Robot Artist

Dhruv Mangtani and Justin Tian

ASR Second Semester Project - Final Paper

May 1st, 2020

Abstract

We designed and built a robot artist that is capable of creatively drawing new art. We trained a machine learning model to generate lifelike images and designed a printer that moves a writing utensil horizontally and vertically to draw these images. By the end of the project, the artist succeeded in drawing the images generated by the algorithm. It can print at up to 5.70 cm/s, and deviates from drawing a straight line by 0.367 mm on average. Qualitatively, the machine learning model succeeded in creating realistic art comparable to that produced by a human.

Table of Contents

Introduction

Motivation

As technology replaces and disrupts various jobs, both white and blue collar, many wonder what its limit is. Although creative tasks appear to be safe from automation, we explore the hypothesis that physical art can also be created by robots. The inspiration for this project came from the developing trend of non-conformist art that defies art's classical definitions. For example, a banana duct-taped to a wall was once valued at $120,000 (at least, before another artist decided to eat it) [1]. If art is entirely up to interpretation, we wanted to stretch its boundaries by producing a new form of art. The purpose was to determine whether a machine is capable of producing realistic, physical art without human aid, meaning that the artist could simply continue drawing new images without instruction. This level of autonomy was meant to simulate a human artist. If it can do so, there appears to be no reason that its products should not be considered authentic, given the open-ended and creative nature of the field. We realized that the technology behind AI "creativity" is quite new, so there is still plenty of space for exciting research into the topic. We had in mind how interesting it would be to watch a robot draw out a realistic image that had never been seen by the world before. Additionally, we saw the potential for making the artist design aesthetic in itself by constructing the entire project with wood, rather than traditionally used materials like metal and plastic. This project also presented a fantastic opportunity to learn more about mechanical design and engineering while implementing a new computer science algorithm.

History

The method of art generation, Generative Adversarial Networks (GAN), was created by a research scientist at Google Brain in 2014. Its applications in the industry range from security (e.g. fraud detection) to image editing. However, it has also been used for more recreational purposes such as generating fake images of people that look real or creating cartoon characters [2]. While researchers have explored generating such images digitally, the robot artist will go one step further and physically draw the image, thus imitating a human artist. This idea is not entirely new—in 1973, artist Harold Cohen created an algorithm called AARON to produce drawings. However, the computational power required to perform machine learning was simply not available at the time, so Cohen instead used manually made rules. As such, the algorithm is not considered "creative" because of its formulaic nature (see Figure 1.1). For example, it might consistently place certain images near each other in any drawing, or always draw in a singular style [3].

Since the development of GANs, the creation of higher quality, unique, and creative art has been made possible. The Rutgers University's Art and Artificial Intelligence Laboratory recently created a software that generated art realistic enough to earn exhibitions at art fairs. Their algorithm, AICAN[1], uses a modified GAN trained on historical art that is more adept at making "creative" art, rather than generating images similar to the ones it is trained on. To accomplish this, they focused on minimizing the difference between the images created and the classic art distribution, but also maximizing the difference between their art and the established styles of art. This allowed the neural network to learn how to effectively create stylistically unique art that still looked real [4] (See Figure 1.1) (See Theory section for more details).

[1] Artificial Intelligence Artist and Collaborative Creative Partner

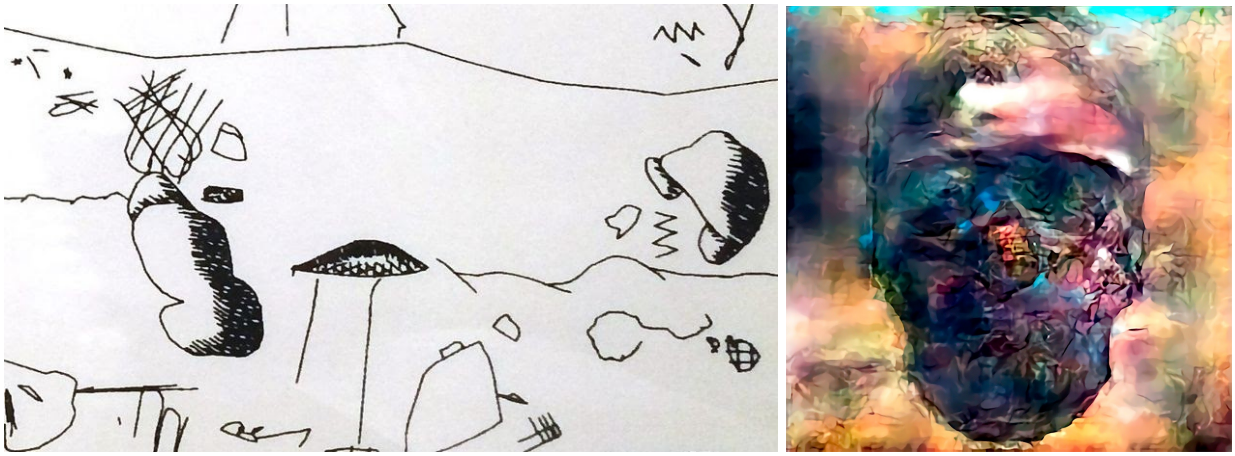


Figure 1.1: Art generated by AARON (L) vs art generated by AICAN (R) [5, 4]

Of course, our intention is not to aid robots in displacing every job in the world. While images created by AI are similar to those made by humans, man-made art will still have importance in the future because of the scarcity factor (e.g. copies of the Mona Lisa cost a lot less than the original version). A robot can always recreate the same drawing, but humans can only reproduce their creative work during their lifetimes, if it is even possible to do so.

The robot artist will be using the Cartesian coordinate plane to navigate the drawing medium. The Cartesian coordinate system was created by French philosopher and mathematician René Descartes and was first published in 1637 [6]. It is said that Descartes came up with the coordinate plane system while watching a fly on the ceiling from his bed. Descartes wondered what the best system would be to describe the location of the fly as it moved around the ceiling. He decided that one corner of the ceiling would be used as a reference point. He could then specify how far away the fly is by measuring its distance from the reference point in the horizontal and vertical directions. These two measurements are the coordinates of the fly and are unique for every location on the ceiling. Descartes' invention of the coordinate plane is significant because it created a link between algebra and geometry. Shapes could then be described algebraically, opening up many new avenues in the world of mathematics [6, 7]. For the printer, the Cartesian system is useful because coordinates can be used to describe an exact location on the drawing medium that the writing utensil must be moved to.

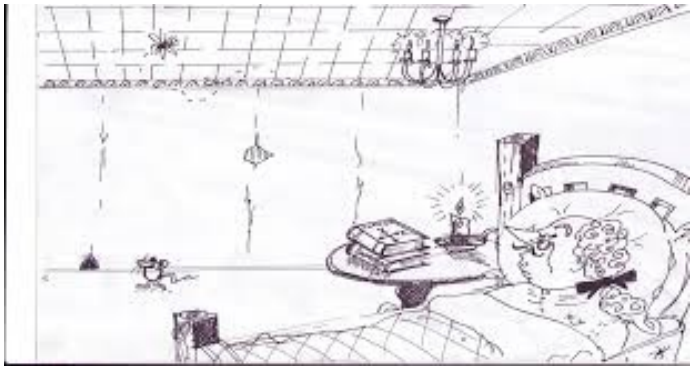


Figure 1.2: Descartes Watching a Fly on the Cartesian Grid [6]

Commercial printers also rely on the Cartesian grid system to navigate their drawing tools. The printer we created is most similar to a dot-matrix impact printer. Dot-matrix printers work by moving vertically and horizontally across the drawing medium, impacting the paper with an ink-soaked print head. The first printer of this kind was a teletypewriter called the Hellschreiber, invented by Rudolf Hell in the 1920s. The Hellschreiber printed by splitting each line of text into columns broken into pixels. It would then iterate through each column by hammering dots onto the paper medium. Hellschreibers were used during World War II by the German military to help carry out the role of the Enigma machine in transmitting secret messages. They were also useful for printing newspapers because they were relatively simple and had a few number of moving parts[8].

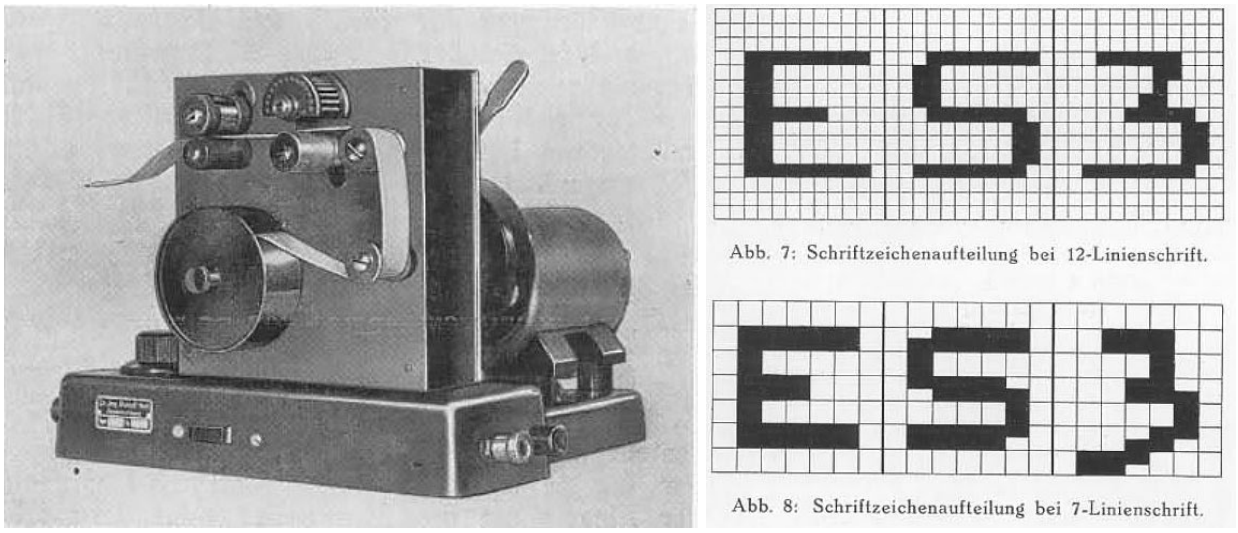


Figure 1.3: Hellschreiber (L) and printed characters (R) [9]

By 1957, IBM was manufacturing and marketing its first commercial dot-matrix printer. In the 1970s and ‘80s, these quickly became the most popular printers for use with computers due to their versatility and low cost [10]. By the 1990s, they were replaced with inkjet printers that print faster and more precisely by shooting ink droplets at specific locations on the paper medium. Today, dot-matrix printers have become nearly obsolete. Still, they are not extinct because they are cheaper than inkjet and laser printers and are reliable enough to work for many years without expensive replacements [11].

Figure 1.4: IBM-4224 Dot Matrix Printer [12]

Design

Printer

The robot artist printer design takes inspiration from the conveyor belt systems of both 3D and laser printers. We decided upon horizontal and vertical movement of the writing utensil instead of moving the paper medium itself. This design fits our vision for the printer system, since we want the printer to be modular and portable, so that it can be placed on any surface and start drawing.

The movement of the writing utensil for drawing is dependent upon three motors: two controlling vertical movement (y-axis) and one controlling horizontal movement (x-axis). The motors drive conveyor belts. Although we could have only used one motor to control vertical movement, we decided to use two motors to guarantee stability of movement across the y-axis. Stability is especially important, since the two motors control the movement of a relatively heavy wooden bar (See Figure 2.3). The x-axis motor and conveyor belt is installed in this wooden bar. An attachment that holds a writing utensil is secured on the conveyor belt, so that spinning the conveyor belt results in horizontal movement of the writing utensil (See Figure 2.4). This wooden bar is attached by both ends to the y-axis conveyor belts (See Figure 2.2). The y-axis belts are able to slide the wooden bar up and down, resulting in vertical movement of the writing utensil. The y-axis motors and belts are also encased by wooden boxes. All wooden pieces are laser cut on ¼ inch wood.

To stabilize the movement of the horizontal bar, pairs of stainless steel rails are added on either side of each conveyor belt (see Figure 2.2). The rails run through the x-axis bar and limit possible wobbling of the bar. Dry lubricant was used to address the issue of friction.

To attach the conveyor belts to separate parts, we originally wondered if we could simply nail the attachments onto the belts. We found inspiration from the conveyor belt system of a Creality Ender-3 3D printer. Instead of leaving the conveyor belt as a closed loop, the belt is cut so that there are two open ends (Figure 2.1). The open ends are threaded into slits on the attachment and fastened so that the attachment moves with the belt. This design ensures stability. The Creality Ender-3 3D printer uses small brass clips to hold the conveyor belts in place. However, these clips were not available, so instead, the belts were fastened by tying each end into a knot. The downside to this approach was that it was difficult to tie the knot so that the belt would be tight around the motor and bearing. This difficulty led to problems in the final design, as the rightmost vertical bar motor would occasionally be unable to pull the conveyor belt because the tension in the belt was too loose (to be discussed further in Next Steps).

