Input Gate:** This is where it is decided how valuable the current input is to solve the task and what new information should be used as the network's current cell state, given h_{t_1} and x_t. Thus, just like the Forget Gate, this gate takes in 2 inputs, the previous hidden state and the new input data. However,

before the Input gate comes into play, a new memory network which uses the tanh activation function plays a role first. This new memory network can be pictured on the right of the Input Gate in fig. 2.6.

h_{t_1} and x_t are passed through the tanh activation function along with a weight W_c which combines them to output a "new memory vector" containing information from the new input data given the context from the previous hidden state. We know that tanh is not a sigmoidal function since it can output values between -1 and 1, so this new memory vector can also be negative.

Following this step, the Input Gate then takes in the same 2 inputs and passes them through a sigmoidal activation function along with a weight W_i, outputting a vector of values between 0 and 1. This vector is then pointwise multiplied with the new memory vector. Thus the output of the Input gate is given by:

$$i_t = \tanh(W_c \cdot [h_{t-1}, x_t] \otimes \sigma(W_i \cdot [h_{t-1}, x_t])$$

Where W_C is the weight matrix for the input gate. The resulting output is then added with the current cell state, thus updating the current information in the network. It is important to note that this step adds a vector to the cell state rather than multiplaying.

- **Output Gate:** Once the current cell state has been updated by the 2 previous gates, the role of the Output Gate is to compute the new hidden state which will be used in the next iteration of the network. Once again, the input in this gate are the previous hidden state and the new input data. These 2 inputs are also passed through a sigmoidal activation function along with a weight

matrix W_o. Thus the output of the Output Gate is given by:

$$o_t = \sigma(W_o \cdot [h_{t-1}])$$

In parallel, the current cell state c_t is passed through the tanh activation function, outputting a vector of values between -1 and 1. This vector is referred to as the "squished cell state". This squished cell state and the sigmoidal vector from o_t are then pointwise multiplied, which results in the new hidden state:

$$h_t = \tanh(c_t) \otimes o_t$$

The reason for which the network does not simply output the updated cell state is because the cell state is the accumulation of all the iterations of the network, and the next iteration does not require that much information. A good analogy for this would be someone giving a thorough explanation of how typhoons happen when simply asked whether it will rain or not.

In summary, the Forget Gate decides which pieces of the long-term memory should now be forgotten (have less weight) given the previous hidden state and the new data point in the sequence. Following that, the Input Gate updates the cell state by adding information that is considered worth keeping. Finally, the Output Gate prepares for the next iteration by deciding the next hidden state. The 3 gates in LSTMs are neural networks with their respective weights and activation functions, which get updated through BPTT just as an RNN.

The steps described above are repeated at each iteration of the network, but the actual output of the network once the iterations are done can't be the hidden state, since we are actually interested in all the accumulated information in the cell state. For that purpose, at the very last step of the network, there is a linear layer

that converts the last hidden state to the desired output. This operation occurs only once, which is why it is not included in the diagram of fig. 2.6

2.5.1 Solving the Vanishing Gradient Problem

Now that the overarching structure of LSTMs has been explained, it is important to explain what the inclusion of the gates, the cell state and memory cells bring in terms of improvement over classic neural network architectures, most notably RNNs.

The biggest advantage of LSTMs is how they solve the vanishing gradient problem. In architectures such as RNNs, the hidden state at time step t is a function of the hidden state at time step $t-1$ and the input at time step t, mathematically represented as:

$$h_t = \sigma(h_{t-1}, x_t)$$

The problem with this is that when the model is trying to learn long-term dependencies, the gradients of the weights become very small, and as the gradients are used to update the weights during the training process, the updates are very small and slow, causing the model to converge very slowly as the gradients begin to vanish. The gradients eventually completely vanish and the weights of the model are frozen.

LSTMs solve this problem through the memory cells that can retain information over a longer period of time and the control exerted by the three gates.

As explained earlier, the input gate controls the flow of information into the memory cell, the forget gate controls the flow of information out of the memory cell, and the output gate controls the flow of information out of the memory cell and into the output.

The input gate is controlled by the current input and the previous hidden state, the forget gate is also controlled by the current input and the previous hidden state, and the output gate is controlled by the current input, the previous hidden state, and the current memory cell state.

By including these gates, LSTMs are able to selectively choose which information to keep or discard, and thus avoid the vanishing gradient problem. The gates allow the network to selectively let information through, which means that the gradients are less likely to vanish, as the network can choose to keep important information and discard the less important information.

Additionally, the memory cells, by themselves, are not affected by the non-linearity of the activation function, which means that the gradients flowing through them are less likely to vanish or explode, as they are not affected by the saturation of the activation function.

Therefore, LSTMs use the gates and memory cells to solve the vanishing gradient problem by selectively choosing which information to keep, and allowing the gradients to flow through the memory cells, rather than saturating them with non-linear activation functions.

One important thing to note about the vanishing gradient problem is that it is most commonly encountered in networks which use a sigmoidal activation function. We know that the sigmoid function's derivative can be expressed in terms of itself:

$$\sigma(x)' = \sigma(x)(1 - \sigma(x))$$

The maximum value of this derivative is 0.25, which occurs when $x = 0$. As the value of x gets larger (either positive or negative), the derivative of the sigmoid

function gets smaller and smaller, approaching 0. This means that the error gradient will also get smaller and smaller as it is propagated backwards through many layers, eventually vanishing to zero.

To address this issue, alternative activation functions such as ReLU have been developed. The choice of the used activation function is therefore a key aspect to consider when building a neural network.

CHAPTER 3

Convolutional Neural Networks

Now that the core concepts of machine learning have been explored, our interest shifts to architectures which are more relevant to generation of text and image, beginning with Convolutional Neural Networks (CNN).

CNNs are a type of neural network commonly used for image and video analysis, but can also be used for speech recognition, digital art and other applications. Compared to other types of neural networks, such as FFNs or RNNs, CNNs are specifically designed to work with high-dimensional input data, such as images.

Through the use of filters (also referred to as kernels), CNNs are able to learn and extract features from input data [11]. These filters are contained in layers called convolutional layers and essentially help identify patterns such as edges or shapes to help with data analysis.

3.1 Convolutional Layers

Convolutional layers are the building blocks of CNNs. These layers apply filters to input data to extract important features. The filters are essentially small matrices of weights that are learned during the training process. This application of a filter on an input is referred to as convolution.

In computer vision, an image is typically represented as an assembling of pixels in the form of a matrix and a number of channels, which is what enables the representation of color. For example, for an RGB image, the number of channels would be three, while a grey scale image would only have a single channel with pixels whose value in the matrix could fluctuate between 0 for black and 255 for white, as seen in fig. 3.1

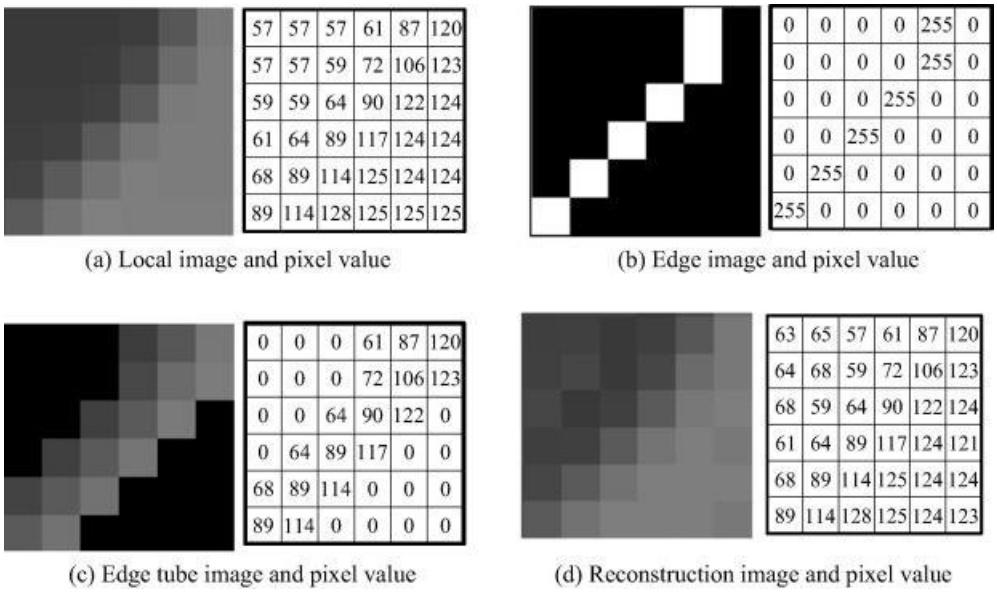

Figure 3.1: Representation of a Grayscale Image in Computer Vision.

The convolution operation involves sliding the filter over the input image, and computing the dot product between the filter and the corresponding piece of the image, before storing the result of the dot product in an output matrix called the feature map. For example, if the image is represented by a 9×9 matrix, and the filter by a 3×3 matrix, then, we would take the dot product between the filter and the top-left 3×3 patch of the input image matrix, then store the output in the top-left position of the output matrix. The patch is then shifted one column to the right and this operation is repeated until the entire image matrix has been covered by the filter.

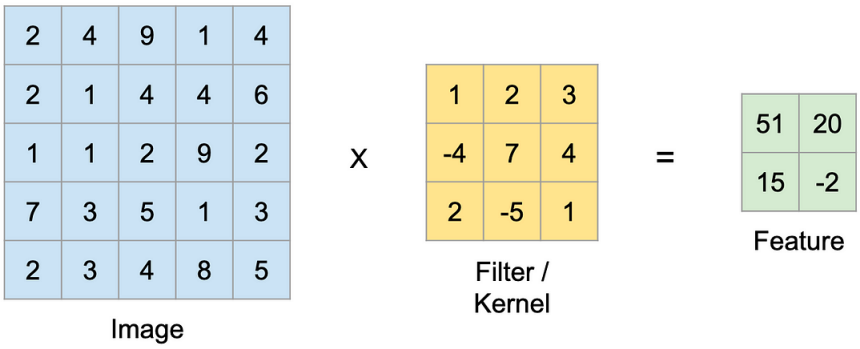

Figure 3.2: Convolution: Application of a Filter on an Input Image

More mathematically, given an input image, represented as a matrix I with dimensions $W \times H \times C$, where W is the width, H is the height, and C is the number of channels. A filter, represented as a matrix F with dimensions $w \times h \times C$, is applied to the input image. The output of this operation is the feature map, represented as a matrix M with dimensions $(W - w + 1) \times (H - h + 1) \times 1$, assuming a single channel output.

To generalize, the output of the feature map at a given position (i, j) can be expressed as the following:

$$M_{i,j} = \sum_{a=1}^{w} \sum_{b=1}^{h} \sum_{c=1}^{C} I_{i+a-1,j+b-1,c} F_{a,b,c}$$

Where $M_{i,j}$ represents the value of the feature map at position (i, j), and $I_{i+a-1,j+b-1,c}$ represents the value of the input image at position $(i + a - 1, j + b - 1)$ and channel c. $F_{a,b,c}$ represents the weight of the filter at position (a, b) and channel c.

Through training of the CNN, the weights of the filters in the convolutional layers are adjusted and enable the network to output feature maps which are revealing of various elements in the input image such as shapes, edges or textures. Some CNNs are specifically trained to detect more complex shapes such as eyes, humans or different kinds of plants.

3.2 Subsequent Layers

Following the convolutional layer, the resulting feature map is passed through other layers before being output by the network. The subsequent layers include the following:

- Activation layer: This layer applies a non-linear activation function to the output of the previous layer. The activation function introduces non-linearity into the network, which is essential for capturing complex relationships between features. The most commonly used activation function for CNNs is the ReLU function, defined as:

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

- Pooling layer: This layer downsizes the output of the previous layer by taking the maximum, average, or other summary statistic over small regions of the feature maps. The aim of the operation is to reduce the spatial dimensions of the feature maps while preserving the most important features.

- Fully connected layer: This layer takes the downszied output of the previous layer and passes it through a fully connected neural network, which performs linear transformations to finalize the given task.

The output of these layers is then passed to a final output layer which can perform different operations depending on the task at hand. The layer can consist of a single node for a binary-classification problem or can output an entire image matrix for a generation task.

CHAPTER 4

Transformer Neural Networks

Tranformers [21] are a machine learning architecture which specializes in sequential data inputs and is mainly used in tasks such as NLP tasks. Its original conception advocated for the use of the attention mechanism which had already been used in previous models, but the results it generated far exceeded expectations and transformers are now a state-of-the-art architecture in a variety of artificial intelligence fields, specifically NLP.

Transformer networks are a very powerful tool as they make up for the shortcomings of RNNs and LSTMs due to their ability to take in entire sequences as inputs while learning the complex relationships between the elements of sequence using a mechanism known as self-attention.

More specifically for this paper, transformers have really interesting applications in both text and image generation and will thus be used in the construction of the illustrated storybook generator.

4.1 Attention mechanism

The concept of attention is the core component of the transformer architecture. As the name suggests, the attention mechanism aims to take in an input sequence, such

as a sentence, and selectively attend to different parts of the sentence. Essentially, the mechanism aims to compute how much each word must pay "attention" to other words in the sentence. Given how languages work, words are not always related to what comes after and before them. A word can sometimes refer to another word that came many words prior, which is why this is important for NLP tasks, where the order and relationships between different elements of the sequence are often critical for accurate predictions. By selectively attending to different parts of the input sequence, the attention mechanism allows the transformer to model these relationships more effectively.

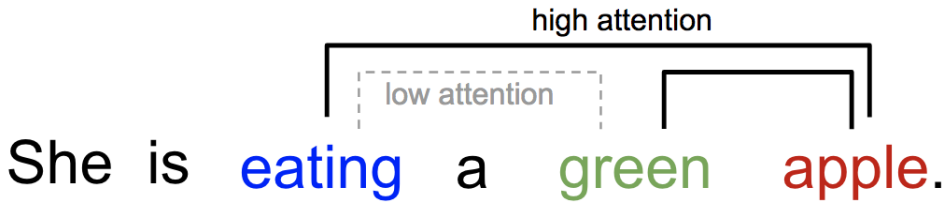

Figure 4.1: Attention Applied to a Sentence

In the transformer architecture, the attention mechanism is used in two key components: the self-attention mechanism and the multi-head attention block [21]. In the self-attention mechanism, the model attends to different parts of the input sequence to compute a weighted representation of the sequence, which is then used in subsequent layers of the transformer. In the multi-head attention block, multiple self-attention operations are performed in parallel to capture different aspects of the input sequence, allowing the model to capture more complex relationships between different parts of the sequence.

