University of Bergen
Department of informatics

Exploring Ways of Creating an AI Drawing Assistant

Author: Christian Mehl Wergeland
Supervisor: Troels Arnfred Bojesen

UNIVERSITETET I BERGEN
Det matematisk-naturvitenskapelige fakultet

Abstract

Making a digital drawing assistant is a complicated problem for the simple reason that we are trying to make a computer program understand and create something that are as abstract as a drawing. Thus, there have been many different approaches to what a drawing assistant is supposed to do and what part of the drawing process it is designed to assist with.

In this thesis, we will explore ways to improve the drawing process, both in terms of speed and quality of the outcome. This will be done with the use of neural networks that will try to learn what the drawer intends to draw from what they has previously drawn, as well as their assessment of that.

The final drawing assistant created in thesis managed to reliably predict the drawer's intention and can be used to create simple drawings and illustrations.

Acknowledgements

I would first like to thank my supervisor Troels Arnfred Bojesen who helped me extensively throughout the thesis, not only with giving me directions and advice on the thesis, but also by creating a good social environment. I would also like to thank my parents and brother, who always was there to support and encouragement me throughout my study period and through the process of writing this thesis. Thank you.

Christian Mehl Wergeland
08 June, 2022

Contents

List of Figures

1 Introduction

1.1 Motivation

A teacher or lecturer might have discovered that using illustrations, figures and maybe even a mascot character has greatly improved the learning and understanding of the students. But creating new drawings for every class might fast become more time consuming than just writing lecture slides in plain text. To easen the burden, one could try to use some of the tools that exist in some programs, like the shapes in Microsoft Pain. But these can quickly become limiting and are quite often just as slow, or slower, to use than just drawing by hand.

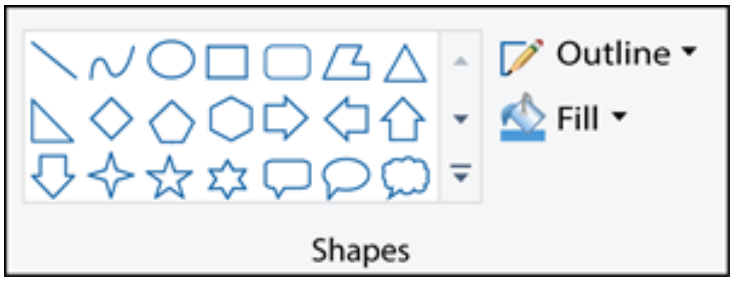


Figure 1: The shapes tools in Microsoft Paint.

But what if we could use machine learning as a help and make it easier to draw and keep the style of ones drawings? What if machine learning could be used as "support wheels" so one can focus on the drawing and not on having to redraw every time something does not turn out the way it was imagined?

Earlier machine learning assisted drawing tools usually fall in to one of two different approaches: using machine learning to continue a drawing, and using machine learning to predict what we are drawing. An example of the first category is sketch-rnn [1] which uses a neural network trained on millions of drawings collected from the Quick, Draw! [2] data set to finish a drawing of a pre-selected object.

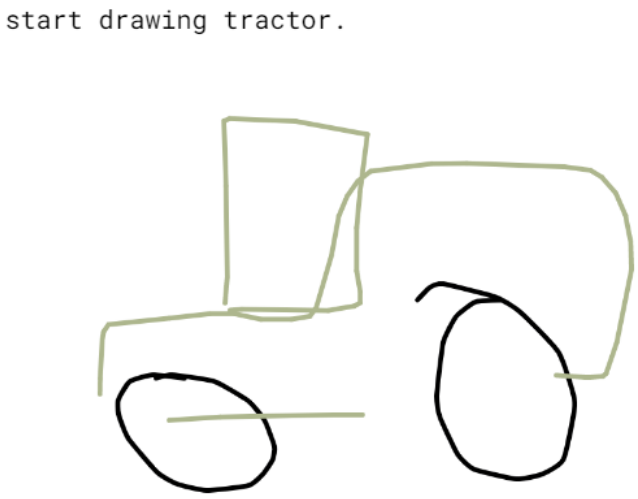


Figure 2: Sketch-rnn suggesting a completion of the tractor (green) and from the 2 wheels we drew (black).

In figure 2 we can see how sketch-rnn tries to finish a drawing of a tractor based on the two wheels. Every couple of second the demo tries a different continuation of the tractor. The problem with this approach is that we need to select from a list of already learned objects before starting to draw, and the machine learning algorithm does not do anything with what we have already drawn.

An example of the second approach is AutoDraw [3], where we can start to draw, let's say a house, and then AutoDraw will start to give predictions, from which we can select a suggested figure among some candidates. An example of this is shown in figure 3. This is closer to what we would like to achieve in this thesis, but most of the shapes are to specific, and too finished, therefore lacking the personal style of the drawer. What we end up with is mostly the same as with the first approach, where we are limited to a set of pre-defined objects.

