FAKE IT TILL YOU MAKE IT:
AN EXPLORATION OF
GENERATIVE ADVERSARIAL NETWORKS

Henry Swaffield

Adviser: Professor Daniel Scharstein

A Thesis
Presented to the Faculty of the Computer Science Department
of Middlebury College
in Partial Fulfillment of the Requirements for the Degree of
Bachelor of Arts

May 2018
ABSTRACT

Generative Adversarial Networks (GANs) are an incredibly exciting approach for efficiently training computers to learn many features in data, as well as to generate realistic novel samples. Thanks to a number of their unique characteristics, some experts believe they may reinvent machine learning. In this thesis I explore the state of the GAN, focusing on the mechanisms by which they work, the fundamental challenges and strategies associated with training them, a selection of their various extensions, and what they may have to offer to the greater machine learning community. I also consider the broader idea of building machine learning systems comprised of multiple neural networks, as opposed to using a single network. Using the state of the art progressive growing of GANs approach, I conducted experiments where I generated painting-like images that I believe to be the most authentic GAN-generated portrait paintings. I also generated highly realistic chest X-ray images, using a progressively grown GAN trained without labels on the NIH’s ChestX-ray14 dataset, which contains 112,000 chest X-ray images with 14 different disease diagnoses represented; it still remains to be seen whether the GAN-generated X-ray images contain clear identifying features of the various diseases. My generated results further demonstrate the relatively stable training of the progressive growing approach as well as the GAN’s compelling capacity for learning features in a variety of forms of image data.
ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to Professor Daniel Scharstein, my adviser for this thesis, for always being a friendly and inspiring mentor ever since I have attended Middlebury College and taken 4 courses with him, but especially this semester as the person who provided much trusted guidance and feedback through the planning, research, writing, and presentation stages of this work.

Thank you, Peter Bromley, for being another GAN enthusiast with whom I could think through ideas and discuss points of confusion.

Thank you, Professor Michael Linderman, for your efforts managing Gattaca, and for your gracious assistance helping not only myself, but many others to use it. Without that machine, I would not have been able to run any of the computationally demanding experiments that I did.

Thank you, Professor Christopher Andrews for so considerately showing me of one of the coolest GAN extensions I read about, and in doing so, opening my mind to future areas of work related to optimizing GANs to produce output with a variety of new qualities.

Thank you, Professor Amy Briggs, my academic adviser, for helping me realize that computer science was the area of study for me.

Thank you, to my friends and family for always being there, and to everyone else who is kind.
# TABLE OF CONTENTS

1 **Introduction**  

2 **Machine Learning**  
2.1 Solving problems with machine learning  
2.1.1 Machine learning for regression  
2.1.2 Machine learning for classification  
2.2 Training machine learning models  
2.2.1 Gradient descent  
2.2.2 Bias, variance, overfitting and the importance of data  

3 **Neural Networks**  
3.1 Neurons  
3.1.1 Biological neurons and the one learning algorithm  
3.1.2 Artificial neurons  
3.2 Feedforward neural networks  
3.3 Backpropagation and neural network gradients  
3.4 Deep learning  
3.5 Convolutional neural networks  
3.6 The elegance of convolutions: edge detection, an example  

4 **Generative Adversarial Networks**  
4.1 Adversarial arms race  
4.1.1 Two may be better than one  
4.1.2 Detectives and art forgers  
4.2 Defining the GAN objective  
4.2.1 A brief discussion of probability distributions  
4.2.2 The cross-entropy loss function  
4.2.3 Discriminator and generator objective functions  
4.2.4 The GAN minimax game  
4.3 Fundamental GAN training dilemmas and solutions  
4.3.1 When the signal is lost: vanishing gradients  
4.3.2 Training instability  
4.3.3 Mode collapse  
4.4 Leveraging the latent space and other uses of GANs  
4.4.1 Conditional GANs  
4.4.2 Latent space logic  
4.4.3 Semi-supervised learning
# LIST OF FIGURES

1.1 Two GAN-generated celebrities that never existed .................. 2  
1.2 GAN-generated portraits of men and women .......................... 3  
1.3 GAN-generated chest X-rays ........................................ 3  
2.1 Model parameters define hypotheses .................................. 8  
2.2 Underfit, generalizable, and overfit models ......................... 13  
3.1 A biological neuron ..................................................... 16  
3.2 An artificial neuron ..................................................... 18  
3.3 Common activation functions ......................................... 20  
3.4 Limitations of linear decision boundaries ............................ 21  
3.5 A feedforward neural network ......................................... 22  
3.6 Convolutions and feature maps ....................................... 29  
3.7 Edge detection example ............................................... 31  
4.1 The probability distribution of possible sums from rolling two dice ... 39  
4.2 The data flow in a GAN ................................................. 41  
4.3 The objectives of $D$ and $G$ .......................................... 45  
4.4 Generator cost functions as related to discriminator performance ... 49  
4.5 Diverse generated output vs. mode collapse ........................ 54  
4.6 Text to image synthesis with conditional GANs ....................... 59  
4.7 Multiple problems solved with the same conditional GAN approach ... 61  
4.8 Latent space arithmetic for meaningful concepts .................... 65  
5.1 Less detailed data is easier to generate .............................. 69  
5.2 The DCGAN architecture ............................................... 70  
5.3 Progressively growing $D$ and $G$ .................................... 70  
5.4 Portraits generated by Jones and Bonafilia .......................... 71  
5.5 Generated paintings without progressive growing at different training iterations ............................................................. 72  
5.6 Generated handwritten digits with progressive growing at different training iterations ........................................................ 74  
5.7 Generated portrait paintings with progressive growing at different training iterations ........................................................ 76  
5.8 Generated portraits selected for subjective qualities .................. 77  
5.9 Generated chest X-ray images with progressive growing ............ 79
CHAPTER 1

INTRODUCTION

Generative Adversarial Networks (GANs) are an incredibly exciting approach for efficiently training computers to become aware of many features of reality, and they may eventually reinvent machine learning (ML) and artificial intelligence (AI), disciplines already growing steadily in influence over much of technology and society.

Yann LeCun, a prolific inventor of ML systems and the chief AI scientist at Facebook, referring to GANs, wrote, “this, and the variations that are now being proposed are] the most interesting idea in the last 10 years in ML, in my opinion” [34].

Perhaps like the origins of many of humanity’s great creative triumphs, the idea of the GAN came to Ian Goodfellow one night, while he was with friends at a bar [48]. Despite being told his idea would never work by those he was with, inspired, Goodfellow went home that night and implemented a working version of his elegant idea before he went to bed. Since then, in 2014, GANs have evolved to take on many forms, with a great variety of capabilities and potential applications, but they all leverage the core of Goodfellow’s initial idea, which was to produce more powerful ML systems by pitting individual ones against each other in a competition where they are each mathematically incentivized to improve [21].

In the GAN’s original form, the competition occurs between two adversaries. The adversaries, or competing players, are distinct ML systems. One is tasked with judging the realism of provided data samples while the other produces fake data with the goal of tricking its adversary, the other player. As time progresses, both players improve such that the GAN’s generated samples are eventually novel, realistic, and ideally varied. The players are most often very sophisticated systems, that utilize many recent advancements from the field of ML, allowing them to learn nuanced patterns and for their interactions to produce impressive results [17, 52, 30].
Figure 1.1: Two GAN-generated celebrities that never existed in the real world, demonstrating the capacity of GANs to create novel, realistic data samples. Achieved by Karras et al. [30], of Nvidia, these results were the first megapixel GAN-generated images to show such great variety and realism, achieved through a process of progressively growing the GAN to work its way up to larger image sizes. Figure from [30].

GANs are unique in the way they learn to represent many aspect of reality, which they can take advantage of to produce novel, realistic data. Typically, this means GANs are provided some real data and then they are used to generate new data that is realistic, as judged by its learned understanding of what realism means. Theoretically GANs can be used with any type of data, though most research in the area focuses on generating images, as does mine here.

Most people are initially shocked when they see how broadly applicable GANs are. To get an idea of what sorts of features a GAN can become aware of and how realistically it can replicate reality, see examples of GAN-generated content in Figures 1.1, 1.2, and 1.3. For now, take these examples as an indication of the power of GANs, which can also be utilized in many ways beyond generating images or data, to be discussed later.
Figure 1.2: GAN-generated portrait paintings of men and women with varying characteristics. These painting-like images were also generated by a GAN, which can detect unlabeled features in data, like the fact that some portraits may come in a round form, and that figures are often positioned at varying angles.

Figure 1.3: GAN-generated chest X-ray images, showing varying positioning of fictitious subjects with distinct characteristics.
To deeply understand the many ways that Generative Adversarial Networks may be transformative, it is useful to first consider the state of machine learning, particularly its broad reach, its most commonly used general strategies, as well as fundamental challenges that practitioners of ML face when they try to apply it to any sort of problem.

2.1 Solving problems with machine learning

Machine learning is a boldly burgeoning area of computing and artificial intelligence that enables the creation of highly-effective systems with applications to a myriad of tasks, within many domains. Machine learning involves creating and refining models to recognize patterns in data, gaining insights into the phenomena the data describe. When this process, called training, is well executed, using abundant and high-quality data, the resulting model not only represents the data it has previously seen, a finite set of specific examples of a phenomenon, but also the abstract rules that describe unseen examples. This ability for models to accurately describe a phenomenon beyond the examples they have been trained on, the ability to learn useful patterns, gives them tremendous utility. What differentiates systems that leverage well-trained ML models from those that do not, is that ML techniques can perceive structure in data that, in many cases, no human would be able to discover or express concisely. This powerful ability enables computers to efficiently recognize and encode the patterns of a phenomenon, even if no human could understand it, and is, in a nutshell, the reason for allowing them to learn, rather than manually specifying how they should accomplish some task.

The defining idea behind machine learning is that automatically learning useful models side-steps the need for a human programmer to explicitly define the logic behind a given task, opening up a world of possibilities for those that are equipped to effectively
employ it. In a sense, machine learning expands the capabilities of people building systems by allowing them to create software that no person could comprehend, let alone create, using only manual methods. No longer must a team explicitly discover and express the logic of some task, but instead they can provide data, specify the inputs and outputs of the system, and let learning algorithms fine-tune how the system solves the task, provided they can mathematically evaluate its performance. Andrej Karpathy communicates the transformational potential of this approach, saying, “[machine learning] is almost a new computing paradigm because normally humans write code but here, in this case, optimization writes code and sometimes it can do it better than you” [49].

Learning algorithms often work through optimizing a metric related to a model’s current or potential performance, and they are widely applicable. ML models are increasingly at the core of many systems in fields that traditionally process abundant data, like finance, but their use also propels new inventions, like autonomous cars, sufficiently safe for mainstream use [33, 4]. A particularly exciting application of ML and one that inspired a portion of my experimentation, to be discussed in Chapter 5, was Stanford ML Group’s recent ML system that supposedly diagnoses patients more accurate than human radiologists, given chest X-ray images [53]. Machine learning is constantly taking on new forms and disrupting new fields, even ones that might be thought of as purely human, like the generation of art, which will be explored here. Thanks to concurrent advancements in software and hardware, the pattern recognition capacity of ML models has steadily improved. Coupled with constantly expanding data availability that results from nearly every aspect of society becoming computerized, the usefulness of ML is blossoming. Some liken its emergence to the advent of electricity [47]. Simply put, machine learning’s many forms represent extraordinary opportunities.
2.1.1 Machine learning for regression

Machine learning models have had the most success answering specific questions for provided examples of input data, as they map input A to an answer B; when this training occurs using labels that provide the correct answers for each example, it is called supervised learning, as opposed to unsupervised learning, which identifies structure in data without such labels [47]. Supervised ML models are simply mathematical functions of widely varying kinds that, for each possible input, provide an output. How a model does that is determined by the its parameters, just as a line’s slope determines, in part, how an input value of \( x \) maps to an output \( y \). ML works by setting a model’s parameters in a way that the function performs the desired task.

Borrowing from Andrew Ng, a simple model to predict the sale price of a given home demonstrates, at a high level, how many ML models work and how they are similar to human experts [45]. To predict the sale price of a home, human experts require data, as they clearly must know fundamental characteristics of the house being considered. Presumably its size and number of bedrooms would matter, and with more information the experts’ predictions would probably improve. As an example, knowing the age of the home is probably more useful than not knowing that information. Not only could that variable determine the price in a discernible way, but it may also enhance the predictive power of the other known features; perhaps having numerous new bathrooms could be more valuable than fewer new bathrooms or as many older bathrooms. By seeing many different homes and the prices they eventually sold for, real-estate experts develop a sense for which features of a home are worth considering, how they relate to each other, and how to compute a price given those trends. In other words, experts learn useful patterns for the purpose of predicting price. Looking to the future, they can more confidently predict an answer than someone who lacks their related experience seeing many examples of home sales. Supervised ML models learn in a conceptually similar
A machine learning model may be a function that takes input data pertaining to a specific example, variables called features, with which it computes a hypothesis, or answer. Models, like a house price predictor, that output answers in a continuous range perform regression. How a model computes an answer is determined by its parameters, the coefficients, or weights, of the model’s input features. These parameters encode the model’s representation of observed patterns, from the ultra-simple, such as the probable trend that square footage and price are positively correlated, to the more nuanced. Models may compute simple linear equations, like $\text{price} = 50 \times \text{s}$ where $s$ is a home’s square footage, or they may be much more complicated functions, possibly involving products and higher powers of features.

In supervised learning, a model’s output for each training example is evaluated against previously-known correct values, or ground truth labels, using a human-specified cost function. Also called a loss function, the cost function provides a measure of performance by quantifying the severity of the errors in the model’s hypotheses [42]. The model’s parameter values are optimized during training such that the cost function decreases over time for the available training examples.