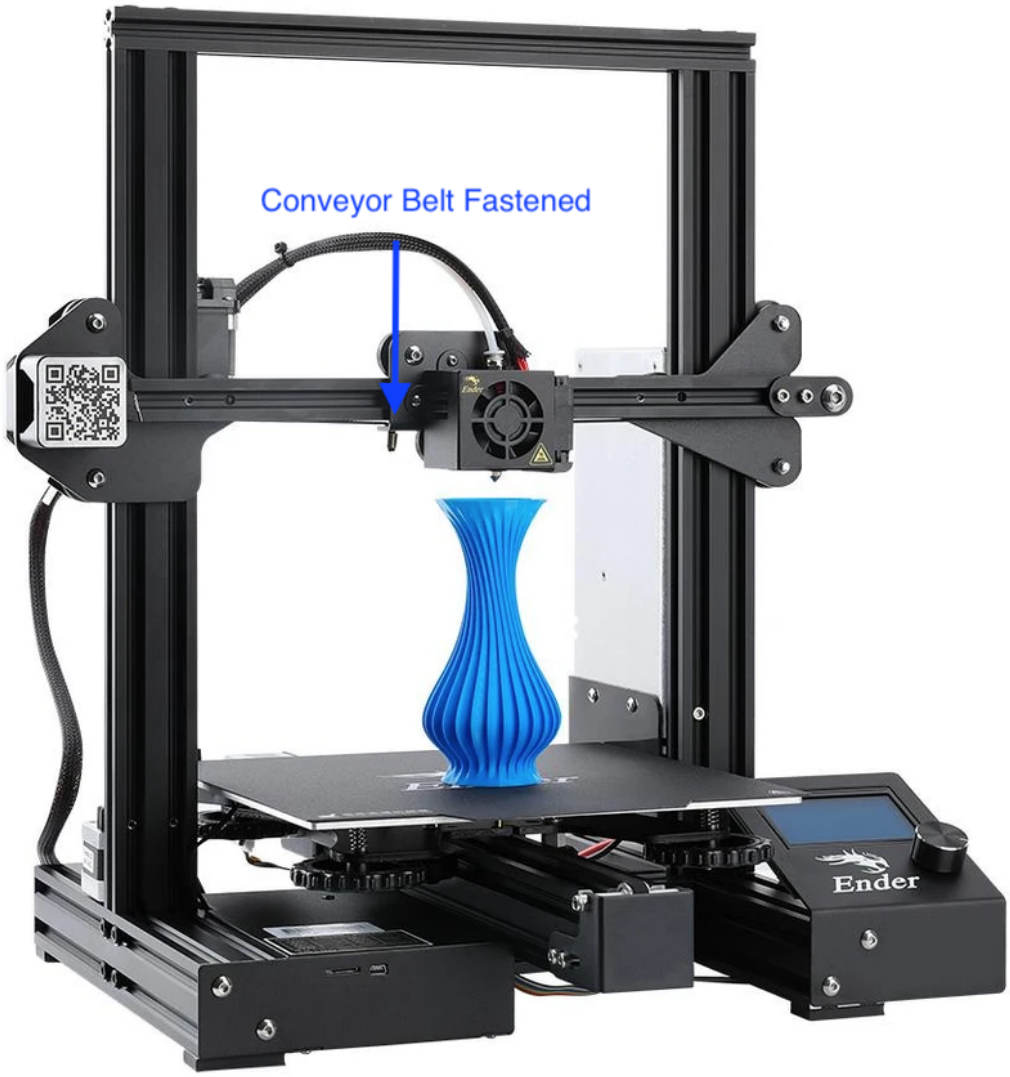


Figure 2.1: Creality Ender-3 3D Printer [13]

Similarly to an impact printer, the robot artist printer relies on pressing the drawing utensil at certain coordinates. A solenoid was used to press and retract the utensil. The solenoid is attached to six stacked slabs of wood each with a hole in the center. A writing utensil can be placed inside the hole, and a rubber band holds it in place. When the solenoid pushes down, it pushes the wooden slabs down along with the writing utensil, therefore creating up and down movement.

Drawings

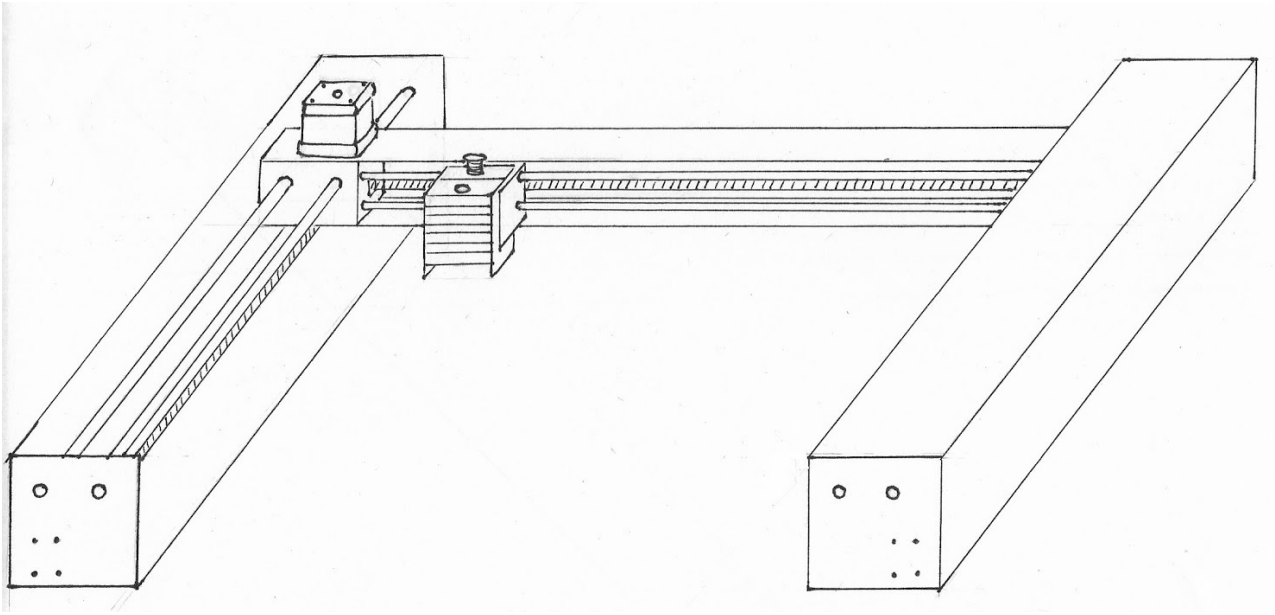


Figure 2.2: Overview of Design

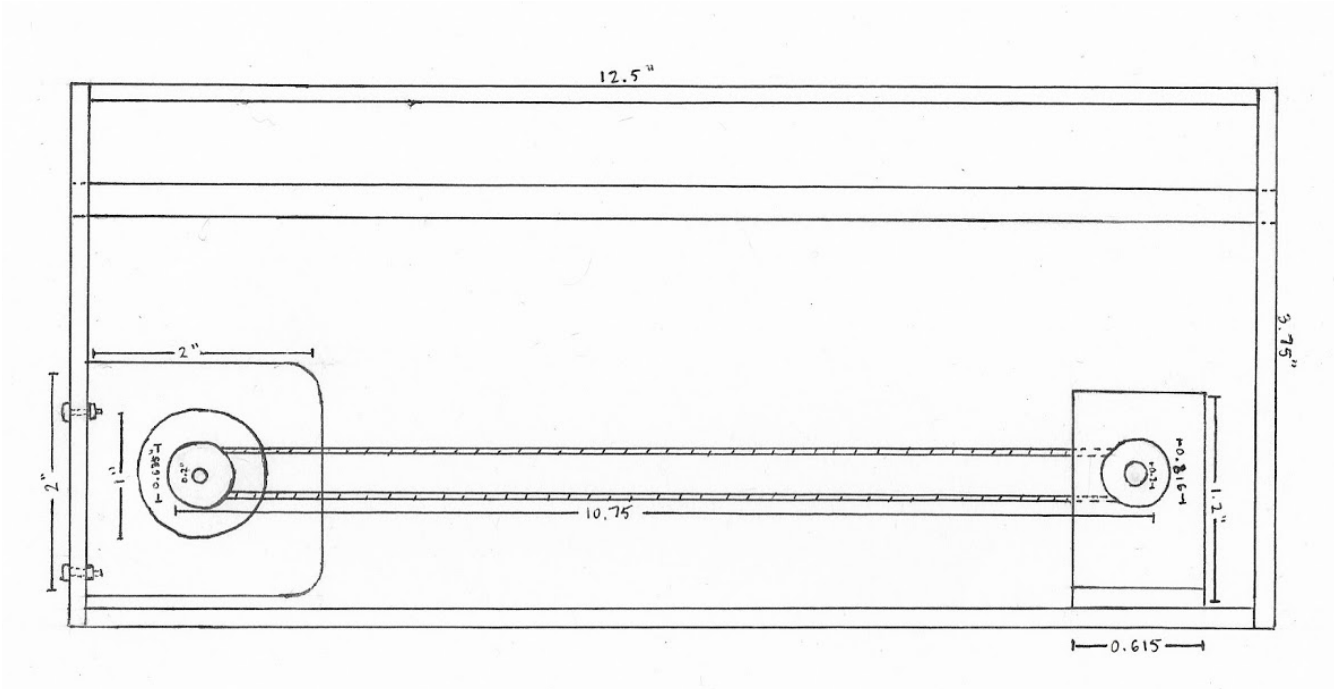


Figure 2.3: Y-Axis Design

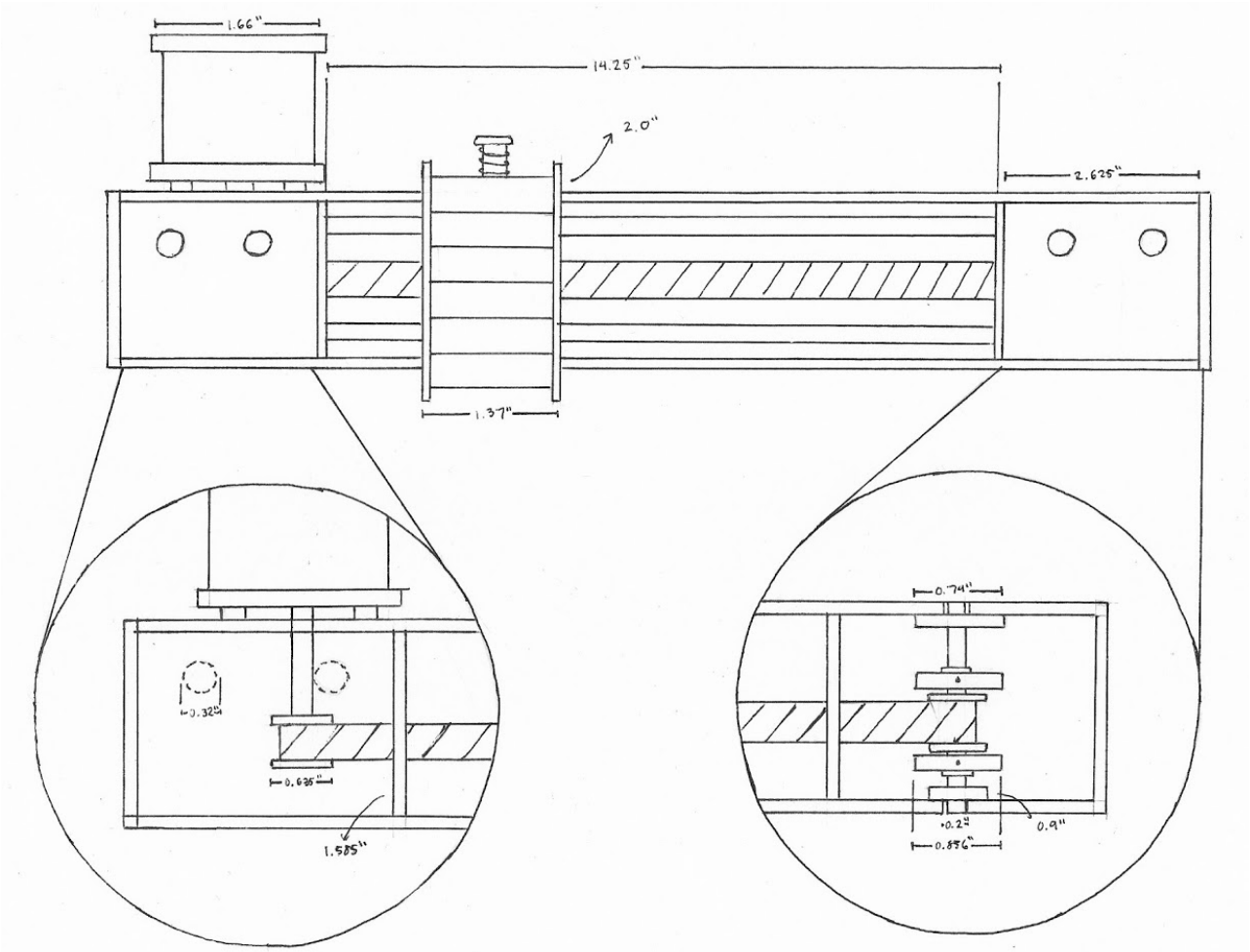


Figure 2.4: X-Axis Bar Design

Circuit

All components of the Robot Artist are driven by the Arduino Mega. The three stepper motors we used are wired to Sparkfun Big Easy Drivers and connected to a power supply that outputs 6.9V and roughly 2.1A (Figure 2.5). Each driver is connected by three wires to the Arduino: STP, which controls the microsteps taken by the stepper motor, DIR, which controls the direction in which the stepper motor turns, and a ground wire. One problem that came up when using the Sparkfun Big Easy Driver is that the driver would quickly overheat. From the power supply, it was evident that the 1.3 A drawn by each driver was too high. This problem was fixed by turning the potentiometers on the drivers. By turning the dial on the potentiometer counterclockwise, the amount of current that each driver draws from the power supply is decreased. The decrease in current comes at a price, however. When the current decreases, so does the maximum rotational speed of the stepper motor. A balance with the potentiometer was found, where the drivers drew enough current to power the stepper motors at a high enough rotational speed while limiting the current so that the drivers would not overheat.