4.2 Self-Attention Mechanism

In transformers, the self-attention mechanism computes a weighted sum of the elements in a sequence for each position in the sequence. The weight for each element is computed using a score function that measures the relevance of the element at a given position to all other elements in the sequence.

The score function is computed using "keys", "values", and "queries". These concepts can be a bit abstract, but one of the better ways of explaining them is by using the example of a search engine. Let's take YouTube as an example, when you search for a video, the text you type in the search bar is the query, the query is then mapped to a series of keys which can be a video title, the text in the description or even the comments in the videos. After the query is successfully mapped to a series of keys, YouTube redirects to a page with a series of videos, which are the values you are looking for. In summary, YouTube is using a user-specified query to search for videos that have keys (e.g. titles, tags) that are relevant to the query, before then returning the values (actual video suggestions) back to the user.

More specifically, the self-attention mechanism in a transformer is applied by using a query vector, a set of key vectors, and a set of value vectors, that are all learned parameters of the model.

The intuition behind this is that we can think of the key and value as representing the content of any element in the input sequence, while the query represents the importance of every element to the task at hand. The model then computes a similarity between each query and key. The resulting similarity is used as "attention weights" which are then associated with the values.

By doing so, the model is able to assign higher weights (more importance) to elements in the input sequence that are a better fit for the query, and lower weights for less relevant elements. Overall, this allows the encoded elements of the input sequence to have more context about the elements around them.

More mathematically, to compute the attention weights, we first compute a score for each key vector based on its similarity to the query vector. These are all learned parameters. The most common way to do this is to use the dot product between the query vector and each key vector:

$$\text{score}(q, k_i) = q \cdot k_i$$

Here, q is the query vector, and k_i is the i-th key vector. i ranges from 0 to n, which is the number of words in the input sequence. The higher values indicating greater similarity. We can also note that this dot product is the same thing as the matrix multiplication QK^T, where Q and K are the matrices populated with the sets of queries and keys.

Once the scores for all the key vectors have been computed, the result is scaled by a factor of $\frac{1}{\sqrt{d}}$, where d is the model's encoding dimension. For example, if the model encodes words using vectors of dimension 256, then the the scaling factor is $\frac{1}{\sqrt{256}}$.

This scaling is done in order to remedy for large values obtained from the dot products which can then lead to instability in the training process. When d is considered small, this step can be skipped as the model is less prone to the vanishing gradient problem.

After the scaling, a softmax function is applied to obtain the attention weights. The softmax function is used for normalization and ensures that the attention weights are positive and sum to 1. It is defined as such:

$$\alpha_i = \text{softmax}\left(\frac{\text{score}(q, k_i)}{\sqrt{d}}\right)$$

$$\alpha_i = \frac{\exp\left(\frac{\text{score}(q,k_i)}{\sqrt{d}}\right)}{\sum_{j=1}^{n}\left(\frac{\text{score}(q,k_j)}{\sqrt{d}}\right)}$$

Where, α_i is the attention weight for the i-th key vector, and n is the total number of key vectors.

Finally, the attention weights are multiplied with the value matrix V, which is also a learned parameter, in order to compute what is known as the scaled dot-product attention:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d}}\right)V$$

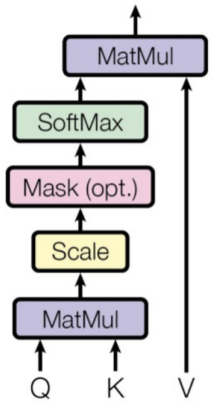

Figure 4.2: Scaled Dot-product Attention

This process can be repeated for each position in the input sequence, allowing the transformer to selectively focus on different parts of the input when generating output.

4.3 Multi-Head Attention

The multi-head attention block, as seen in fig. 4.3, is an extension of the self-attention mechanism in which multiple attention heads are used to capture different aspects of the input sequence. The idea is to split the query, key, and value matrices into h different matrices, and perform self-attention on each of them in parallel. Each attention head has its own set of learnable weight matrices.

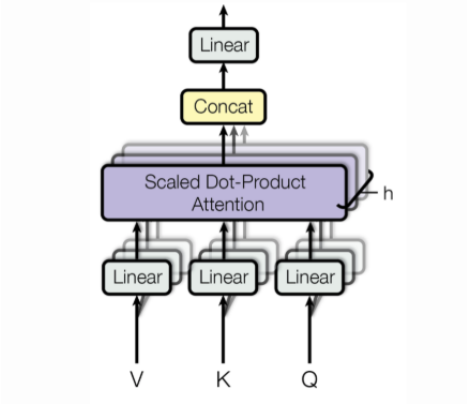

Figure 4.3: Multi-head Attention Block

The first step is to perform self-attention on each head, which will output h output matrices which we will refer to as C_i.

$$\mathbf{C}_i = \text{Attention}(Q_i, K_i, V_i)$$

However, since the input to the multi-head attention block is a single sequence in the form of a matrix, it is not ideal that the output is a set of matrices. For that reason, the h output matrices go through a concatenation layer which outputs a single matrix C which contains all the information extracted from the h attention heads. This matrix is then linearly transformed using another learnable weight matrix W_O, to obtain the final output of the multi-head attention block:

$$\mathbf{Z} = \mathbf{C}W_O$$

The final output matrix $\mathbf{Z} \in \mathbb{R}^{n \times d}$ contains a representation of the input sequence that captures multiple different aspects of the sequence, as captured by the different attention heads.

By allowing the model to attend to different parts of the input sequence in different ways, the multi-head attention block helps the transformer to model complex relationships between different parts of the input sequence.

4.4 Positional Encoding

In the previous sections, we have discussed the concepts of self-attention and multi-head attention, and in doing so, it became clear that the the attention mechanism used in the transformer architecture does not have any inherent notion of position or order of the elements in the input sequence. This is in contrast to other sequential models like RNNs, which have an inherent notion of order due to their recurrent connections.

However, when dealing with NLP tasks such as text generation, order is not something that can be completely ignored, and in order to incorporate information about the position of each element in the sequence, the input sequence is passed through a positional encoding layer.

The positional encoding layer takes place before the input sequence is passed through the multi-head attention, which allows the transformer to capture positional information in a way that is compatible with the attention mechanism. This allows the model to effectively model complex relationships between different parts of the input sequence, even when the order of the elements in the sequence is critical to the task at hand.

Positional encoding is a fixed encoding of the position of each element in the sequence, which is added to the element's representation. This allows the transformer model to attend to elements in the sequence based on their position, as well as their content.

There are a few different ways to implement positional encoding, but a common method is to use sine and cosine functions of different frequencies:

$$PE(pos, 2i) = \sin(pos/10000^{2i/d})$$

$$PE(pos, 2i+1) = \cos(pos/10000^{2i/d})$$

Where, *pos* is the position of the element in the input sequence, i is the embedding position for each element, and d is the chosen dimension for the embeddings in the model.

We can see from the PE expression that the aim is to use a sine function for even embedding positions and a cosine function for odd embedding positions. Additionally, the magnitude of the sinusoidal functions changes as i changes, for a fixed *pos*.

The idea behind the formula is that each position in the sequence should have a unique encoding that captures its position information. The choice of the specific frequency and phase for each sine and cosine function ensures that each position in the sequence has a unique encoding. The factor of 10000 in the denominator is an arbitrary choice that sets the scale of the encoding values.

Example 4.1. We will present an example of how the sentence "I am happy" would be encoded using the positional encoding layer, using 4-dimensional embeddings:

First, we tokenize the sentence into a sequence of word embeddings:

$$I = \begin{bmatrix} I_1 \\ I_2 \\ I_3 \end{bmatrix} = \begin{bmatrix} 0.3 & 0.5 & -0.1 & \dots \\ -0.2 & 0.8 & 0.6 & \dots \\ 0.7 & 0.2 & -0.4 & \dots \end{bmatrix}$$

where I_1, I_2, and I_3 are the embeddings for "I", "am", and "happy", respectively, and each row corresponds to a single embedding dimension.

Next, we apply the positional encoding function to each embedding:

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

For example, for the first embedding dimension of the first word ("I"), we have:

$$PE_{0,0} = \sin\left(\frac{0}{10000^{0/4}}\right) = 0$$
$$PE_{0,1} = \cos\left(\frac{0}{10000^{1/4}}\right) = 1$$

So the final encoded vector for the first word ("I") is:

$$E_1 = \begin{bmatrix} 0.3 & 0.5 & -0.1 & \dots \end{bmatrix} + \begin{bmatrix} 0 & 1 & 0 & \dots \end{bmatrix} = \begin{bmatrix} 0.3 & 1.5 & -0.1 & \dots \end{bmatrix}$$

We repeat this process for each word in the sequence to get the final encoded sequence:

$$E = \begin{bmatrix} E_1 \\ E_2 \\ E_3 \end{bmatrix} = \begin{bmatrix} 0.3 & 1.5 & -0.1 & \dots \\ -0.2 & 1.5 & 0.8 & \dots \\ 0.7 & 1.5 & -0.4 & \dots \end{bmatrix}$$

where each row corresponds to a single word in the sequence, and each column corresponds to a single embedding dimension.

Once PE is computed for every i and every pos in the input sequence, the resulting matrices are added to the initial word embeddings, which introduces a dimension of order in the embeddings.

4.5 Closing Facts

In conclusion, transformers have set themselves apart as the leading architecture in NLP by providing a more efficient way of processing sequential data. The key innovation of the transformer is the use of the self-attention mechanism, which allows the model to focus on important parts of the input sequence while

ignoring irrelevant information. Transformers also incorporate positional encoding to preserve information about the position of each element in the sequence.

The architecture has massively outperformed previously used models such as RNNs and LSTMs, almost making them obsolete. While there is the issue of transformers being potentially computationally expensive, their scalability as well as undoubted performance make them a no-brainer choice for any language task. In coming years, it is clear that transformers will continue to cement themselves as the go-to architecture for sequential data and may even branch out to fields broader than NLP.

In regards to this project, it is clear that using a transformer architecture for a text generation model is a very good fit and has the potential to yield very good results.

CHAPTER 5

Generative Adversarial Networks

With the purpose of this paper being to cover new technologies involved in generative ML frameworks through the construction of a illustrated storybook generator that uses text synthesis and image generation, it is natural to cover the topic of generative adversarial networks (GAN [6] arguably the most prominent generative model architecture in deep learning.

Just as the word "adversarial" suggests, GANs are networks which leverage the contest between two models: a generator model and a discriminator model. Given a certain data distribution, the generator is tasked to generate fake data, which is then sent to the discriminator whose role is to discern the fake data from the real data. Essentially, the generator and discriminator are adversaries trying to deceive the other and as both models learn through training, the aim is to reach a point where the fake data generated is good enough to be indiscernible from the training data, thus always deceiving the discriminator.

5.1 Discriminative and Generative Methods

As explained above, GANs are a combination of 2 models, a generative model G and a discriminative model D. These 2 models are predictive models which

are inspired by the 2 main methods used in ML to make predictions. The most popular one is the discriminative method, which is used in most common neural network architectures, where the model learns the patterns of a desired output, given the training data. More mathematically, a discriminative model aims to learn the conditional distribution of the target variable Y given an input variable X.

$$P(Y|X = x)$$

On the other hand, generative models aim to learn the joint probability distribution of the input variable and output variable.

$$\begin{aligned} P(X, Y) &= P(X|Y)P(Y) \\ &= P(Y|X)P(X) \end{aligned}$$

Therefore, when the model is tasked with making a prediction it uses Bayes' theorem and computes the conditional probability of the Y given X:

$$P(Y|X) = \frac{P(X, Y)}{P(X)}$$

The biggest advantage of generative models is that they can be used to generate new instances of data, since these models also learn the distribution of the input data, whereas discriminative models only learn the distribution of the desired output.

5.2 Architecture Overview

We will now take a high-level look into the architecture of a GAN, as seen in fig. 5.1.

G and D, which are the main components of the model are simply neural

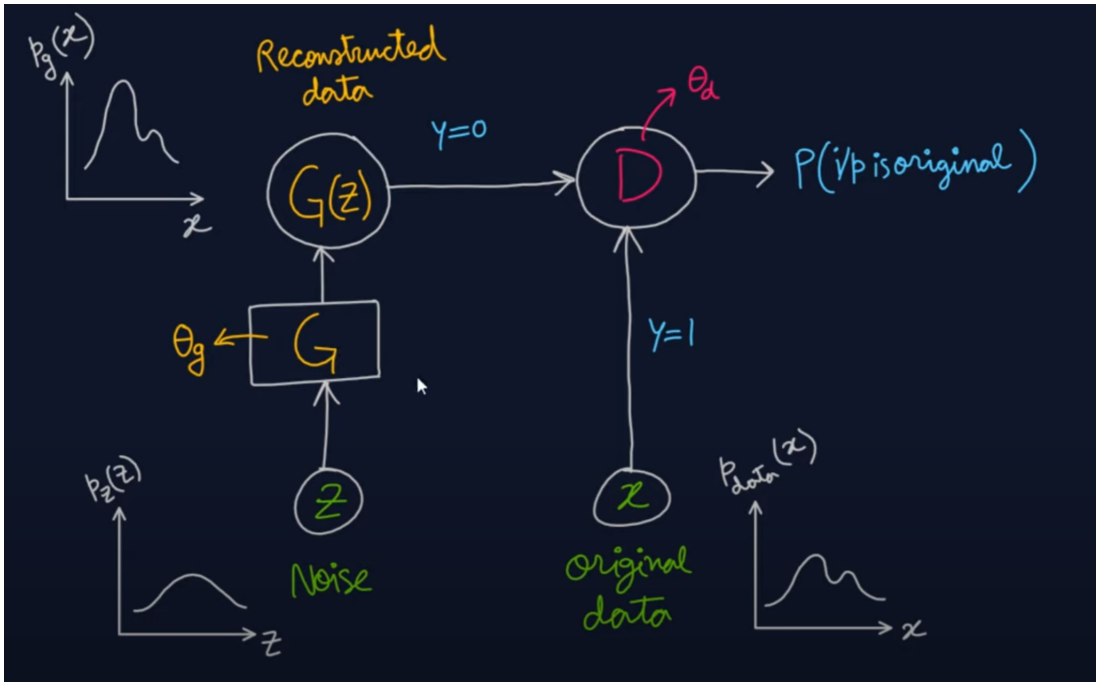

Figure 5.1: GAN Intuitive Overview

networks, with θ_g and θ_d being their respective weight matrices. The training data follows a distribution $P_{data}(x)$ and there is also a noise distribution $P_z(z)$.