If each tunable model parameter were a dimension in a space then every point in this space would correspond to a different model function, or a different way of mapping inputs to outputs. This space is called the hypothesis space. For any possible assignment of model parameters, corresponding to a point in the hypothesis space that defines a model function, there is a measure of how well the corresponding model performs, some function of the cost function, often simply its negative, called the objective function. In training, the objective is to search the hypothesis space for the best performing model, indicated by finding optima of the objective function.

For a single parameter home price prediction model, $\text{price} = \Theta_0 \times \text{s}$ where $\Theta_0$ is a
Figure 2.1: The relationship between parameters and output for a very simple function is demonstrated to illustrate how a model’s parameters directly affect its hypotheses and error. Different slopes and y-intercepts, the two parameters of a linear model $y = mx + b$, should achieve varying degrees of error, the difference between true data points and the model’s hypotheses. Prediction error for a given example is highlighted in green. Intuitively, a very steep or slight slope would over or undershoot each example and thus serve as a bad model parameter, for the current y-intercept. The goal of optimization is to find the parameter values that minimize error. In picking model parameters that describe the data well, the model encodes the rules that can apply to unseen examples. Figure from [6].

As parameter optimization occurs during the training process, the model develops a more useful representation of how input features relate to one another and how they affect the output answer just as human experts do in their experience. In this way, supervised ML models replace the need for a specific, tailored solution to many problems, by providing strategies for efficient pattern recognition in a variety of data [62]. As discussed, an ML model can effectively learn much of what a real-estate expert learns, and it can similarly eliminate the need for a programmer to figure out how to explicitly solve
other problems, if they can be solved with well-designed models trained on sufficient data, as discussed in Section 2.2.

2.1.2 Machine learning for classification

It is often most practical for machine learning models to express their output not in terms of a number from some continuous range, as in regression, but in terms of a value from a finite set of discrete options. These problems involve taking in an input and classifying to which category that input most likely belongs, and as such, they are called classification problems. The machine learning systems that are used to solve this expansive set of problems, the trained models or functions that classify provided inputs, are called classifiers. These systems are ubiquitous because many questions with real world implications boil down to classification problems. This practical need to express output in terms of discrete values is well demonstrated by spam email detectors, a canonical example of a classifier system. For the purposes of blocking all suspected spam emails, an email client ultimately requires a “yes” or “no” answer, from its spam classifier, when it determines whether or not to block an email. Real value outputs, such as 42.1 and 42.1111 are not only are less convenient, but they give the illusion that there is some relevant quality being assessed in a continuous way, which there is not for the spam filter’s ultimate purposes. There can be continuous output relating to a probability of an input corresponding to a class, indicating a confidence level, but ultimately the answer to a classification problem is a classification for an input, as the most confident classification would be chosen. Similarly, face recognition technology, now present in smartphones, must classify whether the user is the phone’s owner or not, before it can unlock the phone. Expressed as they are here, these two problems only have two possible output values, making them examples of binary classification. When there are more than two classes to consider the problem is called multiclass classification. In
essence, classifier models map input data to another coordinate space where they make classification decisions. How a given input example is classified is determined by where the input is mapped to, in relation to the model’s learned decision boundaries.

2.2 Training machine learning models

The automated process of searching the hypothesis space for the best assignment of model parameters, as measured by a human-specified objective, is called training. While ML model training eliminates the need for humans to explicitly program certain tasks, it requires its own difficult work, designing models and acquiring large datasets.

2.2.1 Gradient descent

While training, a learning algorithm iteratively updates a model’s parameters, a vector \( \Theta \), such that the objective function is optimized, commonly using a form of gradient descent. In the most basic form, this process aims to minimize the cost function, \( J \), reducing the difference of the model’s hypotheses compared to correct values, for the available training examples. Gradient descent is an algorithm for optimizing the objective function, and as the name implies it does this by descending the gradients, or slopes, of the objective function.

Gradient descent works by using the derivatives of the objective function of the model’s current parameters with respect to each trainable parameter, over a batch of training examples, to guide updates to each parameter towards optima. These derivatives provide a signal for how much error on average a parameter is currently causing, whether it should be increased or decreased, and by how much. The process stops when the model’s performance converges or stops changing, at some value. Gradient descent includes a parameter, \( \alpha \), referred to as the learning rate or step size that scales each
parameter update, impacting the time training takes to complete and how well it escapes local minima (minuscule updates may converge at poor optima) [56]. In the original version of gradient descent, \textit{batch gradient descent}, the model produces hypotheses for every available training example per epoch, where each prediction is scored by the loss function, $J(\Theta)$. These error values are summed, and the derivative $\frac{\partial}{\partial \Theta_j} J(\Theta)$ is used as the gradient to guide the simultaneous updates of each parameter:

$$\Theta_j := \Theta_j - \alpha \frac{\partial}{\partial \Theta_j} J(\Theta).$$  

At the local optima of a function its gradient levels out, approaching 0. Gradient descent stops when this convergence occurs, as it notices when subsequent iterations no longer yield improvements.

Now that it is common to train ML models on immense datasets, it can be impractical to only update a model’s parameters after evaluating error on a batch of all training examples. The other end of the spectrum from batch gradient descent is \textit{stochastic gradient descent}, or SGD, that works by randomly shuffling the order of training examples and updating $\Theta$ after evaluating the gradient after each individual example [45, 56]. \textit{Mini-batch gradient descent} is in between SGD and whole batch gradient descent, as it computes gradients and updates $\Theta$ after evaluating a subset of training examples, of batch size $b$. Mini-batches allow for parameter updates $\Theta$ at a high frequency for more quick training than batch gradient descent, while still averaging the gradients over multiple training examples, potentially providing more useful signals per model update [45].

Training some types of models whose gradients are much more difficult to approximate can be extremely difficult as they sometimes flatten, or vanish, before a satisfactory optimum is reached, as they often do for deep neural networks and GANs, to be discussed later.
2.2.2 Bias, variance, overfitting and the importance of data

Ironically, some models that perform very well on their training data are actually useless, and their low loss function score, for training set examples, actually indicates that they were trained poorly. Often resulting from having too many model parameters and too little data, overfitting is a fundamental problem in machine learning. Overfitting occurs when a sufficiently complex model has enough parameters to merely memorize the training set, and its noise; in this sense, overfitting occurs when optimization works too well, and the models it produces have too much variance, or variation in output for similar inputs [13, 15]. Overfitting produces models that are useless because they do not represent generalizable patterns that would apply to novel examples, the goal of machine learning. That said, it is of course desirable to model complex patterns in data, and so it is also important to avoid underfitting, which produces models that have too much bias, or overly simplistic representations of the training data. Bias and variance are often thought of as representing a foundational trade-off in machine learning, following from the intuition that complex models with high variance can easily overfit, or memorize specific training data, while less complex models simply don’t learn as well and thus are less likely to overfit, but more likely to underfit, as shown in Figure 2.2.

Occam’s Razor says that simpler hypotheses may better explain fundamental patterns of phenomena with fewer false assumptions, and regularization is a machine learning approach to reduce excessive complexity [59, 56]. As an example, gradient descent can include an additional term that penalizes a model’s complexity, a scalar hyper-parameter $\lambda$, that weights the relative importance of minimizing the cost function and simplifying the model [64]. The update rule for a general example of regularized gradient descent is

$$\Theta_j := \Theta_j - \alpha \frac{\partial}{\partial \Theta_j} J(\Theta) + \lambda \frac{\partial}{\partial \Theta_j} P(\Theta),$$

(2.2)

where $P$ abstractly represents a function penalizing increased model complexity, and $J$
Figure 2.2: Given the same dataset for training, more complex models are better able to fit the data they are exposed to, but if they are too complex the training process will not discover useful, generalizable patterns, but instead it will overfit, or merely memorize the data. If a team already has labels for their dataset it would not be useful to create an overfit model that merely echoes their own data. Similarly an underfit, low-accuracy model, that does not describe general trends well, would also lend little use. Figure adapted from [29].

represents the model’s cost. Regularization is employed in a variety of ways and may prevent overfitting, but it does not provide a perfect solution to this trade-off.

The trade off of a model’s expressiveness and predictive power highlights the importance of having a large and representative dataset, on which a complex model could learn highly-nuanced yet generalizable patterns [56]. Unfortunately, having sufficient data is almost always an issue. To build the best ML systems having more training data never hurts, but gathering this data, unlike many other challenges related to machine learning, potentially requires immense expenditures and collaboration with many parties in the external world. Even if large datasets are available, labeling a sufficient number of training examples, potentially millions, can be infeasible, as the process often involves individuals labeling training examples, one by one.

Generative Adversarial Networks are a very unique approach to ML. The way they learn, as well as what they can create, allows them, in some cases, to mitigate or bypass the fundamental challenges that come with collecting a large, labeled dataset. Addressing the challenges associated with acquiring sufficiently large and varied training sets, GANs have been used to generate novel training data [61]. GANs have also been used
through a process called *semi-supervised learning* to almost completely eliminate the need for labels, when training classifiers, in some cases reducing the required number of labeled examples by over 99%, discussed more thoroughly in Section 4.4 [58, 19].
CHAPTER 3

NEURAL NETWORKS

Ostensibly similar to the systems in our biological brains, artificial neural networks (NNs) are a broad class of ML models that have had great success in many areas. Though their use still requires relevant problems to be expressed in terms of optima, their complexity and flexibility differentiates them other learning systems [25]. Notably, NNs can learn and utilize abstract representations of raw data, eliminating the costly need for human experts to manually engineer features (if there were a suitable expert at all) [35]. Advancements in data availability and computational power have made this approach, representation learning, even more powerful, as the neural networks themselves have improved. Convolutional neural networks are one of the greatest innovations in the history of neural networks, making possible many computer vision tasks. The NN tools discussed here are not only much more capable pattern recognizers that are taking over ML, but they are also the models that are combined to form the most powerful compositions, like GANs.

3.1 Neurons

Neural networks are truly extraordinarily varied in size, shape, and workings, though they all harness the power of individual learning units latticed together to form more powerful learning systems.

3.1.1 Biological neurons and the one learning algorithm

Biological brains, which many human brains consider to be capable learning systems, are made up of networks of nerve cells. Nerve cells, or neurons, receive and send electrical signals. In addition to enabling the transmission of sensory information, mo-
Figure 3.1: Biological neurons receive and send signals, firing when they have received sufficient signal to meet the threshold of an action potential. Figure adapted from [1].

tor control, and other vital systems, neurons are the fundamental working unit of our brains [2]. The numerous parts of the brain, which specialize in specific tasks, are each made up of neural networks [2]. As outlined in Figure 3.1 neurons are elongated cells that receive input signals from and transmit signals to other neurons in a directed way, via dendrites and axons, respectively. They only transmit when they have a sufficiently strong signal.

Incredibly, the brain’s application-specific regions are flexible enough to process new forms of data such that they can learn and mimic the functionality of other regions, suggesting that there is a single learning process, or algorithm, embodied in the physical processes of our brains [24]. Biological neural networks might even be powerful enough to learn to do anything, even if they must go against millions of years of evolution that perfected them for some other function. Among other examples, the work of Melchner et al. [70] supports this exciting idea. They rewired the brains of ferrets such that visual information, signals from the eye’s retina, would be sent to the auditory thalamus, which naturally processes sound, and found that the ferrets could still see, though less well.

Among other incredible ideas stemming from findings like these, is the one learning algorithm hypothesis, the idea that there exists an all-purpose learning algorithm, physically manifest in the workings of biological brains, and potentially able to be run
Jeff Hawkins, the founder of Palm Computing and now a neuroscience researcher, believes the path to powerful artificial intelligence is to understand this learning algorithm, or what he calls the neocortical algorithm. He writes: “as long as we can decipher the neocortical algorithm and come up with a science of patterns, we can apply it to any system that we want to make intelligent. And one of the great features of neocortically inspired circuitry is that we won’t need to be especially clever in programming it” [23, p. 62]. If researchers can develop a sophisticated enough understanding of the one learning algorithm, if it exists, then all that is required for AI is implementation. Luckily, when designing neural architectures, we can look to nature for ideas, as biological neural networks are everywhere. Biological neural networks provided new training data can evidently learn a variety of tasks, and the computational models they inspire can too.

### 3.1.2 Artificial neurons

Artificial and biological neurons have a few characteristics in common. Like biological neurons, artificial neurons, which will be referred to as neurons, are also organized within networks and similarly have directional input and output signals. Artificial neurons must be activated by input signals, because as functions they require an input to produce an output, like a biological neuron only fires when it has received signals. Also like their biological counterparts, artificial neurons may have input and output signals from and to numerous other neurons. Their networks can recognize patterns and learn tasks. While they do not exist in physical space, like biological neurons, they can also be thought of as having similar shapes.

Beyond these abstract similarities, it is not clear that networks of artificial neurons fundamentally capture the learning power of our brains, which are themselves not well understood [24]. That said, even if the engineers that build NN systems know little
Figure 3.2: An artificial neuron’s activation is calculated by summing its inputs multiplied by the neuron’s corresponding parameters, to produce a net input, which is then passed through the activation function. The output of the a neuron, or its activation, is the output of its activation function. Figure from [57].

An artificial neuron is an individual learning unit, with numerical input values, analogous to signals received from biological dendrites, parameter weights, and an output value, analogous to those sent through biological axons. For each input signal a neuron receives, the neuron has its own parameter weight to scale it by. Inputs may come directly from raw data, such as pixel color values in a digital photo, or the output of a previous layer of neurons, which will be discussed later [59, 45]. These trainable parameter weights serve the same purpose as the model parameters of other machine learning models. A neuron’s output, or activation, is computed by multiplying each input value by its corresponding weight parameter, summing them, and then passing the sum through an activation function, as illustrated in Figure 3.2 [67, 45].