A circuit was also needed for the solenoid. After much testing, the solenoid was determined to function stably at around 11-13V. Since the maximum arduino output voltage is 5 V, it cannot individually power the solenoid. Instead, an external power source is required to

power the solenoid. The 5V Arduino output acts as an electrical signal that turns on a switch, allowing current to flow from the external power source to the solenoid. Traditionally, a transistor would be used to complete this task; however, only a MOSFET was available. MOSFETs are used for high current applications, while bipolar junction transistors are used for low current applications. The circuit was wired so that a digital pin on the Arduino was connected to the gate pin of the MOSFET, with a resistor wired from the gate to ground (called a pull-down resistor). This pull-down resistor (10K Ohms) keeps the MOSFET turned off. Without the pull-down resistor, the pin driving the MOSFET will be an input pin until the Arduino can configure it to be an output pin. At this time, the pin will be floating, which can cause the MOSFET to be on. The pull-down resistor keeps this from happening, and ensures that the MOSFET only turns on when the 5V Arduino signal is received. The drain pin of the MOSFET is wired to the negative terminal of the solenoid, while the source pin is connected to both the negative terminal of the external power source and the Arduino ground. The positive wire of the solenoid is connected to the positive terminal of the external power source. Whenever the Arduino digital pin is set to HIGH, a 5V electrical signal is sent to the MOSFET and turns it on, allowing current to flow between the negative terminals of the solenoid and the external power supply. When this happens, the solenoid triggers, lowering the writing utensil onto the paper. The circuit can be seen in Figure 2.5.

The external power supply used to power the solenoid was a makeshift 12V battery. Since an actual 12 volt battery was not available, a 9V battery was wired in series with two 1.5 volt batteries. The two 1.5 V AA batteries were taped on top of the 9V battery, and wires were secured to the positive terminal of one AA battery and the negative terminal of the other AA battery. Nevertheless, the makeshift 12 volt battery worked in powering the solenoid.

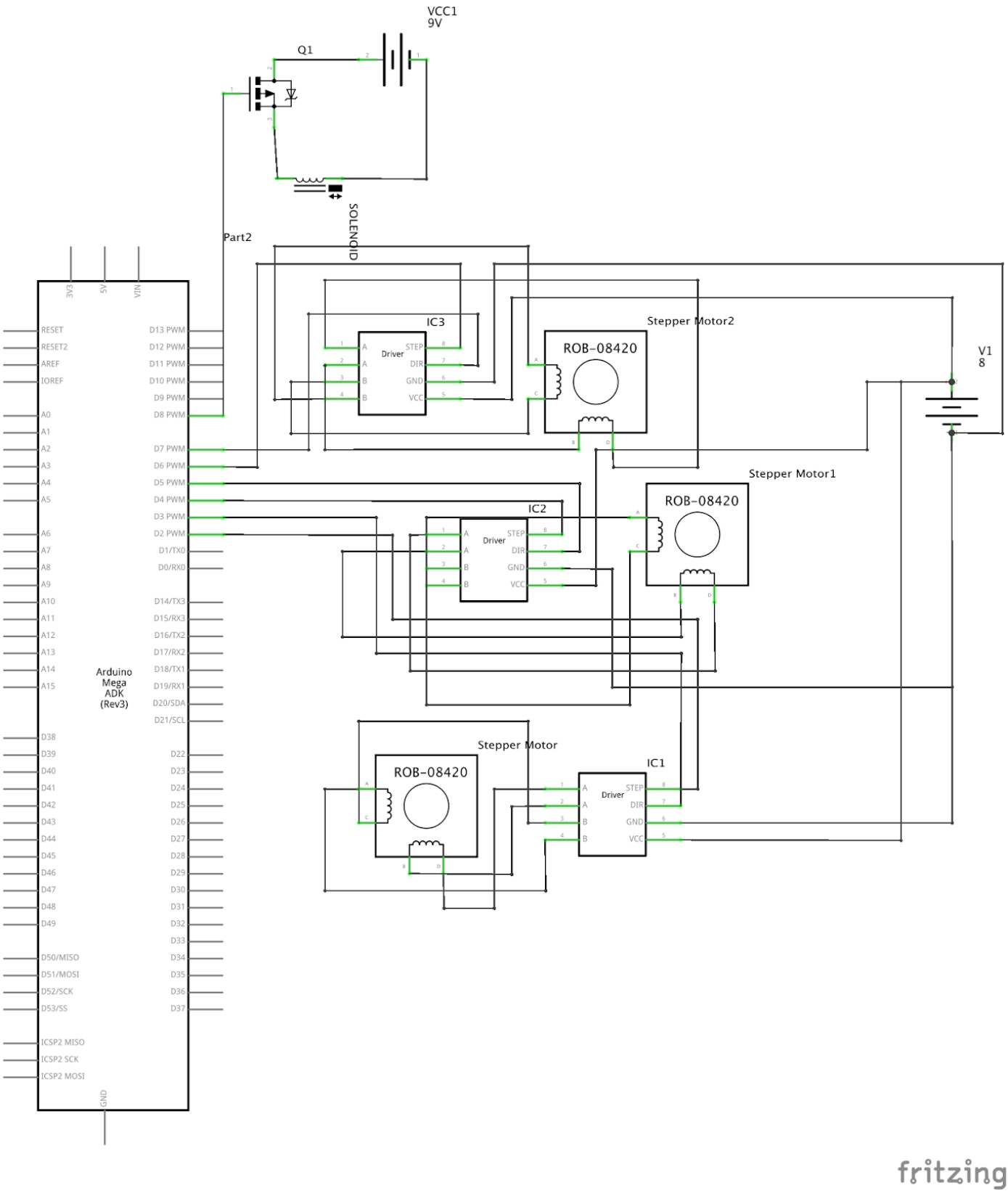


Figure 2.5: Schematic Circuit Diagram with 3 Stepper Motors and Drivers and 1 Solenoid

Code

The code for the robot artist is split into three files. The first file (in Python) is for training the GAN network and storing the trained model. The second file (in Python) is designed to generate new artwork using the trained model created by the first program. The third file (in Arduino code) prints the generated image. The following are descriptions of each file.

Machine learning libraries allow developers to design models without worrying about recreating and optimizing training algorithms (with their respective equations) to perform efficiently. The PyTorch library was chosen because of its relative simplicity, popularity, built-in linear algebra libraries, and multi-thread processing optimizations. The code used to train the GAN was slightly modified from a tutorial created by the PyTorch developers (see Appendix D). The GAN was trained on the CelebFaces Dataset which consists of 202,599 human faces [14]. After training the model, the code converted it into a "state_dict" which is a Python dictionary that represents the parameters it learned during training [15]. The state_dict was then saved into a file and uploaded with the API code to AWS S3 (see *Server Architecture*).

The end-to-end art-generating process is as follows (see Figure 2.6). First, the printer makes an HTTP request to the online API created using the AWS APIGateway service. The API makes a call to AWS Lambda and the code and state_dict are retrieved from AWS S3 (see *Server Architecture*). The code works by importing the state_dict to recreate the trained GAN. It then produces a random noise vector. This random noise is fed to the GAN generator network, which produces a new piece of art (see *Generative Adversarial Network Architecture*) of size 60x60 pixels. The API code converts the art into a binary bitmap, where every pixel is represented by a 0 or 1. The bitmap is returned back to the printer over the same HTTP connection.

The 2D array bitmap is then "stepped" through by the printer. The printer starts with the first row of pixels and progresses through each one using a for loop. If the current pixel is a 1, the Arduino writes *HIGH* to the solenoid which receives current via the MOSFET and pushes the pen down onto the paper. If it is a 0, the Arduino writes *LOW* to the solenoid and the current switches off, retracting the pen. After each row is complete, the x-axis bar moves the solenoid back to the left-most position, and the pair of y-axis stepper motors moves the x-axis bar down the paper by a pixel. Because the stepper motors are precise to 0.8 rotational degrees, the individual movements by each of the y-axis stepper motors appear simultaneous, and a straight line can be drawn when the pair moves in coordination. The stepper motors are moved by modulating the input to their drivers' STP ports on the Arduino Mega. The direction is similarly flipped by modulating their DIR ports.

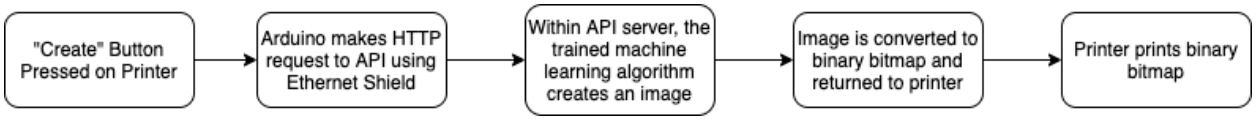


Figure 2.6: End-to-End Printing Process

Server Architecture

In order to build the API, three AWS services are used: S3, Lambda, and APIGateway. AWS Lambda is a serverless computing service, meaning that developers can create functions in the cloud without building dedicated servers. The Lambda function generates a new image when it is called. In order to build an API for the function, we used APIGateway. Specifically, a REST API was created because Amazon does not make Lambda functions compatible with HTTP APIs

[16]. Lambda functions can be used without an API; however, this method is much more difficult given the lack of AWS Arduino libraries. The API allows us to make a URL for the Lambda function, allowing the Arduino to simply make an HTTP request to the URL (via the Ethernet shield). When a request is made to the API, a new serverless instance is booted up by AWS, and the Lambda function runs. Lambda limits code packages that are directly uploaded to their service to 10MB in size when zipped. Our package is much larger, around 200MB when zipped because it includes the PyTorch library and the state_dict of the trained GAN. As such, we needed to store the package in S3 (Simple Storage Service)[2]. The Lambda function is linked to the code package in S3 and downloads it whenever the function is initiated by an API call.

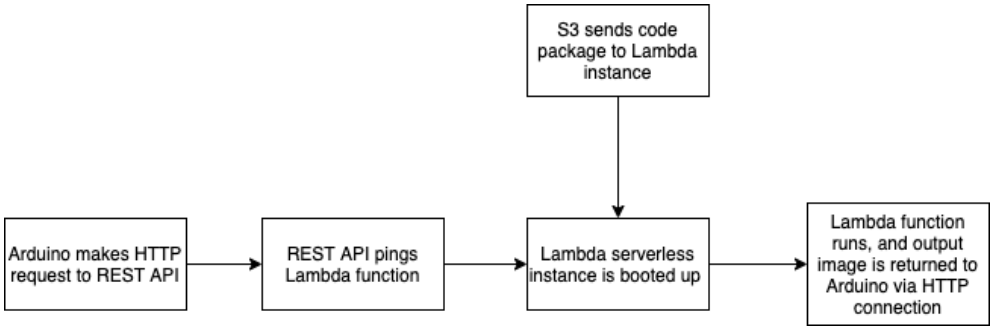


Figure 2.7: Interaction between AWS services

Theory

Stepper Motor

One might wonder why a stepper motor is used for our printer instead of a traditional DC motor. DC motors have a major limitation—there is no way to precisely control how many times a regular DC motor spins. Stepper motors are brushless DC motors that move in steps, as the name implies. The rotor is made from two discs shaped like little gears. One of these discs is a magnetic north pole, while the other is a magnetic south pole. The teeth of these two discs alternate (as shown in Figures 3.1-3.3). Then, multiple coils of wire that are organized in "phases" (groups) are oriented around the rotor. Although the number of wire coils differ depending on the type of stepper motor, the four coil stepper motors we used had coils oriented ninety degrees from each other. Opposite coils are paired together in a phase, and when the phase is energized, the electromagnetic coils pull teeth towards them. Then, the other phase is energized, turning the rotor again. In this way, the motor rotates, moving one step at a time [17] (Figure 3.1-3.3).

[2] S3 allows developers to upload large folders to the cloud.

