Matthew Spillman

Dr. Chanley Small

Honors Independent Study

4 December 2020

Deep Learning: A Journey into Artificial Neural Networks and their Applications

Background: Literature Review of Machine Learning and Deep Learning

Machine learning is a discipline in the field of artificial intelligence which seeks to make machines learn from data. Recently, a class of machine learning algorithms has achieved groundbreaking results in machine learning tasks such as computer vision and natural language processing. However, these algorithms are somewhat poorly understood, and they rely on large amounts of data and processing power. The goal of my independent study is to gain a better understanding of this approach to machine learning, called "deep learning", and determine its usefulness when applied to more modest datasets.

AI and Machine Learning

Computers are programmable calculators. They can manipulate numbers, store the results, and perform new calculations based on the results of previous ones. Humans write programs which control how these calculations are carried out. The field of artificial intelligence (AI) seeks to make these programmable computers perform tasks which typically demand human intelligence (Nilsson, 2014, p. 1). Early successes in the field of AI involved "problems that are intellectually difficult for human beings but relatively straightforward for computers – problems that can be described by a list of formal, mathematical rules" (I. Goodfellow et al., 2016). These problems are easy to define and they leverage the ability of computers to perform rapid calculations. For example, the computer that beat world chess champion Garry Kasparov in 1997 was programmed to search the game tree for better moves, and it evaluated 50 to 100 million chess positions per second (Campbell et al., 2002). Using similar brute-force methods, modern computers can outperform humans in many such easily-defined tasks.

However, many of the most challenging AI problems are “easy for people to perform but hard for people to describe formally”, such as image and speech recognition (I. Goodfellow et al., 2016). When humans can’t describe a problem formally, they can’t design algorithms which utilize computers’ processing power effectively. To program a computer to classify images of animals, for example, a programmer would have to consider all possible combinations of poses, camera angles, lighting conditions, and backgrounds, which is essentially impossible. In these cases, it is better to provide lots of data about the problem and let an algorithm learn a solution on its own. In the animal classification example, the programmer could supply an algorithm with a large dataset of animal images and labels indicating which animal is in each image, and an algorithm could learn what each animal looks like automatically. This approach is called machine learning, or ML (Géron, 2019, p. 4). There are three main types of machine learning: supervised learning, unsupervised learning, and reinforcement learning.

Supervised learning is the most common type of machine learning (LeCun et al., 2015, p. 1). It involves giving the machine labeled data (Géron, 2019, p. 8). For example, an audio clip could be labeled with the words the clip contains, or an area’s demographic/geographic information could be labeled with the housing prices in the area. An algorithm is trained on the labeled dataset. Then, when presented with an unlabeled data instance, it attempts to predict the label that should go with that instance. Classification tasks typically require supervised learning (Géron, 2019, p. 8).

Unsupervised learning involves unlabeled training data (Géron, 2019, p. 10). Though the data does not have labels, unsupervised algorithms can model the distribution of the data. This often helps find distinct groups and outliers. Unsupervised learning is useful in data visualization and anomaly detection, among other things (Géron, 2019, p. 12).

Reinforcement learning is significantly different from supervised and unsupervised learning. In reinforcement learning, the algorithm, called an "agent" in this context, is able to take actions in an environment, like a robot in a room or a player of a board game. The algorithm is given rewards or penalties for its actions, the criteria of which are decided by the programmer. The algorithm attempts to learn what it needs to do to maximize its reward. A common application of reinforcement learning is teaching robots how to walk (Géron, 2019, p. 14). Additionally, the first program to beat human professionals in the game of Go primarily used reinforcement learning (Silver et al., 2016).

Deep Learning and Neural Networks

Traditional ML techniques have contributed significantly to the development of useful software, notably driving the development of spam filters (Géron, 2019, p. 3). However, many of these techniques "were limited in their ability to process natural data in their raw form" (LeCun et al., 2015, p. 1). Typically, human programmers had to hand-design algorithms which extracted the raw data's most useful information before applying machine learning techniques. This often "required careful engineering and considerable domain expertise" (LeCun et al., 2015, p. 1). Recently, though, a class of machine learning algorithms has achieved state-of-the-art results without the use of hand-designed feature extractors. These algorithms apply a series of learned transformations to their input data, with each transformation extracting more complex features from its input. The first transformations automatically convert the raw data to a more useful representation, thereby reducing the need for hand-designed feature extractors. This set of techniques has been dubbed "deep learning" (LeCun et al., 2015, p. 1). Deep learning has beaten previous ML records in image recognition and speech recognition, and has "produced extremely

promising results" in the field of natural language processing (LeCun et al., 2015, p. 1). For example, when Google Translate switched to a deep learning model in 2016, translation errors decreased by 60% (Wu et al., 2016, p. 1).