Biological neurons do not fire continuously, only when they have received enough signal to surpass a threshold, and artificial neurons, being superficially similar, have functions of inputs to determine their output activations, activation functions [67].
tivation functions simulate the logic of a biological neuron, regarding whether its inputs surpass a threshold to fire a signal. In actuality, most activation functions do not give a binary output, but some are steep curves that, for most input values, output values near distinct limits, like 0 and 1, or -1 and 1 [67].

Activation functions perform a few important roles that are essential for a neuron’s usefulness to a neural network. Considering the output of numerous neurons, it would be difficult for a network to know which neurons should have the most say in a decision, if they were not all on the same scale. Activation functions are a way a neuron ensures that its output remains in a specific range, such as $-1$ to $1$, as specified by the function being used [39].

As previously discussed, much of ML model training relies on optimizing parameters, with respect to the gradients of cost functions. As will be discussed more thoroughly in Section 3.3, a neuron’s parameter updates rely on the optimization procedure being able to take derivatives of the models, and so it is important to use differentiable activation functions or functions whose slopes can otherwise be known, with common examples shown in Figure 3.3 [35]. Non-linear activation functions are used because they allow neural networks to approximate more complex functions than linear functions could, as discussed next.

### 3.2 Feedforward neural networks

As alluded to before, the activation functions being used are non-linear. This is important for training useful NN models because if each neural unit in the network used linear activation functions then the whole network itself would reduce to a linear function of its inputs, as a combination of other linear functions [20, 67]. A primary motivation for using neural networks, as opposed to a linear model, is that they encode sophisticated, nonlinear decision boundaries that can learn relationships between data that linear func-
Figure 3.3: Three commonly used activation functions: Sigmoid, TanH, and ReLu. They are all non-linear, while Sigmoid and TanH are differentiable, allowing the neural networks that use them to learn complex patterns through gradient descent. ReLu is not actually differentiable, but thanks to its simple nature, in practice, an optimization algorithm can always compute its gradient, because it is just one of two defined slopes, depending on the value of $x$. ReLu, or the rectified linear unit, activation function has been most commonly used recently for its simplicity and training performance [35]. The TanH, or hyperbolic tangent, activation function is used in a wide array of tasks, and though very similar to the sigmoid function, it has a steeper gradient and it performs better in most tasks [67, 45]. The Sigmoid, or logistic, activation function performs less well than TanH and ReLu, except for NNs performing binary classification [45]. Figure from [39].

Layering neurons into neural networks enables models to learn more sophisticated patterns than their individual neuron units could learn alone. **Feedforward neural networks** are models whose inputs pass through multiple non-cyclical stages, or layers, consisting of neurons that transform the output of the previous layer, in parallel [59, 20]. The first layer of a network simply feeds raw input data into the network, for which it is called the *input layer*. The output of the network comes from a layer of neurons that...
perform one last activation of the computations of the model, and as such it is called the output layer. In a sense, the input and output layers of a NN are the most clearly visible and comprehensible, as they output human-recognizable data, either input or a model’s final output. They are also directly interacting with the outside world because they define the form of the function’s inputs and outputs, and as such these layers are specifically designed to fit the external needs of an ML system. The vast majority of learning, however, occurs in layers between the input and output layers, a sequence of hidden layers not directly visible to the outside world and whose output is not suitable for the needs of the whole network [20]. Feedforward neural networks pass the activations of layers of neurons forward through a defined sequence of such hidden stages, of which there can be a varying amount from one to more than 100. As a network learns, the hidden layers are essentially defining useful features for subsequent layers, notably the output layer, so they can make decisions with increasingly abstract considerations [62].

In a NN with $L$ layers, the $l$-th layer is comprised of specific neurons, made unique by their differing parameters. During forward propagation, each neuron in layer $l$ takes in the activation of the previous layer in parallel, a vector $a^{l-1}$, and outputs a scalar, its own activation [20]. The activation of the $l$-th layer, $a^l$, is simply a vector of the activation outputs of each of its neurons where the activation of the $i$-th neuron in layer $l$ is stored in the $i$-th index of the activation $a^l$, $a^l_i$. 

Figure 3.4: Recognizing some conceptually simple relationships between features, such as XOR, is impossible for a model that can only use a linear decision boundary on input features. Figure adapted from [11].
Figure 3.5: A feedforward neural network with 3 input values, 1 hidden layer, and two output values. This neural network is fully connected; each neuron of a layer receives and weights every output of the previous layer. $W_{41}$, $W_{42}$, $W_{43}$ highlight the trainable weights that neuron 4, the neuron at the top of the hidden layer, has to scale output from neurons 1, 2, and 3, of layer 1. Each neuron has weights performing the same role, though they are not highlighted. $z_4$ represents the net input, or weighted sum of the inputs from layer 1, and $a_4$ is node 4’s activation. The input propagates forward through the network until it is output by the final layer, giving the final output for the whole function. Figure adapted from [68].

As shown in Figure 3.2, each neuron multiplies its input by its own parameters, takes the sum, and passes that value through its activation function. Formally, for the $i$-th neuron in layer $l$, with its activation function $g_l^i$, its weighted input $z_l^i$, its parameters, $\Theta_l^i$, and the previous layer’s activation, $a_l^{l-1}$, its output activation is:

$$a_l^i = g_l^i(z_l^i) \text{ with } z_l^i = \Theta_l^ia_l^{l-1}.$$  \hspace{1cm} (3.1)

Input values are propagated forward through the network, until they are eventually output by the final layer, $a^L$, demonstrated in Figure 3.5. Clearly, neural networks can quickly become complicated functions. It may seem difficult to calculate useful derivatives, for the purposes of optimizing the model but luckily backpropagation provides a tried-and-true way to efficiently compute the gradients [62, 59, 20].
3.3 Backpropagation and neural network gradients

To use gradient descent to train a neural network, the optimization procedure must take the derivatives of the model’s objective function, with respect to each neuron’s trainable parameters, to guide the model’s parameter updates. Because a NN involves many learning units, each of which contribute to the model’s output, there are many sets of parameters that are contributing differently to the network’s overall error. The suboptimal assignment of model parameters in one layer, resulting in error, affects all others, in a sense. In a network with many hidden layers, the activations of neurons in earlier layers propagate forward through numerous subsequent layers, which themselves likely contain neurons with sub-optimal parameter assignments. A dramatic example could be the output layer of a NN exaggerating the errors from earlier layers, by making a classification decision based on information from a subtle, previous error. In this way, the error attributable to one neuron is a function of its own imperfections as well as the flaws of all subsequent layers, which may transform its error. Neural networks are composite functions, meaning they are composed of functions, in sequence, that operate on the output of a previous function, or layer’s activation. The chain-rule in calculus allows for the calculation of derivatives of composite functions, by chaining together the derivatives of the functions of which it is composed. For example, if \( y = F(x) = P(G(x)) \), and we denote \( z = G(x) \), then

\[
\frac{\delta y}{\delta x} = \frac{\delta y}{\delta z} \cdot \frac{\delta z}{\delta x}.
\]

(3.2)

Similarly computing derivatives in terms of the products of others, backpropagation is an algorithm for computing the partial derivatives of a neural network’s cost function, \( J \), with respect to its parameters. If one can compute the gradients of a model with respect to its trainable parameters, then its representation of the data can be improved through parameter updates, as discussed in Section 2.2.1 [50]. Incredibly, backpropagation works sufficiently well in the real world to train even extremely complex NNs,
including GANs.

During training, after a network computes a hypothesis for an input example, through forward propagation, the error resulting from each neuron is computed going backwards, giving the name backpropagation, or backprop for short. The error attributable to a neuron \( n \) is computed by summing the gradients of the neurons in next layer, multiplied by corresponding model weights of the neurons in the next layer (the weights that scaled \( n \)’s activation during forward propagation). These values are also multiplied by the gradient of that neuron’s activation function. For a NN with \( L \) layers, the first layer for which error is computed is the output layer, layer \( L \), giving the whole error of model, or how far away the NN’s prediction is from where it should be, as indicated by the training set labels [45]. Next, backpropagation computes the gradients of error resulting from the nodes in layer \( L-1 \), and continues backwards through the whole network. The result of this process is that the partial derivative of the cost function with respect to each neuron’s weights is computed, allowing for gradient descent to be used. This process enables the training of NNs even with many hidden layers, increasingly dominant model functions in the field of machine learning called deep neural networks [59].

### 3.4 Deep learning

Further fueling the hope that artificial neural networks may be universally applicable, in addition to the superficial similarities between artificial and biological neurons, is the fact that artificial NNs can theoretically learn any pattern, just as proponents of the one learning hypothesis might argue our brains have a single algorithm responsible for all learning.

Mathematically, it is proven that there exists a neural network using non-linear activation functions that can compute any continuous function, even if the set of considered networks is restricted to those with a single hidden layer [67, 50]. In other words, re-
gardless of the function being modeled, if we use enough neurons we can always build a network that approximates it arbitrarily well [50]. This means if there is a rich-enough dataset describing some phenomenon and a complex enough NN model being used to model it, the model can learn patterns no human has ever perceived, as it approximates the behavior of the underlying function, some coherent set of rules that dictates the relationship between inputs and outputs. If any function can be computed with a neural network, why couldn’t the workings of biological brains be modeled? This universality theorem applies even if a function has many inputs, and a neural network to represent it must have at most one hidden layer [50]. So why make models with many hidden layers?

Deep neural networks, NNs with a large number of layers, are preferable to shallow networks because of their ability to more abstractly represent hierarchical models of features [21]. They may more quickly and efficiently recognize patterns in data, initially basic features, on which they build additional levels of increasingly specific features. To illustrate this concept, a deep classifier on image data might compute the most fundamental features in the first layers, such as edges, followed by layers that identify shapes, aided by previously identified edge features, and eventually very specific features that can inform decisions based on even richer representation of the input, e.g., concentric circles inside an oval finally being recognized an eye [45]. Hierarchical representations are useful for processing many forms of data, beyond visual data [45, 21]. This should not come as a surprise because most everything in existence is composed of some sort of subcomponents, and similarly could be represented with varyingly precise descriptors. Data coming from the physical world should logically share those properties.

Another way to phrase the advantage of deep NN representations, inspired by circuit theory, is the fact that they much more efficiently use a given number of neurons. To compute the same function, shallow networks may require exponentially more neurons
than their deeper counterparts, which have the advantage of building up increasingly complex features, rather than needing to do all work with a single layer [45].

Intuitively, this should not be surprising, as answering numerous smaller and related questions can be more efficient than tackling a single larger question. For example, the function taking the XOR between $n$ binary inputs, equivalent to taking the set’s parity, can be computed by enumerating and evaluating all $2^n$ pairs of input values, or it can be equivalently expressed with $\log(n)$ comparisons, much more efficiently using each neuron, if multiple layers are allowed [46]. As previously discussed the deep learning approach works by first answering smaller questions on which it builds more informed answers. For this parity problem that means a NN could compute the parity first of pairs, then of groups of 4, then groups of 8, and eventually the whole set, as it combines two outputs to double the size of the set considered, per neuron, where the doubling occurs every layer. The shallow approach, in this case, must consider all possible pairings in one layer.

For many problems, there are computational limits that currently render the use of shallow networks, where many deep networks would succeed, impractical or impossible [35]. In other cases it may just work better to use a deep NN due to the fact that some functions can be more efficiently computed in a hierarchical way. These advantages are why deep learning is taking over, along with its great branding, of course [45].

### 3.5 Convolutional neural networks

Learning patterns in images, and other data organized in grid-like structures, is a hugely impactful area of ML that has been propelled forward by deep learning approaches, especially convolutional neural networks (CNNs) [20, 36]. A CNN is simply a NN that has at least one layer performing convolution operations, explained below. It is possible to classify images using fully-connected NNs, but doing so is problematic for
a few reasons, and less consistent with the elegant deep-learning idea of building up increasingly sophisticated feature representations of the data.

A digital image is a matrix of pixels with numerical values, either a grayscale brightness value or RGB color values. It is important to remember that the number of pixels grows with the product of both image dimensions. The multidimensional nature of image data means that even for small images, but especially richer images, there would need to be many neurons in the first layers of a NN, if each of its input-layer neurons corresponded to a pixel. That would immediately result in a very large number of parameters in the network, and for meaningful training to be accomplished, an immense training set would be required, because more complex models require more training data to avoid overfitting [36]. Learning directly on pixel data in this way is inherently wasteful and misguided, as it does not take advantage of a few tremendously helpful ideas.

When recognizing the most basic and fundamental features that could be detected in an image, of which a given pixel may be a part, it is probably less useful to consider the pixels that are located in distant positions. Consider the problem of detecting edges. When deciding whether a pixel in the middle of an image, for example, is located on an edge in the encoded image, it is reasonable to not consider the pixels located far away, on the outer edges of the image. This is because they, intuitively, would be part of separate structures that do not affect the decision of this local feature. Similarly, when detecting whether a particular area of pixels represents an eye, in an image, it would probably also be less useful to consider values of far away coordinates, because an eye, like most other structures, should be contained within a fixed area. If there was no drawbacks to having too many parameters this issue might be less important, but because it is of great practical interest to reduce a model’s parameters, as well as to make its training work easier, it’s better for the computations performed on pixel values to be sparsely
Recognizing and representing features locally may also improve performance when the image is altered, because their individual roles would likely change less than a single NN that must understand the whole scene; for example, the problem of edge detection probably changes less when an image is cropped, than does the work of a fully-connected layer, which would maybe have to re-learn everything. Another benefit of this approach may be that it reuses its feature extractors across the whole input, further reducing the number of required parameters.