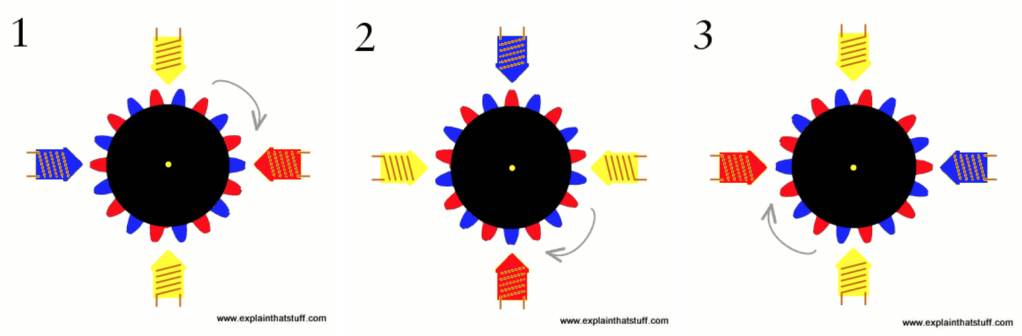


Figure 3.1-3.3: Diagram of the Inner Workings of a Stepper Motor
From left to right: how the Stepper Motor spins [18]

Because of its ability to move in microsteps, stepper motors are used to achieve precise positioning or a controlled speed. However, the limitations of stepper motors are that they are inefficient and have no feedback, meaning that they do not record position, unlike servo motors.

There are two types of stepper motors: unipolar and bipolar. In unipolar stepper motors, the phases are always energized the same way. One coil will always be positive, and the other will always be negative. The disadvantage with unipolar drivers is that there is less torque because only half the coils are energized at the same time. Bipolar drivers, on the other hand, reverse current flow through the coil. This means that the coils are energized with alternating polarity, so all the coils can work to turn the motor [19].

A stepper motor driver is used along with the stepper motor itself (See Circuit Diagram 2.5). A stepper motor driver is a driver circuit that sends current through phases in pulses to the motor. The stepper motor driver that is used for the printer is a microstepping stepper motor driver. The advantage of a microstepping driver is that it gives very fine motion resolutions, which is desired for the printer. A microstepping driver essentially increases and decreases current along a sine wave, so no pole is ever completely off [17].

Solenoid

The solenoid is used to lift and push the drawing tool on and off the drawing medium. The type of solenoid used is a linear electromechanical actuator solenoid. A solenoid converts electrical energy to mechanical energy where an electromagnetic coil repels a magnetic actuator into linear motion [20]. The magnetic field produced by the electromagnetic coil is directly proportional to the current and the density of turns. The circuit for the solenoid involves a MOSFET, a power supply, and the Arduino Mega. The MOSFET and the power supply are used to amplify the current from the Arduino digital output, triggering the solenoid without burning out the Arduino pins. The solenoid can then be controlled from the Arduino code. Refer to Figure 2.5 for the circuit.

Cartesian Grid

The concept of using a pair of perpendicular axes for printing was inspired by the Cartesian grid. The x-axis is commonly a horizontal line, while the y-axis is a vertical line. An axis is a reference line, and the two axes used in the Cartesian coordinate system intersect at the origin. The two axes also divide the plane into four different quadrants, where the first quadrant contains a set of numbers that are both positive.

For the printer, a Cartesian coordinate system is also used in a different way. A two dimensional array can act as a Cartesian system, where a point can be uniquely described by two numbers. Specific locations can be designated where the printer will make a mark on the paper, and others where the printer will not. However, one difference between two dimensional arrays and the coordinate system is that in the coordinate system, a positive set of coordinates is above and to the right relative to the origin (first quadrant). With two dimensional arrays, the first quadrant is below and to the right of the origin. This is because the set of numbers used in a two dimensional array represent the row number and column number. Row numbers increase downwards, while column numbers increase towards the right. When the printer was set up, this concept of the first quadrant being downwards and to the right was kept in mind. When the printer is about to begin drawing an image, it is placed at its own origin. If the drawing area of the printer is said to be a coordinate plane, then we would want the origin to be the highest point to the left.

Neural Networks

The artist uses a neural network architecture known as Generative Adversarial Network (GAN). Invented recently, GANs are used to generate images, text, or any other kind of data that is similar to the data it is trained on. For example, a GAN trained on images of dogs can generate new pictures of dogs that aren't real but look like they could be.

A standard neural network receives training data, which consists of inputs and their respective outputs. It then learns to generate the correct output given new input. Neural nets are constructed from "neurons", which are found in different layers that are interconnected by "synapses." Synapses are valued by their weights (the coefficients of the function produced by training). The layers are organized in this sequence: input layer, hidden layers, and output layer. At the start of the training process, the weights are all initialized to random values. The network then starts forward propagation—the input vector is dot multiplied by the matrix of weights attaching it to the first hidden layer (see Figure 3.4). An activation function, the sigmoid function in the case of classification[3], is applied to the outputs of this multiplication, resulting in values between 0 and 1. The same process is then applied with the output vector and the weights matrix of the next hidden layer, until the output layer is reached. The multiplication of weights and inputs is depicted in the equations in Figure 3.4.

[3] The sigmoid function is a type of activation that converts an input number into a value between 0 and 1. This is useful for classification because the outputs can be rounded to binary.

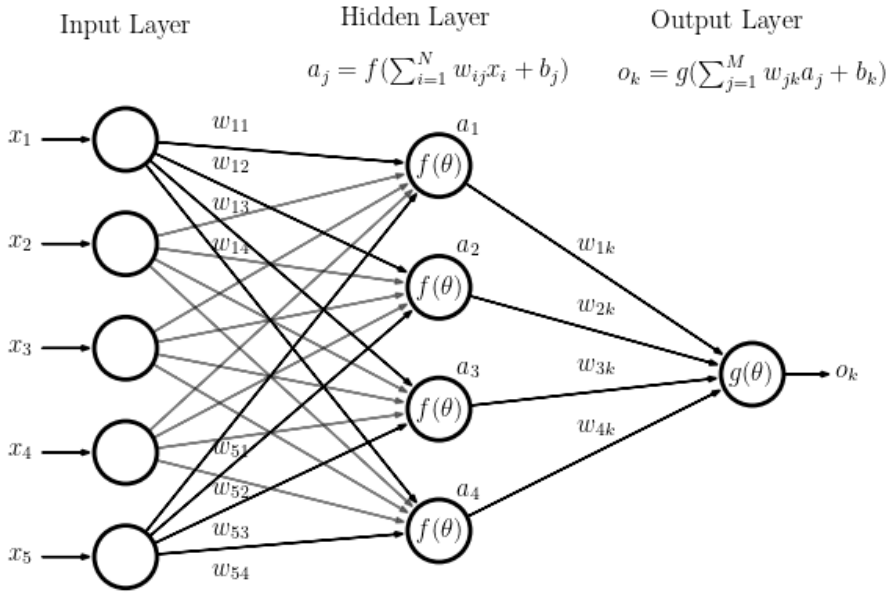


Figure 3.4: Example Neural Network, where equations represent dot multiplication of weights with inputs [21]

The network then evaluates the final output by calculating the error. If it predicted 0.3, but the actual output (from the training data) was 1, the error would be 0.7. A loss function is then applied to the error, and this loss is used to update the weights matrices of the network via a process called backpropagation. During backpropagation, each weight is adjusted by the calculated loss, and the magnitude of this adjustment can be amplified or reduced depending on the learning rate, a parameter that is determined before training is initiated. The final result is that the network is composed of a highly complex polynomial function which can be applied to any input vector and produce a desired output. The optimization algorithm is discussed in detail in Appendix C.

GAN Architecture

A GAN consists of two separate neural networks, a discriminator and a generator. The generator's job is to create new images that closely match the subject of the training data. The discriminator's job is to distinguish the fake images created by the generator from actual images. Both networks rely on each other to improve: the generator is optimized to fool the discriminator into classifying its fake images as real pictures, and the discriminator is optimized to not be fooled. The core parameter of a neural network is the loss function, which evaluates the network's performance so it can improve. While the discriminator attempts to minimize its loss in recognizing fake images, the generator's goal is to maximize the loss of the discriminator. Both networks are trained competitively in this way (see Figure 3.4) [22].

The type of GAN used for the robot artist is a Deep Convolutional GAN (DCGAN). This means that both neural networks are based on convolutional layers, which have been proven to be especially effective for dealing with image data [23]. Convolutional layers are useful because they allow us to map inputs into outputs of a different size, either adding or cutting information from the inputs (see Figure 3.5) [24]. In the discriminator, the convolutional layer is trained to

cut out useless information from the input images, resulting in an output that is particularly useful for analyzing the image's authenticity. Think of this process like filtering out anything that doesn't help the discriminator identify an image as fake or real. In the generator, the convolutional layers take an input (such as the initial random noise we feed the network) and map it to an output that more closely resembles real images in the training dataset.

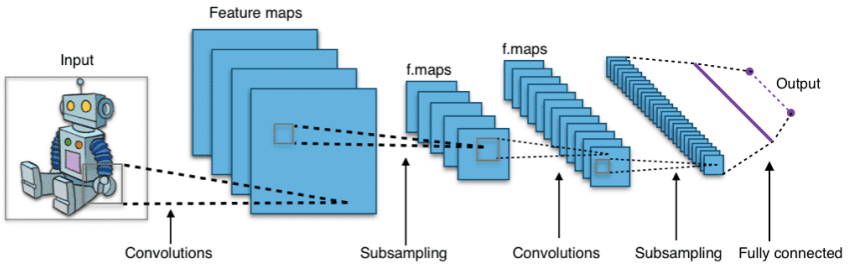


Figure 3.5: A convolutional network maps the image input matrix into an output that is more useful for the machine learning task, which is classification in this particular case [25].

For the purposes of this project, the architectures for both the generator and discriminator were created using a tutorial published on DCGANs [26]. Both networks are built with five convolutional layers, each one modifying the size of the inputs by applying filters (which are fine-tuned during training). In the generator, these layers expand the random noise vector input into grids of 4x4, 8x8, 16x16, 32x32, and 64x64 pixels, in that order. In this way, a randomly generated vector of size 100 is converted into a 64x64 pixel matrix. In the discriminator, the same process occurs, but starting with a 64x64 pixel grid and ending with a binary scalar output—0 or 1 (indicating whether the image is real or fake). The optimization algorithm used for both networks is known as Mini Batch Gradient Descent. Mini Batch works via the following two steps. The training dataset is split up into "batches," each containing 128 images. The losses and the mean of their gradients are then calculated for the entire batch, and the weights of the network are updated using the mean (see Appendix C).

GAN Training

The training process is as follows (see Figure 3.4). First, the discriminator is trained on a dataset of real images, essentially learning what a real image looks like. The generator then creates a dataset of fake images. At the beginning, the generator's fake images look like random noise because it has not yet learned how to make realistic art. These images are fed into the discriminator. The loss of the discriminator is used to update its weights; i.e. the discriminator is trained on these images, learning that they are fake. Then, the fake images are labeled as real. They are fed into the discriminator and the loss is calculated. However, this calculated loss is not used to update the discriminator weights because the images were labeled as real, when they are

actually fake. Instead, the purpose of this calculated loss is to update the generator's weights [26]. The reason we do this is to make the generator minimize the discriminator's loss when it is being fooled; i.e. make the generator maximize the true loss of the discriminator. This method accomplishes the "Inverted Loss" task in Figure 3.4. Finally, once the generator's weights have been updated with the discriminator's loss, the training process restarts, marking the end of an epoch. Eventually, as the generator's weights are tuned to maximizing the loss of the discriminator, its outputs begin to resemble real images.