At every iteration in the training process, a randomly sampled input x from the training data is fed to D and a noise vector z is fed to G. Initially, the input noise z contains no information, and after being passed through G, an output $G(z)$ is created, representing the reconstructed data generated by G. Essentially, in a traditional GAN, G is tasked with generating data from scratch, and as this process continues, a distribution $P_g(x)$ is created for the reconstructed data. The reason why the variable x is chosen to describe the distribution of the generated data $G(z)$ is because the domain of the original data is the same as that of $G(z)$.

More mathematically, to learn the generator's distribution $P_g(x)$ over the training data x, a prior on input noise variables following $P_z(z)$ is defined, then a mapping to data space $G(z;\theta_g)$ is defined, where G is a neural network which takes in the weight θ_g and a randomly sampled noise value z. Similarly, we define a neural network $D(x;\theta_d)$, which takes in the weights θ_d and a randomly sampled element x

from the training data. The output of D represents the probability that x came from the data rather than P_g. This output is a single binary scaler, with 0 meaning that the input came from generated data and 1 meaning that the input came from the original training data.

5.2.1 Minimax Game

The GAN architecture is best represented by the following statement: D and G are playing a two-player minimax game. A two-player minimax game is a game where one player wants to maximize their probability of winning, while the second player aims to minimize the first player's probability of winning. This specific two-player minimax game between D and G has a value function $V(G, D)$ [6]:

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

This function is very similar to the Binary Crossentropy Loss function L, which is often used when training a binary classifier such as D.

5.2.2 Binary Cross Entropy

Binary cross entropy [24] is a measure of the distance between two probability distributions. It is commonly used in classification tasks as the loss function, where the goal is to predict the probability of a certain class (often represented as a binary value, either 0 or 1) and the predicted probability is compared to the true label using the cross entropy loss.

The binary cross entropy loss is defined as:

$$\mathcal{L} = -(y \log(p) + (1 - y) \log(1 - p))$$

where y is the true label (0 or 1) and p is the predicted probability of the positive class (also between 0 and 1).

The loss function has the property that it is always non-negative, with the value approaching 0 as the predicted probability gets closer to the true label. In other words, the loss function is minimized when the predicted probability is equal to the true label.

If $y = 1$, the loss is equal to $-log(p)$. This means that if the predicted probability is close to 1 (e.g. $p = 0.9$), the loss will be small (e.g. $-log(0.9) = 0.1$). On the other hand, if the predicted probability is far from 1 (e.g. $p = 0.1$), the loss will be large (e.g. $-log(0.1) = 2.3$).

If $y = 0$, the loss is equal to $-log(1-p)$. This means that if the predicted probability is close to 0 (e.g. $p = 0.1$), the loss will be small (e.g. $-log(1 - 0.1) = 0.1$). On the other hand, if the predicted probability is far from 0 (e.g. $p = 0.9$), the loss will be large (e.g. $-log(1 - 0.9) = 2.3$).

We then have the following expression:

$$\mathcal{L} = -\sum y \ln \hat{y} + (1 - y) \ln(1 - \hat{y})$$

In the above expression, y is the input value, which can be 0 or 1, and $\hat{y}$ is the predicted probability of the model. So we have the following 2 cases:

- When $y = 1$, then the input came from the original training data, so $\hat{y} = D(x)$ and we get: $\mathcal{L} = -\sum \ln(D(x))$.

- When $y = 0$, then the input came from the generated data, so $\hat{y} = D(G(z))$ and we get: $\mathcal{L} = -\sum \ln(1 - D(G(z)))$.

We take away the summation terms and add the 2 cases, which gives us:

$$L = \ln(D(x)) + \ln(1 - D(G(z))$$

While this expression has begun resembling $V(G, D)$, it is only valid for a single data point since we are not using expectation. Expectation is the average value of an experiment if it is performed a large number of times. A common example is a heads or tails game that is performed a large number of times, the average probability of each outcome averages out to be $\frac{1}{2}$ as long as the coin is not biased. More mathematically, expectation, in a discrete sense, is the sum of the product of every outcome with its probability:

$$\mathbb{E}(x) = \sum xp(x)$$

We then take the expectation of the previous expression:

$$\mathbb{E}(\mathcal{L}) = \mathbb{E}[\ln(D(x))] + \mathbb{E}[\ln(1 - D(G(z)))]$$
$$\mathbb{E}(\mathcal{L}) = \sum p_{data}(x) \ln(D(x)) + \sum p_z(z) \ln(1 - D(G(z)))$$

However, we are assuming that the distributions are continuous, so rather than taking the sum, we need to take the integral.

$$\mathbb{E}(L) = \int p_{data}(x) \ln(D(x))dx + \int p_z(z) \ln(1 - D(G(z)))dz$$

We have now derived the non-optimized value function $V(G, D)$:

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

5.2.3 Optimizing the value function

Since the goal is to minimize the loss, we want to find the global minimum for the value function. Intuitively, since the generator wants to minimize the discriminator's probability of winning, the distribution of the generator's generated data needs to replicate the distribution of the training data, so as to deceive the discriminator's predictions. More mathematically, we want to prove that P_g converges to P_{data} at the global minimum of the value function. In other words, we want to show that $P_g = P_{data}$ at the global minimum of $V(G, D)$.

However, as per the minimax game, we first fix G and the aim is to find the value of D for which V(D, G) is maximized, we have the following:

$$V(D, G) = \int_x P_{data}(x) \ln(D(x)) + P_g(x) \ln(1 - D(x)) dx$$

We can see that the above expression is of the form: $a \ln(x) + b \ln(1 - x)$. Finding the global maximum of this function can be done by differentiating it and finding the critical point. By doing so we get that the global maximum of this function is found at $x = \frac{a}{a+b}$.

Therefore, for a fixed G, the optimal discriminator is:

$$D_g^*(x) = \frac{p_{data}(x)}{p_g(x) + p_{data}(x)}$$

We can thus replace $D(x)$ by the optimal $D^*(x)$ in the expression and we get:

$$\min_G V(D, G) = \mathbb{E}_{\boldsymbol{x} \sim p_{\text{data}}} \left[\ln \frac{p_{\text{data}}(\boldsymbol{x})}{P_{\text{data}}(\boldsymbol{x}) + p_g(\boldsymbol{x})} \right] + \mathbb{E}_{\boldsymbol{x} \sim p_g} \left[\ln \frac{p_g(\boldsymbol{x})}{p_{\text{data}}(\boldsymbol{x}) + p_g(\boldsymbol{x})} \right]$$

We now want to find the value of G which will minimize the value function. Intuitively, what we want to prove here, is that the probability distribution of G will

be the same as the probability distribution of the data, since the optimized generator should output data that is not discernable from the real data.

Therefore we want to measure the distance between the $p_g(x)$ and $p_{data}(x)$. The method which is used to compute this distance is known as the Jensen-Shannon divergence.

5.3 Jensen–Shannon divergence

The Jensen-Shannon divergence (JSD) is a measure of the similarity between two probability distributions. It is a symmetric version of the Kullback-Leibler divergence (KLD), which is commonly used for the same purpose.

JSD is defined as the average of the KLDs between each distribution and their average.

We first have the average M of two distributions, P and Q:

$$M = \frac{P + Q}{2}$$

The KLD between P and Q is defined as:

$$KL(P||Q) = \sum_i P(i) \ln \frac{P(i)}{Q(i)}$$

We then take the average of the KLD between both P and Q with M to get the JSD.

$$JS(P||Q) = \frac{KL(P||M) + KL(Q||M)}{2}$$

$$JS(P||Q) = \frac{1}{2}\left(\sum_i P(i) \ln \frac{P(i)}{M(i)} + \sum_i Q(i) \ln \frac{Q(i)}{M(i)}\right)$$

Since in the case of GANs, we are not dealing with discrete distributions, we take the expectation of the distribution rather than the sum.

$$JS(P\|Q) = \frac{1}{2}\left(\mathbb{E}_{\boldsymbol{x}\sim P} \ln \frac{P}{M} + \mathbb{E}_{\boldsymbol{x}\sim Q} \ln \frac{Q}{M}\right)$$

It is important to notice that if we set $P = p_{\text{data}}$ and $Q = p_g$ and $M = \frac{p_{\text{data}} + p_g}{2}$, the expression above is very similar to the value function with the optimized $D^*(x)$.

In fact, We have:

$$\min_G V(D,G) = \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}}\left[\ln \frac{p_{\text{data}}(\boldsymbol{x})}{P_{\text{data}}(\boldsymbol{x}) + p_g(\boldsymbol{x})}\right] + \mathbb{E}_{\boldsymbol{x}\sim p_g}\left[\ln \frac{p_g(\boldsymbol{x})}{p_{\text{data}}(\boldsymbol{x}) + p_g(\boldsymbol{x})}\right]$$

$$\min_G V(D,G) = \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}}\left[\ln \frac{\frac{p_{\text{data}}(\boldsymbol{x})}{P_{\text{data}}(\boldsymbol{x}) + p_g(\boldsymbol{x})}}{2}\right] + \mathbb{E}_{\boldsymbol{x}\sim p_g}\left[\ln \frac{\frac{p_g(\boldsymbol{x})}{p_{\text{data}}(\boldsymbol{x}) + p_g(\boldsymbol{x})}}{2}\right] - 2\ln(2)$$

$$\min_G V(D,G) = 2\left(JS(p_{\text{data}}\|p_g)\right) - 2\ln(2)$$

So the generator wants to minimize the expression above. We know that any given JSD cannot be negative, as the lowest possible value is 0 which occurs when the 2 distributions are identical. In other words, the minimum of the value function above occurs when $p_{\text{data}} = p_g$.

We have thus proven that at the global minimum of $V(D,G)$, we get $p_{\text{data}} = p_g$, which is equivalent to saying that the distribution of the real data is identical to the distribution of the generated data.

5.4 Training overview

In fig. 5.2, we can see the training process of a GAN. At the beginning of the training process, the distribution of the the generated data and the real data are not overlapping as we can see in the first image. The two distributions being distinct at the start is to be expected since G has not yet received the gradient message from the comparison with the training data. We also see that the probability of the output of D is very random.

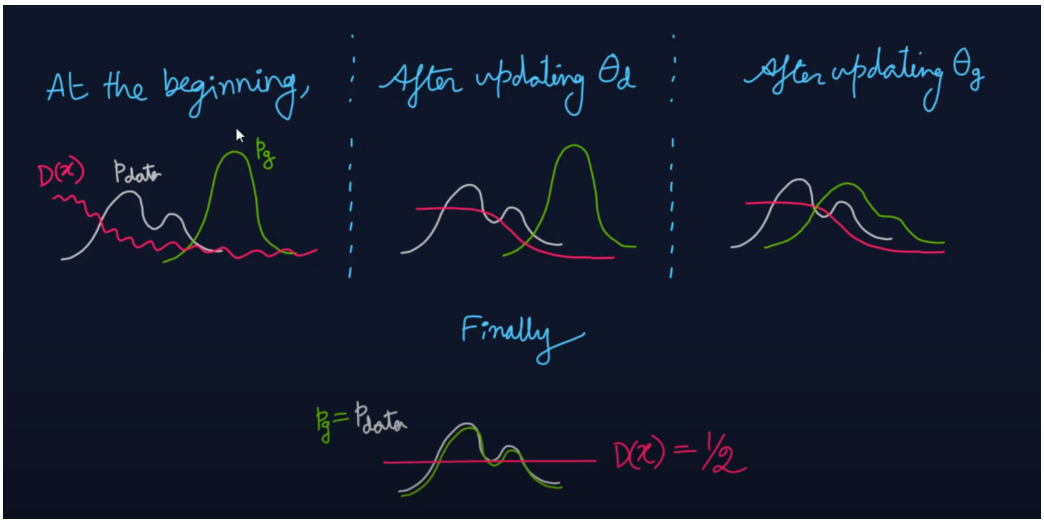

Figure 5.2: Intuitive Representation of GAN Training

As the adversarial interactions between G and D continue and their respective weights get updated, we see that D learns to correctly differentiate between the fake and real data.

As G becomes better at generating "realer" data, the distributions of the real and fake data start getting closer and there is a clear overlap.

Finally, once G is fully trained, it can generate data almost identical to the training data, causing the 2 distributions to fully overlap and making the discriminator unable to discern between real and fake data. This is the GAN's equilibrium state as seen in the bottom image.

5.5 Closing Facts

In conclusion, through their adversarial training, GANs are able to produce really satisfying results on various kinds of generation tasks. While they are most popularly used for image generation, and will be in this project as well, GANs are highly flexible and can be implemented for video, or sound generation as well. This flexibility is rooted in their ability to learn complex and high-dimensional distributions, thus enabling them to capture the underlying structure and patterns of the data, which allows them to generate new and realistic samples of whatever training data is fed.

Moreover, unlike transformers, GANs, in their simplest form, are not the best fit for handling sequential data and can thus struggle with NLP tasks, but there have been examples of variants such as Recurrent GANs (RGANs) and autoregressive GANs (AGANs) that have been able to remedy this weakness to some degree.

CHAPTER 6

A primer on VQ-GAN

VQGAN (Vector Quantized Generative Adversarial Networks)[5] is a generative architecture used for high-resolution image synthesis and will be the crux of the image generation component of this project. By building on already effective image generation models such as VQVAE[12], further optimizing them in a GAN context and leveraging the expressive nature of transformers, VQGAN has yielded groundbreaking results and has cemented itself as the state-of-the-art, standalone method in the field of image generation.

6.1 Image Perception Theory

One of the most fascinating aspects of VQGAN is its interpretation of how we perceive images. As humans, when asked to describe an image, we often use distinct and discrete features or symbols [10], these can include shapes, actions or color descriptions. On the other hand we tend to struggle with continuous and precise representations since we don't perceive images as a succession of pixels.