CNNs augment the capabilities of NNs by performing convolutions, or weighted sums of nearby pixels, with sliding feature selectors called filters, or kernels. CNNs perform convolutions through a series of steps and before feeding that data into fully-connected layers, resulting in richer features than raw pixel values, and more efficient representations. Sometimes the output of a convolution operation is called a feature map, which nicely communicates the purpose of a convolution [20]. Convolutions use a filter’s parameters to take the weighted sum of the pixels around a pixel, to quickly extract some feature information and map it over the input, such as where there are vertical edges. Filters operate over a local receptive field, and implicitly reflect the idea that input pixels have sparse interactions because they are usually much smaller than the total input size [20]. The same filter is reused across the whole image, and it slides by a fixed amount, known as its stride. This is advantageous because it eliminates the need to have the same feature detection capabilities needlessly stored in many redundant parameters. As will be discussed, this idea makes sense. For example, an edge detector shouldn’t care where in an image an edge is located; it simply looks for edges, all through the image. This approach is much simpler than having an edge detector for
Figure 3.6: Convolutional neural networks use series of sliding filters to extract meaningful and relevant data from an image, or other matrix-data, through convolution operations. These pixels have sparse interactions, because each pixel is only considered as it relates to the few other pixels in its immediate surrounding. Figure from [54].

every pixel.

Depending on how the layer is setup, the layer’s output may be the same size as its input, or it may be smaller. Often convolutions are followed by pooling layers that simplify the convolution’s output through various operations, such as setting each entry of the feature-map to be the value to be the highest value around it, producing a condensed feature map [50]. These layers normally do not have any parameters, as they just perform an operation like taking the max or average of a selection. The intuition behind this technique is that the exact location of features is less important than their positions relative to each other [50]. Another result of the combination of convolution and pooling layers is that one can now accept input of any size, because they operations are reused many times across the input.
3.6 The elegance of convolutions: edge detection, an example

A single convolution layer may have numerous filters, each outputting its own feature map. As a concrete example, I will discuss a basic edge detection implementation. To detect a vertical edge with a 3 x 3 convolution filter, the system should highlight where the colors abruptly change between two close vertical selections. The following matrix is an implementation of that idea, resulting in values close to 0 when colors do not abruptly change, but non-zero values when they do change, identifying the coordinates of vertical edges [44]:

\[
\begin{bmatrix}
1 & 0 & -1 \\
1 & 0 & -1 \\
1 & 0 & -1 \\
\end{bmatrix}
\]  

(3.3)

Horizontal edges can be similarly detected, in a very rudimentary way, with the following filter matrix:

\[
\begin{bmatrix}
1 & 1 & 1 \\
0 & 0 & 0 \\
-1 & -1 & -1 \\
\end{bmatrix}
\]  

(3.4)

These feature maps can still be rendered as images, where values close to 0, areas where the feature is not detected, are rendered as black, and bright areas denote the detected feature’s location (see Figure 3.7).

CNNs are an elegant extension to the ideas of deep learning, as they efficiently build feature maps for subsequent network components to build on. Rather than diving right into the pixel data, in the hopes of finding all relationships between all pixels, they map out feature spaces, which are locally focused, caring little about the specific values of the pixels. They are incredibly flexible as they handle transformation invariance well, i.e., their components will still be able to extract the same features if input is scaled or transformed in some other way [44]. Given all of their desirable properties
Figure 3.7: The combination of simple vertical and horizontal edge detection, achieved by summing the two edge detectors, Filters (3.3) and (3.4). With two very simple convolutions, even provided raw input data, a convolutional layer can provide a decent outline of where various objects exist, which could inform subsequent feature detection.
it is not difficult to see how CNNs have become central to much of computer vision. Unsurprisingly, they are core components of most GAN designs.
CHAPTER 4
GENERATIVE AdVERSARIAL NETWORKS

Individual neural networks can clearly achieve impressive and practical results as they may learn functions that encapsulate a conveniently described phenomenon, but there are cases where it may be advantageous to simultaneously train multiple networks rather than just one. Approaching a problem with multiple ML models may allow more flexibility and conceptual freedom in how to solve it, as the problem can be re-phrased as a collection of smaller and more tractable tasks. Generative Adversarial Networks are a shining example of this idea, as they leverage individually powerful tools to create even more sophisticated learning functions.

For many problems, how to choose a suitable objective function with which to train a model is often unclear. GANs create dynamic objective functions that optimize tricky and seemingly subjective qualities, by creating an adversarial game between two players.

They can be used to generate realistic data, of any form, though much focus is placed on images. Beyond images generation, the way they learn features, in an unsupervised way, may be very useful for other applications, like efficiently training classifiers, creating smart feature extractors, and even general purpose data processors.

4.1 Adversarial arms race

How one objectively defines the goal of a learning system is a fundamental and necessary task in machine learning. Doing that with the limited freedom provided by a single-network approach would be extremely difficult for some problems [72, 27, 30]. To demonstrate this point, I ask the reader to define, or to at least describe a strategy for, a function that scores how realistic an image seems, to be used in a system that generates novel, realistic images.
4.1.1 Two may be better than one

There are a number of reasons for why it is difficult to write a loss function for scoring apparent image realism, but intuitively it’s difficult because it requires a human to explicitly define an objective function for evaluating a subjective question [72]. Unlike a supervised learning problem, e.g. determining whether a labeled image belongs to class “tiger” or class “school bus,” how real an image seems is ambiguous. As a result, a loss function for apparent image realism should probably be fundamentally different, in some way, from one that would work in a setting where the model can be mathematically evaluated with a straightforward loss function, like squared error. This challenge highlights the ingenuity of systems like the GAN, that leverage multiple neural networks to express training objectives of more vague problems that may be difficult to frame with only quantitative metrics.

As previously established, machine learning can reveal structure and patterns in data that no person could ever discern, so why shouldn’t humans leverage those capabilities to more effectively do machine learning? Because ML models have the capabilities to explain complex phenomena, even ones that humans could not, it is reasonable to consider how ML models could replace the work of humans in the tasks of creating more capable ML systems, difficult tasks like defining a model’s objective or assembling a labeled training dataset.

In the case of GANs, which can directly deal with problems such as whether or not an image looks real, one neural network acts as an adaptive loss function [30, 27, 21]. In this way machine learning not only enhances the capabilities of people seeking to do specific tasks, but it enhances the expressive power of machine learning practitioners building systems to accomplish more abstract goals. By cleverly using multiple neural networks, GANs bypass the need for a human to solve the seemingly paradoxical challenge of finding an objective function for the subjective question of whether an image
looks real or not, enabling systems that can achieve impressive results, with a straightforward strategy, to be discussed next.

### 4.1.2 Detectives and art forgers

GANs are provided training data and through a process which can be conveniently thought of as a game, the systems improve their understanding of reality to better discern whether any given sample is real or if it has been artificially generated. In answering that question, the system is really indicating whether the considered sample is compatible with its understanding of how real data should look, which it learns from the available training data. As a GAN improves those capabilities, it also improves its competing ability to generate more and more realistic samples, which may ultimately be startlingly good, as shown in Figure 1.1 [30]. GANs can generate many different types of data and have a variety of uses, and designs, all of which exploit a general adversarial approach to estimating generative models [21, 17].

A GAN has two learning components that compete against each other in a game, as adversaries. One player is called the generator, and is the network that actually generates a GAN’s novel samples. The generator competes against the other player, a network called the discriminator that simply decides whether a given sample is real, meaning it came from the training data, or if it was generated by the generator, its adversary. During training, the two adversarial networks are updated in sync, so that they improve at their respective jobs, eventually converging to a stalemate where an well-trained discriminator can’t differentiate between training data and the generator’s output. In practice, these two players are almost always deep neural networks, though the framework only requires they be differentiable functions [21, 17].

To clarify their interactions, a GAN’s competing players can be thought of as detectives and art forgers [21]. In this game, the goal of the forgers is to generate fake art that
is indistinguishable from authentic art, and the goal of the detectives is to confidently
discriminate between the fake and real examples, with the highest possible average accu-

racy. The forgers and detectives are analogous to a GAN’s generator and discriminator
networks, respectively.

The rules of the game, embedded in the GAN training procedure, dictate that only
the detectives get to see any real art, which serves as training data for building the
detectives’ understanding of what real art is like. Logically, it is also useful to show the
detectives, in training, numerous examples of fraudulent art, which are constantly being
made. In a supervised manner, the detectives are taught to discriminate between real and
generated art, during their time at the police academy. Naturally, police training only
improves the detective’s understanding of the problem, though it does involve the work
of the forgers.

The forgers know that all art they generate is false, and so their goal is to always
convince the detectives that their art is real. How confidently they can make the detec-
tives wrongly believe that their generated art is authentic is their measure of success,
and they are constantly experimenting to improve that score. The result of their experi-
menting and interactions with the detectives is that the forgers develop an understanding
of what real art is like, which is how they ultimate are able to produce fake examples
that are not perceptibly different from the real thing. Don’t forget that for the forgers,
making realistic art is made more difficult by the fact that they never get to access any
real art, only the detectives do, and neither party willfully shares its improved under-
standing with the other, its adversary. Furthermore, a player can only refine its own
understanding, without directly affecting the understanding or actions of its adversary;
detectives in the police academy cannot magically make art forgers less creative, and
forgers cannot simply compel the detectives to ignore various details of their fake art.
This constraint ensures that the forgers are actually developing an understanding of how
to make realistic art, as opposed to merely copying it, which anyone could do. These effective art forgers highlight a special aspect about GANs. The design of the game results in systems that develops a sophisticated understanding of the world around it, without being able to copy it [17, 60].

4.2 Defining the GAN objective

More impactful than the data GANs have generated is the way they learn to do it. This section describes how GANs define an objective function for the vague task of generating realistic data, made possible by expressing training as a game played by two adversaries.

4.2.1 A brief discussion of probability distributions

To meaningfully articulate what GANs actually learn, it is important to first discuss probability distributions, and several related concepts, because what GANs ultimately model are the probability distributions that describe whatever phenomenon generated a training dataset.

A sample space is the set of all possible outcome values for some phenomenon, such as the set of possible sums of two rolled dice. Barring extraordinary circumstances that would invalidate the roll, every time two standard dice are thrown their sum will take on a value coming from the sample space: \{2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}. The probability of a dice roll coming from \{2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\} is 1, and is equivalent to the sum of the respective probabilities of each independent possible outcome. The way this total probability is allocated over the sample space is its probability distribution [10]. Figure 4.1 illustrates this probability distribution, by graphing its probability mass function, a function that gives the probability of a sample from a discrete set. Continuous
probability distributions are similarly described by *probability density functions*, a classic example being the bell curve that describes the Gaussian, or normal, distribution.

Like the dice roll, a digital photo is also just a sample, coming from the space of all possible images, which is the set of all possible assignments for a matrix of image pixels, of the available color values. As with the dice roll, there exists a probability distribution that generally describes the likelihood of each outcome from this sample space. Whenever a photographer takes a picture of a human face, they are sampling from the same set of possible photos, but this probability distribution would probably be different from the general case. Intuitively, the set of photos that are more likely if the photographer focuses on a face, are the ones that we would recognize as depicting a face. Embedded in these probability distributions that we can only estimate with samples, is a representation of the rules of reality, like the notion of edges or the fact that typical photos tend to not produce images that are completely random noise [7].

To generate realistic data samples, GANs estimate these complex probability density functions that describe reality, like the function that gives the likelihood of a particular photo of a human face [7]. Using its sophisticated estimations of these functions, a GAN can guess which images are realistic, just as we can tell that a dice roll sum of 7 is plausible but that 500/500 rolls of 2 would be highly unlikely and therefore unrealistic, given the distribution in Figure 4.1.

### 4.2.2 The cross-entropy loss function

The machine learning task of classification is the process of estimating a function that computes from which probability distribution a given sample most likely came from, where each probability distribution corresponds to a classification category. Cross-entropy is a commonly used cost function for classification tasks. For binary classi-
Figure 4.1: The probability mass function of the probability distribution of possible sums from rolling two dice, where each die has outcomes 1-6, of equal likelihood. Figure from [65].

\[ C = -\sum_{i=1}^{N} y_i \log(h(x_i)) - \sum_{i=1}^{N} (1 - y_i) \log(1 - h(x_i)) , \]

where \( N \) is the total number of samples, \( y_i \) represents the label for the \( i \) – \( th \) example, and \( h(x_i) \) is the model’s hypothesis on the \( i \) – \( th \) input example [50, 60]. Ground truth labels will either be 0, or 1, and so only one term will penalize the function per hypothesis. This is reasonable because in the binary classification task, the model can only be wrong in two ways: the first term penalizes false negatives, and the second term penalizes false positives. When a model’s hypotheses are farther away from ground-truth labels the cross-entropy values are much higher, and when a model’s hypotheses are close to where they should be, it approaches 0, enabling cross-entropy to work well as a loss function [50]. Because \( \log(0) \) is undefined, the use of this cost function implies that the model being trained is outputting the probability of an example being in a certain class, rather than just the most likely class, so \( h(x) \) is between 0 and 1.
This approach is more informative than if the model gave a binary output, and allows for more precise learning. Specifically, it enables the model to be more severely penalized when it is very confident about an incorrect prediction, and less penalized when it is unconfident about an incorrect prediction. Another nice characteristic is reflected in the function’s non-linear gradient, as it allows training to avoid the common learning slowdown problem, so it works well in practice [50]. Ideally, through training, the model will learn representations that allow it to very confidently make accurate predictions. As will be discussed, a primary task in training a GAN is to train its discriminator in a supervised way, where cross-entropy plays a central role in defining the objectives of both GAN players.

4.2.3 Discriminator and generator objective functions

To generate novel data that are similar to a set of real data, the objective of GAN training is to estimate the probability distribution from which the real data was sampled, \( p \). As discussed, a GAN has two distinct adversarial learning functions, the generator, \( G \), and the discriminator, \( D \), each having their own model parameters \( \Theta_G \) and \( \Theta_D \), respectively.