Deep learning models transform their inputs using algorithms called neural networks. Neural networks consist of a series of layers, where each layer contains some "neurons" activated conditionally by the neurons in the previous layer. The degree to which a neuron activates is controlled by the strength of the connections between that neuron and the previous layer's neurons. These strengths, or "weights", as well as "bias" terms associated with each neuron, determine the manner in which each layer transforms its input (Hecht-Nielsen, 1989). A neural network with many layers is considered "deep", hence the name "deep learning". Although deep learning only recently came to the spotlight of ML research, neural networks have been used since at least the 1980s, when researchers discovered how to train them (LeCun et al., 2015, p. 3). The training algorithm, called "backpropagation", involves feeding a labeled input into the network, then calculating the difference between the network's output and the target label. Using calculus, this error can be *propagated back* through the network to automatically determine how each weight and bias must be changed to reduce the error (Hecht-Nielsen, 1989).

Though the backpropagation algorithm was discovered as early as the 1980s, neural networks "were forsaken by the machine-learning community" for much of their existence because they did not perform as well as other algorithms (LeCun et al., 2015, p. 3). Researchers thought they got trapped in poor local minima and that creating a good automatic feature extractor was impossible (LeCun et al., 2015, p. 3). In reality, neural networks failed due to a myriad of factors, including poor weight initialization, insufficient compute, and the "vanishing

and exploding gradient problem" caused in part by poor choice of the neurons' activation functions (I. Goodfellow et al., 2016, p. 286). In 2006, a group of researchers discovered a way to pre-train networks in an unsupervised fashion such that the "weights of a deep network could be initialized to sensible values" (LeCun et al., 2015, p. 4). This revived interest in deep neural networks, and subsequent research revealed and solved many of the issues which arise when training using backpropagation. Now, neural networks and deep learning have "been widely adopted by the computer-vision community" and other machine learning fields (LeCun et al., 2015, p. 4).

The Drawbacks of Machine Learning and Deep Learning

Despite the widespread adoption of deep learning within the machine learning community, the behavior of neural networks is still somewhat poorly understood. Because neural networks often consist of millions of learned weights and biases, their predictions "can be difficult to interpret and can have counter-intuitive properties" (Szegedy et al., 2014, p. 1). Recent research found that neural networks can be fooled into labeling unrecognizable images as recognizable objects with 99.99% confidence (Nguyen et al., 2015, p. 1). Furthermore, correctly classified images can be imperceptibly altered such that a neural network misclassifies them with extremely high confidence (Szegedy et al., 2014, p. 1). These altered images, called "adversarial examples", raise questions about the security of using deep learning models in important applications. Such models could potentially be exploited by attackers using adversarial examples. In fact, many classes of machine learning models are susceptible to adversarial examples, making it a problem with ML in general as well (I. J. Goodfellow et al., 2015, p. 1).

Another point of concern is that deep learning relies on potentially unsustainable amounts of data and computational power. In the field of computer vision, there is "unanimous agreement" that the recent revolution came about primarily due to larger datasets and more powerful processors, rather than the improvements made to neural network optimization (Sun et al., 2017, p. 1). The neural networks used in computer vision tasks (such as image classification) use a specialized architecture introduced in the early 1990s which "achieved many practical successes during the period when neural networks were out of favour" (LeCun et al., 2015, p. 4). This architecture has remained mostly unchanged since then, while hardware improvements have increased its performance. However, it is becoming increasingly difficult to make processors more powerful. Historically, the number of transistors on a silicon chip doubled approximately every two years, a principle known as Moore's law. Now, however, chip components are nearly so small that "quantum uncertainties" would make transistors "hopelessly unreliable" (Waldrop, 2016). With "no obvious successor to today's silicon technology", the raw power of processors is likely to grow more slowly in the future (Waldrop, 2016).