For example, when looking at fig. 6.1, we would use the terms "dog", "white and brown", "green background" or "grass". A sequence of discrete representations such as "a white and brown dog laying on green grass" allows us to understand

the relationships between the words in the sentence. We understand that "laying on" is the dog's action and "white and brown" is its description. The relationships between discrete symbols are referred to as long-range dependencies [10].

Figure 6.1: long-range Representation: Brown and White Dog on a Green Grass Background

While transformers have made it possible to model these long-range dependencies in sequential inputs, it is not the easiest feat for computer vision models. It is more natural for computers to perceive images as a continuous representation of adjacent pixels as is done in CNNs. Each pixel is represented by a matrix of color values and by assembling these pixels together, an image can be interpreted and reconstructed. This is known as a pixel-based approach which learns visual parts and local interactions [7]. We can see a representation of this in fig. 6.2, where a CNN learns how to compose pixels at varying layers of abstraction: pixels become edges, edges become shapes, and shapes become parts.

In summary, there are two complementary theories of perception, the first one being through discrete symbols and long-range dependencies (using transformers), and second being through local-interactions and continuous pixel representations (using CNNs). VQGAN leverages both these theories by combining CNNs and transformers for high-resolution image synthesis.

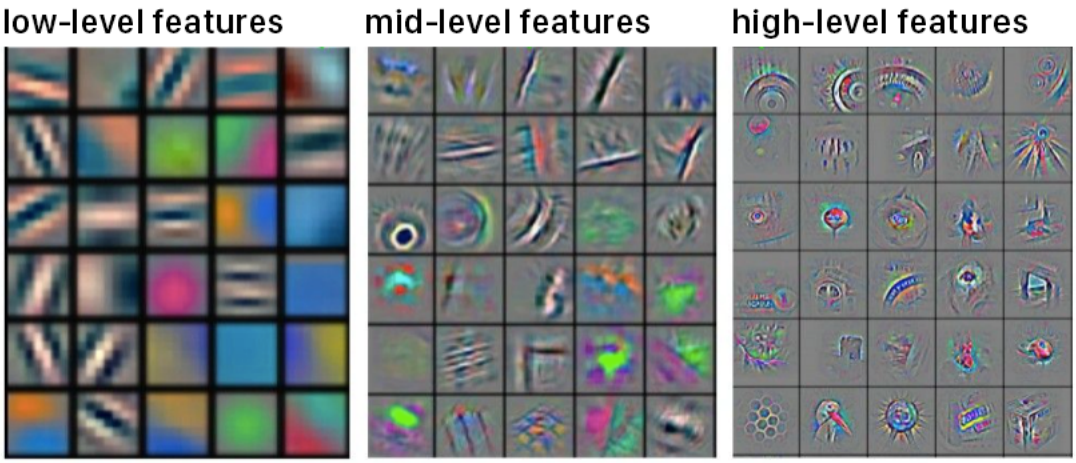

Figure 6.2: A CNN feature map showing features at different levels [10]

6.2 Autoencoders

Before providing an overview of VQGAN, it is important to introduce the various architectures on which it was built, starting with the autoencoder. The autoencoder is a neural network architecture used for tasks such as data compression and pattern recognition. As seen in fig. 6.3, The autoencoder consists of two CNN components, an encoder and a decoder. The former maps the input data to a lower-dimensional representation called a latent code, while the latter maps the latent code back to the original input space. Essentially, if an image is passed as an input, it is encoded as a series of latent codes, which then go through the decoder which is tasked with decompressing and reconstructing the latent codes into an image. The set of inputs which are passed through the encoder form what is called the latent space.

In the context of generative models, the latent space is a concept used to describe a space that is used to represent the structure or the underlying factors that generate the data. It is called "latent" because it is not directly observed in the data, but rather is inferred from the data. For example, in the case of image generation, the latent space might be a 100-dimensional vector, and the data space might be a 32x32x3 tensor representing an RGB image. The generator would take as input a point

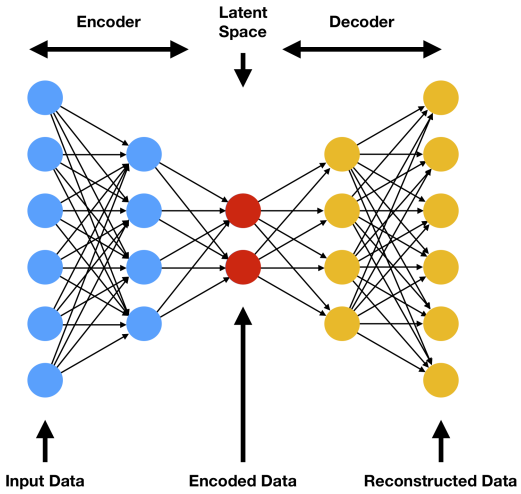

Figure 6.3: Autoencoder Architecture

in the latent space (i.e., a 100-dimensional vector) and output a 32x32x3 tensor representing an image.

Through unsupervised training, the goal of the autoencoder is to minimize the difference between the input and the reconstructed output, this difference is known as reconstruction loss.

Due to the simplicity of their construction, autoencoders present some limitations, the main one being the lack of of control over the latent space. Due to this lack of control, the encoder can take in 2 very similar images and their respective representations in the latent space would be very far apart, which makes it difficult to interpret learned representations of images. This lack of control over the latent space makes it harder to generate new data samples from the given training data. For this reason, the Variational Autoencoder (VAE) was introduced.

The VAE, as seen in fig. 6.4 is a type of autoencoder that adds a probabilistic twist to the encoder's output in order to control the latent space representations and learn meaningful representations. The VAE does this by modeling the distribution of the latent space using a probabilistic approach. Specifically, the VAE assumes

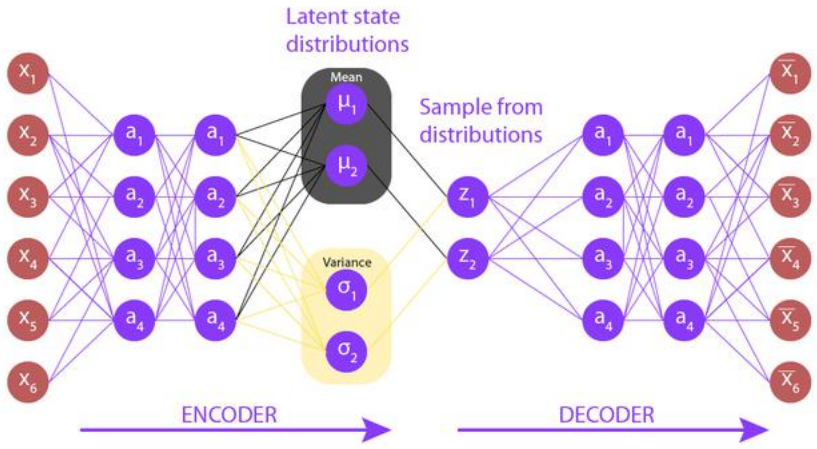

Figure 6.4: Variational Autoencoder

that the latent code is drawn from a Gaussian distribution with a mean and variance learned by the encoder. This distribution can be controlled by adjusting the mean and variance parameters, allowing for more control over the properties of the latent space. Essentially, instead of directly mapping the input to a deterministic latent code, the VAE encoder maps the input to a probability distribution over the latent code. The decoder then maps samples from this distribution to the input space, allowing the VAE to generate new data samples that are similar to the training data.

The VAE training objective is to minimize the sum of the reconstruction loss which was mentioned earlier, and the KL divergence between the latent code distribution and a prior distribution. The prior distribution is typically chosen to be a standard Gaussian distribution, which encourages the latent code to have a simple and smooth distribution.

Compared to a traditional autoencoder, a VAE has several advantages, such as the ability to generate new data samples, control over the distribution of the latent code, and the ability to perform disentangled representation learning, which allows the model to learn meaningful and interpretable representations of the

data. However, VAEs can be harder to train and may require more computational resources than traditional autoencoders.

More mathematically, we can describe the autoencoder and the VAE like so: In a traditional autoencoder, the encoder learns to map an input image x to a fixed lower-dimensional latent representation z, and the decoder learns to map z back to a reconstructed image x'. This can be expressed as:

$$z = encoder(x)$$

$$x' = decoder(z)$$

On the other hand, the VAE treats the encoder as a probabilistic model that maps an input image x to a probability distribution over the latent space, represented by a mean vector μ and a variance vector σ^2. Specifically, the encoder maps x to the parameters of a multivariate Gaussian distribution:

$$q(z|x) = N(z; \mu, \sigma^2)$$

where N is the probability density function of the multivariate Gaussian distribution, z is a sample from the latent space.

The decoder then learns to map a latent sample z back to a reconstructed image x':

$$x' = decoder(z)$$

During training, the VAE aims to minimize the difference between the reconstructed image x' and the input image x, while also regularizing the latent space to follow a standard Gaussian distribution with $\mu = 0$ and unit standard deviation. This is achieved by minimizing the VAE loss, which is a combination of two terms:

- Reconstruction Loss: Measures the difference between the reconstructed image x' and the input image x. This is typically measured using a loss function such as mean squared error.

- KL Divergence Loss: Measures the difference between the learned distribution over the latent space and a standard Gaussian distribution. This encourages the learned distribution to be close to a standard Gaussian distribution, which has several desirable properties such as independence between dimensions and ease of sampling.

The KL divergence loss is given by:

$$KL(q(z|x)||p(z)) = \frac{1}{2}\sum(1 + log(\sigma^2) - \mu^2 - \sigma^2)$$

where p(z) is a standard Gaussian distribution, and the sum is taken over the dimensions of the latent space. Here, the KL loss is used to regularize the latent space, by encouraging the learned distribution $q(z|x)$ to normalize around the origin of the latent space.

The term $(\mu^2 + \sigma^2)$ can be viewed as a representation of the distance between the $q(z|x)$ and the origin of the latent space, and penalizes the learned distribution from being too far away from the origin. Therefore, by minimizing this term, the VAE pushes the learned distribution closer to the origin of the latent space, which results in easier generalization and interpretation of the learned representation.

6.3 VQVAE

The VQVAE architecture is an extension of the VAE architecture that introduces a vector quantization layer between the encoder and decoder. This layer maps each continuous-valued latent vector to a discrete codebook entry, which is a vector from

a pre-defined set of discrete vectors. The vector quantization technique will be covered in more details in section 6.4.4.

As seen in fig. 6.5, the codebook acts as the latent space in the network, encouraging the encoder to learn a discrete representation of the input rather than a continuous one, and forcing the decoder to reconstruct the input using the discrete entries in the learned codebook. The construction and purpose of the codebook will be covered in further details when we formally explain VQGAN in section 6.4.

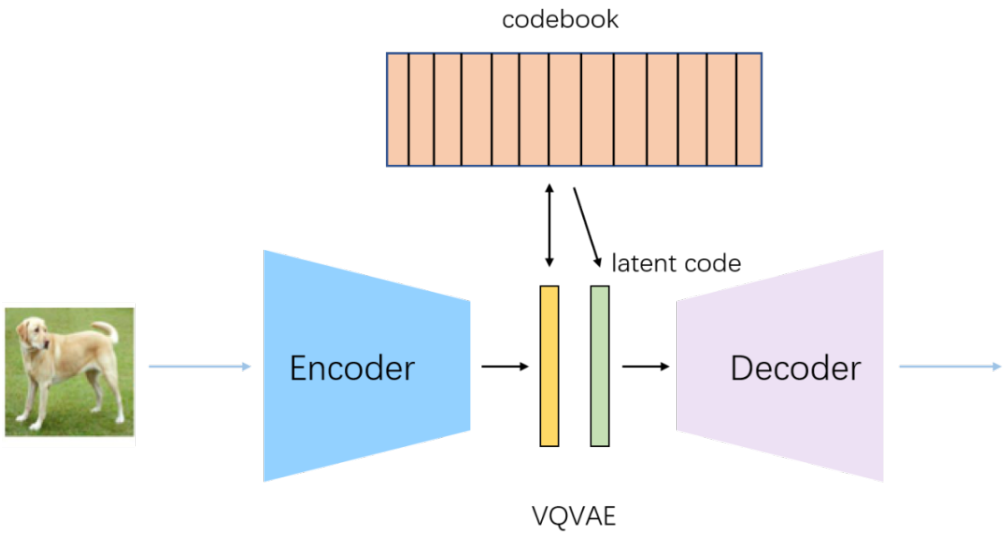

Figure 6.5: Vector Quantized Variational Autoencoder

During training, the VQVAE learns to minimize the same reconstruction and regularization terms (with KL divergence) as the VAE but with an additional commitment loss term [12], which is is used to encourage the encoder to "commit" to its current choice of discrete code. More specifically, it encourages the continuous latent codes to be close to their corresponding discrete code from the learned codebook, which helps ensure that the encoder produces a meaningful and consistent representation of the input image.

The commitment loss term can be formulated as follows:

$$L_{\text{commit}} = \beta ||z_e - \text{sg}(\text{round}(z_e))||^2$$

where z_e is the output of the encoder, which consists of a sequence of continuous latent codes, sg() is the stop gradient operation, meaning that the gradient of the loss with respect to the codes is not back propagated through to the encoder. round(·) rounds the continuous codes to the nearest discrete code in the codebook, and β is a hyperparameter which controls the strength of the commitment loss, similarly to a learning rate parameter.

The commitment loss term encourages the encoder to produce latent codes that are close to their nearest discrete code in the codebook. This can help improve the quality and consistency of the latent representations produced by the encoder, which can lead to better reconstruction performance and a more meaningful latent space.

Overall, By learning discrete rather than continuous representations, which are more structured and more easily interpretable, and by introducing a commitment loss which ensures a better representation of the latent space, VQVAE has shown to be a qualitative upgrade over VAE and has been used as the building block for newer breakthrough architectures such as VQGAN.

6.4 VQGAN

VQGAN replicates VQVAE as it uses an encoder-decoder architecture and a quantized discrete latent space, however it introduces an adversarial context by adding a discriminator. Essentially, it is a combination of VQVAE and GAN. Moreover, VQGAN further distinguishes itself from other image generation methods through

its incorporation of transformers in order to exploit their highly promising sequential learning capabilities [5] and introduce them to high-resolution image synthesis.