The goal of training \( G \) is to define its function mapping from a latent space, \( p_z \), to another probability distribution, \( p_g \), serving as the GAN’s approximation of \( p \). The latent space, or latent prior, is typically a high-dimensional Gaussian distribution, where each sample \( z \) provides a random noise vector to the generator function. \( G(z; \Theta_G) \) defines the generator function with the parameters \( \Theta_G \), on a random input \( z \), outputting a sample from \( p_g \) [21]. When the model is trained well, \( G(z) \) is realistic, and therefore difficult to detect as being fake, or not being a sample from \( p \). The signal that guides the generator’s training is how well its fake samples trick \( D \). With this intuition, it should not be surprising that the generator’s loss function is closely related to \( D \)’s.

\( D \) is a classifier that simply predicts whether a given sample came from \( p \) or \( p_g \).
Figure 4.2: Abstraction of the data flow between the probability distributions and learning functions in a GAN. The classification task for $D$ is simply to discern whether a given input $x$ came from $p$ or $p_g$, where $x$ would be $G(z)$ for some latent vector $z \sim p_z$. As $D$ learns this task it provides $G$ an adaptive loss function. $G$ never receives real data as input, so it does not have the opportunity to merely mimic any real data, it actually learns strategies to fool $D$. Figure adapted from [66].
$D(x)$ is the probability that $x$ came from $p$ rather than $p_g$ [21]. As shown in Figure 4.2 the architecture of the system allows the GAN’s training loop, but not $D$, to know from which distribution each training example comes from, either $p_g$ or $p$. All real data is known to be real, and all generated data is known to be fake, and so $D$ can be trained in a supervised way, with a standard classification loss function. No manual labeling is required because the GAN’s training control flow can effectively label each sample when it’s evaluated by $D$, because it retrieves it from either real data or generated data.

As a supervised classifier, $D$’s cost function is simply the previously discussed binary cross-entropy function, Equation 4.1, with a few GAN-specific substitutions, resulting in the function

$$J^D(\Theta_D, \Theta_G) = -\frac{1}{2} \mathbb{E}_{x \sim p}[\log D(x)] - \frac{1}{2} \mathbb{E}_{z \sim p_z}[\log (1 - D(G(z)))] ,$$

(4.2)

which formalizes $D$’s cost with respect to the output of not only $D$, but also $G$—because all fake data is the output of $G$. Substituting for $y_i$, the two classification cases for $D$, the first term corresponds to cases where $x \sim p$, or $x$ is a sample from the real data distribution, and the second for when $x \sim p_g$, or when $x$ is fake, sampled from the generator’s distribution. As discussed before, all fake data was the output of the function $G$ on some input vector $z$, and similarly all real data is just a sample from $p$.

The objective is actually to optimize for the expected value of the equation, and so the sums are replaced by their expected values, or the average value expected over many examples, $\frac{1}{2}$. Intuitively, the expected value is $\frac{1}{2}$ because if $G$ and $D$ are optimized then $G(z)$ will be realistic enough that $D$ might as well flip a coin to make its prediction if $\frac{1}{2}$ of the data is fake [26, 60, 21].

In summary, training $D$ tries to minimize this Equation 4.2, and in doing so its classification capacity between the two classes, real and generated data, improves, like any other binary classifier. Unlike most supervised learning tasks, the design of the systems eliminates the need for labels, in this original formulation. Another unique
aspect of the cost function is that its second term, corresponding to the case where data is fake, includes the output of its adversary, \( G \), which has separate model parameters that \( D \) cannot access. Owing to that, \( D \)'s training is closely tied to \( G \)'s training, and the two can actually use the same cost function, to be discussed next.

Unlike the function to evaluate the classification performance of \( D \), which could be trained like any other supervised classifier, outside of the GAN context it would not be clear what function should score how realistic \( G \)'s output is. The way GANs bypass that difficulty is by making \( G \)'s quantifiable objective simply to trick \( D \), i.e., to reduce \( D \)'s prediction accuracy—precisely the opposite objective of \( D \). Training \( G \) simply seeks to maximize the function that \( D \) seeks to minimize, or equivalently \( D \) seeks to maximize whatever function \( G \) tries to minimize. Concisely conveying the direct competition between these two players, is this relationship between their cost functions; if the cost function for \( D \) is \( J^D \), then \( G \)'s cost function is \( J^G = -J^D \).

### 4.2.4 The GAN minimax game

The goal of GAN training is for its players, \( D \) and \( G \), to minimize their respective loss functions, \( J^D(\Theta_D, \Theta_G) \) and \( J^G(\Theta_D, \Theta_G) \), where both are a function of the output of the two players. Because \( D \) and \( G \) can only update their own parameters, they cannot minimize their objective function in a regular, unperturbed manner, as it is also directly dependent on the strategy of their adversary. Ian Goodfellow cleverly solved this problem by reframing the GAN’s overall objective as finding a *Nash Equilibrium* between these two adversaries [21].

Game-theorists define a Nash equilibrium as a “set of strategies, each belonging to a player, where no player has an incentive to change their strategy, given what other players are doing” [63]. When the players are learning functions, as they are in the case of GAN training, the strategies in question are the player’s model parameters. Much
like a human player’s strategy, each player’s model parameters determine their behavior, and interactions with the other player, as shown in Figure 4.3. Specifically, the Nash equilibrium being sought is the set of parameters \( \{ \Theta_D, \Theta_G \} \) that define the optima of \( J^D \) with respect to \( \Theta_D \) and the optima of \( J^G \) with respect to \( \Theta_G \) [17]. The idea of a Nash equilibrium is clearly applicable to this application, as the players’ loss functions are at local optima, with respect to their own trainable parameters. Conveniently, the Nash equilibrium of GAN training occurs in the situation where \( p_g \) models \( p \), because in no other scenario could \( G \) better fool \( D \) [17]. This specifically occurs when \( D \) has the optimum parameter values, where \( D^*(x) = \frac{p(x)}{p(x) + p_g(x)} \), or that \( D \) accurately encodes the probability density function for real and generated data [21, 8]. Under the assumption that \( G \) generates highly realistic samples, and with the GAN design decision that half of \( D \)’s input samples come from real data, \( D^*(x) \) will be 50%; in other words, \( D \) will be stuck making random guesses, where any deviation from that strategy would result in inferior performance. \( G \) will continue to produce samples that are indistinguishable from real data, because any deviation from that strategy would be counter to its objective.

In the original formulation of the system, the game is a minimax game, as \( G \) tries to minimize the same function that \( D \) tries to maximize, the negative of Equation 4.2, as expressed here:

\[
\min_G \max_D J^G(\Theta_D, \Theta_G) = \mathbb{E}_{x \sim p}[\log D(x)] + \mathbb{E}_{z \sim p_z}[\log (1 - D(G(z)))] \tag{4.3}
\]

GAN training uses Stochastic Gradient Descent to train \( D \) and \( G \) in distinct, but simultaneous steps [17, 43]. Returning to the metaphor of detectives and art forgers, one epoch of GAN training consists of a step the “police academy”, and another step where the forgers improve, done simultaneously [43]. As outlined in Figure 4.3, in each iteration, \( D \) receives one example of fake data and one example of real data. For generated data, \( x_f \sim p_g \), \( D \)’s gradient is computed in the direction to guide parameter updates towards \( D(x_f) = 0 \). On real data input \( D \)’s gradient guides parameter updates towards
Figure 4.3: The objectives of $D$ and $G$ are with reference to their own model parameters. In both cases, $x \sim p$ and $z \sim p_z$, $\Theta_D$ affects the error, because the GAN objective is a function of how well $D$ classifies data. Only in the second case, where $D$ takes in fake data, $G(z)$ for $z \sim p_z$, does $\Theta_G$ affect the error. When data $x$ is real $D$ wants $D(x)$, the predicted probability that $x \sim p$, to approach 1, where $G$ has no direct involvement in that process. When data $x$ is fake, $x = G(z)$ for $z \sim p_z$, and so $G$ is involved in this case, as it seeks to maximize $D(x)$ while $D$ tries to minimize it. Figure adapted from [17].
returning $D(x_{data}) = 1$, for real data, $x_{data} \sim p$. Before any parameter updates occur to $D$, $G$ computes its gradient that guides $G$ towards minimizing $\log(1 - D(G(z)))$. After both players have computed the gradients of their loss functions, with respect to their own parameters, the two perform a simultaneous update of their strategies, $\{\Theta_D, \Theta_G\}$. In the next iteration of training the two players, on average, will be more formidable adversaries.

In practice there are numerous versions of this original approach. Some advocate performing varying numbers of steps per player, per round of training, though no approach clearly works best for GANs, generally.

### 4.3 Fundamental GAN training dilemmas and solutions

The fact that deep neural networks are so commonly trained is pretty incredible when one considers that some may have billions of parameters. Even more subtle is the process of adversarially training two deep neural networks, where one must use the other, in some capacity, as its cost function. Though Ian Goodfellow’s original formulation of the GAN is very elegant and theoretically robust, the task of actually training GANs to estimate an unknown probability distribution that describes a rich sample space is extremely difficult, and plagued with a few fundamental issues, in addition to many of the classic challenges of training other ML models.

Below we explore the perennial GAN training challenges of vanishing objective function gradients, general training instability, and low poor diversity of generated outputs, in extreme cases mode collapse. In addition to finding new uses for GANs, much research in GANs still simply focuses on how to address these difficulties. As will be discussed, in many cases it is impractical to use the minimax or other basic GANs, and so what normally occurs is that researchers invent new methodologies to combat their own particular instances of common GAN problems, to achieve their desired results.
4.3.1 When the signal is lost: vanishing gradients

The minimax formulation of GAN training allows for theoretical proofs that show the optimality of the Nash equilibrium solutions to GAN training, but actually training a GAN in this way, as specified by Equation 4.3, unfortunately can be impractical [17]. In many situations, “unhealthy competition” between $D$ and $G$ can lead to $G$’s gradient saturating, or disappearing as a result of the loss function leveling out [30].

Because $G$ only can update its own parameters, $G$ can only make use of the gradient $\frac{\partial}{\partial \Theta_G} J^G$, and so the first term of Equation 4.3 does not provide any useful information for $G$’s training, as it does not include $G$. This is not surprising because as shown in Figure 4.3 when $x$ is real, i.e., $x \sim p$, the case corresponding to the first term of the minimax formulation, $G$ is not involved in its processing. The minimax game ensures that only the second term, the term corresponding to error resulting from false positives, provides a signal to $G$’s gradient. Problematically, in the minimax version of GAN training, as soon as $D$ confidently classifies $G(z)$ as fake, or $D(G(z))$ approaches 0, $G$’s gradient saturates, providing no indication of how to improve, and training fails [17].

This premature learning slowdown would not occur in more traditional use cases of the cross-entropy loss function, such as supervised classification, because its gradients normally vanish only when the trained model is nearly perfect [50, 17]. In the case of minimax GANs, this is an issue because $G$’s gradients vanish when $D$, a supervised classifier, is highly accurate, rather than when both $D$ and $G$ are nearly optimized. The practical implications of this dynamic require anyone using this approach to take great care to balance the training of $G$ and $D$, which is extremely difficult [8].

To address this problem $J^G$ can be re-written in a way to avoid gradient-saturation, while still maintaining its goal to force $D$ to make more false-positive classifications, as shown in Figure 4.4. The new, non-saturating formulation of the training game still takes the approach of defining $J^G$ as being the opposite of $J^D$, but rather than simple
negating $J^D$ it instead flips the target labels that guide the cross-entropy loss [17, 60, 26]. As previously discussed, for training $G$, it is not beneficial to take the gradient of the term that lacks $G$, so it is omitted from this new equation,

$$J^G = -\frac{1}{2} \mathbb{E}_{z \sim p_z} [\log (D(G(z)))]$$

(4.4)

that no longer saturates when $D$ effectively classifies $G(z)$ as being fake [17]. Ian Goodfellow motivates this heuristic by saying that cross-entropy is an effective classification loss function that only saturates, or stops learning, when guesses are nearly perfect, but the minimax game has two players, where $D$ may easily become highly effective before $G$ improves, stopping progress. By flipping the labels of the original $J^D$ equation, the negative of Equation 4.3, this approach is essentially maximizing the likelihood that $D$ is mistaken, rather than minimizing the likelihood it is correct, as in the minimax game [17]. When $G$ uses the non-saturating heuristic loss function, it is no longer seeking to minimize the function that $D$ tries to maximize, and so it is technically no longer minimax game, but the same adversarial relationship between the two players is maintained, and so it can yield impressive results [21].

Traditional non-GAN generative models are based on the idea of maximum likelihood, and GANs too can be formulated with that principle, resulting in additional game formulations with more manageable gradients than those in the minimax game, as demonstrated in Figure 4.4 [8, 17, 22].

In the context of generative models, which estimate true probability distributions, maximum likelihood learning seeks to assign a model’s parameters such that the probability of available training data, samples from the real probability distribution being modeled, are maximized [17]. As in the first two GAN formulations, the training process estimates a probability distribution describing real data, but now the high-level, expressed goal is to create a model that minimizes the distance between $p$ and $p_g$, often as measured by KL-divergence [8]. KL-divergence is one way of computing how dif-
Figure 4.4: The cost functions of three formulations of $G$, as they relate to $D$’s performance. When $D$ is optimally trained, relative to its cost, $D(G(z))$ approaches 0, because it can confidently discriminate between real and fake data. As shown in this graph, when $D$’s training quickly outpaces $G$’s the slope of $G$’s function vanishes and training halts. The training strategy of a GAN system is dependent on subtly nuances in these cost functions, such as whether or not the GAN should train $D$ and $G$ in a balanced way. Figure from [17].