Additionally, the datasets and metrics used to train ML algorithms introduce problems of their own. For example, one algorithm widely used in the U.S. healthcare system attempted to predict patients' future healthcare costs, which would then be used to refer higher-cost patients to special care programs (Obermeyer et al., 2019, p. 1). This assumes that the cost of a patient's care is a good estimator of the patient's health risks. However, at a certain level of health, black patients' healthcare costs are on average $1801 less per year due to a number of factors related to poverty and discrimination by primary care providers (Obermeyer et al., 2019, p. 4). Consequently, the algorithm correctly assigned lower cost predictions to black patients than to white patients of similar health. This led to the white patients being enrolled in special care

programs more often. In this instance, the metric used to train the algorithm was poorly chosen, so its recommendations were biased. In other cases, the dataset is fundamentally flawed. An analysis of facial recognition algorithms found that one trainable and several non-trainable algorithms perform consistently worse on female, black, and young (18 to 30 years old) faces (Klare et al., 2012, p. 1). While the performance of the algorithms suggested that all these cohorts are inherently harder to recognize, the performance of the trainable algorithm increased for black, young, and middle-aged faces when trained only on faces belonging to those cohorts (Klare et al., 2012, p. 10). In this case, the distribution of the data adversely affected performance in a biased manner. These examples demonstrate how ML models trained using bad data or bad evaluation metrics can silently replicate biases which exist in the real world.

Original Component

GitHub repository containing all source code: https://github.com/MSpill/Independent-Study-ML

RNN folder contains source code for part 1, GAN folder contains source code for part 2.

Part 1: Implementing a Neural Network and Backpropagation

To further familiarize myself with neural networks and the backpropagation algorithm, I chose to implement a full neural network training pipeline from scratch as the first part of my independent study. Most real-world deep learning is aided by code libraries which handle the implementations of the networks and backpropagation automatically. This removes unnecessary work and allows programmers to focus on managing the training data and the high-level network structure. However, to understand the challenges and pecularities of neural network training, I wanted to understand the backpropagation algorithm. The best way to understand this algorithm is to implement it.

Before implementing backpropagation, though, I had to implement a neural network. For the architecture, I chose to implement a modified neural network called a recurrent neural network, or RNN. RNNs attempt to address a weakness of normal neural networks. Normal neural networks require that their input be a fixed size: if the input layer is 7000 units wide, the input to the network *must* consist of 7000 numbers. This is problematic if you want to process data of an unknown or variable size (for example, text or video). RNNs resolve this issue by allowing any number of data points to be sent through the network sequentially. Instead of completely resetting when a new data instance is sent through the network, RNNs maintain an internal state which allows them to "remember" some information from previous inputs. This is achieved through the use of *recurrent connections*: each layer's output depends not just on the

output of the layer before it, but also on its own previous output. Normal feedforward weights control how each layer processes the previous layer's output, and recurrent weights control how each layer processes its own previous output. This allows the network to process sequential data by evaluating each data instance in the context of earlier data instances.

I chose this architecture for two reasons. First, I suspected that there would be few resources describing in detail the backpropagation variant for RNNs (called "backpropagation through time"). This meant I would have to derive the necessary formulas and implement them myself. Second, vanilla (traditional) RNNs are notoriously difficult to train. The more inputs the network receives in one run, the deeper its "unrolled" version will be, and this depth slows training in the absence of advanced techniques. I hoped implementing this variant would allow me to encounter the vanishing gradient problem and similar backpropagation issues in practice.