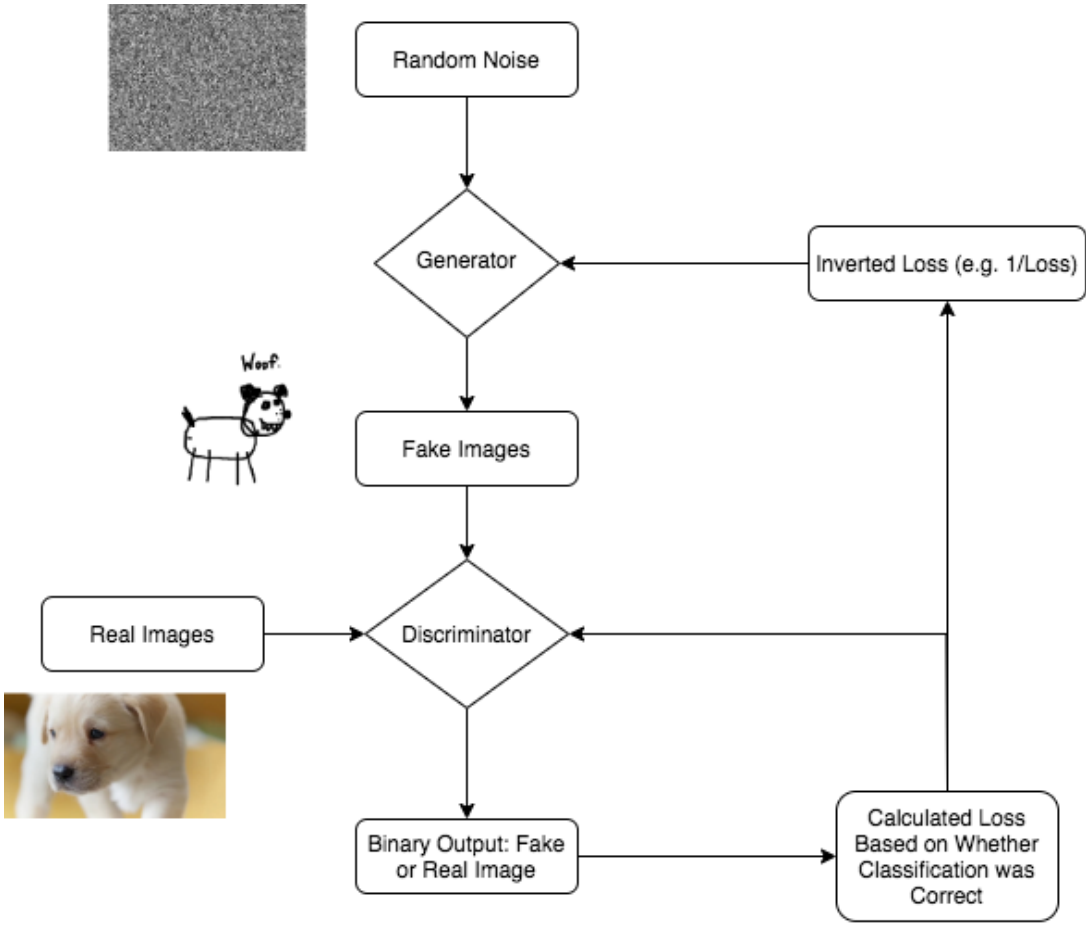


Figure 3.6: Training Process of GAN Discriminator and Generator

Results

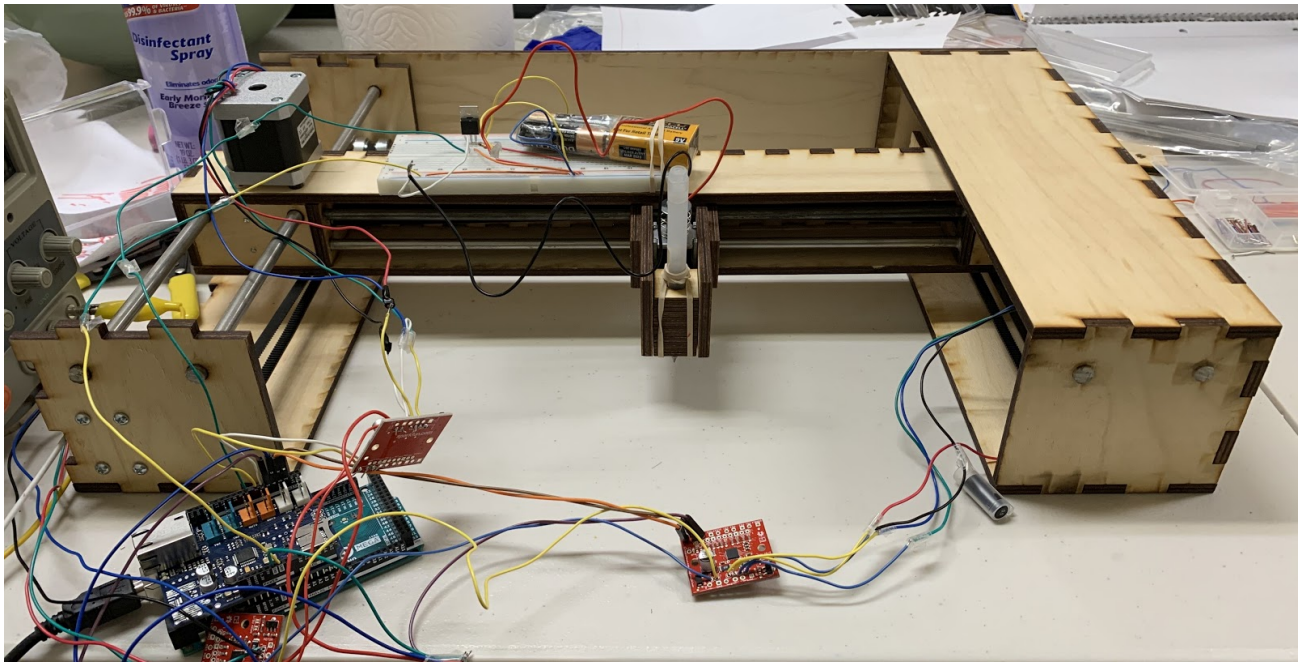


Figure 4.1: Robot Artist

The Robot Artist successfully generated and printed new artwork (see Figures 4.2, 4.3). Due to the limitations of the current situation with COVID-19, we were unable to successfully incorporate the API call into the Arduino code (see Conclusions), although we did manage to make the separate components work. As such, we were able to generate images and manually import them into the Arduino.

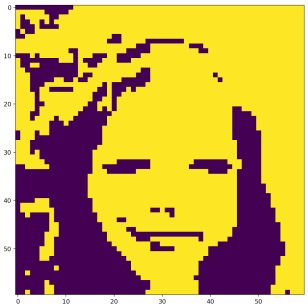
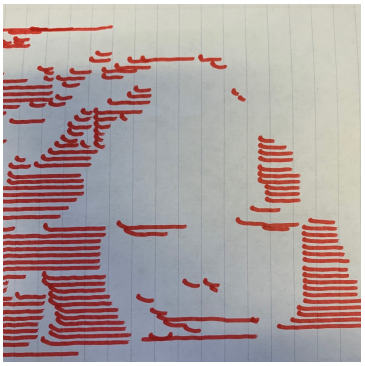
Color Image	Monochrome Image	Printed Image
		

Figure 4.2: First Generated Face and Final Printout

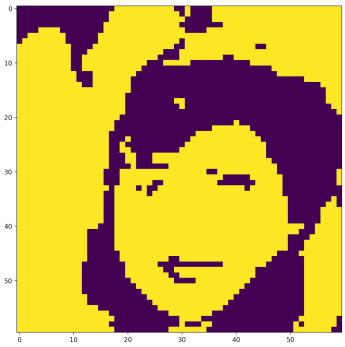
Monochrome Image	Printed Image
	

Figure 4.3: Second Generated Face and Final Printout

Time to Train	Processor	# Cores	Speed
4hr	Intel Core i9	8	2.4 GHz

Figure 4.4: GAN Training Specs

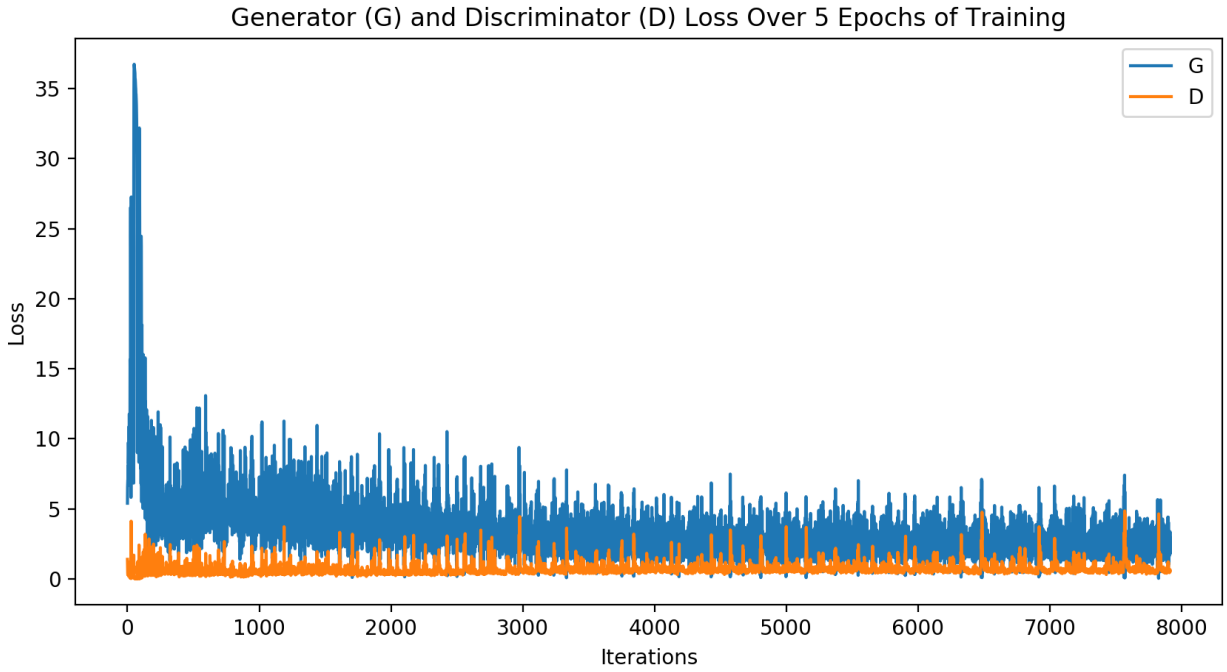


Figure 4.5: Generator and Discriminator Loss Over 4 Hours of Training

Delay (Microseconds)	Trial #1(s)	Trial #2 (s)	Trial #3 (s)	Average (s)	Average Speed (cm/s)
100	2.21	2.21	2.16	2.19	5.70
200	4.30	4.32	4.15	4.26	2.94
300	6.11	6.21	6.33	6.22	2.01
400	8.12	8.18	8.08	8.13	1.54
500	10.02	10.11	10.35	10.2	1.23

Figure 4.6: Time (s) that it takes for the printer to draw a 12.5 cm line at different delays

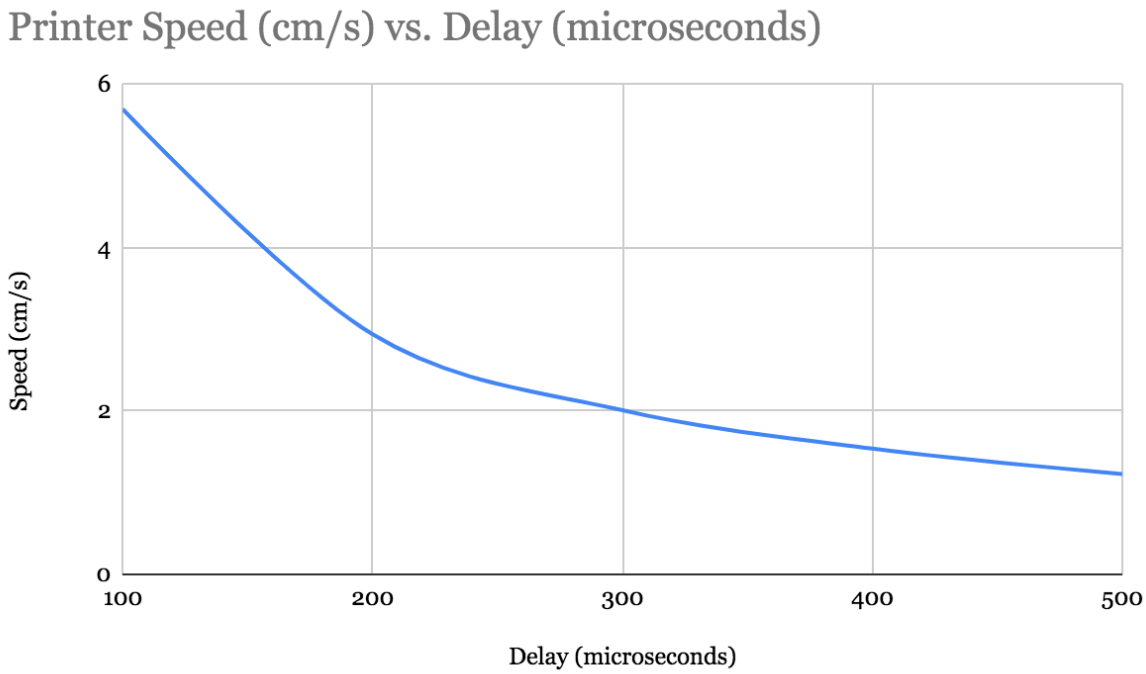


Figure 4.7: Speed vs. Delay

Line numbers	Trial #1 (mm)	Trial #2 (mm)	Trial #3 (mm)
1-2	5.14	4.95	5.93
2-3	6.27	6.26	6.22
3-4	6.53	6.24	6.22
4-5	6.16	6.43	6.32
5-6	6.46	6.59	6.34
6-7	6.23	6.03	6.34
7-8	6.27	5.72	6.59
8-9	6.16	6.24	6.28
9-10	6.17	6.41	6.59
Mean	6.15	6.10	6.31
Standard Deviation	0.40	0.50	0.20

Figure 4.8: Mean and Standard Deviation of distances between horizontal lines drawn (i = 1000)

Conclusions

We trained the two GAN neural networks, the generator and discriminator, over 5 epochs on a Macbook Pro (see Figure 4.4), meaning that the training dataset of images was iterated over 5 times. Due to the competitive nature of the networks, as programmed into their loss functions (see Machine Learning in Section 4), their losses decreased over each epoch until converging on relatively low values, as depicted in Figure 4.5. By the end of the training process, the generator network was capable of creating images that were realistic enough to fool the discriminator into classifying them as real. Qualitatively, we also observed that these images were extremely realistic (see Figure 4.2). The loss for both networks drops over 5 epochs, but after about 3 epochs the change in loss was negligible. As such, if we were to repeat the lengthy training process, we would reduce the number of epochs.