Different two probability distributions are, as a function of the ratio between the likelihood of all possible samples given the two corresponding probability density functions. The KL-divergence between the probability distributions $P$, describing real samples, and $P_g$, $G$’s estimation of the real distribution, is expressed in Equation 4.5, with $\chi$ representing the space of the possible data that could be sampled by $D$, e.g., as all possible images of a some format [9]. $P(x)$ and $P_g(x)$ are the probability density functions corresponding to $P$ and $P_g$, which provide each distribution’s probability of a sample $x$ being observed.

$$KL = (P||P_g) = \int_{\chi} P(x) \log \left( \frac{P(x)}{P_g(x)} \right) dx$$ (4.5)

When the KL-divergence between two probability distributions is 0 they are identical, and as this value increases the relationship between them becomes more random, as they are increasingly different [51].

In the context of GANs, When $P(x) > P_g(x)$ the likelihood of a sample coming from real data is higher than the likelihood it came from generated data, meaning that $P_g$ is not sufficiently covering $P$ [8]. Intuitively, this ratio being larger indicates that $G$
is not representing all “types”, or modes of data represented in the real training data. As an example, if a face generator only generated faces with blue eyes, then it is probably more likely that an image of a green-eyed face came from real data. When $P(x) < P_g(x)$ and $x$ is a generated sample, $x$ is unrealistic [8]. If the same face generator generated an image $x$ that simply lacked coherent structure, $P_g(x)$ would probably be much higher than $P(x)$, which we would expect to be quite low, because photos taken of people tend to not produce incoherent globs of color. If $KL = (\mathbb{P}||\mathbb{P}_g)$ is used as a GAN cost function, it penalizes the first case much more harshly, when $G$ does not produce all modes of data, rather than the second, when $G(z)$ is highly unrealistic for most $z$ [8]. To have any hope of generating realistic data, it is therefore better to use $KL = (\mathbb{P}_g||\mathbb{P})$ as a cost function. While this approach may generate some realistic results, it is no longer incentivized to produce many different types of results, making the task of generating diverse output difficult, as will be discussed later. There are a variety of GAN formulations that either minimize KL-divergence, or another measure of divergence between $\mathbb{P}_g$ and $\mathbb{P}$ [9, 21, 17, 51].

Returning to the original and cleanest version of GANs, it has been shown that the minimax game actually optimize a separate measure of divergence, Jensen-Shannon divergence, a divergence that nearly averages $KL = (\mathbb{P}||\mathbb{P}_g)$ and $KL = (\mathbb{P}_g||\mathbb{P})$ [8]. That is a nice quality because it requires less of a trade off between two desirable GAN output characteristics, output variety and realism, but remember, the minimax game is normally very difficult to train, despite its theoretical advantages.

Seemingly minor differences in the cost functions of either player can have large implications on training, beyond just whether or not a certain gradient vanishes, as it determines the whole strategy of a team’s approach. As an example, how $D$ is trained in relation to $G$ varies across projects. In training the minimax GAN, the system must ensure that neither player’s training outpaces the other, whereas for other systems, like
Wasserstein-GAN (WGAN), that need not be a consideration. Wasserstein-GAN optimizes yet another divergence metric, the *Earth-Mover* distance, and given WGAN’s resulting characteristics, their strategy is actually to optimize $D$ quickly [9]. There are many formulations of the GAN game that take take the basic idea of using two competing networks much further, overcoming many of the vanilla GAN’s shortcomings to achieve more impressive results.

### 4.3.2 Training instability

The comparatively straightforward problem of gradient saturation is just one way GAN training may not converge to an optimum for both $D$ and $G$. Finding a Nash equilibrium where both $D$ and $G$ perform well, the objective of GAN training, is a very difficult problem [58, 17, 52].

While the adversarial interactions between $D$ and $G$ allow for the mathematical specification of an objective for $G$, solving the seemingly paradoxical problem of defining an objective functions for subjective criteria, they are fundamentally difficult to optimize because the progress of one network may detract from the progress of the other. Specifically, a parameter update to $D$ that lowers $J^D$ might increase $J^G$, while updates to $G$ might similarly decrease $J^G$ while increasing $J^D$, clearly shown in Equation 4.3. Sometimes these interactions between $D$ and $G$ can cause “gradient descent to enter a stable orbit”, or a repeating cycle that never converges to the desired equilibrium [58]. That results in training where generated samples are never optimal, instead switching which imperfections they contain.

Using other cost functions for $D$ and $G$ that slightly disentangle their objectives can provide practical advantages in addressing this problem, while still maintaining enough of the adversarial relationship, as when $J^G$ was defined with Equation 4.3 or 4.4. One example approach to address this problem is to rephrase $G$’s objective as trying to min-
imize the difference between the output of an intermediate layer in $D$ between when it receives real and generated data [58].

Their are numerous GAN implementations intended for a variety of use cases, and while they suffer from common high-level problems in many cases unique solutions must be concocted to address these challenges, meaning there are many formulations of cost functions, but no one cost function is best for every purpose [17]. Another example of a simple addition to the cost functions that may have the effect of preventing non-convergent orbits, is to add a term to each player that contains the difference between each parameter’s current value and its average historical value [58]. Intuitively, if a stable orbit is thought of as being a circle then this new term, the difference between the average historical parameter values and the current parameter values, will be larger when the radius of that orbit is larger, because the average of the orbit is at its center. This historical average component, being a positive term of the model’s cost functions, should be minimized during optimization. Recall that the designers of a ML system must specify their model’s objective function to optimize for, and that in addition to evaluating a model’s output it should also be designed in a way that makes the training process more successful, just as regularization as a method to prevent over-fitting does, for instance. Salimans et al. [58] found that the addition of this historical average term, which penalizes non-optimal orbits around some point, allowed various games, such as the minimax game, to converge when they otherwise did not. Just as the original GAN formulation cleverly achieves the goal of specifying an objective function for scoring the vague quality of data realness, there exist numerous enhancements that address other difficult problems. In many cases just tweaking the cost functions being used can yield dramatic improvements, allowing a GAN to succeed where it otherwise would have failed. This also highlights how the original GAN framework really is just a framework, rather than a precise specification for how to actually achieve the best re-
sults. There is much area for creativity and improvement within the general strategy of pitting generators and discriminators against each other, particularly when it comes to finding ways for them to compete in more productive ways.

4.3.3 Mode collapse

In addition to vanishing gradients and finicky training instability, *mode collapse* can also prevent the training of a useful GAN. Mode collapse is a situation where the generator’s output lacks sufficient variation, resulting from $G(z)$ mapping to the same point for many different values of $z$, and it very commonly occurs in GAN training [21, 58, 17, 16]. When this occurs it means that $G$ only produces a small number of distinct outputs, as it finds very tight spaces that fool $D$, for a time. Mode collapse is signified by nearly identical output, as shown in Figure 4.5. Ideally $G$’s parameters should not cause large sections of the latent space to map to nearly indistinguishable outputs, because it means that the generative model being trained is only capable of making a small number of sufficiently unique samples, unlike the probability distribution that governs real data. The problem is common enough that GANs for a while were only considered to be useful in applications where variation of output was not necessary [17].

Mode collapse intuitively arises from the fact that in many GAN formulations $D$ does not consider the variation of $G$’s output in its objective, and so $G$’s task of tricking $D$ does not hinge on the diversity of its output. One cause for this problem is that $D$ often considers one sample at a time, and so when deciding whether or not a sample is real, it does not have the added information of seeing that $G(z)$ is nearly identical for many $z$ [58]. One way to address this problem is to provide $D$ additional information, in the form of other samples, to inform its classification. In feeding the discriminator a mini-batch of samples along with statistics describing the mini-batch, *minibatch discrimination* allows for the variability of samples to be a criterion that $D$ considers [58].
Figure 4.5: As shown on the right, mode collapse occurs when $G$ maps too much of its input space to a very small space of outputs, resulting in images that visibly lack diversity. This example demonstrates mode collapse of a conditional GAN, which has two inputs, a latent space vector, varying on the horizontal axis, and an image of a bag outline, varying on the vertical axis. MAD-GAN is an effective approach to counteract mode collapse, and its generated output clearly has much more variation than other approaches. $G$ takes the outline of a bag to generate photo-like images of imaginary bags they could describe. In this case a mode is a way or style of mapping an outline, on the left, to a photo version. Mode collapse is most clearly visible in the coloring of samples on the right, where different latent space vectors, corresponding to different output columns, map to indistinguishable outputs. MAD-GAN does not cause the latent space to collapse to a single mode, as reflected in the greater texture and color variation of its samples. Figure from [16].
If $D$ is fed numerous random samples it can see when multiple are nearly identical, and so $G$, as a player seeking to fool $D$, must no longer produce output that collapses to a single mode.

In practice, the approach of providing $D$ additional information so that $G$ is incentivized to generate variation is achieved in a number of ways, and is also just one way of achieving diverse results. Karras et al. [30], of NVidia, use a simple version of the mini-batch discrimination approach by computing the standard deviation of each feature of the samples in each minibatch, and feeding those statistics into $D$, with the images. A separate approach adds a regularization term to $G$ that encourages diverse outputs through orthogonalization [30]. By encouraging $G$ to produce a mini-batch of samples, high dimensional vectors, that are orthogonal to each other, the likelihood of mode collapse is greatly reduced, as its outputs would span a larger portion of the real sample space.

In addition to creating new objective functions and providing additional information to either player, researchers have also made more fundamental changes to the GAN game, by adding new players. Ghosh et al. [16], of Oxford University and IIT Kanpur, rethought the GAN game to prevent mode collapse by adding additional generators. Their general GAN framework, *Multi-agent Diverse GAN*, adds additional generators, where all generators share the task of fooling a common discriminator. As before, $D$’s objective is a guiding signal for all generators. To incentivize diverse generated output, $D$’s objective also contains the goal of identifying from which generator a given sample comes from, if it is a fake sample [16]. Initially this approach may seem counterproductive, as if it would cause all generators to converge to the same mode, to prevent $D$ from being able to distinguish them, yielding little benefit. Though they acknowledge that scenario is possible, Ghosh et al. demonstrated that the random initialization of each generator’s model weights makes that scenario highly unlikely [16]. Randomly initial-
izing NN model parameters is standard practice in ML, as it prevents different neurons from redundantly computing the same features, as they would if they all had the same parameter values [45]. The more likely scenario, they found, is that each generator initially produces samples that approximate different areas of the true data distribution, because they are distinct functions mapping latent input to generated samples (whereas if the parameters were not initialized randomly, they would compute the same function). As a result of using random initialization, $D$’s most plausible route to dominating the generators would be if it forces each generator to collapse to its own, distinguishing mode [16]. Being common adversaries of $D$, each generator’s objective is to not let that happen. The generators are therefore incentivized to not only produce realistic samples, but also to avoid doing it in a way that may provide clues to $D$ as to which generator generated a given sample, preventing mode collapse, as shown in Figure 4.5 [16].

The opportunity to not only rethink the architecture of a single NN, but the interactions between them, as well, allows for much creativity. Both players of a GAN are rich tools in and of themselves, each coming with a number of design considerations that determine their capabilities, while those decisions also affect the performance of the overall GAN system, that they comprise. It’s fascinating to see how random initialization of model weights, which is itself beneficial to the performance of a single neural network, for its symmetry-breaking effect on parameters, is also beneficial as it breaks the symmetry of output of distinct generators in a MAD-GAN. Clearly, many desired attributes can be incorporated into the objective of a GAN, but they can also be used from completely different purposes, and more fundamentally redesigned.

### 4.4 Leveraging the latent space and other uses of GANs

GANs are intriguing beyond the realistic data samples they generate, as much of their allure comes from the way they compute outputs with recognizable features, in a struc-
tured way, only given unlabeled inputs. A traditional GAN’s generator, \( G \), receives a vector \( z \) as its input, sampled from a latent prior, normally a high-dimensional Gaussian distribution. As will be discussed here, by providing a GAN additional inputs, or by simply exploring its latent space, we can better control the outputs of GANs, allowing them to be utilized with much greater control.

### 4.4.1 Conditional GANs

Soon after GANs were invented by Ian Goodfellow [21], back in 2014, various extensions to augment their capabilities emerged, including conditional GANs [41] that define mappings between two input variables and generated samples, enabling many new GAN use cases. A traditional GAN learns probability distributions such that its generator, \( G \), can map a latent space vector, \( z \), to one hopefully realistic sample. A conditional GAN learns a conditional probability distribution that allows \( G \) to map a new variable, \( y \), to samples, varying with its latent space input, \( z \). An example conditional GAN’s output is shown in Figure 4.5, where \( y \) is the image outline of a bag and the alternative photo-like outputs correspond to different latent space entries. \( y \) could be as simple as a class label or something much richer. The conditional GAN’s input \( y \) can take on many forms, including text, images, and any other type of digital data, but whatever it is it provides auxiliary information to guide the generation process, often steering inputs to different areas in the output space [41]. In practice, conditional GANs allow for an incredible variety of mappings between different types of data that not only allow for greater control of generated output, as providing class labels might, but also an entirely new way to take advantage of \( G \), as it can be used to enhance or process any data in many ways.

One can think of conditional GANs as being regular GANs that take requests, in the form of their auxiliary input, \( y \). Just as there are many possible outcomes to any request one could make to a person, conditional GANs encode many possible outcomes
to a given request, $y$, in the latent space, or another source of noise. For each latent space vector $z$, conditional GANs provide a distinct answer to the request $y$. In some cases the input $y$ can literally be a request, or a description of what the desired output should contain, not unlike how one might ask an artist to include specific elements in their work. In the most basic version, the auxiliary input could be a class label to dictate what class the generated output should fit into, as judged by $D$, where $D$ also classifies the samples it sees [55, 41]. A conditional GAN that simulates hand drawn digits could use $y$’s value to determine the number its generated images represent. With traditional GANs dictating what form $G$’s output takes requires understanding how a vector $z$ is transformed into a sample, often achieved through passing many examples into $G$. Using a conditional GAN, simply setting $y = 3$ can be enough to generate an image of that number, a much easier process. More impressively, text to image synthesis, as achieved by Reed et al. [55], uses conditional GANs to generate images from more abstract image descriptions. Just as there are numerous photos that can be described by most reasonable photo descriptions, they can produce numerous outputs for each input $y$, shown in Figure 4.6.