Prior to VQGAN, transformers had rarely been used for vision tasks due to the challenge of converting images into sequential data. While there have been efforts to apply transformers for image classification tasks [4], VQGAN has completely reshaped the perception of how transformers can be used effectively outside of NLP.

6.4.1 Overview

From a high-level overview, VQGAN follows a relatively simple process: The encoder, which is a CNN, takes in an image as an input and encodes it. The image encoding is then quantized using vector-quantization, which replaces the image encoding into a sequence of the nearest codebook vectors extracted from a learned codebook. The codebook vector sequence is then passed to the decoder, which is tasked with reconstructing the input image. The decoder is also a CNN. The output of the decoder is then passed to the discriminator which aims to determine if it is real or reconstructed. The process is then repeated until the discriminator can't discern real and reconstructed images.

This is similar to a classic GAN architecture, with the difference being that the generator in this case is made of an encoder, a codebook and a decoder. While the generator may seem identical to VQVAE, there are differences. In VQGAN, the encoder is a pre-trained CNN with frozen weights, which means that during training, the gradient message does not affect the encoder. Another key difference is the fact that there are two separate phases of training. The first training phase aims to optimize the generator, and the second training phase introduces the discriminator network which engages with the generator in adversarial training for further optimization. Finally, the transformer network is introduced after the two training phases in order to learn the long term dependencies between the

codebook vector sequences in order to predict codebook sequences which would be reconstructed into better quality images.

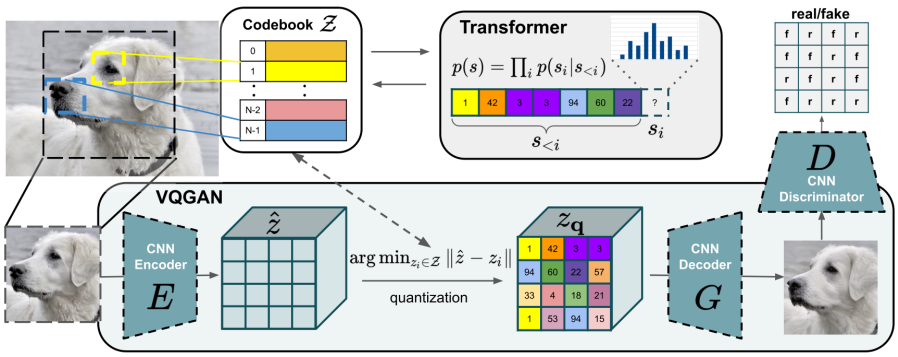

Figure 6.6: VQGAN Architecture

6.4.2 Pre-adversarial Training

The pre-adversarial training phase of VQGAN aims to optimize the generator, which involves the construction and optimization of the codebook and the decoder with what is called the vector quantization loss:

$$\mathcal{L}_{\text{VQ}}(E, G, \mathcal{Z}) = \|x - \hat{x}\|^2 + \left\|\text{sg}[E(x)] - z_{\mathbf{q}}\right\|_2^2 + \left\|\text{sg}\left[z_{\mathbf{q}}\right] E(x)\right\|_2^2$$

To clearly explain this loss function, we can explore each of the three terms individually:

- $|x - \hat{x}|^2$ refers to the reconstruction loss, which just as in VQVAE, refers to the squared difference between the input image X and its reconstruction $\hat{x}$. In other words, it measures how well the generator and encoder are able to produce an output that closely matches the original input.
- $\left|\text{sg}[E(x)] - z_{\mathbf{q}}\right|_2^2$ refers to the commitment loss which is also seen in VQVAE. This term measures the squared Euclidean distance between the output of the encoder $E(x)$ after quantization and the selected codebook vector $z_{\mathbf{q}}$. The aim

being to encourage the generator to reduce the distance between the encoded vectors and the selected codebook vectors.

- $\left\|\text{sg}\left[z_{\mathbf{q}}\right] - E(x)\right\|_2^2$ refers to the the codebook loss, which encourages the model to use codebook vectors that are representative of the input distribution. Specifically, it measures the squared Euclidean distance between the output of the encoder and the reconstructed image after the quantized codebook vector is passed through the generator.

We can also see that both the commitment and codebook loss terms use the stop gradient function, sg, in order to prevent the gradients from flowing through the quantization function, which helps to stabilize the training and improve the quality of the generated images.

The reason for that is because the quantization function takes the output of the encoder, which is a continuous-valued vector, and maps it to the closest codebook vector in a discrete set of possible values, using the nearest neighbor algorithm. In other words, the quantization function is not continuous and therefore not differentiable, meaning that the gradients cannot be back propagated. In VQGAN, the stop gradient function is used on both the output of the encoder, $E(x)$, and the selected codebook vector, $z_{\mathbf{q}}$ to prevent the gradients from flowing back to the quantization function.

This method is not only used in VQGAN and VQVAE, as it is commonly found in neural network architectures which use non-differentiable activation function.

Example 6.1. Given a function $f(x) = x^2$ that we want to minimize with respect to x, we could use gradient descent and compute the gradient of f with respect to x:

$$\frac{df}{dx} = 2x$$

We would then update x by subtracting a multiple of the gradient from its current value, by introducing the learning rate α:

$$x_{new} = x - \alpha \frac{df}{dx}$$

However, suppose we want to modify the function f by passing it through a non-differentiable function, like the stop gradient operation. We might define the modified function as follows:

$$g(x) = \text{sg}(f(x))$$

To compute the gradient of g with respect to x, we would apply the chain rule:

$$\frac{dg}{dx} = \frac{dg}{df} \cdot \frac{df}{dx}$$

However, because the sg operation stops the gradient from flowing through f, the first term in this equation is zero:

$$\frac{dg}{df} = 0$$

This means that the gradient of g with respect to x is simply the gradient of f with respect to x:

$$\frac{dg}{dx} = \frac{df}{dx} = 2x$$

In other words, the sg function eliminates the gradient of g and does not allow it to backpropagate.

Now that the process of the pre-adversarial training has been covered, it becomes clearer that the loss terms used depend on the codebook actually being populated given that an image cannot be reconstructed without the codebook sequence being passed to the decoder. The following sections will cover how the codebook is constructed using k-means clustering and how vector quantization is applied using the nearest neighbor algorithm.

6.4.3 codebook Construction

The codebook is populated using the k-means clustering algorithm, which is a method for partitioning a set of data points into k clusters based on the mean distance between the points and the cluster centroids. The data points in this case are the image encodings that are output every time an image passes through the encoder. The centroid of these image encodings is then inserted into the codebook.

A centroid is a point that is the average of all the points in a cluster. Each cluster has one centroid. One can think of the centroid as the "representative" of a cluster, as it is the point that is closest to all other points in that cluster. The centroid is essentially characterizing the cluster with a single point rather than with all its individual points.

In the context of k-means clustering, the centroid is calculated iteratively. The centroid begins as an arbitrary position and is recalculated at each iteration until the algorithm converges.

Example 6.2. Suppose, X is a set of data points, c_1 and c_2 are the initial centroids, C is the codebook. For each iteration, we assign each data point to the cluster with the nearest centroid. We repeat this until the centroids don't change anymore, indicating convergence. Then we set the centroids of the clusters as the codes of the

codebook.

$$X = \begin{bmatrix} x_1^1 & x_2^1 \\ x_1^2 & x_2^2 \\ \vdots & \vdots \end{bmatrix} c_1 = \begin{bmatrix} c_{1,1} \\ c_{1,2} \end{bmatrix} c_2 = \begin{bmatrix} c_{2,1} \\ c_{2,2} \end{bmatrix} C = \begin{bmatrix} c_1 c_2 \end{bmatrix}$$

1. Assign each data point to the cluster with the nearest centroid

$$\text{for } x \in X :$$

$$d_1 = \sqrt{(x_1 - c_{1,1})^2 + (x_2 - c_{1,2})^2}$$

$$d_2 = \sqrt{(x_1 - c_{2,1})^2 + (x_2 - c_{2,2})^2}$$

$$\textbf{if } d_1 < d_2 :$$

$$cluster(x) = 1$$

$$\textbf{else:}$$

$$cluster(x) = 2$$

2. Re-compute the centroids of the clusters

$$c_{1,1} = \frac{\sum_{x \in X, cluster(x)=1} x_1}{|X|}$$
$$c_{1,2} = \frac{\sum_{x \in X, cluster(x)=1} x_2}{|X|}$$
$$c_{2,1} = \frac{\sum_{x \in X, cluster(x)=2} x_1}{|X|}$$
$$c_{2,2} = \frac{\sum_{x \in X, cluster(x)=2} x_2}{|X|}$$
$$C_1 = \begin{bmatrix} c_{1,1} \\ c_{1,2} \end{bmatrix}$$
$$C_2 = \begin{bmatrix} c_{2,1} \\ c_{2,2} \end{bmatrix}$$
$$C = \begin{bmatrix} c_1 c_2 \end{bmatrix}$$

Once the algorithm converges and outputs a centroid, the codebook is populated with the computed centroids and this process is repeated until the codebook is fully populated. As for the dimension of the codebook, it is a hyperparameter of VQGAN that the user can specify before retraining the model. In the original paper [5], the authors chose a size of 8192 distinct vectors for the final codebook.

6.4.4 Vector Quantization

After the codebook is populated, VQGAN can apply vector quantization to replace the encoded representation of the input image by a sequence of codebook vectors. This sequence is computed by applying the nearest neighbor algorithm which computes the codebook vectors which are most similar to the encoded input image.

Example 6.3. Given an image encoding represented by the continuous latent space z that is represented by a 2-dimensional vector, and a codebook C with two codes

c_1 and c_2. The encoder maps z to the nearest code in the codebook using nearest neighbor search. This is done by calculating the distance between z and each code in the codebook, and selecting the closest code.

$$z = \begin{bmatrix} z_1 \\ z_2 \end{bmatrix} \qquad C = \begin{bmatrix} c_1 & c_2 \end{bmatrix} \qquad c_1 = \begin{bmatrix} c_{1,1} \\ c_{1,2} \end{bmatrix} \qquad c_2 = \begin{bmatrix} c_{2,1} \\ c_{2,2} \end{bmatrix}$$

$$d_1 = \sqrt{(z_1 - c_{1,1})^2 + (z_2 - c_{1,2})^2}$$
$$d_2 = \sqrt{(z_1 - c_{2,1})^2 + (z_2 - c_{2,2})^2}$$

$$\begin{cases} z_{quantized} = c_1 & \text{if } d_1 < d_2 \\ z_{quantized} = c_2 & \text{otherwise} \end{cases}$$

This quantization of the continuous image encoding is the core component of architectures such as VQVAE and VQVAE since it allows for generation of more creative images. This creativity is rooted in the fact that the compression of the continuous latent space into a discrete space causes information loss, which the decoder then has to compensate for, resulting in uncommon and surprising variations.

6.4.5 Adversarial Training

Once the generator is optimized, the adversarial training begins as VQGAN introduces a discriminator network. In this training phase, the generator G, composed of the encoder, codebook and decoder, is trained in competition with a discriminator network D that is tasked with distinguishing real images from

reconstructed ones. D is a CNN that takes an image as input and outputs a binary value indicating whether the image is real or fake.

The discriminator employed in VQGAN is called a patch-based discriminator which works by dividing the input image into smaller patches and feeding each patch through the discriminator network separately. This approach allows the discriminator to capture more detailed information about the image, as opposed to using a single output for the entire image. The output of the patch-based discriminator is a $n \times n$ matrix, where each element represents the discriminator's output for a particular patch in the image.

The training uses a GAN loss defined as follows:

$$\mathcal{L}_{\text{GAN}}(E, G, \mathcal{Z}, D) = [\log D(x) + \log(1 - D(\hat{x}))]$$

where x is a real image from the dataset, $\hat{x} = G(E(x))$ is the reconstructed image obtained by passing x through the encoder E, the quantization function $\mathcal{Z}$, and the generator G, and D is the discriminator network.

The first term in the GAN loss, $\log D(x)$, represents the log probability assigned by the discriminator to the real image x. The generator network tries to minimize this term by generating images that are more realistic and that are more likely to be classified as real by the discriminator.

The second term in the GAN loss, $\log(1 - D(\hat{x}))$, represents the log probability assigned by the discriminator to the generated image $\hat{x}$. The discriminator network tries to maximize this term by correctly identifying the generated images as fake. The generator network tries to minimize this term by generating images that are more realistic and that are more likely to be classified as real by the discriminator.

Following the adversarial training phase, the generative capabilities of VQGAN are further enhanced and the model can be used for high-resolution image generation,

which in theory should produce better results than other models such as VQVAE. However, VQGAN goes even further by introducing transformers in order to learn how to generate codebook sequences that would increase the quality of the reconstructed image even more.

6.4.6 Learning Image Composition with Transformers

Through the quantization function and the construction of the codebook, it is now possible for VQGAN to represent images in terms of a sequence of codebook vectors. And whenever it comes to learning sequential inputs, there are no better architectures than transformers.

In fact, after the codebook is constructed and the generator is optimized. VQGAN uses a transformer network to learn how to represent images as sequences of codebook entries. For a given codebook sequence s with a number of tokens, the transformer is trained to learn how to predict the next token in the sequence. In other words, given the indices of the tokens that come before the current index, the transformer learns to predict the probability distribution of likeliest next indices.

For example, given a token sequence $s = [2, 5, 2, 7]$, where each token represents a codebook vector, to predict the token after the first index (i.e., the value 5), the transformer takes as input the first index (i.e., the value 2) and predicts the probability distribution of possible values for the next index, and so on until the sequence is complete.

The probability distribution is denoted by $p\left(s_i \mid s_{<i}\right)$, which represents the likelihood of observing the next sequence entry s_i given the entries at previous indices $s_{<i}$. The likelihood of the full token sequence is then computed as the product of the likelihoods of the individual indices:

$$p(s) = \prod_i p\left(s_i \mid s_{<i}\right)$$

The transformer loss is expressed as:

$$\mathcal{L}_{\text{Transformer}} = \mathbb{E}_{x \sim p(x)}[-\log p(s)]$$

By minimizing the average negative log probability of the token sequence s given the original image x, the transformer learns to generate token sequences that better represent the original image. The negative log-likelihood is used instead of the likelihood itself because it is easier to optimize and penalizes large errors more strongly.