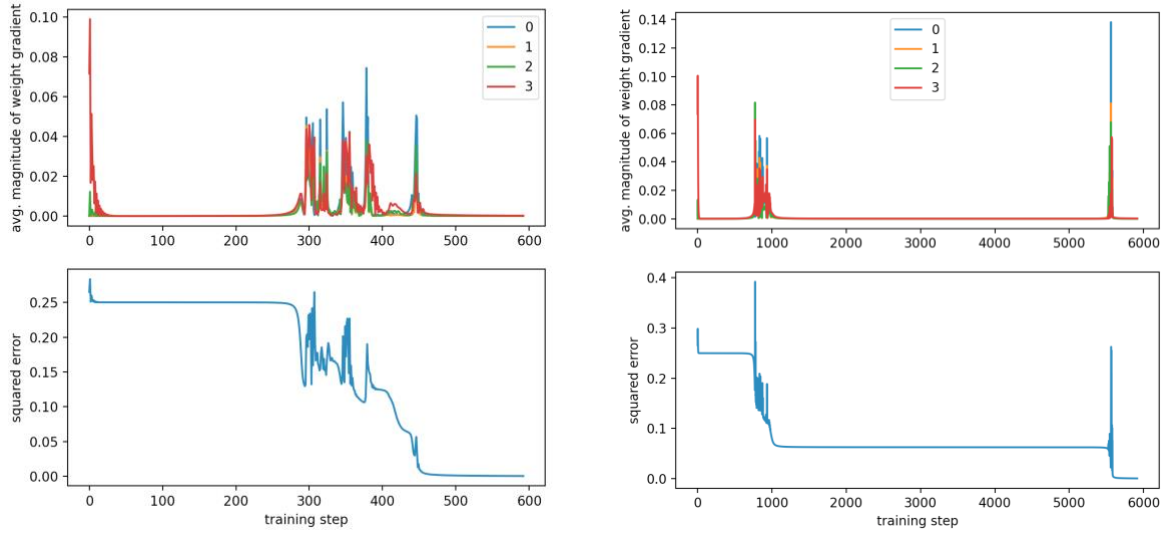
Implementing the RNN proved fairly easy. The fundamental formulation of neural networks is simple, and the addition of recurrent connections did not significantly complicate the implementation. I used the Python programming language, because Python is home to many useful data processing libraries and, once I reached part two of my original component, machine learning libraries. While writing the RNN, I used the numpy library to handle matrix multiplication. Matrix multiplication is not difficult to program, but numpy does it efficiently, which sped up the network and its training. Ultimately, the RNN implementation was done in a week.

Next, I tackled the predictably more challenging task of RNN training. After a few days of taking partial derivatives, I roughly determined the general procedure for finding weight updates. Then, I spent several weeks translating this procedure into code and testing on simple

tasks. Once I ascertained that the training code was functional, I began training RNNs on three text generation tasks of varying difficulty.

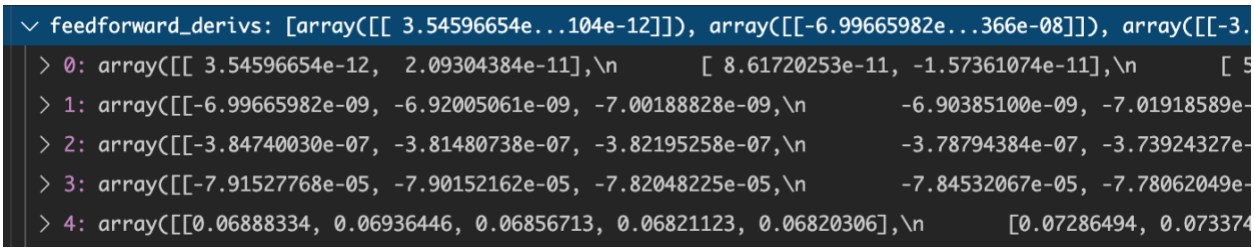
The first text generation task was by far the easiest: the input to the network was either a one or a zero, and the goal was to output a one if the previous two inputs were ones, and zero otherwise. The next task was more difficult: given a genetic base sequence, predict the next base. Although accuracy would always be low, the RNN might use base frequencies and common base triplets to make its prediction. The third task was similar to the second, but the sequence to predict from was a segment of C code instead of genome data.

Ultimately, the RNNs I trained only succeeded in the first task, where inputs were ones and zeros. During training, many issues with backpropagation became apparent. The training was prone to long plateaus of near-constant error followed by potentially unstable periods of rapid change. In the two graphs below, error and weight update magnitudes are tracked as the RNN trains twice on the first task.



In both graphs, the error remains nearly the same for the majority of the training time, then decreases suddenly. In the left graph, error goes from 0.25 to 0 in a very unstable 150 training

steps, whereas in the right graph there are two plateaus at error values of 0.25 and 0.0625. The upper graphs show that the weights were changed the most during these intermediate periods and were barely changing while error plateaued. These plateaus may be a result of the vanishing gradient problem. A closer inspection of weight gradient magnitudes in a larger network reveals that weights closer to the output receive much larger updates than those far from the output:



The numbers in the fifth row (labeled 4) represent changes to weights closest to the output. They average around 0.07. The row above represents weight updates one layer further back. Their average magnitude is approximately $7*10^{-5}$, which is 1,000 times smaller than 0.07. Similarly, each subsequent row has values 100 to 1,000 times smaller than the row below it. These vanishing weight updates make training very slow, because the first layers aren't able to learn any useful feature extraction. In fact, the slow-training first layers seem to inhibit the faster-training layers by needlessly mangling the input data.