Rather than producing a single sample with the greatest likelihood, as assessed by a model’s representation of some phenomenon, GANs learn probability distributions that describe realistic data and then effectively take individual samples from them, producing sharp and nuanced results. They can be thought of as making choices that other models can’t, like whether a generated image of a face will have glasses or not, as opposed to producing a blurry, ghosted intermediate between the two possibilities. For each latent vector $z$, there is a corresponding data sample, that does not represent the average of all possible examples, as traditional models trained with least squared error would, but just one within an acceptable set of samples [34]. The way GANs are trained means that conditional GANs as multi-purpose tools can also produce sharp, useful, and realistic
Figure 4.6: Conditional GAN-generated photo-like images to match a provided text description, with different photos corresponding to different latent vectors. To achieve their results Reed et al. combined recent advancements in natural language processing and conditional GANs to create a system that can sometimes provide numerous reasonable outputs for an image description [55]. While the bird examples are varied, like real photos, the bottom two examples show much less diversity of generated output. The lower right generator has collapsed to a single mode, generating flower images that are dramatically less varied than real photos that could match the description “this white and yellow flower have thin white petals and round yellow stamen.” Figure from [55].
outputs for a variety of tasks.

If the auxiliary input to $G$ is a photo, or some other rich data, fascinating translations can be computed between different data formats, as the conditional GAN’s function can be thought of as a flexible data processor, suitable for completing many tasks. Learning mappings between any sort of input and output, allows for a generator that can compute any sort of function that produces new data from input data, replacing the need to explicitly program these transformations.

Image super-resolution is an example of such a mapping that has been achieved with conditional GANs. After much training experience seeing pairs of low and higher resolutions images, SRGAN estimates the conditional probability distributions that describe plausible high-resolutions images, given a lower resolution image that should look similar, allowing a given photo to be scaled up in a reasonably realistic way [38]. To be discussed later, I used SRGAN to smooth some of the output of my GAN experiments.

Isola et al. [27], found that many different problems traditionally solved with complicated, application-specific algorithms ultimately consist of mapping pixels to pixels, and that the same conditional GAN system, if trained on appropriate data, could be used to perform a wide variety of tasks, as shown in Figure 4.7. One interesting design decision they made is that their GAN’s generator did not have an explicit latent space input, $z$. Instead they introduced noise, for variation of outputs, by using dropout, which alters outputs by randomly eliminating various neurons in $G$ [27]. Their examples generate complex, photo-like outputs from simplistic inputs, which is a task GANs are clearly well suited for. Because much of ML relies on recognizing patterns in relevant structures, it seems that the opposite problem, reducing complicated inputs to simple outputs, could also be extremely useful, like how they reduce an aerial photo to a map-like representation, as shown in Figure 4.7. An example of that, the task of image segmentation, which could be applied in a variety of systems like autonomous vehicles, is not yet
Figure 4.7: The results of a single conditional GAN approach applied to multiple problems, labeled above each input and output photo pair. Rather than explicitly programming a solution to each problem individually they used the same conditional GAN, trained on different data. Figure from [27].

achieved as well by conditional GANs as it is by other available methods, but it will be interesting to see if that changes, as GANs improve [27].

Conditional GANs as a strategy to process data can be applied in a variety of settings, including as a step in a larger ML system. Researchers at Apple, seeking to train a model that could detect where iPhone and iPad users’ eyes were looking on the device screens, realized it would be too expensive to label a sufficiently large training set [19]. To address their problem they generated synthetic training data, and used a conditional GAN to make their fake data look more real [61]. Importantly, they generated synthetic data in a way that meant they would not need to subsequently label it. Using a conditional GAN as a refiner of their synthetic data allowed them to achieve consistently better results than if they used the raw synthetic data, and their approach, which they call simulated + unsupervised learning eliminates the expensive need to label training data. Using this strategy to refine data could also be applied to real data, where due to a variety of circumstances, certain training examples might be less useful. Perhaps one could generally augment the capabilities of less powerful instrumentation by using a conditional GAN that translates low quality data to higher quality data, in way conceptually
similar to image super-resolution.

Conditional GANs have been used in other situations where acquiring certain types of data is difficult. Companies unable to perform road tests of autonomous-vehicles during the night, or in inclement whether, have used transformations of driving data acquired during safe conditions to create training data that describe more dangerous conditions, allowing for much safer, simulated training [19]. Conditional GANs have also been used to map other types of data, data that are not measurements produced by instrumentation.

As an example of their broad potential applicability, researchers at Amazon have used conditional GANs as a way to simulate human behavior. They trained a conditional GAN, eCommerceGAN, to map a user’s current cart of items to a probable cart of items they could checkout with, as a way to make more effective product suggestions [32]. Adversarial training allows for more expressive power, and the introduction of auxiliary inputs, allows conditional GANs to be tools for accomplishing a greater variety of tasks.

4.4.2 Latent space logic

The way a GAN’s generator translates samples from its latent space to realistic data is incredible for many reasons, like the fact that $G$ never accesses training examples, but what some believe to be the true promise of adversarial training is that they become aware of real world phenomena that are not explicitly labeled. Once again quoting Yann LeCun, the man who created the convolutional neural network, Facebook AI research, and much more, he wrote “there are many interesting recent development in deep learning... The most important one, in my opinion, is adversarial training (also called GAN for Generative Adversarial Networks)... perhaps more interestingly, the generator can be seen as parameterizing the complicated surface of real data: give it a vector Z, and it maps it to a point on the data manifold.” [34]. More exciting than the fascinating
images generated is how GANs seem to learn much about reality and its structure, in an unsupervised way.

One of the GAN’s most fun traits is that similar inputs to $G$ map to similar sample outputs, allowing for smooth transitions between samples in the output space as small changes are made to input vectors $z$, called *latent space interpolations*, of which multiple example videos are linked in the Appendix. Interestingly, some property or concept can be varied as one moves in a particular direction in the latent space, because $G$ translates varying characteristics of $z$ to realistic outputs.

Considering a GAN that generates faces, like one created by Radford et al. [52], if people depicted in $G$’s output sometimes wear glasses, but not always, then $G$ has in a sense learned what glasses are. Because $G$ only maps some values of $z$, a subset of all the possible latent vectors, to output with glasses, there is a some parameter, or composition of parameters, in $G$ that encodes whether a given input $z$ will have glasses. If that occurs, even if the GAN’s training data has no labels at all, $G$ has learned some representation of glasses, and probably many other features as well. The logic describing whether glasses appear may be hidden in the immense complexity of a deep neural net with a large input space, but by looking at the way many inputs map to outputs, researchers have been able to reverse engineer features.

After producing many generated face image samples, one can cluster them by any recognizable concept, like the sex of the depicted individuals or the presence of glasses, and by averaging the latent vectors that were mapped to those examples in each cluster, a representation of the concept for which they were sorted is computed. *Latent space arithmetic* of these averaged vectors, which represent some abstract concept, can then be combined to produce latent space vectors that map to samples with the corresponding combinations of properties, as shown in Figure 4.8. If output with specific properties was desired there are other ways to search the latent space, still relying on the fact
that $G$ produces outputs with common characteristics when provided input with shared features.

Volz et al. [69] used GANs to produce new levels like the ones found in Mario and other classic 2-dimensional video games, and then used a fitness function to search the latent space for inputs that would generate levels with desired properties. For the first phase of their work, they trained a GAN to produce realistic samples, encodings of tiles in a grid that fit the trends of Mario levels. As discussed, without an understanding of how the latent space maps to output it is difficult to control what sort of output is achieved. To produce levels that have various desired properties, perhaps a certain difficulty as measured by the performance of ML models that play the game, they treated their trained generator as a “compact and robust genotype-to-phenotype mapping”, where a game’s genotype is a latent vector and its phenotype is the output game, which allowed them to simulate evolution in their search, mediated by a fitness function of the desired properties [69]. Just as natural selection causes the increased abundance of genes that result in physical characteristics that improve an organism’s fitness, they were able to search the space of possible games for ones that better fit their desired properties [69].

### 4.4.3 Semi-supervised learning

As discussed in Section 4.4.2, GANs can develop representations of a variety of abstract concepts, without any labels. *Semi-supervised learning* takes advantage of that property for the purpose of training a classifier, while requiring much fewer labeled examples [58]. Unlike many GAN use cases, semi-supervised learning is primarily concerned with training an effective discriminator, to be used separately, as opposed to creating a generator that either produces random realistic data, or one that performs some mapping between data. The fundamental reason for why one would use semi-supervised learning
Figure 4.8: Latent space arithmetic on generated faces by Radford et al. [52] shows that a generator can develop a representation of abstract concepts like the differences between male and female faces as well as the presence of glasses. Each column of pictures on the left represents images clustered by some concept, men wearing glasses, men without glasses, and women without glasses. By averaging the latent space input that was mapped to the output of these clusters, and performing arithmetic with those resulting averages, the concepts themselves can be added and subtracted. The vector for the concept man with glasses subtracting man without glasses disentangles men from glasses, and the resulting difference amazingly encodes the presence of glasses. The vector representing glasses, when added to a vector of women without glasses results in output of women wearing glasses. There are 9 examples of a woman with glasses because the researchers added noise, to demonstrate interpolations around the resulting vector for women with glasses [52]. Arithmetic in the output space, shown on the bottom, does not result in logical images, as abstract concepts in raw data are not as tractable as they are when a deep net can be used to represent them. Figure from [52].
as opposed to supervised learning, is that it requires far fewer labeled training examples, which can be enormously expensive or impossible to sufficiently annotate [31].

While training a GAN, both players develop representations of numerous abstract concepts, such that $D$ can effectively recognize samples as coming from distinct clusters, regions in a representation space where close points represent samples that share common and meaningful characteristics [19]. The bulk of the learning occurs in this stage, requiring no labels. As impressive as that process is, it is not quite enough to train a classifier, because $D$ still has no understanding of how to map the various clusters to some specified set of labels. A small portion of labeled training examples then allow $D$ to identify which clusters correspond to each label, rather than learning what the labels mean in a purely supervised way. Using the labels to steer prior, unsupervised learning is much more efficient, over 99% more efficient in the case of classifying handwritten digits from the MNIST dataset. Rather than using all 60,000 labels, Salimans et al. achieved comparable performance using only 100 [58, 19, 18].

If semi-supervised learning is applied broadly such that it could similarly reduce the number of labeled training examples required for other traditionally supervised learning tasks, ML would take another huge leap forward. Drastically reducing the fraction of training examples that need to be labeled would be hugely beneficial because the abundance of unlabeled training data is much more vast than the amount of labeled data, for a wide variety of problems [31].

Not only would that advancement reduce a costly challenge of ML, labeling data, but it will also make much more existing data usable for training. Consider how much data exists, and how little of it has been labeled or annotated by a person. All of a sudden, any photo on the Internet would be able to provide insights, in an unsupervised way, while labeled datasets, like the ones that are commonly relied upon today, could simply provide the signals to help classifiers label the clusters they identify. In this way, ML
would be less dependent on labeled data for learning, instead only needing it to label the concepts it has already learned in an unsupervised way. It will be interesting to see how unsupervised learning continues to take on new uses and applications, as more capable GANs and other other creative systems are conceived.
I experimented with GANs to explore the generation of realistic and high resolution novel data in three categories of images: handwritten digits, portrait paintings, and chest X-rays.

5.1 GAN implementations that generate high resolution images

Training GANs to converge and produce realistic, high resolution images is much more difficult than generating lower resolution images, in part because there is more to learning required to do it, in addition to much more substantial memory requirements. In terms of the GAN game, it is easier for a discriminator to distinguish real from fake images when they are larger because there are more features that the generator must get right [30]. To illustrate this point, consider what real photos of paintings would look like if downscaled to a resolutions of $32 \times 32$ or even $4 \times 4$ pixels, as shown in Figure 5.1. To produce new samples that are realistic, given those low resolution samples as a reference, the GAN really only needs to figure out the general shape of faces, and other basic features like what color combinations are likely. To produce higher resolution portraits, the generator must learn to create finer features like noses, eyes, hairstyles, styled clothing, and many more.

To achieve high-resolution results, many models learn basic features and then refine the images they create in subsequent steps. This is done in a variety of ways, but most rely on convolutions, notably \textit{fractionally-strided convolutions} that create larger outputs from inputs, to create images in $G$ [17]. Deep convolution GAN, or DCGAN, created by Radford et al. makes $G$ a network with many convolution layers, not using any pooling layers, as a strategy to build up larger and larger output, outlined in Figure 5.2 [52].
Figure 5.1: Images of paintings at resolutions $4 \times 4$, $32 \times 32$, and $256 \times 256$. The left two are generated, and the right example is a real painting, with many fine features. Just as deep learning models build hierarchical representations, many GANs that generate high resolution samples first learn more simple features [12, 17, 30]. The left two samples were generated early in training. Rightmost image from [5].

Most GANs today are based on DCGAN, though there are other approaches for similarly learning simple features on which more complicated features are learned [17].

The Laplacian GAN, or LAPGAN, was the first GAN that could generate high-resolution output, and it works slightly differently from DCGAN, though it too builds up images in a sequence of steps [17]. LAPGANs, invented by Radford et al., use a sequence of separate conditional GANs that map low resolution images to higher resolution images, through convolutions, ultimately producing larger outputs than were previously possible with single GAN approaches [12].