6.4.6.1 Sliding Attention

One of the challenges faced by the authors of VQGAN was the computational demand and complexity of training the transformer on bigger images and larger sequences. In fact, transformers are known to have quadratic computational complexity with respect to the input sequence length, since for any given sequence element, it has to compute the attention score for all other elements in the sequence. The authors worked around this challenge by using a variant of the attention mechanism called sliding attention.

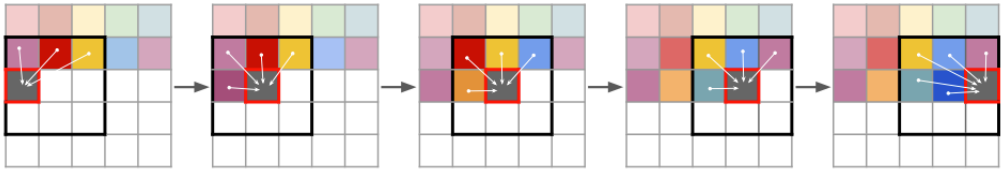

Figure 6.7: Sliding Attention [5]

Sliding attention involves processing an image in a sequence of non-overlapping tiles or patches, with each patch covering a fixed size of the image. This is very similar to how filters are applied to input images in a convolutional layer.

Each patch is processed independently by a transformer network with attention mechanism, and a separate output is produced for each patch. These outputs are then concatenated in order to form the final attention matrix for the output image.

6.5 Closing Facts

VQGAN is a powerful generative model that has demonstrated impressive results in a variety of image synthesis tasks. The model leverages the 2 main image perception theories through its use of transformers to model long-range dependencies, and CNNs to learn a pixel-based approach.

One of the key features of VQGAN is its two-phase training process, which allows it to effectively balance the trade-off between fidelity and diversity in generated images. In the first phase, the model trains an encoder-decoder architecture with a quantization function and a codebook of discrete values. In the second phase, VQGAN introduces a discriminator network which further optimizes the model and enhances the quality of the generator's output.

Furthermore, VQGAN also introduces the use of transformers to learn codebook sequences for image synthesis. This allows the model to take advantage of the highly expressive nature of transformers and to learn more complex relationships between image features and codebook vectors. By using sliding attention, VQGAN is able to avoid high computational cost and efficiently capture short and long-range dependencies in the image while generating codebook sequences, resulting in more coherent and visually appealing generated images.

Overall, VQGAN is a highly effective and creative generative model that has pushed the boundaries of image synthesis. Its two-phase training process, use of transformers, and quantization techniques have shown how combining proven architectures can yield unprecedented results and has opened the door for many

possibilities. As such, it is a valuable addition to the growing field of image generation and holds great promise for future research and development.

CHAPTER 7

A primer on CLIP

CLIP (Contrastive Language–Image Pre-training) [14] is a pre-training method for language models developed by Openai that uses images and their associated text captions to learn representations of language. The goal of CLIP is to learn a general-purpose image-text encoder that can be fine-tuned for a variety of natural language tasks such as image captioning, text-to-image synthesis, and visual question answering. It is considered a groundbreaking architecture thanks to its ability to connect text and images, a feat considered to be very complex prior to the conception of CLIP.

The intuitive idea behind CLIP is to contrast the representations of images and text in a way that encourages the model to learn a common representation of the two inputs. This is done by training the model to predict whether a given image and text pair is "matched" or "mismatched".

In the context of this paper, we will explore CLIP's application in text-to-image synthesis, where the pre-trained CLIP model is fine-tuned by adding an image generator on top of the image-text encoder. The generator is trained to generate images that match a given text description. The pre-trained CLIP model provides a strong prior for the text-image relationship, which can improve the quality and diversity of the generated images. This specific use case is what garnered a lot of

attention towards CLIP, notably with the implementation of VQGAN+CLIP [3] which has yielded impressive results.

7.1 Image and Text Encoding

The first step of the CLIP model is the encoding of input images into a dense vector representation using a cNN. Once the image is encoded, its most important features are extracted and the network outputs a numerical representation of the image.

Following image encoding, CLIP takes on the text encoding task. This is very similar to the previous step, with the main difference being the use of a transformer (or an RNN) rather than a CNN to convert a text sequence into a dense vector representation.

These contrastive pre-training steps can be implemented in different ways as the task of encoding text and images can be done using various pre-trained CNNs and transformers, which gives the CLIP model a lot of flexibility and allows users to test different combinations.

7.2 Text/Image Similarity

Following the image and text encoding steps, CLIP measures the similarity between the text and image encodings using cosine similarity.

Example 7.1. Given an image encoding $I = [0.2, 0.3, 0.5, 0.1]$ and a text encoding $T = [0.6, 0.2, 0.1, 0.5]$, in order compute the cosine similarity between I and T, we first need to calculate the dot product of I and T:

$$I \cdot T = 0.21$$

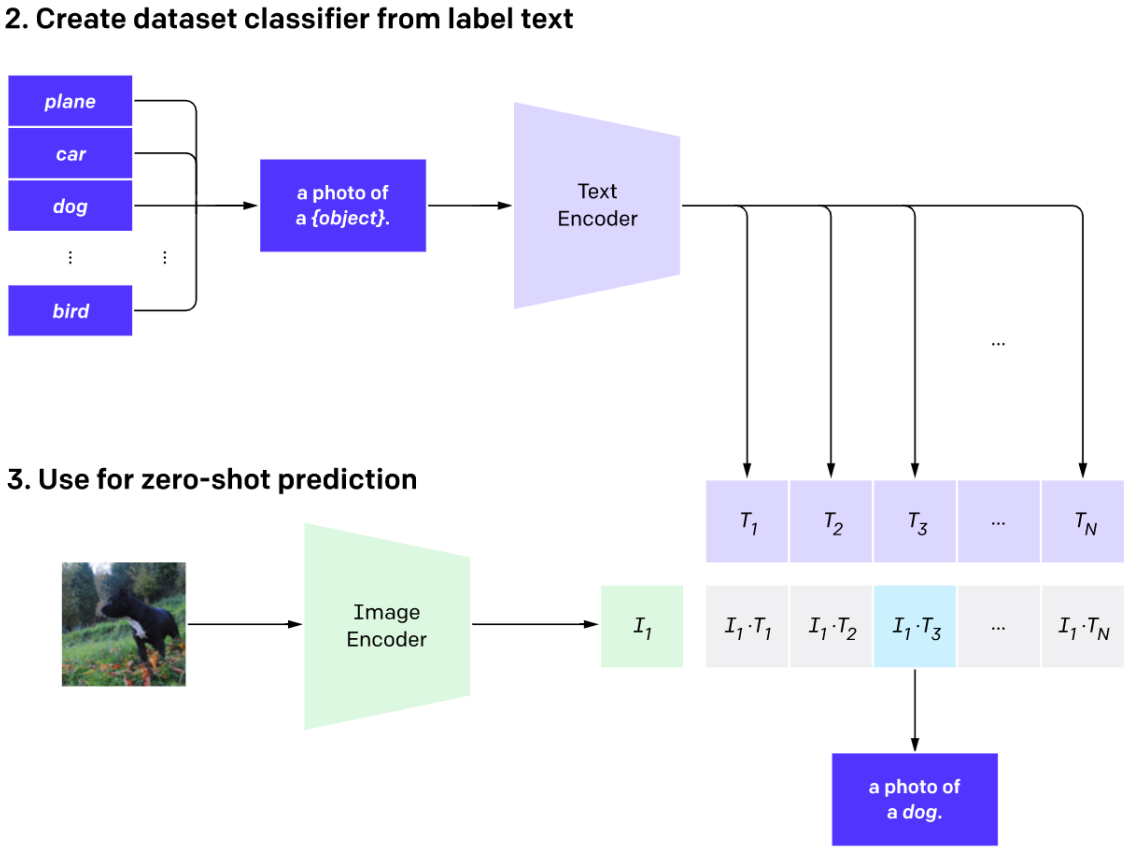

Figure 7.1: Contrastive Pre-Training

Next, we need to calculate the Euclidean norms of I and T:

$$\begin{aligned} ||I|| &= \sqrt{(0.2)^2 + (0.3)^2 + (0.5)^2 + (0.1)^2} \\ &= \sqrt{0.04 + 0.09 + 0.25 + 0.01} \\ &= 0.54 \\ ||T|| &= \sqrt{(0.6)^2 + (0.2)^2 + (0.1)^2 + (0.5)^2} \\ &= \sqrt{0.36 + 0.04 + 0.01 + 0.25} \\ &= 0.71 \end{aligned}$$

Finally, we can compute the cosine similarity using the formula:

$$\text{cosine similarity}(I, T) = \frac{I \cdot T}{||I||\,||T||} = \frac{0.21}{(0.54)(0.71)} \approx 0.546$$

The closer the value is to 1, the more similar the text and image encodings are.

7.3 Combining with VQGAN

In content generation and more specifically image synthesis, CLIP is commonly used in an adversarial context as the discriminator. This is done by coupling CLIP with another generation model, most commonly VQGAN.

In this context, CLIP takes in an image generated by VQGAN and a text prompt as inputs. It then calculates the similarity between the two and sends the result back to VQGAN. VQGAN continues to generate images until the similarity score output by CLIP is small enough. An intuitive representation of VQGAN and CLIP's interaction can be seen in fig. 7.2.

Essentially, the goal of VQGAN is to minimize the difference between a generated image x and a target image y, while the goal of CLIP is to maximize the similarity between a generated image x and a text prompt t.

In summary, CLIP has introduced a new set of possibilities for generative models such as VQGAN. It was originally a model meant to form a connection between images and text and its main use case was to find images that best fit a given caption. By pairing an image generatiion model with CLIP, text-to-image synthesis has been pushed to new heights, as can be seen with the popularity garnered by models such as VQGAN+CLIP or even DALL-E [15] which also uses CLIP.

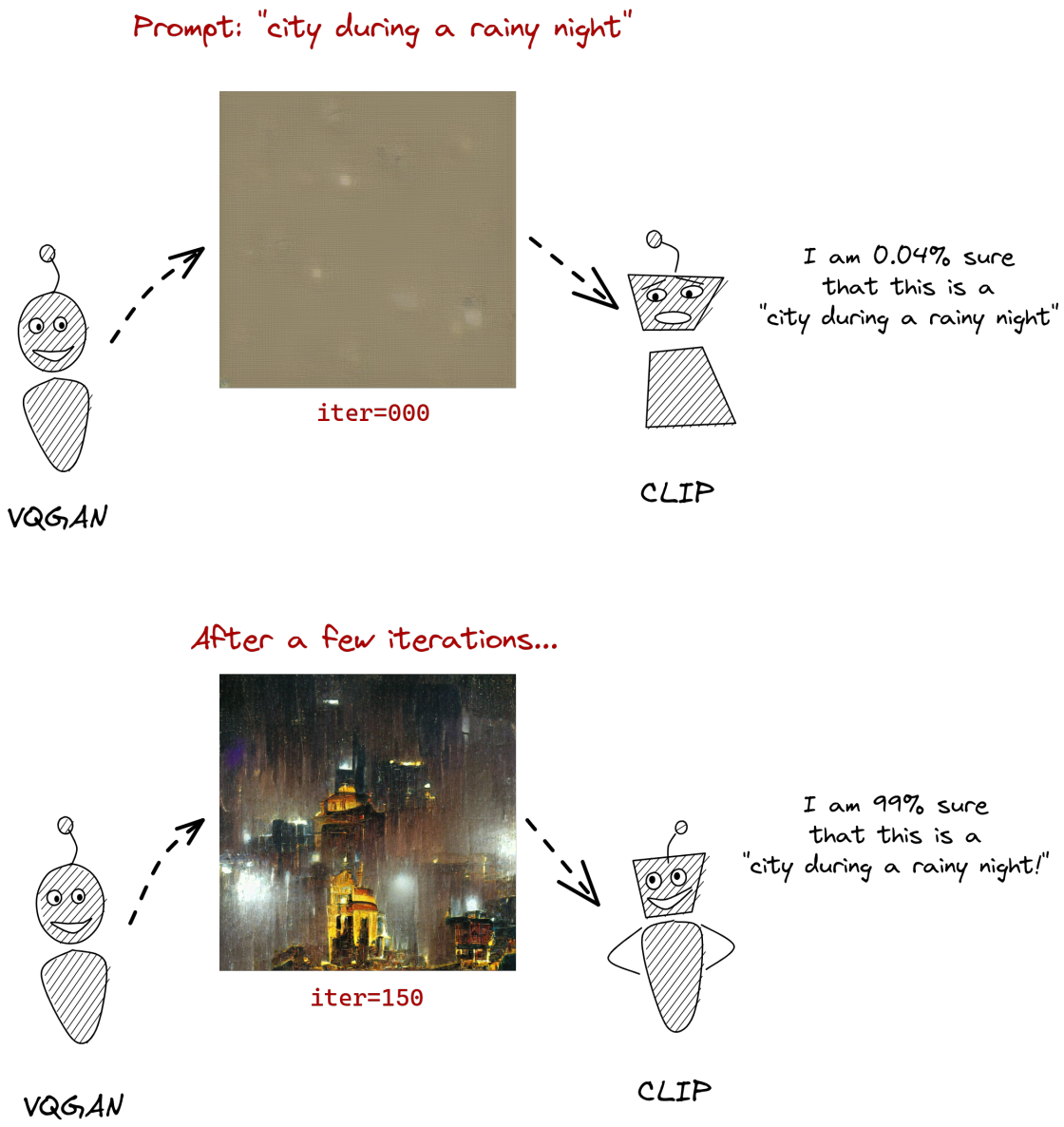

Figure 7.2: Intuitive Explanation of VQGAN+CLIP [10]

CHAPTER 8

Implementation

In this section of the paper, the implementation of the illustrated storybook generator will be covered. This generator will have two main components, each using the generative models explored in previous chapters. The first component is a text generator that uses transformer models and is fine-tuned on training sets containing descriptive text. The text generator will be used to generate short stories that will iteratively be given as inputs to the second component: an art generator using an implementation of VQGAN+CLIP for text-to-image synthesis.

The two components will then be combined in an interface where users can use the generated text to generate illustrations. The output will then be sent to a database which is pulled by and displayed in the interface, allowing users to view the illustrated short story.

8.1 Text Generator

Text generation is a prominent form of AI that has become increasingly popular in recent years. Through their versatility and adaptability to different everyday life situations, text generators have become an incredibly powerful tool. These models can be used in a wide range of applications, including automated content creation,

chatbots and text summarizing, making them very appealing to individuals and companies alike.