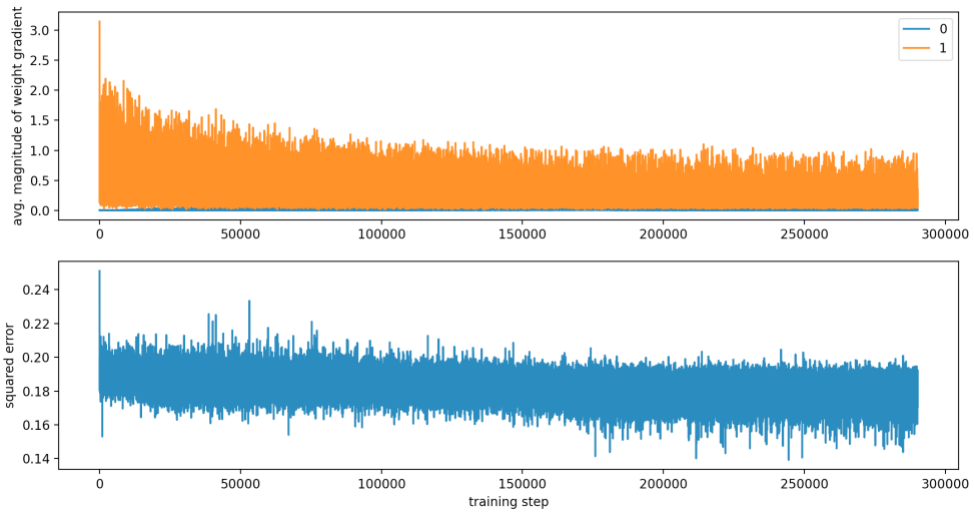
Using the backpropagation equations, I can demonstrate why these vanishing gradients occur. The update equation for a weight closest to the output is this:

$$-\frac{\partial E}{\partial w_{mn}} = -2(O_n - T_n) * \frac{\partial}{\partial w_{mn}} f\left(b_n + \sum_{j=1}^{J} w_{jn} H_j\right)$$

$$= -2(O_n - T_n) * f'\left(b_n + \sum_{j=1}^{J} w_{jn} H_j\right) * H_m$$

The details of each factor are unimportant (without more definitions, that expression would be impossible to interpret). However, it is important to note the magnitude of each factor in the last expression. $(O_n - T_n)$ represents the difference between a neuron's expected output and actual output. Since neurons on the output layer are typically constrained to output values between 0 and 1, this factor will be between 0 and 1. $f'\left(b_n + \sum_{j=1}^{J} w_{jn} H_j\right)$ represents the derivative of the neuron's activation function. For sigmoid neurons (the type I used) that derivative is always between 0 and 1. Finally, H_m represents the output of a neuron one layer back. As with a neuron on the output layer, this is likely to be between 0 and 1. Therefore, each factor in this equation is between 0 and 1. The further from the output we go, the more factors these equations contain. Each factor is less than one, so the more of them we multiply together, the smaller the product will be. Because there are more of these small factors in the updates for far-back weights, those updates become vanishingly small.

Because of these training challenges, none of my RNNs were able to perform the genome base prediction or C source code prediction tasks. For the C prediction task, the error never noticeably decreased in several days of training. For the base prediction task, error did seem to decrease slightly over time, indicating the network may have been learning some patterns:



Still, base sequence samples generated from the RNN never had nucleotide frequencies matching real genome data.

By implementing a modified neural network structure, I furthered my understanding of backpropagation and encountered its difficulties firsthand. Unfortunately, the vanishing gradient problem, which plagued neural networks for decades, also prevented my RNNs from learning meaningful relationships. Still, I had expected and hoped the vanishing gradient problem would demonstrably affect my training. With my journey into the weeds of neural network implementation complete, I moved on to the next part of my original component.

Part 2: An Image Generation Model Trained on Westminster's Campus

After investigating the inner workings of deep learning models and their training algorithms, I wanted to investigate at a higher level how data and training time could affect the performance of these models. I also wanted to collect my own training data, which I felt would make my work more original. To meet both of these needs, I decided to train a generative model on images of Westminster's campus. Generative models suited my needs in this case because they operate on unlabeled data. Rather than make predictions, which usually requires labeled data, generative models seek to emulate their training data. This eased my workload because I could simply collect data without needing to go back and manually label each example.