To train my GANs, I ultimately used the *progressive growing* approach, invented by Karras et al., because it elegantly and effectively builds up detailed representations, by adding new layers to both $D$ and $G$ as training progresses, outlined in Figure 5.3 [30]. Furthermore, I thought that progressively growing GANs clearly embodied the deep learning idea of first learning fundamental features to inform the learning of more complicated structures, at an especially deep level; not only do their early layers naturally learn more simple features, as commonly occurs, but in their approach training data is actually modified as training progresses. Both real and fake data fed into $D$ increases in resolution through time, starting out small [30].
Figure 5.2: The DCGAN architecture of the generator for modeling images from the LSUN dataset [52]. Numerous convolutional filters are applied to a 100 dimensional input vector to produce feature maps. Through 4 fractionally-strided convolutions, features are flattened into structures that are increasingly large in 2-dimensions, resulting in a $64 \times 64$ image. Figure from [52].

Figure 5.3: The progressive growing approach of $D$ and $G$ explicitly forces $D$ and $G$ to learn simple features before more complicated ones. As training time progresses, shown on the horizontal axis, the resolution of data samples that $D$ and $G$ take in and generate, respectively, increases. When the resolution of the network increases new layers are added to the pre-existing $D$ and $G$ architectures that double the capacity, allowing much quicker and more stable training [30]. Figure from [30].
Figure 5.4: Selected $64 \times 64$ generated portraits by Jones and Bonafilia [28]. These paintings come from a collection of their generated paintings that were automatically selected for their realness as portraits, as scored by the discriminator of their model [28]. Though they have aesthetically pleasing qualities, they lack important details like discernible facial features, which most would expect to see in a portrait painting. Figure adapted from [28].

This approach also achieved the most satisfying GAN results I have ever seen, in terms of resolution, sharpness, and realism, the fictitious celebrities shown in Figure 1.1. I chose to use their technique because of it is well suited to generating large images, and a core aspect of my goal was to produce larger and more detailed generated portraits than I had ever previously seen. In addition to the ability to generate high-resolution images, their approach includes improvements for training stability, training speed, and variation of output, which made the choice even clearer [30].

5.2 Generating images

My primary goal was to generate relatively high resolution, aesthetically pleasing, and realistic painting images. I focused on producing realistic portraits, and I subjectively believe the results I achieved to be the most realistic GAN-generated portrait paintings, sharper than prior attempts I have seen, like those shown in Figure 5.4.
Figure 5.5: Generated paintings without progressive growing at different training iterations, organized by row. The top row are generated samples after 999 training iterations, while the lower three are samples generated at 10999, 100999, and 199999 training iterations, ordered from top to bottom. Unlike the progressive approach, training here does not initially limit the scope, and so $G$ has the enormously difficult task of tricking $D$ with $64 \times 64$ samples, without prior experience at lower resolutions. Training has not converged to provide any recognizable objects or structure. Total training time was approximately 34 hours.

5.2.1 Generating paintings without progressive growing

Initially inspired by Jones and Bonafilia [28], I first experimented with their code, to see if I could achieve higher quality results, using superior hardware. Jones and Bonafilia, who were seniors at Williams College as they completed their research, explain they were limited to producing $64 \times 64$ images because they used a GPU with only 2GB of memory. My initial hope was that using the the Middlebury College Computer Science Department’s Titan Xp GPU, which has 12GB of memory, I would be able to generate at least $128 \times 128$ images.

The results I achieved using their code, shown in Figure 5.5 do not provide evi-
idence of training convergence or very much learning of defined structures. As Jones and Bonafilia acknowledge, the example images they published, of which some of their portraits are shown in Figure 5.4, were specifically selected because they were the most realistic. My takeaway was that this approach was not effective, and would be even less effective at producing larger images, and decided to change my strategy.

5.2.2 Generating handwritten digits with progressive growing

My workflow, for generating each category of images, consisted of acquiring a dataset, processing the images, training a GAN with progressive growing on Gattaca, a workstation run by the Computer Science department, and then exploring the results. I used the publicly available implementation of progressive GAN growing by Karras et al. [30], making minor modifications, to train my GAN models, as well as to generate samples and latent space interpolation videos, to which there are links in the Appendix.

The 60,000 MNIST hand-written digit images came from [37]. The 70,000 WikiArt painting images, of which 15,000 were portraits, came from [5]. The 112,000 NIH chest X-ray images came from [71].

Data preprocessing consisted of resizing images to a resolution that is a power of 2, so that progressive doubling would result in the proper resolution, then processing them to make the datasets either all grayscale or all RGB. After that, I converted the datasets to TFRecords. TFRecords are an efficient data format for processing large datasets, used by TensorFlow, the ML library my models were built with [3]. Through conceptually straightforward, this process took longer than I would have liked, both in terms of runtime, running for many hours, and debugging time. All preprocessing was run on Gattaca.

I first generated MNIST digits, to test if I had properly set up everything. I was pleased with the results, because they were realistic, varied, and demonstrated progres-
Figure 5.6: Each row contains generated $32 \times 32$ handwritten digit samples at different training iterations, going down as more training time has elapsed. The top row is before any training, and clearly no structure is represented. The next rows contain samples after 621, 1182, and 7350 training iterations, from top to bottom. As shown, simpler features are learning earlier in progressive training, like basic trend that the images are mostly black, with a little bit of white and gray in the middle. Later, the general shapes of digits are learned, and eventually they are much sharper, realistic samples. These samples were $32 \times 32$, rather than the normal MNIST size of $28 \times 28$ because progressive growing requires output to be achievable through a series steps that double the resolution, so output dimensions must be a power of 2. Total training time was 21 hours, 35 minutes.

5.2.3 Generating paintings with progressive growing

I trained a GAN to generate $256 \times 256$ portrait paintings with the same approach as discussed in Section 5.2.2, except using the portrait painting data. $D$ and $G$ were much
more complex than in the prior experiments, as $D$ had 23,052,353 parameters and $G$ had 23,044,309 parameters. Training was run for over a week, but the results did not look as if they had converged fully, as results may have been cyclically improving and worsening in various ways, as shown in Figure 5.7. Even with slight training instability, I am quite pleased with the results, because they recognizably depict people in a portrait painting style. In some cases I then used SRGAN, a pre-trained conditional GAN, to scale up the results, so they would look smoother at higher resolutions.

The progressive growing approach appears to have worked well for this task. Even at the very low resolution of $4 \times 4$ $D$ and $G$ seem to have learned that various dark shades of brown and gray often make up much of the image, while lighter hues closer to the skin colors of the depicted figures are concentrated towards the middle of the frame. In many cases the output builds up and then loses recognizable structure, but much less commonly than my prior results, in Figure 5.5.

Overall I am extremely pleased with the results, because I subjectively find many generated samples to be more realistic and authentic than prior attempts at generated portrait paintings, with some favorites displayed in Figure 5.8. This is not to say they are the clearest GAN-generated renditions of people that have a non-photographic style, but I find them to be the most consistent with the traditions of portrait painting.

I am not claiming my results are the most artistic GAN-generated paintings. Looking at the those generated by the CAN, or Creative Adversarial Network, invented by Elgammal et al. [14], I feel their model best simulates deliberate human work and creativity. That said, their most compelling generated paintings, are very abstract, and were explicitly intended, in their model’s objective, to be different from established painting styles, whereas I hoped mine would fit into identifiable genres [14].

One aspect of this experiment that could been a source of image noise is the fact that there are less restrictive rules that dictate what a portrait painting may be like, than
Figure 5.7: Generated $256 \times 256$ portraits from different stages in training, organized by row, going down as more training has occurred. The top row is before training, where no structure has been learned. The second row contains samples generated after 480 training iterations, where each training iteration comprises of $G$ producing 1000 images. From the third row downwards, samples were generated after 4047, 6008, 11680, and 12000 training iterations. As is shown by the lower two rows, training did not always improve generated output. Total training time was 8 days, 13 hours, 27 minutes.
Figure 5.8: A collection of generated portraits selected for the completely subjective quality of how much they pleased me. I selected images that seemed the most realistic, as the top right and bottom left do to me. The top left and lower right were chosen because they have other interesting characteristics, while still being relatively well-formed. The two of the middle row were chosen because they are stylistically unique from most generated samples and have nice features, despite their noisy components.
there are of what a photo of a person’s face may contain, for example. Paintings are created by artists, rather than digital cameras, engineered apparatuses that presumably have a more fixed and mathematical way of representing the world. Paintings are less sharp and may vary even more in style, than standardized datasets of digital images. Further making the process difficult is that many of the portraits in my dataset were only included because they contained a person, or part of a person. In some cases training data would depict a person far away in a scene, or simply a body part; one portrait in the training set was a close up of a foot, which likely did not provide much use for the task of generating facial portraits. Not only was modeling paintings difficult because of their seemingly more organic nature, but also much of my data consisted of examples that were of a style so different that they could provide little benefit.

To explore the limits of GAN realism, I tried generating chest X-rays with GANs trained on a much larger and more standardized dataset of sharp medical images.

5.2.4 Generating chest X-rays with progressive growing

As discussed in Chapter 4, GANs can produce sharp, realistic results, as they become aware of concepts without needing labels, which may eventually allow them to be useful feature extractors for other machine learning tasks. I trained a GAN to generate 512 × 512 grayscale chest X-ray images, and they are my most realistic, and sharp results, shown in Figure 5.9. I attribute the realism of these samples to the fact that the training data were generated in a mechanical way, all produced by X-ray imaging devices focusing on the same region of the body. There were also many more training examples, 112,000 [71]. Training stability was much improved as the vast majority of outputs look plausible, with well defined structures.

Deep convolutional NNs given chest X-ray images can diagnose patients more accurately than human radiologists for a variety of conditions [53]. I thought it would
Figure 5.9: GAN-generated $512 \times 512$ chest X-rays after 9400 training iterations, where each iteration $G$ generates 1000 images. Total training time was 7 days, 8 hours, 5 minutes.
interesting to see if GANs could learn the identifying features of diseases, in an unsupervised way, especially because labeling medical data requires much more expertise than the already expensive labeling tasks of most other machine learning. Additionally, the largest available datasets were labeled using natural language processing, and so they may contain errors [71].

I invite all readers, especially those with radiology experience, to view the latent space interpolation video of this model, linked in the Appendix, to see if any individual frames of the video, each analogous to a single X-ray, look as if they are realistic and whether they may contain features that signify various medical conditions, like pneumonia, or the presence of tumors.
CHAPTER 6

CONCLUSION

At this point it should be clear that Generative Adversarial Networks are an incredibly clever and diverse framework for building powerful machine learning systems that are not only capable of generating novel, realistic data samples of many forms, but also useful for much more. They can essentially parameterize reality in the way that they map a latent space to their output space that estimates the real data space [34]. Though much emphasis in this thesis was placed on image data, GANs are theoretically applicable to any kind of data. Conditional GANs might soon be used for a great variety of data processing functions, as they simply map data inputs to realistic outputs. Semi-supervised learning and other approaches that may leverage the unsupervised learning capacities of GAN games, might provide a way to overcome fundamental limitations currently facing machine learning practitioners today, dramatically expanding the availability of training data that is usable, by making the required number of labels for a given task much fewer.

GANs take advantage of advancements in machine learning and deep learning, to allow for superhuman expressive power, in defining a model’s objective. Machine learning models are often superior at finding certain structure in data, and as a result of that it is often preferable to “program without explicitly programming”, i.e., to let the power of optimization of ML models automatically find the best solutions to problems. Considering that idea, it is reasonable that we may use ML to define our model’s objectives, as the discriminator does for the generator, in a GAN. This approach allows GANs to evaluate previously difficult to define objectives, like the apparent realness of a data sample.

Having great freedom to alter a GAN, in terms of the cost functions used, the architectures of individual players as well as that of whole GAN, the utilization of auxiliary information, and other enhancements, researchers have incentivized generators to create output to not only be realistic, but also varied, and creative, by introducing relatively
straightforward dynamics into their GAN games, specifically adding terms to their cost functions that promote the desired characteristics [16, 14]. We are still at the dawn of the GAN, and it will be fascinating to see what new GAN strategies emerge, and what sorts of structure they may be able to learn, as well as what technologies they may enable.

I personally believe GANs represent an exciting step in the direction of a world where the interconnected and numerous constituents of our ML systems are not just individual neurons, but neural networks. As GANs show, connecting and utilizing individually powerful models, in a larger system, allows for more abstract problems to be tractable in the language of optimization. Maybe large networks of neural networks, or neural network networks, will become common and powerful tools, as they are comprised of more capable learning units than individual neurons. I also think this idea seems more consistent with a basic understanding of how the brain works. Many deep NNs simply map an input to output, but it’s not clear that biological brains act in that way. Intuitively, it seems more likely that the brain is like a GAN, lacking a single input and output, but comprised of multiple networks, which may have their own inputs and outputs, with complex interactions between them.

Returning to Ian Goodfellow’s original idea, isn’t it funny that the introduction of adversarial relationships to neural networks is what gives the otherwise less inspiring models a human-like capacity to create? What other interactions are out there?
APPENDIX A

VIDEO RESULTS: LATENT SPACE INTERPOLATIONS

Please view these example videos of latent space interpolations of my trained models:

(a) QR code link to Chest X-rays GAN latent space interpolation video

(least interesting):
https://youtu.be/NBxIigNCryk

(b) QR code link to Portrait GAN latent space interpolation video:

https://youtu.be/OouuNmFud78

(c) QR code link to MNIST GAN latent space interpolation video:

https://youtu.be/0nZ5XJuhfHY
BIBLIOGRAPHY


[40] Cade Metz. AI is transforming google search. the rest of the web is next, Feb 2016. https://www.wired.com/2016/02/ai-is-changing-the-technology-behind-google-searches/.


[71] Xiaosong Wang, Yifan Peng, Le Lu, Zhiyong Lu, Mohammadhadi Bagheri, and Ronald M. Summers. Chestx-ray8: Hospital-scale chest x-ray database and bench-