However, it is arguable that accessibility is what has been at the root of the rise in popularity of text generators. Models such as chatGPT [8] or T-5 [19] are now accessible to the public and allow individuals with no background in programming to use these text generating programs for various tasks.

Given the purpose of this research, which is the exploration of the capabilities of generative models, the main focus of this section will be to take a look into the possible implementations of text generators and how their results can be improved through transfer learning [25].

8.1.1 Common Implementations

While the implementations of text generators are numerous, with each offering advantages and drawbacks, one architecture that stands far ahead of others for text generation tasks is the transformer.

RNNs have often been used for simplistic text generation or translation tasks, but their limitations were quite clear as they can only interpret and process sequences in an ordered manner and are unable to deal with long sequences due to their inability to preserve older inputs in memory. While LSTMs solve this short-term memory issue and retain older inputs, they still offer the same drawback of always keeping track of the order in the sequence, which is not great for text generation where meaning can depend on more complex contextual relationships.

There have also been efforts of combining a CNN with an RNN or an LSTM to extract and learn long-range relationships by using filters, but CNNs remain much better suited for image classification tasks and struggle with extraction of more subtle contextual information in text sequences.

Transformers overcome all the drawbacks of the aforementioned architectures by

taking the input text sequences all at once, leveraging the self-attention mechanism and retaining the concept of order through positional encoding layers.

Recently, various text=based generative models from the Generative Pre-trained Transformer (GPT) series have used the transformer architecture to generate coherent and contextually valuable text and have yielded unprecedented results.

In the following sections, a possible implementation of a text generator will be presented using the GPT-2 model, a highly advanced and versatile language model with over 1.5 billion parameters.

8.1.2 Transfer Learning

Transfer learning [25] is a technique that involves using knowledge gained through solving one problem to solve another similar problem. In other words, it refers to the transfer of knowledge amongst tasks. This technique is often used in order to "fine-tune" an already existing model with trained parameters. In the context of NLP, transfer learning can be used to retrain a pre-trained model on a new training set in order to achieve results that fit a different task better.

More specifically, given a source task T_s and a target task T_t, transfer learning aims to improve the performance of T_t by leveraging knowledge acquired during the training of T_s.

It is typically assumed that the source and target tasks are related in some way. It is common to represent the source and target tasks as probability distributions $P_s(X, Y)$ and $P_t(X, Y)$, where X is the input space and Y is the output space. We can assume that the marginal input distributions $P_s(X)$ and $P_t(X)$ are similar, but the conditional distributions $P_s(Y|X)$ and $P_t(Y|X)$ may differ.

In other words, $P_s(X)$ and $P_t(X)$ represent the probability of encountering a particular input in the source and target tasks, and it is assumed that these

distributions are similar because the inputs to the tasks are usually similar when it comes to transfer learning.

On the other hand, $P_s(Y|X)$ and $P_t(Y|X)$ represent the probability of encountering a particular output given an input in the source and target tasks, and it is assumed that these distributions may differ since the desired results of the target task are different from those of the source task.

The main appeal of this approach is how it can save time and resources by not having to construct and train a model from scratch to accomplish a new task.

For example, if a text generator can already generate conversational text, but we would like to generate text that resembles the lyrics of a song, transfer learning can be used to retrain the model on a training set which would contain song lyrics, rather than building an entirely new model to generate song lyrics.

Transfer learning involves initializing the pre-trained weights of a model and then training the model on a new training set while updating the weights of the output layers and possibly the last few layers of the model. The earlier layers are typically kept frozen during fine-tuning in order to preserve the knowledge learned from the source task.

8.1.2.1 Data Gathering

When fine-tuning a pre-trained model such as GPT-2, the first step is to gather data that fits the task of interest. For this research paper, the goal is to build a text generator that will then be part of a storybook generator, where the generated text will be fed as a prompt into an art generator. It is therefore more desirable for the text to be descriptive, rather than conversational or informative.

During this process, various training sets that were deemed descriptive enough were gathered, from movie summaries and children stories to painting analysis.

Once the data is gathered, we need to set up the coding environment. Fortunately,

GPT-2 and other related models can be fine-tuned and deployed on various programming languages due to their compatibility with the Hugging Face's Transformers library [23], which provides a unified API for various language models.

While it is possible to use JavaScript or even Ruby for this purpose, Python is the most commonly used option due to the wide range of libraries and frameworks, such as PyTorch and Tensorflow, that are available for ML tasks. Additionally, Python is compatible with the Google Colab environment, which provides a cloud-based environment that allows access to powerful computing resources without the need for expensive hardware, resulting in faster and more efficient training of models.

8.1.2.2 Training

Once the data is gathered and the environment is set up, the transfer learning process can begin by first loading the dataset. Some libraries such as Keras and Hugging Face provide access to datasets but it is more advisable to find custom training sets that fit the given task. The dataset can be loaded directly into Python but it is usually loaded into a Pandas dataframe.

Once the dataset is loaded, the following step involves loading a tokenizer in order to tokenize the vocabulary of the dataset. The vocabulary is the set of all words used in the dataset and the tokenizer is tasked with converting each element into a vector token that can be understood by a language model. This is essentially the text encoding step that is found in transformers.

The Hugging Face library provides tokenzierS which are specific to the used pre-trained model. Since GPT-2 is chosen in this case, its corresponding tokenizer is used.

After the dataset is tokenized, the resulting text tokens are used to create a PyTorch dataset which can then be used for training. This dataset can be modified through truncation, padding or even filtering.

The next step is to split the dataset into training, validation, and test sets. The training set is used to train the language model, the validation set is used to evaluate the model during training, and the test set is used to evaluate the final performance of the model.

The pre-trained GPT-2 model is then loaded through the Hugging Face Library and then retrained on the PyTorch dataset. The training can be optimized by choosing different loss functions, learning rates and training epochs.

Once the training is done, it is good practice to save it locally in order to be able to simply load it at any other time rather than having to rerun the entire training process. Once saved and loaded, the resulting model can be evaluated by generating text and comparing it to the generated text of the model without fine-tuning.

8.1.3 Using the model

The Hugging Face library provides the *generate()* function which generates text from the fine-tuned model. This function works by predicting the next token in the sequence given the previous tokens. It takes in a prompt as input and iteratively generates tokens given the previously generated tokens until a maximum sequence length is reached, or until a special end-of-sequence token is generated. Essentially, this function leverages the self-attention mechanism used in transformers to learn how to predict what word should be generated given the context of the previous words found in the text sequence.

This function can also take in other parameters, such as:

- maximum length: controls the maximum length of the generated text sequence.
- minimum length: controls the minimum length of the generated text sequence.
- temperature: controls the creativity of the generated text. A lower value generates more conservative text that resembles the training data, while a

higher value generates more varied text that may diverge more from the training data.

- topk: controls the number of tokens to consider at each step of the generation process. Only the top k tokens with the highest probability are considered.

- repetition penalty: controls the tendency of the model to repeat the same tokens. A higher repetition penalty discourages the model from repeating tokens that it has already generated.

- length penalty: controls the tendency of the model to generate longer or shorter sequences. A higher value encourages the model to generate longer sequences, while a lower value encourages shorter sequences.

Since the aim of this text generator is to generate a sequence of text which can be used as a short story, to then be fed as a series of prompts to a text-to-image synthesizer, resulting in an illustrated storybook, we have to guide the text generation using these various parameters and some helper functions in order to generate text that fits the required task.

For that purpose, we created a helper function which truncates generated text to make sure it ends on a period. Additionally, parameters such as the maximum length, the number of sequences, and the creativity of the text (e.g., temperature) were left as user inputs for the final generation function.

8.1.4 Results

The capacity of transformers in text generation is truly showcased in these pre-trained models. They are able to generate grammatically correct and coherent text that can be used in all kinds of domains. However, these models also present some limitations. In fact, given the generation process' reliance on choosing the next

sequence through probability, the process is highly non-deterministic, which leads to some instances where the generated text is far from ideal.

Another big area of concern is the potential for bias and harmful language, as these models learn from large datasets of human-generated text that may contain problematic or offensive content. While transfer learning on cleaner training sets can remedy this issue, it does not eliminate it completely given that the knowledge gained through the initial training still remains in the neural network. One way to work around this is to ban the use of certain words during the generation process by hard coding it, but that can cause incoherence and inaccurate grammar.

Finally, the purpose of the text generator was to output text that is highly descriptive in order to generate visually appealing images, however, the generated text does not always fit this criteria and is essentially at the mercy of the given prompt. When given an especially ambiguous prompt, the output text can be incoherent and very much not ideal for image generation.

8.2 Art Generator

The art generator component of the illustrated story book interface uses the VQ-GAN+CLIP architecture hosted through Google Colab. Given the popularity of the model, there have been many implementations on various repositories. These repositories already combine VQGAN and CLIP effectively and are also hosted through Google Colab which is very convenient as it allows users with no programming background to simply write their text prompts and run the entire notebook in order to generate images.

For a more hands-on implementation, it is possible to use the pre-trained models provided by the creators of VQGAN, which can be found at the following repository: *https://github.com/CompVis/taming-transformers.git*, and to then set up the

CLIP model which can be found at this repository: *https://github.com/CompVis/taming-transformers.git*. Using the two models as well as the other dependencies specified in their repositories, a VQGAN+CLIP implementation can be made from scratch. However, this method is very inefficient as it is very easy to have conflicts in dependencies as well as RAM issues that may arise when the models are downloaded. Therefore, it is not advised to opt for this implementation.

For the purpose of this research, we will use a pre-built VQGAN+CLIP implementation, many of which can be found in public Colab notebooks, the most popular ones are gathered at the following page: *https://ljvmiranda921.github.io/notebook/2021/08/11/vqgan-list/*.

These implementations are very streamlined and effective and simplify the process to a few user inputs. In fact, users can input their desired prompt, as well as the dimensions of the image, and in some cases the implementations allow to specify the desired quality of the image as well as the maximum number of iterations, in other words, the number of feedback loops between VQGAN and CLIP. Following that, users can run the notebook and an image will be generated. Given that the generated images are aimed to create illustrations for a short story generated by the text generator, the image generation function parses every sentence and inputs them in an array. Every sentence in the array is then fed to VQGAN+CLIP to generate a corresponding image.

In regards to transfer learning, it is possible to retrain the VQGAN model used on a different image dataset. However, that would require finding a tailor made image dataset, which are not easy to find or compile, and there can always be the issue of images not being in the right format or having inappropriate dimensions, causing issues during the fine-tuning process. However, given that VQGAN+CLIP involves prompt-driven generation, we can simply make sure that every prompt has a prefix or suffix attached to it in order to maintain a desired style. For example,

it is possible to add the prefix "a figurative painting of" to every prompt, which would give similar results to a model fine-tuned on a figurative art dataset.

8.2.1 Results

VQGAN+CLIP's generative capabilities are highly diverse. Through its unique combination of state-of-the-art architectures, the model is able to output artwork that is both creative yet faithful to the given prompt. However, just as with the text generator, the model is not without its limitations.

The model requires really high computational power, and there is a clear balancing act between the quality of the output image and the time it takes to render. This drawback is further exacerbated given that the goal is to generate multiple images for an illustrated story book. Moreover, the non-determinism factor can be seen even more strongly here, as two similar prompts can result in very different outputs if not enough detail is provided. Additionally, it is clear that the more details provided in the prompt, the better the image quality will be, so the output image is once again at the mercy of the input prompt.

8.3 User Interface

Now that both components of the illustrated story book are implemented, the remaining task is to combine them in a user interface. This poses a significant challenge since both models were implemented in Google Colab. In order to use the models to generate images, the user has to manually run the Colab notebooks.

This is an accessibility issue that is commonly found with ML models. In fact, most models are built by people who understand the method behind them and therefore understand how to use them, which presents an issue when sharing them

with users who simply want to get the output. This accessibility issue also makes it harder to collaborate as well as get significant feedback from users.

8.3.1 Gradio Interface

The best way to overcome this obstacle is through the Gradio library [1]. Gradio is a Python library that makes it very easy to generate interfaces for machine learning model. Instead of requiring users to interact with a model through code or a command-line interface, Gradio provides a visual interface that allows users to easily input data, see the results of the model, and explore how changing input values affects the output. Gradio provides a range of in-built input and output components, such as check boxes, sliders, text boxes, and image galleries, that make it easy to build interactive and visually appealing interfaces.

The library is mainly used by calling the *gradio.Interface()* function. This function can generate different kinds of interfaces, as seen in fig. 8.1. For this research, we will use an input(s)/output(s) interface. This interface takes in three inputs: a function, input types, and output types. The function can be of any type as long as there is a defined return type. The input types are essentially the function parameters and the output types are what the function returns. For example, for the text generator, the model generation can be put inside a function which takes three parameters: a prompt, the number of generated words, and the number of sequences. The Gradio inputs could then be a text box for the prompt, and sliders for the number of words and sequences.

Through Gradio, the generated interfaces can be shared using URLs and sent for various users to try. However, these URLs expire after a period of time or cease to work if the computer from which the interface was generated is shut down. This issue can be solved by using Hugging Face Spaces, which is a web-platform allowing users to deploy ML interfaces and have them hosted indefinitely. Through

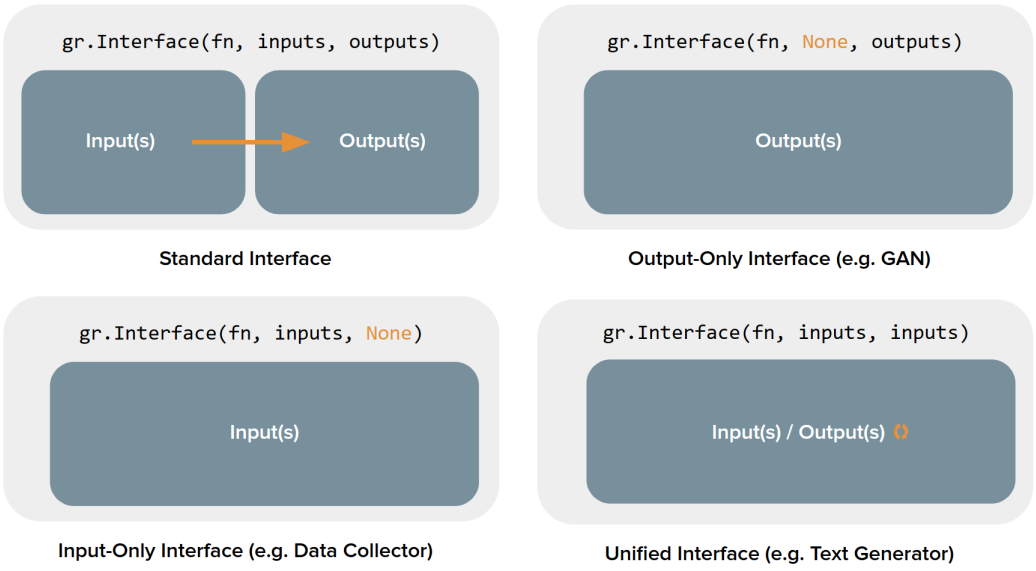

Figure 8.1: Types of Gradio Interfaces

Hugging Face Spaces, the Gradio interfaces which were generated through the Colab notebooks can be hosted without having to constantly rerun the notebooks.