The creation of the API was not without its errors and debugging efforts. Of these, the primary issue was dealing with AWS Lambda's hard cap on the size of the deployment package containing the PyTorch library, the state_dict, and the code itself. While S3 allowed us to increase the limit from 10MB (zipped) to 50MB (zipped), our zipped package remained at around 200MB. We determined that the PyTorch library was the primary culprit, at around 600MB in unzipped size. Realizing that we only used the neural network sublibrary, we deleted the majority of its files and modified the package requirements to salvage the neural network library by itself. This reduced the total package size to under 50MB, and we were able to create the API. Due to the stay-at-home order, we were unable to physically work together over the

final phase of our project: integrating the API call with the Arduino code. While we succeeded in programming the Ethernet shield to make HTTP requests, we couldn't develop and debug the code to parse the response into a JSON object and extract the 2D grid into an Arduino array. See Next Steps for our plan to complete this.

Given that we were unable to properly integrate the API response, we decided to hard-code the 2D array that represents the generated image. The GAN was trained to produce 64x64 pixel images, but when we placed them in the code, the Arduino returned a memory error due to the size of the code file. To solve this, we scaled down the output images to 60x60 pixels (the maximum sized array we could hard-code). However, when we do fix the API call, we will revert back to the original 64x64 resolution.

From the mechanical scope, the printer functions well, but that isn't to say that it is perfect. While drawing the generated images, we noticed that the lines it drew were not equally spaced. After observing this precision error, we decided to test the spread of line spacing. The printer was programmed to draw ten straight lines horizontally, moving down a set amount of steps (i = 1000) after each line is drawn.[4] Then, using a caliper, the distance between the first line and the next was measured, the second line and the third line, and so on. The standard deviation of the distance between the two lines was calculated to represent the consistency/precision of the two stepper motors that work together to move the horizontal bar vertically. After executing this experiment with three trials, we found the average standard deviation to be 0.367 millimeters. This means that on average, the space between two lines drawn differs from the mean by 0.367 millimeters. Although this result isn't as low as we hoped it might be, it is still a very small spread, meaning that the stepper motors are very precise.

Aside from the stepper motors themselves, there are other areas where the printer could be improved. When we look at the images drawn by the printer in Figures 4.2 and 4.3, we see that each line drawn by the printer starts with a small "hook." The hook occurs due to the fact that the attachment that holds the writing utensil has some leeway on the rails, meaning that it is able to bend downwards and upwards slightly. Normally, because of the weight of the attachment, it sags downwards so the writing utensil is tilted slightly upwards (relative to the drawing direction on the paper). When the solenoid pushes down, however, the force applied by the writing utensil on the paper is enough to overcome the gravitational force acting on the whole attachment, straightening the writing utensil at an angle perpendicular to the paper. This causes the writing utensil to correct itself, tilting slightly downwards. This same concept also explains why the lines drawn by the printer drift upwards at the end of each line.

To complete a full drawing of a two dimensional array of size 60x60 pixels, the printer took around eight and a half minutes. However, the time it takes for the printer to complete a drawing depends on a number of factors. First of all, the more points that the printer has to mark, the more time it will take. Each time the solenoid presses down, a delay of two hundred milliseconds is processed to make sure that the printer does not start moving before the solenoid has completely lowered the writing utensil. Another factor is the speed at which the motors move. The speed of the motors is controlled by delay statements after writing HIGH and LOW consecutively in the Arduino code. The larger the delay, the slower the stepper motors turn. The fastest speed of the stepper motors is achievable by removing the delay statement altogether; however, the faster the stepper motor turns, the more current it draws. As we learned, the more

[4] While conducting this experiment, certain design errors were ignored (right vertical bar stepper motor not always pulling the conveyor belt, crooked horizontal bar). Errors are discussed in Next Steps.

current it draws, the faster the stepper motor driver heats up. For these reasons, the relationship between delay (microseconds) and speed (meters/second) was tested. In the experiment, the delays 100, 200, 300, 400, and 500 microseconds were tested (Figure 4.6). The results indicate that each 100 microsecond increase in delay results in roughly a two second increase in the time it takes to draw the 12.5 centimeter line. The speed, on the other hand, does not have linear change. This is expected, however, since a constant number divided by a variable with linear change should produce a rational equation (as seen in Figure 4.7).

Additionally, three different writing utensils were tested with the goal of drawing in a way that most closely resembled the generated image. The first writing utensil we tested was the classic pencil. However, the pencil brought up a multitude of problems. First of all, the lead would break if too much pressure was applied on the pencil by the solenoid. Next, the pencil would get more and more dull as the image was drawn, which means that lines drawn later would be wider and less distinct. An extremely sharp pencil would draw very thin lines, which would lead to large gaps between lines because of the resolution of the image. This last problem became a recurring issue, as any thin tip pencil or pen would create too much white space between lines. The obvious solution was to use a sharpie or other type of marker with a thicker drawing point. However, we were limited by the size of the hole used to fit the writing utensil, as this issue was not considered during our design process. In the end, the writing utensil we used for our final drawings was a thin Crayola marker, which drew lines thicker than that of a pencil or pen, especially when more force was applied. The white space problem remained, still evident in Figures 4.2 and 4.3. But, the marker minimizes this problem into a minor annoyance, since it covers up a majority of the white space that was not covered by pens or pencils.

Next Steps

Due to the stay-at-home order, there are a few steps that we would still like to implement when we have the chance to work together. The robot artist is now functional and able to print out the pictures generated by the machine learning algorithm. The next step to be taken is to determine the best way to parse JSON responses from the API request. We were able to make the Ethernet shield ping the API server, so once we accomplish this, the artist will be able to retrieve images from the server.

There are also a few areas of improvement concerning the design of the printer. First of all, as mentioned in the Design section, the stepper motor on the right vertical bar does not consistently pull the conveyor belt the amount it should. This error is due to the fact that the conveyor belt is not secured tightly enough over the stepper motor shaft and the bearing opposite to the shaft. Because the conveyor belt does not tightly hug against the spinning motor shaft, the conveyor belt often slips off the teeth of the motor shaft's bearing. This error results in the conveyor belt not turning when the stepper motor is. The temporary solution used when printing the images in Figures 4.2 and 4.3 was to manually push the horizontal bar down. To fix this issue, one simple solution would be to remove the bearing holder (rectangular piece with circle in the middle in Figure 2.3) and move it to the right so that the conveyor belt would be tighter.[5]

Another improvement that could be made is to fine-tune the movement of the stepper motors even more. The Big Easy Driver offers options for lower step resolutions that could be experimented with. The resolution currently being used for all three stepper motors is the full

[5] See Figure 2.3

step resolution; however, it could be adjusted to the one-sixteenth step resolution option. This would make each step even smaller, which would allow more precise movements. The pixel grid created by the machine learning algorithm could be larger and more detailed. Additionally, with the modified precision of the stepper motor, it is highly likely that we would be able to eliminate the extra white space between each line. Nevertheless, increasing step resolution leads to decreased torque, so we would have to find a balance between a high enough step resolution with a high enough torque.

Acknowledgements

We would like to thank Dr. Dann and Mr. Ward for helping us with various parts of the artist, from finding the best lubricant for the metal rails to understanding how the laser and 3D printers in the lab worked. In particular, we want to thank Dr. Dann for personally delivering parts to Justin's house over quarantine.

Bibliography

[1] https://www.cbsnews.com/news/banana-art-basel-performance-artist-eats-banana-today-taped-to-wall-that-had-sold-for-120000-2019-12-07/

[2] https://www.analyticsvidhya.com/blog/2019/04/top-5-interesting-applications-gans-deep-learning

[3] https://www.americanscientist.org/article/ai-is-blurring-the-definition-of-artist

[4] https://www.engadget.com/2019/02/12/aican-doesnt-need-human-help-to-paint-like-picasso/

[5] https://computerhistory.org/blog/harold-cohen-and-aaron-a-40-year-collaboration/

[6] https://www.projectmaths.ie/documents/T&L/IntroductionToTheCartesianPlane.pdf?strand

[7] https://wild.maths.org/ren%C3%A9-descartes-and-fly-ceiling

[8] https://en.wikipedia.org/wiki/Hellschreiber

[9] http://www.hffax.de/history/html/hellschreiber.html

[10] https://en.wikipedia.org/wiki/Dot_matrix_printing

[11] https://yourmileagemayvary.net/2020/01/30/why-do-they-still-use-dot-matrix-printers-at-airports

[12] https://datatechcomputer.com/printers/ibm-4224-dot-matrix-printer/

[13] https://www.creality3dofficial.com/products/creality-ender-3-pro-3d-printer

[14] http://mmlab.ie.cuhk.edu.hk/projects/CelebA.html

[15] https://pytorch.org/tutorials/beginner/saving_loading_models.html

[16] https://docs.aws.amazon.com/apigateway/latest/developerguide/http-api-vs-rest.html

[17] https://www.motioncontroltips.com/faq-what-are-stepper-drives-and-how-do-they-work/

[18] https://www.explainthatstuff.com/how-stepper-motors-work.html

[19] https://techexplorations.com/blog/arduino/blog-the-difference-between-unipolar-and-bipolar-stepper-motors/

[20] https://www.electronics-tutorials.ws/io/io_6.html

[21] https://www.astroml.org/_images/fig_neural_network_1.png

[22] https://papers.nips.cc/paper/5423-generative-adversarial-nets.pdf

[23] https://medium.com/@jonathan_hui/gan-dcgan-deep-convolutional-generative-adversarial-networks-df855c438f

[24] https://adeshpande3.github.io/A-Beginner%27s-Guide-To-Understanding-Convolutional-Neural-Networks/

[25] https://www.unite.ai/wp-content/uploads/2019/12/Typical_cnn-1.png

[26] https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html

[27] https://medium.com/@dmitrijtichonov/debunking-loss-functions-in-deep-learning-4b9abc4c8d4c

[28] https://towardsdatascience.com/nothing-but-numpy-understanding-creating-binary-classification-neural-networks-with-e746423c8d5c

[29] https://srdas.github.io/DLBook/GradientDescentTechniques.html

Appendix A

Part Description	What needed for	Cost	Where you'll buy it
Conveyor belt and bearings: bearing attaches to motor	attach to motors for x and y direction movement of pen	$14.99	Amazon
3 Stepper Motors	Spin conveyor belts	$17.95 (Already have 2)	Sparkfun Website
SN754410 Quadruple Half-H Driver	Stepper motor driver	$19.95	Sparkfun Website
10K Ohm Resistor	Solenoid circuitry	-	Borrowed from Dr. Dann
~12V Solenoid	Up/down movement of writing utensil	-	From Whitaker Lab
MOSFET	Solenoid circuitry	-	Borrowed from Dr. Dann
Two AA Batteries, one 9V Battery	Solenoid circuitry	-	Borrowed from Dr. Dann
Multipurpose 304/304L Stainless Steel Rod, ¼" Diameter, 2 Feet Long	Horizontal bar - rod	$4.34	McMaster-Carr
½" Diameter Stainless Steel Rod	Vertical bar - rod	-	From Whitaker Lab

Appendix B

In Figure 2.4, the leftmost and rightmost rectangular pieces with two holes in them were for keeping the horizontal bar stable on the rods that were put through these holes. There was no other way to secure those rectangular pieces except to drill a bracket onto the rectangular piece with the holes, and then drill that bracket onto the bottom of the horizontal bar. However, with drilling comes imperfect measurements and alignments. The problem that arose from drilling the rectangular pieces in was with alignment. The horizontal bar has four sets of two holes (eight holes total), four on the front side and four on the back side (front side shown in Figure 2.4, back ones are not shown). The back holes were needed to guarantee stability of the horizontal bar on the rods during up and down movement. Laser cutting these holes, however, took much more

effort and trial and error than we thought it would. Since the wooden rectangular pieces with the front holes were already drilled on, the location of the back holes needed to be exactly parallel to these holes, or else the horizontal bar would not be exactly perpendicular to the vertical bars. If this happened, the horizontal bar could be tilted upwards relative to the left vertical bar, which means that the image drawn will be tilted counterclockwise. This problem needed to be avoided at all cost, but making sure the back holes were perfectly aligned with the front holes turned out to be a huge difficulty. As previously mentioned the front holes are on a wooden rectangular piece that was secured onto the horizontal bar using a bracket, which means that human error resulted in the wooden rectangular piece not being perfectly flush to all edges of the horizontal bar. This also means that the holes on the wooden rectangular piece were at a location different from where we intended them to be. Due to this, the measurements we made would not be able to perfectly match the back holes to the front ones. To fix this problem, we were forced to use a strategy of trial and error, where we would print out the piece with the back holes, assemble the horizontal bar, and see whether or not the rods would be at an angle or not. If they were, then the next piece would be adjusted accordingly. After repeating this process over and over again (around seven attempts), a satisfactory layout was finally achieved, where the angling of the horizontal bar relative to the left vertical bar is minimal. That said, relative to the left vertical bar, the horizontal bar still is not perfectly perpendicular. The horizontal bar still is tilted slightly upwards, which explains why the images drawn by the printer are tilted counterclockwise. Additionally, the right vertical bar is also not perfectly perpendicular, and is also tilted slightly counterclockwise. There is no simple way to fix this problem now, as the whole printer is assembled and the parts cannot be removed and realigned anymore. The misalignment of the printer really is just a design flaw. By laser printing the wooden rectangle piece as a section of the horizontal bar (using Makerbox), a lot of the human error that comes with drilling could have been avoided, and the front and back holes would have been more aligned.