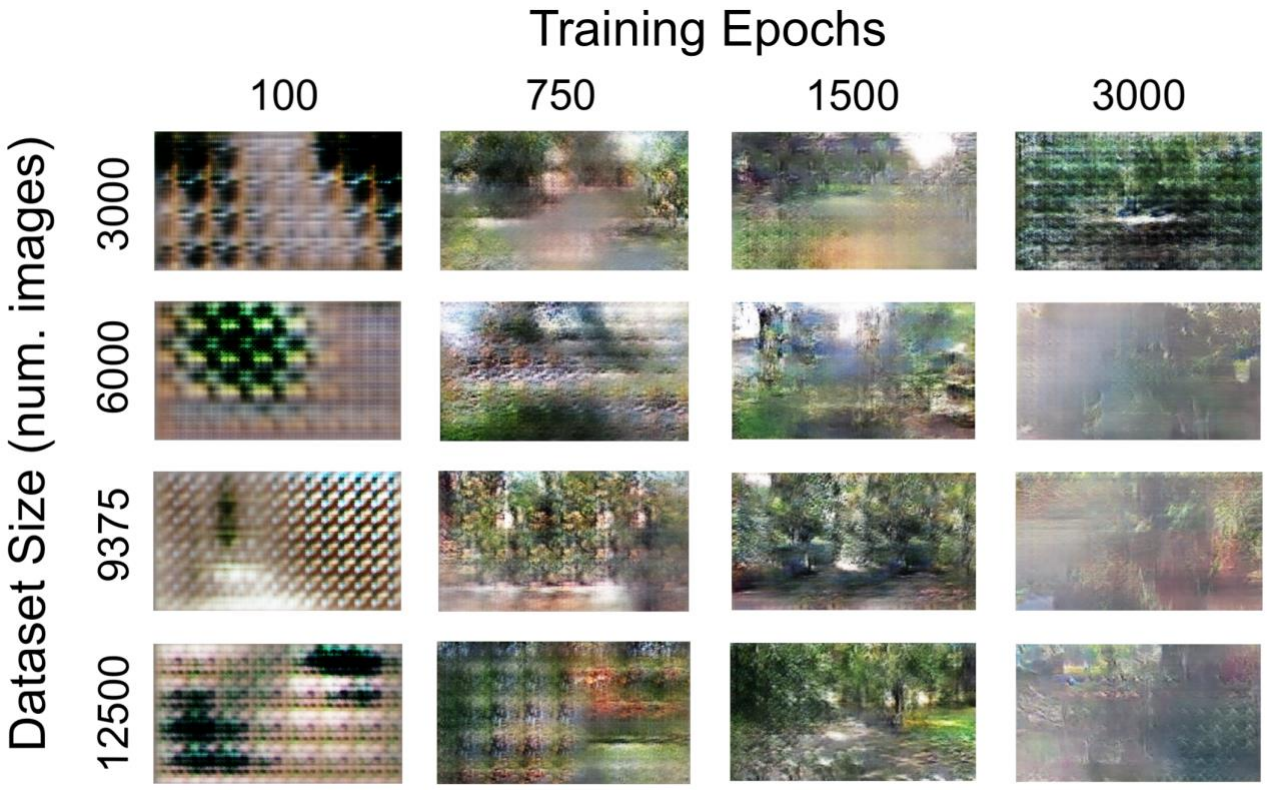
To collect my images, I walked around Westminster while taking timelapses with my phone. All images were taken outside and did not include people when possible. Once I took my timelapses, I wrote some code to extract each image and put them all in one folder. Finally, I reduced their resolution from 1920 by 1080 to 240 by 135, to make training easier. My final dataset consisted of over 12,500 unique images of the outdoor areas of Westminster's campus.

As I collected data, I began to set up the generative model. For the model's architecture, I chose to use generative adversarial networks (GANs), a popular architecture for image generation. GANs come in two parts: a generator and a discriminator. The generator network takes a random seed (often 100-dimensional) as input and outputs an image. The discriminator network takes an image as input and outputs a prediction of whether that image is real. The discriminator is trained to distinguish between real images and those generated by the generator, and the generator is simultaneously trained to fool the discriminator. By competing, both networks get better over time.