With the models publicly hosted, there are numerous ways of combining them in a user interface. The most straightforward way is to simply create a web page using frameworks such as raw HTML, React or Flask, and embedding the two hosted spaces. This page can then be styled with CSS.

This method does present some drawbacks because the two interfaces are their own separate entities, meaning that manipulating the outputs and inputs of each interface through Javascript or HTML is not possible. This crosses off options such as automatically feeding the output of the text generator to the image generator, or even creating custom buttons that facilitate the user experience. The only manipulations users can perform are through the embedded Gradio interfaces.

8.3.2 Database storage

The final step to complete the interface is to code a way to display the generated images alongside their respective prompts. In order to do so, a database must be

implemented. There are now many ways of storing all kinds of output types in easily accessible database tables. Firebase by Google offers various services such as hosting real-time databases for web development projects, making it a great fit for this task. Additionally, Firebase has an integrated API which allows users to access databases through Python.

After setting up a Firebase account and creating a project, users have access to different database options. For this specific project, it is preferable to use the cloud storage option since it can take in bigger files. Once the storage is set up, it can be authenticated in the Python script through the given tokens, and then, every time an image is generated, we can use the API to push the generated image alongside its prompt to the storage.

This way of implementing storage makes use of the fact that Gradio interfaces are only concerned with the input and output of a function. Anything that happens in between does not change the final result. For that reason, the code which pushes images to the database can simply be added to the image generation function without having to change the Gradio interface at all. Once a user generates an image using the Gradio interface, the image will be pushed to the Firebase storage.

Displaying the content of the storage database can be done using Javascript in the file containing the embedded interface. Using the Firebase API, it is possible to display every image alongside its title. After adding styling elements, the displayed images and captions are displayed as seen in fig. 8.2.

8.3.3 Results

The final interface leverages the Gradio library for ML deployment, Hugging Faces Spaces for live hosting of interfaces and the Firebase Storage API for storing and displaying the generated images and captions. Users can use the story generation interface by inputting a few words, specifying the length in order to generate a

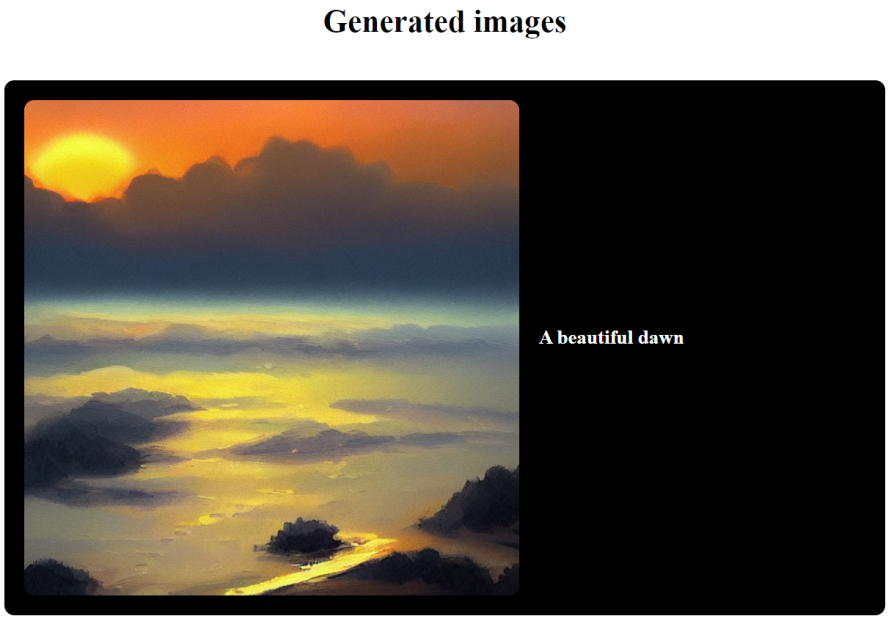

Figure 8.2: Display of Generated Images in User Interface

short story. The story can then be input into the image generation interface, and each sentence will output an image. The images will be displayed in the form of a gallery. However, since the storage used is not a real-time database, users will need to refresh the page in order to view the displayed illustrated story.

While the process is not ideally streamlined, it still accomplishes the task adequately. We could have alternatively opted for a single Gradio interface combining the text generation and the image generation. However, that results in too much unpredictability in results and completely suppresses user intervention.

CHAPTER 9

Conclusion

By mobilizing the tools to generate content which can be applied in various fields, generative models have been at the forefront of a revolution in the ML scene. The sheer ability of these models to create almost human-like text or media is unprecedented and explains the amount of attraction they have garnered.

At their core, these models are based on the neural network architecture, which is inspired by the function of the human brain and its ability to recognize patterns and rewire itself to make better predictions. Through back propagation, neural networks are optimized in order to achieve better results and learn complex and non-linear relationships between inputs and outputs. One of the concepts which underlines the power of neural networks is the universal approximation theorem, which proves that an arbitrary neural network can approximate any function to a certain level of accuracy. Meaning that neural networks can be used to model complex functions, making them a powerful tool for real-world applications.

Despite its undoubted strengths, the basic neural network architecture has clear limitations, which have brought forth variants such as RNNs and LSTMs which aim to overcome these limitations through the addition of a memory component. CNNs are another variant of neural networks that has shown great results in the field of computer vision. Through the use of convolutional layers containing filters

that extract important information from images, CNNs have cemented themselves as a foundational piece of any image detection, classification or generation model.

GANs can be considered a true checkpoint in the history of generative models. By leveraging two neural networks in an adversarial setting, these models have shown the ability to generate reconstructed data that is almost identical to real data. With similarly significant impact, transformers have taken the field of NLP to an entirely new plane. Through the use of self-attention and an ability to take in entire sequences at the same time without losing the concept of order, these models have made complex language task more achievable than ever.

While these models are impressive achievements on their own, combining them for more complex tasks has lead to even greater results, as can be seen with VQGAN+CLIP. This text-to-image synthesis model combines transformers, GANs and CNNs to generate images from a quantized codebook sequence, as well as CLIP's ability to connect text and images to further optimize the prompt-driven generation process.

After exploring the theory behind these models, the ways in which they can be implemented were covered through the implementation of an illustrated storybook generator. This implementation delved into concepts such as transfer learning in order to fine-tune pre-trained models on different tasks and also underlined the accessibility issue that can be faced with some ML models. Libraries such as Gradio and services such as Hugging Face Spaces represent the efforts made to overcome this accessibility issue and allow very easy usage of these complex models. The implementation finally resulted in a user interface containing Gradio interfaces for text and image generation as well as a display of the resulting storybooks.

The topics covered throughout this paper show how much progress has been made in the field of ML and more specifically AI, yet, there seems to be no slowing

down in terms of innovation and it is reasonable to expect much more progress in coming years. Areas where improvements may be achieved include further complexity in models, increased accessibility and quality of datasets or even improved training techniques. However, an area that is often disregarded but deserves attention is the consideration of ethical implications of AI [16].

In fact, as the use of generative ML models becomes more widespread, addressing ethical concerns such as bias and fairness, privacy, and security becomes of increasing importance. Should a model be able to attribute the results to itself, even though its output is derivative of many man-made inputs? How is it possible to monitor the ethics of intellectual property and AI-generated works that mimic specific human creators? Do the drawbacks of AI replacing human workers outweigh the gains?

All these questions are very open-ended but deserve thorough consideration. As the field of generative ML continues to advance, it is essential to focus on developing techniques that can help ensure the ethical and responsible use of these generative models. With careful attention to these issues, the benefits of generative ML can be realized while avoiding potential drawbacks.

References

[1] Abubakar Abid et al. *Gradio: Hassle-Free Sharing and Testing of ML Models in the Wild*. 2019. DOI: 10.48550/ARXIV.1906.02569. URL: https://arxiv.org/abs/1906.02569 (pages 6, 95).

[2] Julius Berner et al. *The Modern Mathematics of Deep Learning*. 2021. DOI: 10.48550/ARXIV.2105.04026. URL: https://arxiv.org/abs/2105.04026 (page 7).

[3] Katherine Crowson et al. *VQGAN-CLIP: Open Domain Image Generation and Editing with Natural Language Guidance*. 2022. DOI: 10.48550/ARXIV.2204.08583. URL: https://arxiv.org/abs/2204.08583 (pages 1, 4, 80).

[4] Alexey Dosovitskiy et al. *An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale*. 2020. DOI: 10.48550/ARXIV.2010.11929. URL: https://arxiv.org/abs/2010.11929 (page 66).

[5] Patrick Esser, Robin Rombach, and Björn Ommer. *Taming Transformers for High-Resolution Image Synthesis*. 2020. DOI: 10.48550/ARXIV.2012.09841. URL: https://arxiv.org/abs/2012.09841 (pages 1, 4, 57, 66, 72, 76).

[6] Ian J. Goodfellow et al. *Generative Adversarial Networks*. 2014. DOI: 10.48550/ARXIV.1406.2661. URL: https://arxiv.org/abs/1406.2661 (pages 1, 3, 45, 48).

[7] Jiuxiang Gu et al. *Recent Advances in Convolutional Neural Networks*. 2015. DOI: 10.48550/ARXIV.1512.07108. URL: https://arxiv.org/abs/1512.07108 (page 58).

[8] Christoph Leiter et al. *ChatGPT: A Meta-Analysis after 2.5 Months*. 2023. arXiv: 2302.13795 [cs.CL] (page 86).

[9] Bernhard Mehlig. *Machine Learning with Neural Networks*. Oct. 2021. DOI: 10.1017/9781108860604. URL: https://arxiv.org/abs/1901.05639 (pages 1, 7).

[10] Lj Miranda. *The Illustrated VQGAN*. 2021. URL: https://ljvmiranda921.github.io/notebook/2021/08/08/clip-vqgan/ (pages 57–59, 83).

[11] Keiron O'Shea and Ryan Nash. *An Introduction to Convolutional Neural Networks*. 2015. DOI: 10.48550/ARXIV.1511.08458. URL: https://arxiv.org/abs/1511.08458 (page 31).

[12] Aaron van den Oord, Oriol Vinyals, and Koray Kavukcuoglu. *Neural Discrete Representation Learning*. 2017. DOI: 10.48550/ARXIV.1711.00937. URL: https://arxiv.org/abs/1711.00937 (pages 4, 57, 64).

[13] Niki Parmar et al. *Image Transformer*. 2018. DOI: 10.48550/ARXIV.1802.05751. URL: https://arxiv.org/abs/1802.05751.

[14] Alec Radford et al. *Learning Transferable Visual Models From Natural Language Supervision*. 2021. DOI: 10.48550/ARXIV.2103.00020. URL: https://arxiv.org/abs/2103.00020 (pages 1, 4, 79).

[15] Aditya Ramesh et al. *Zero-Shot Text-to-Image Generation*. 2021. DOI: 10.48550/ARXIV.2102.12092. URL: https://arxiv.org/abs/2102.12092 (page 82).

[16] Cathy Roche, Dave Lewis, and P. J. Wall. *Artificial Intelligence Ethics: An Inclusive Global Discourse?* 2021. arXiv: 2108.09959 [cs.CY] (page 101).

[17] Robin Rombach et al. *High-Resolution Image Synthesis with Latent Diffusion Models*. 2021. DOI: 10.48550/ARXIV.2112.10752. URL: https://arxiv.org/abs/2112.10752.

[18] Robin M. Schmidt. *Recurrent Neural Networks (RNNs): A gentle Introduction and Overview*. 2019. DOI: 10.48550/ARXIV.1912.05911. URL: https://arxiv.org/abs/1912.05911 (pages 21–22).

[19] Damith Chamalke Senadeera and Julia Ive. *Controlled Text Generation using T5 based Encoder-Decoder Soft Prompt Tuning and Analysis of the Utility of Generated Text in AI*. 2022. arXiv: 2212.02924 [cs.CL] (page 86).

[20] Ralf C. Staudemeyer and Eric Rothstein Morris. *Understanding LSTM – a tutorial into Long Short-Term Memory Recurrent Neural Networks*. 2019. DOI: 10.48550/ARXIV.1909.09586. URL: https://arxiv.org/abs/1909.09586.

[21] Ashish Vaswani et al. *Attention Is All You Need*. 2017. DOI: 10.48550/ARXIV.1706.03762. URL: https://arxiv.org/abs/1706.03762 (pages 1, 3, 35–36).

[22] Thomas Wolf et al. *HuggingFace's Transformers: State-of-the-art Natural Language Processing*. 2019. DOI: 10.48550/ARXIV.1910.03771. URL: https://arxiv.org/abs/1910.03771 (page 6).

[23] Thomas Wolf et al. *HuggingFace's Transformers: State-of-the-art Natural Language Processing*. 2020. arXiv: 1910.03771 [cs.CL] (page 89).

[24] Zhilu Zhang and Mert R. Sabuncu. *Generalized Cross Entropy Loss for Training Deep Neural Networks with Noisy Labels*. 2018. DOI: 10.48550/ARXIV.1805.07836. URL: https://arxiv.org/abs/1805.07836 (page 48).

[25] Fuzhen Zhuang et al. *A Comprehensive Survey on Transfer Learning*. 2019. DOI: 10.48550/ARXIV.1911.02685. URL: https://arxiv.org/abs/1911.02685 (pages 86–87).