Appendix C: Gradient Descent Optimization

As described in the Theory section, neural networks use the backpropagation process to update their weights. The weight adjustment calculation is made using the gradient descent algorithm. The individual adjustments are the gradients which are calculated by finding the partial derivative of the loss function. Both the generator and discriminator in the GAN rely on the loss from binary classification of real vs fake images. This loss function, known as BCE (Binary Cross Entropy) loss is as follows.

$$BCE = -\frac{1}{N}\sum_{i=0}^{N} y_i \cdot log(\hat{y}_i) + (1 - y_i) \cdot log(1 - \hat{y}_i)$$

Figure 9.1: Equation for Binary Cross Entropy Loss [27]

BCE is commonly used in binary classification tasks, and it can be demonstrated why. Figure 9.1 shows an average of all of the individual losses, where *N* is the number of samples (images). *y* represents the true label of the sample, and *y (hat)* is the prediction by the neural network. If *y* is 0, but the prediction, *y (hat)*, is 0.99, the loss is calculated as

$$-(0 * log 0.99 + (1 - 0) * log(1 - 0.99)) = 2$$

*For the purposes of this example, we use 0.99 instead of 1 because we cannot take log of 0.

On the other hand, if the neural net correctly classifies a sample, such as if both the prediction and label are 1, we end up with

$$-(1 * log 0.99 + (1 - 1) * log(1 - 0.99)) = 0.004$$

In this case, the weights would be adjusted very little in comparison to the previous example. Once we calculate the loss, we plug it into the equation for gradient descent:

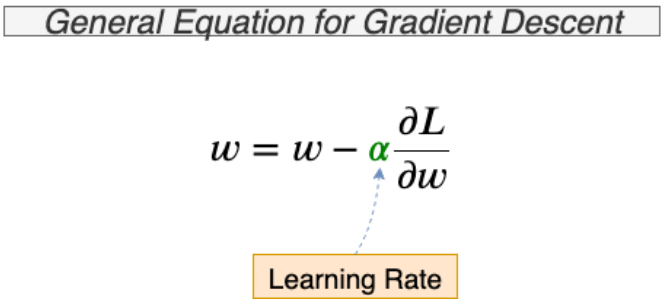


Figure 9.2: General Equation for Gradient Descent [28]

An individual weight *w* is adjusted by the learning rate *a* multiplied by the partial derivative of the BCE loss function with respect to *w*. This results in the loss function approaching its minimum, as visualized in Figure 9.3. Each weight of every synapse is adjusted using gradient descent as the backpropagation algorithm proceeds from the output layer through each hidden layer. This adjustment can be visualized.

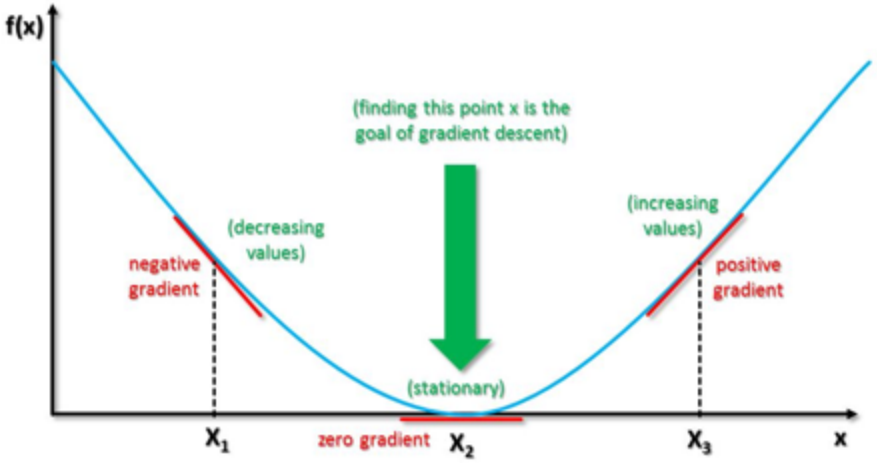


Figure 9.3: Steps of gradient descent until converging, or "descending," on the ideal weight values. f(x) is our loss function, BCE, and the x-axis represents the weight values [29].

Appendix D: GAN and Arduino Code

Training Code - This code implements the architecture and training described in the Theory section

gan.py

```
# Copied and Modified from PyTorch DCGAN Faces Tutorial
# Summary: This code trains the GAN and then saves the generator
network into a state_dict
# It then creates graphs and an animated visualization of
training (not depicted in paper)

from __future__ import print_function
#%matplotlib inline
import argparse
import os
import random
import torch
import torch.nn as nn
import torch.nn.parallel
import torch.backends.cudnn as cudnn
import torch.optim as optim
import torch.utils.data
import torchvision.datasets as dset
import torchvision.transforms as transforms
import torchvision.utils as vutils
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from IPython.display import HTML

# Set random seed for reproducibility
manualSeed = 999
#manualSeed = random.randint(1, 10000) # use if you want new
results
print("Random Seed: ", manualSeed)
random.seed(manualSeed)
torch.manual_seed(manualSeed)

# Root directory for dataset
dataroot = "/Users/dhruvm/Documents/GAN/dataset/CelebA/Img"

# Number of workers for dataloader
workers = 8
```

```
# Batch size during training
batch_size = 128

# Spatial size of training images. All images will be resized to
this
#   size using a transformer.
image_size = 64

# Number of channels in the training images. For color images
this is 3
nc = 3

# Size of z latent vector (i.e. size of generator input)
nz = 100

# Size of feature maps in generator
ngf = 64

# Size of feature maps in discriminator
ndf = 64

# Number of training epochs
num_epochs = 5

# Learning rate for optimizers
lr = 0.0002

# Beta1 hyperparam for Adam optimizers
beta1 = 0.5

# Number of GPUs available. Use 0 for CPU mode.
ngpu = 1

# We can use an image folder dataset the way we have it setup.
# Create the dataset
dataset = dset.ImageFolder(root=dataroot,
                           transform=transforms.Compose([
                               transforms.Resize(image_size),

transforms.CenterCrop(image_size),
                               transforms.ToTensor(),
                               transforms.Normalize((0.5, 0.5,
0.5), (0.5, 0.5, 0.5)),
                           ]))
# Create the dataloader
```

```
dataloader = torch.utils.data.DataLoader(dataset,
batch_size=batch_size,
                                         shuffle=True,
num_workers=workers)

# Decide which device we want to run on
device = torch.device("cuda:0" if (torch.cuda.is_available() and
ngpu > 0) else "cpu")

real_batch = next(iter(dataloader))

# custom weights initialization called on netG and netD
def weights_init(m):
    classname = m.__class__.__name__
    if classname.find('Conv') != -1:
        nn.init.normal_(m.weight.data, 0.0, 0.02)
    elif classname.find('BatchNorm') != -1:
        nn.init.normal_(m.weight.data, 1.0, 0.02)
        nn.init.constant_(m.bias.data, 0)

# Generator Code

class Generator(nn.Module):
    def __init__(self, ngpu):
        super(Generator, self).__init__()
        self.ngpu = ngpu
        self.main = nn.Sequential(
            # input is Z, going into a convolution
            nn.ConvTranspose2d( nz, ngf * 8, 4, 1, 0,
bias=False),
            nn.BatchNorm2d(ngf * 8),
            nn.ReLU(True),
            # state size. (ngf*8) x 4 x 4
            nn.ConvTranspose2d(ngf * 8, ngf * 4, 4, 2, 1,
bias=False),
            nn.BatchNorm2d(ngf * 4),
            nn.ReLU(True),
            # state size. (ngf*4) x 8 x 8
            nn.ConvTranspose2d( ngf * 4, ngf * 2, 4, 2, 1,
bias=False),
            nn.BatchNorm2d(ngf * 2),
            nn.ReLU(True),
            # state size. (ngf*2) x 16 x 16
            nn.ConvTranspose2d( ngf * 2, ngf, 4, 2, 1,
bias=False),
```

```
            nn.BatchNorm2d(ngf),
            nn.ReLU(True),
            # state size. (ngf) x 32 x 32
            nn.ConvTranspose2d( ngf, nc, 4, 2, 1, bias=False),
            nn.Tanh()
            # state size. (nc) x 64 x 64
        )

    def forward(self, input):
        return self.main(input)

# Create the generator
netG = Generator(ngpu).to(device)

# Handle multi-gpu if desired
if (device.type == 'cuda') and (ngpu > 1):
    netG = nn.DataParallel(netG, list(range(ngpu)))

# Apply the weights_init function to randomly initialize all
weights
#  to mean=0, stdev=0.2.
netG.apply(weights_init)

# Print the model
print(netG)

class Discriminator(nn.Module):
    def __init__(self, ngpu):
        super(Discriminator, self).__init__()
        self.ngpu = ngpu
        self.main = nn.Sequential(
            # input is (nc) x 64 x 64
            nn.Conv2d(nc, ndf, 4, 2, 1, bias=False),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (ndf) x 32 x 32
            nn.Conv2d(ndf, ndf * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 2),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (ndf*2) x 16 x 16
            nn.Conv2d(ndf * 2, ndf * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 4),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (ndf*4) x 8 x 8
            nn.Conv2d(ndf * 4, ndf * 8, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 8),
```

```
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (ndf*8) x 4 x 4
            nn.Conv2d(ndf * 8, 1, 4, 1, 0, bias=False),
            nn.Sigmoid()
        )

    def forward(self, input):
        return self.main(input)

# Create the Discriminator
netD = Discriminator(ngpu).to(device)

# Handle multi-gpu if desired
if (device.type == 'cuda') and (ngpu > 1):
    netD = nn.DataParallel(netD, list(range(ngpu)))

# Apply the weights_init function to randomly initialize all
weights
#  to mean=0, stdev=0.2.
netD.apply(weights_init)

# Print the model
print(netD)

# Initialize BCELoss function
criterion = nn.BCELoss()

# Create batch of latent vectors that we will use to visualize
#  the progression of the generator
fixed_noise = torch.randn(64, nz, 1, 1, device=device)

# Establish convention for real and fake labels during training
real_label = 1
fake_label = 0

# Setup Adam optimizers for both G and D
optimizerD = optim.Adam(netD.parameters(), lr=lr, betas=(beta1,
0.999))
optimizerG = optim.Adam(netG.parameters(), lr=lr, betas=(beta1,
0.999))

# Training Loop

# Lists to keep track of progress
img_list = []
```

```
G_losses = []
D_losses = []
iters = 0

print("Starting Training Loop...")
# For each epoch
for epoch in range(num_epochs):
    # For each batch in the dataloader
    for i, data in enumerate(dataloader, 0):

        ############################
        # (1) Update D network: maximize log(D(x)) + log(1 -
D(G(z)))
        ###########################
        ## Train with all-real batch
        netD.zero_grad()
        # Format batch
        real_cpu = data[0].to(device)
        b_size = real_cpu.size(0)
        label = torch.full((b_size,), real_label, device=device)
        # Forward pass real batch through D
        output = netD(real_cpu).view(-1)
        # Calculate loss on all-real batch
        errD_real = criterion(output, label)
        # Calculate gradients for D in backward pass
        errD_real.backward()
        D_x = output.mean().item()

        ## Train with all-fake batch
        # Generate batch of latent vectors
        noise = torch.randn(b_size, nz, 1, 1, device=device)
        # Generate fake image batch with G
        fake = netG(noise)
        label.fill_(fake_label)
        # Classify all fake batch with D
        output = netD(fake.detach()).view(-1)
        # Calculate D's loss on the all-fake batch
        errD_fake = criterion(output, label)
        # Calculate the gradients for this batch
        errD_fake.backward()
        D_G_z1 = output.mean().item()
        # Add the gradients from the all-real and all-fake
batches
        errD = errD_real + errD_fake
        # Update D
```

```
        optimizerD.step()

        ############################
        # (2) Update G network: maximize log(D(G(z)))
        ###########################
        netG.zero_grad()
        label.fill_(real_label)  # fake labels are real for
generator cost
        # Since we just updated D, perform another forward pass
of all-fake batch through D
        output = netD(fake).view(-1)
        # Calculate G's loss based on this output
        errG = criterion(output, label)
        # Calculate gradients for G
        errG.backward()
        D_G_z2 = output.mean().item()
        # Update G
        optimizerG.step()

        # Output training stats
        if i % 50 == 0:
            print('[%d/%d][%d/%d]\tLoss_D: %.4f\tLoss_G:
%.4f\tD(x): %.4f\tD(G(z)): %.4f / %.4f'
                  % (epoch, num_epochs, i, len(dataloader),
                     errD.item(), errG.item(), D_x, D_G_z1,
D_G_z2))

        # Save Losses for plotting later
        G_losses.append(errG.item())
        D_losses.append(errD.item())

        # Check how the generator is doing by saving G's output
on fixed_noise
        if (iters % 500 == 0) or ((epoch == num_epochs-1) and (i
== len(dataloader)-1)):
            with torch.no_grad():
                fake = netG(fixed_noise).detach().cpu()
            img_list.append(vutils.make_grid(fake, padding=2,
normalize=True))

        iters += 1

torch.save(netG.state_dict(), "generator")

plt.figure(figsize=(10,5))
```

```
plt.title("Generator (G) and Discriminator (D) Loss Over 5
Epochs of Training")
plt.plot(G_losses,label="G")
plt.plot(D_losses,label="D")
plt.xlabel("Iterations")
plt.ylabel("Loss")
plt.legend()
plt.show()

fig = plt.figure(figsize=(8,8))
plt.axis("off")
ims = [[plt.imshow(np.transpose(i,(1,2,0)), animated=True)] for
i in img_list]
ani = animation.ArtistAnimation(fig, ims, interval=1000,
repeat_delay=1000, blit=True)

HTML(ani.to_jshtml())

# Grab a batch of real images from the dataloader
real_batch = next(iter(dataloader))

# Plot the real images
plt.figure(figsize=(15,15))
plt.subplot(1,2,1)
plt.axis("off")
plt.title("Real Images")
plt.imshow(np.transpose(vutils.make_grid(real_batch[0].to(device)
[:64], padding=5, normalize=True).cpu(),(1,2,0)))

# Plot the fake images from the last epoch
plt.subplot(1,2,2)
plt.axis("off")
plt.title("Fake Images")
plt.imshow(np.transpose(img_list[-1],(1,2,0)))
plt.show()
```