To implement this more complicated architecture, I decided to use machine learning libraries that would automate much of the work I did for the RNNs in the first part of my original component. I used TensorFlow, the most popular deep learning library for Python. I adapted GAN code from their examples to fit my image dimensions, then began training on the data I had available. Right away, I noticed that this task was much more computationally expensive than training the RNNs had been. One epoch of training (having used every example in the dataset once) took 20 minutes on my home PC and would have taken many times longer on my school laptop. It took several days of nonstop training for any substantial progress to show. Because of this increased computational demand, I decided to use computers from Westminster's VR lab. Those computers have powerful graphics cards, which are very useful for neural network training. On one of those computers, training proceeded 30 times more quickly, with one epoch taking less than 40 seconds. This allowed me to train multiple GAN versions on different subsets of my data.

I trained four GANs: one on the full dataset, one on three fourths of the dataset, one on half the dataset, and one on a quarter of the dataset. I expected the GAN trained on the full

dataset to have the best performance, and I expected the GAN trained on a quarter of the dataset to have the worst performance. The image below summarizes the results. Each row represents one of these GAN's outputs during the course of training. Epoch numbers are adjusted to signify training once on each of a total of 12,500 images.

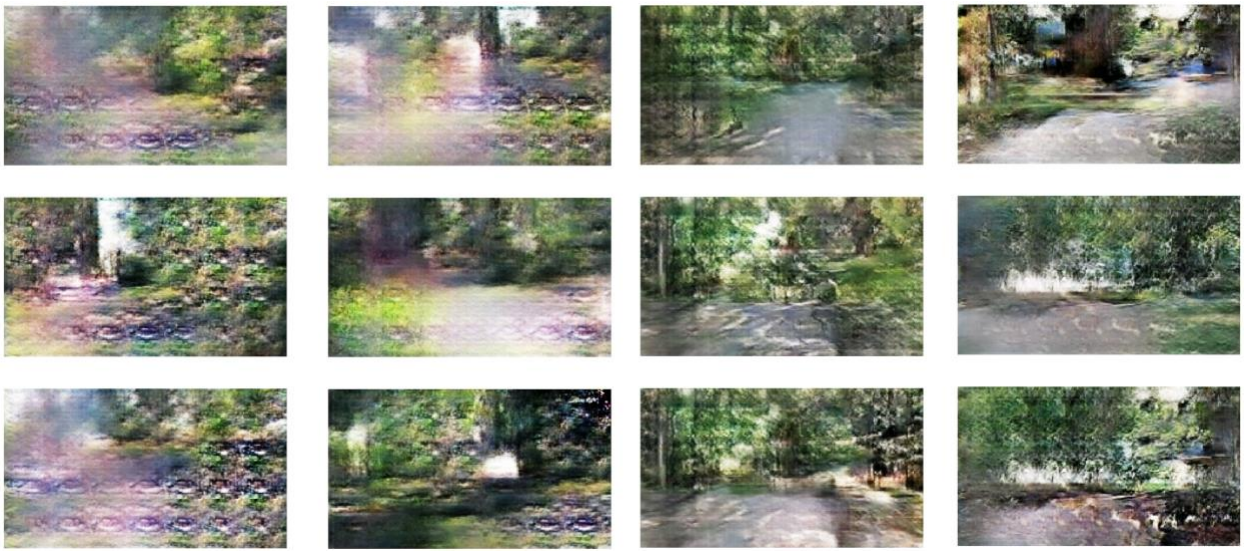


As I expected, the GAN which trained on the full 12,500 images appears to perform the best. However, the performance of the other three GANs is more difficult to judge. Though the GAN which trained on 3,000 images produced the least defined images, the color and composition of its generated images appears more accurate. Overall performance still appears to have increased as the size of the training set increased, though the difference in performance was smaller than I expected.

More interestingly, the performance of each GAN noticeably worsened after 1500 epochs of training. Three GANs collapsed to cloudy images and the other produced an unrealistic uniform green. This is common in classification tasks. When a classifier trains for too long on a dataset, it begins to memorize specific data instances instead of finding patterns, a problem called “overfitting”. This worsens generalization. Still, I was surprised to see a similar phenomenon in a generative context. This could be caused by the discriminator network overfitting the real images and rejecting anything not in the training set. If the discriminator is overfitting, this problem could be alleviated by regularization methods or an increase in the volume of training data.

The worsening performance after 1500 epochs and the apparent inability of any of the GANs to converge to photorealistic images led me to investigate the full training timeline of each GAN. By concatenating the snapshots of generated images at each epoch, I could create videos which served as timelapses of each GAN’s training. In these videos, the instability of GAN training is made even more clear. The style and quality of the outputs is subject to wild swings from epoch to epoch. Much like with RNNs, the inscrutable nature of GAN training can yield results which are difficult to control or improve.