Cloud Code - The handler method in this code is called when the API is pinged by the Arduino
gan_generator.py

```
# Summary: This code is uploaded to S3 and is used in Lambda.
# It imports the trained generator, creates an image, displays
# it, converts it to greyscale, and returns the image to the
API.
# For now, we manually input the 2D array to the Arduino, but
this
```

```
# code will be used when we get the API working with the Arduino
code.

#%matplotlib inline
import argparse
import os

# random library needed to generate random noise vector
import random

# PyTorch library
from torch import stack
from torch import cat
from torch import load
from torch import randn
import torch.nn as nn

# other useful libraries
import numpy as np
import math
import matplotlib.pyplot as plt
from numpy import savetxt
from numpy import asarray

# size of batches during Gradient Descent
batch_size = 128
image_size = 64

# number of color channels
nc = 3

# size of random noise vector inputted to generator
nz = 100

# size of images produced by generator
ngf = 64

# Number of GPUs available. Use 0 for CPU mode.
ngpu = 1

# Generator Network
class Generator(nn.Module):
    def __init__(self, ngpu):
        super(Generator, self).__init__()
        self.ngpu = ngpu
```

```
        self.main = nn.Sequential(
            # input is Z, going into a convolution
            nn.ConvTranspose2d( nz, ngf * 8, 4, 1, 0,
bias=False),
            nn.BatchNorm2d(ngf * 8),
            nn.ReLU(True),
            # state size. (ngf*8) x 4 x 4
            nn.ConvTranspose2d(ngf * 8, ngf * 4, 4, 2, 1,
bias=False),
            nn.BatchNorm2d(ngf * 4),
            nn.ReLU(True),
            # state size. (ngf*4) x 8 x 8
            nn.ConvTranspose2d( ngf * 4, ngf * 2, 4, 2, 1,
bias=False),
            nn.BatchNorm2d(ngf * 2),
            nn.ReLU(True),
            # state size. (ngf*2) x 16 x 16
            nn.ConvTranspose2d( ngf * 2, ngf, 4, 2, 1,
bias=False),
            nn.BatchNorm2d(ngf),
            nn.ReLU(True),
            # state size. (ngf) x 32 x 32
            nn.ConvTranspose2d( ngf, nc, 4, 2, 1, bias=False),
            nn.Tanh()
            # state size. (nc) x 64 x 64
        )

    def forward(self, input):
        return self.main(input)

# function taken from PyTorch library for converting image into
2D array
def make_grid(tensor, nrow=8, padding=2,
              normalize=False, range=None, scale_each=False,
pad_value=0):
    """Make a grid of images.

    Args:
        tensor (Tensor or list): 4D mini-batch Tensor of shape
(B x C x H x W)
            or a list of images all of the same size.
        nrow (int, optional): Number of images displayed in each
row of the grid.
            The Final grid size is (B / nrow, nrow). Default is
8.
```

```
        padding (int, optional): amount of padding. Default is
2.
        normalize (bool, optional): If True, shift the image to
the range (0, 1),
            by subtracting the minimum and dividing by the
maximum pixel value.
        range (tuple, optional): tuple (min, max) where min and
max are numbers,
            then these numbers are used to normalize the image.
By default, min and max
            are computed from the tensor.
        scale_each (bool, optional): If True, scale each image
in the batch of
            images separately rather than the (min, max) over
all images.
        pad_value (float, optional): Value for the padded
pixels.

    Example:
        See this notebook `here
<https://gist.github.com/anonymous/bf16430f7750c023141c562f3e9f2a
91>`_

    """

    # if list of tensors, convert to a 4D mini-batch Tensor
    if isinstance(tensor, list):
        tensor = stack(tensor, dim=0)

    if tensor.dim() == 2:  # single image H x W
        tensor = tensor.unsqueeze(0)
    if tensor.dim() == 3:  # single image
        if tensor.size(0) == 1:  # if single-channel, convert to
3-channel
            tensor = cat((tensor, tensor, tensor), 0)
        tensor = tensor.unsqueeze(0)

    if tensor.dim() == 4 and tensor.size(1) == 1:  #
single-channel images
        tensor = cat((tensor, tensor, tensor), 1)

    if normalize is True:
        tensor = tensor.clone()  # avoid modifying tensor
in-place
        if range is not None:
```

```
            assert isinstance(range, tuple), \
                "range has to be a tuple (min, max) if
specified. min and max are numbers"

        def norm_ip(img, min, max):
            img.clamp_(min=min, max=max)
            img.add_(-min).div_(max - min + 1e-5)

        def norm_range(t, range):
            if range is not None:
                norm_ip(t, range[0], range[1])
            else:
                norm_ip(t, float(t.min()), float(t.max()))

        if scale_each is True:
            for t in tensor:  # loop over mini-batch dimension
                norm_range(t, range)
        else:
            norm_range(tensor, range)

    if tensor.size(0) == 1:
        return tensor.squeeze()

    # make the mini-batch of images into a grid
    nmaps = tensor.size(0)
    xmaps = min(nrow, nmaps)
    ymaps = int(math.ceil(float(nmaps) / xmaps))
    height, width = int(tensor.size(2) + padding),
int(tensor.size(3) + padding)
    grid = tensor.new_full((3, height * ymaps + padding, width *
xmaps + padding), pad_value)
    k = 0
    for y in irange(ymaps):
        for x in irange(xmaps):
            if k >= nmaps:
                break
            grid.narrow(1, y * height + padding, height -
padding)\
                .narrow(2, x * width + padding, width -
padding)\
                .copy_(tensor[k])
            k = k + 1
    return grid
```

```
# handler function is run by the API when the Arduino makes a
request
# event and context are not necessary for this project but are
required by Lambda
def handler(event, context):

    # create generator network for importing the pretrained
state_dict
    netG = Generator(ngpu)

    # load in the trained weights (weights were found in gan.py)
    netG.load_state_dict(load("generator"))

    # random noise input that will be converted to an image
    noise = randn(1, nz, 1, 1)

    # forward propagate with generator to get new image
    fake = netG(noise).detach().cpu()

    # convert image to 2D array using PyTorch function
    img = make_grid(fake, padding=2, normalize=True)

    # resize for displaying purposes
    img = np.transpose(img,(1,2,0))

    # display produced image
    plt.figure(figsize=(15,15))
    plt.imshow(img)
    plt.show()

    # convert numpy array to Python array
    img = img.tolist()

    # scale down image due to memory issues with Arduino
    img = img[:60]
    for i in range(len(img)):
        img[i] = img[i][:60]

    # convert image to greyscale and turn it into a string for
returning to Arduino
    string = "{"
    for i in range(len(img)):
        string += "{"
        for j in range(len(img[i])):
```

```
            # equations/process for color to greyscale
(https://stackoverflow.com/questions/17615963/standard-rgb-to-gra
yscale-conversion)
            clinear = 0.2126 * img[i][j][0] + 0.7152 *
img[i][j][1] + 0.0722 * img[i][j][2]

            csrgb = 12.92 * clinear
            if clinear > 0.0031308:
                csrgb = 1.055 * (clinear ** (1/2.4)) - 0.055
            img[i][j] = round(csrgb)
            if j == len(img[i])-1:
                string += str(round(csrgb))
            else:
                string += str(round(csrgb)) + ","
        if i == len(img) - 1:
            string += "}"
        else:
            string += "},"
    string += "}"

    # save array into text file just in case we need to manually
upload to Arduino
    data = asarray(img)
    savetxt('data.csv', data, delimiter=',')

    response = {
        "statusCode": 200,
        "headers": {
            'Content-Type': 'application/json',
            "Access-Control-Allow-Headers":
'Content-Type,X-Amz-Date,Authorization,X-Api-Key,X-Amz-Security-T
oken',
            "Access-Control-Allow-Origin": '*',
            "Access-Control-Allow-Methods": 'GET,OPTIONS'
        },
        "body": string,
        "isBase64Encoded": False
    }
    return string
```

Arduino Mega Code - Draws out image 2D array manually entered

```
/**
 * Summary: Stepper code reads image array and draws out image
 * using a series of for loops. Given that we were
 * unable to integrate the API,the image array is manually
 * added directly from the generator.
 */

// digital pin assignments for all 3 stepper motors and solenoid

#define stp1 2
#define dir1 3

#define stp2 4
#define dir2 5

#define stp3 6
#define dir3 7

#define solenoid 8

// whether the solenoid is currently drawing
boolean solenoidTriggered = false;

// image size in centimeters
float image_size = 10;

// number of pixels in image
float num_pixels = 60;

// number of steps required to cover one pixel
// 2000 was decided based on the size of the paper
float pixel_iterations = image_size/num_pixels * 2000;

// delay to allow solenoid to finish triggering or retracting
before moving on
int solenoidDelay = 200;

// delay between each step in moving vertically down the paper
int backwardsDelay = 30;

// imaged 2D array
int grid [60][60] = *** manually imported array ommitted to save
space **;
```

```
// changes direction of a stepper motor
void flipDirection(int dir, int val) {
  digitalWrite(dir, val);
}

void setup() {
  // put your setup code here, to run once:
  Serial.begin(9600);
  pinMode(stp1, OUTPUT);
  pinMode(dir1, OUTPUT);

  pinMode(stp2, OUTPUT);
  pinMode(dir2, OUTPUT);

  pinMode(stp3, OUTPUT);
  pinMode(dir3, OUTPUT);

  pinMode(solenoid, OUTPUT);

  // delay to give us time to switch on power supply
  delay(6000);

  digitalWrite(stp1, LOW);
  digitalWrite(stp2, LOW);

  // number of steps to complete a full row
  int iterations = (int) pixel_iterations*num_pixels;

  Serial.println("starting");

  // for loop for vertical axis
  for(int i=0; i<(int) num_pixels; i++) {
    // for loop for horizontal axis
    for(int j=0; j<iterations; j++) {
      // step top bar stepper motor (horizontally)
      flipDirection(dir3, LOW);
      digitalWrite(stp3, HIGH);
      digitalWrite(stp3, LOW);

      // check if pixel is 0 or 1
      if(grid[i][(int) (((float) j)/pixel_iterations)] == 0) {
        // if solenoid isn't already triggered, trigger it
        if(!solenoidTriggered) {
          digitalWrite(solenoid, HIGH);
```

```
          delay(solenoidDelay);
          solenoidTriggered = true;
        }
      } else {
        // if solenoid is already triggered, retract it
        if(solenoidTriggered) {
          digitalWrite(solenoid, LOW);
          delay(solenoidDelay);
          solenoidTriggered = false;
        }
      }
    }

    digitalWrite(solenoid, LOW);

    // shift horizontal stepper motor so that solenoid returns
to
    // left side of axis
    for(int j=0; j<iterations; j++) {
      flipDirection(dir3, HIGH);
      digitalWrite(stp3, HIGH);
      digitalWrite(stp3, LOW);
      delayMicroseconds(backwardsDelay);
    }

    // shift vertical stepper motor pair so that solenoid
proceeds
    // to the next row
    for(int j=0; j<(int) pixel_iterations; j++) {
      flipDirection(dir1, HIGH);
      flipDirection(dir2, HIGH);
      digitalWrite(stp1, HIGH);
      digitalWrite(stp2, HIGH);
      digitalWrite(stp1, LOW);
      digitalWrite(stp2, LOW);
      delayMicroseconds(1500);
    }
  }

// unused
void loop() {}
```