Despite unstable training, saving networks periodically as they train allows them to be restored to their best momentary state. Doing this with the GANs yields results which, while not photorealistic, are certainly reminiscent of the data the models were trained on:



Conclusion

During my independent study, I deepened my understanding of neural networks and backpropagation by implementing a neural network variant from scratch. When I trained models of a modified NN architecture, I encountered the vanishing gradient problem and came to understand why it has hindered neural networks for so long. Next, I investigated how data and training time affect neural network training on a higher level. I collected 12,500 images of Westminster's campus and trained four GANs on different subsets of that data. While performance increased with more training data, the amount of data did not have as large of an impact as I expected. GAN training proved unstable and none of my models converged to photorealistic images, though the best generated images certainly resembled the training data.

Now that I have completed my original component, I have a much clearer perspective on the practical applications of deep learning. While deep learning has driven recent developments in image recognition, natural language processing, and other important fields, it remains difficult to apply to small and unique problems. When time, data, and computational power are abundant, deep learning is most useful. If that is not the case, however, unstable training and inscrutable

results can make deep learning difficult to apply to new datasets. Despite the challenges I encountered while training my models, I came to understand deep learning more thoroughly and I remain fascinated by the field.

References

Campbell, M., Hoane, A. J., & Hsu, F. (2002). Deep Blue. *Artificial Intelligence*, *134*(1), 57–83. https://doi.org/10.1016/S0004-3702(01)00129-1

Géron, A. (2019). *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O'Reilly Media, Inc.

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning*. MIT Press.

Goodfellow, I. J., Shlens, J., & Szegedy, C. (2015). Explaining and Harnessing Adversarial Examples. *ArXiv:1412.6572 [Cs, Stat]*. http://arxiv.org/abs/1412.6572

Hecht-Nielsen. (1989). Theory of the backpropagation neural network. *International 1989 Joint Conference on Neural Networks*, 593–605 vol.1. https://doi.org/10.1109/IJCNN.1989.118638

Klare, B. F., Burge, M. J., Klontz, J. C., Vorder Bruegge, R. W., & Jain, A. K. (2012). Face Recognition Performance: Role of Demographic Information. *IEEE Transactions on Information Forensics and Security*, *7*(6), 1789–1801. https://doi.org/10.1109/TIFS.2012.2214212

LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *Nature*, *521*(7553), 436–444. https://doi.org/10.1038/nature14539

Nguyen, A., Yosinski, J., & Clune, J. (2015). *Deep Neural Networks Are Easily Fooled: High Confidence Predictions for Unrecognizable Images*. 427–436. https://www.cv-foundation.org/openaccess/content_cvpr_2015/html/Nguyen_Deep_Neural_Networks_2015_CVPR_paper.html

Nilsson, N. J. (2014). *Principles of Artificial Intelligence*. Morgan Kaufmann.

Obermeyer, Z., Powers, B., Vogeli, C., & Mullainathan, S. (2019). Dissecting racial bias in an algorithm used to manage the health of populations. *Science*, *366*(6464), 447–453. https://doi.org/10.1126/science.aax2342

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., & Hassabis, D. (2016). Mastering the game of Go with deep neural networks and tree search. *Nature*, *529*(7587), 484–489. https://doi.org/10.1038/nature16961

Sun, C., Shrivastava, A., Singh, S., & Gupta, A. (2017). Revisiting Unreasonable Effectiveness of Data in Deep Learning Era. *2017 IEEE International Conference on Computer Vision (ICCV)*, 843–852. https://doi.org/10.1109/ICCV.2017.97

Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D., Goodfellow, I., & Fergus, R. (2014). Intriguing properties of neural networks. *ArXiv:1312.6199 [Cs]*. http://arxiv.org/abs/1312.6199

Waldrop, M. M. (2016). The chips are down for Moore's law. *Nature News*, *530*(7589), 144. https://doi.org/10.1038/530144a

Wu, Y., Schuster, M., Chen, Z., Le, Q. V., Norouzi, M., Macherey, W., Krikun, M., Cao, Y., Gao, Q., Macherey, K., Klingner, J., Shah, A., Johnson, M., Liu, X., Kaiser, Ł., Gouws, S., Kato, Y., Kudo, T., Kazawa, H., … Dean, J. (2016). Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation. *ArXiv:1609.08144 [Cs]*. http://arxiv.org/abs/1609.08144