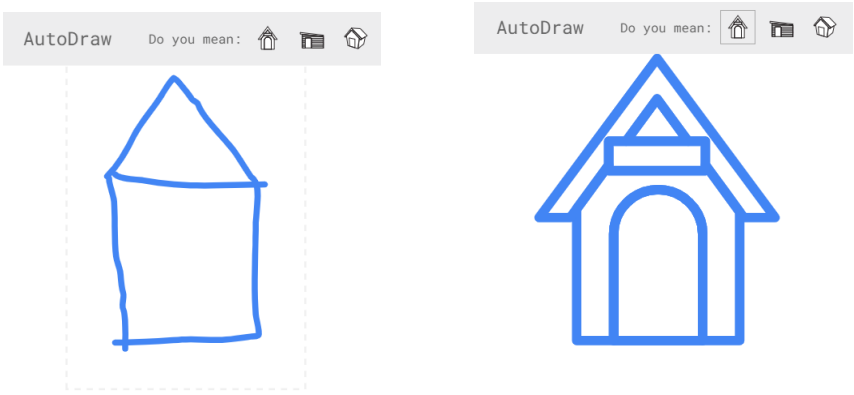


Figure 3: AutoDraw before (left) and after (right) selecting the object closest resembling or representing the one we drew.

1.2 Objective

What we would like to achieve in this thesis is to have an artificial intelligence (AI) drawing assistant that is more of a helping hand, where the AI drawing assistant should make only smaller adjustments as the user draws, and for the most part stay out of the way. An idea of the desired outcome is illustrated in figure 4, where the red line is what is drawn at each step and the AI drawing assistant adjust it to fit better to what is previously drawn and corrects for small errors, noise and unevenness. Note that it does not mean that every line drawn will be strongly influenced by the previous one, so the AI drawing

assistant has to learn to separate drawn objects from each other.

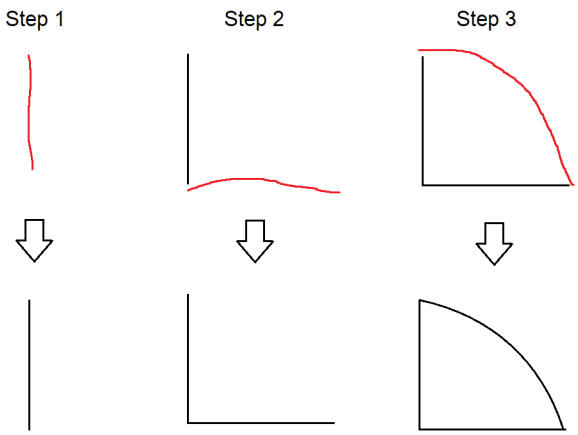


Figure 4: One idea of how the drawing assistant should work at each time step, with the down arrow being the drawing assistant transforming what is drawn.

1.3 Meta-methodology

Even thought the goal of this thesis is clear, to create an usable AI drawing assistant, the path to get there is not, therefore we use a methodology where we use iterative exploration of the problem to gradually get closer to our goal. As we continue to develop our approach to how to create the drawing assistant, we use what we learned in the previous method, to create a better AI.

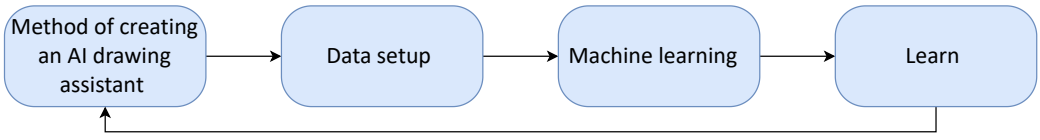


Figure 5: The methodology steps and recursion used in this thesis.

In the first step, we come up with a method for how we can get closer to archive the goal of creating a drawing assistant. Then we figure out what data we need for that method and how we can transform the data into a format that machine learning algorithms can utilize. We now create a machine learning model that is best suited to achieve the goal of the method we came up with, and we try to teach the drawing assistant how to complete the task we wanted the method we came up with to achieve. Finally, we go through what we learned from the approach, what did work and what did not work, and we use what we learn to come up with a new method.

2 Theory

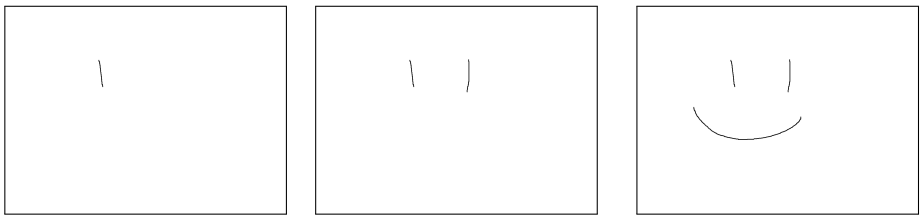
In this chapter, we establish notations and theory for the fields digital drawing and machine learning. (An overview of different machine learning approaches that are relevant to this thesis, will be presented.)

2.1 Drawing theory

In this section we will go through and establish some definitions on different aspects of digital drawings in the context of this thesis.

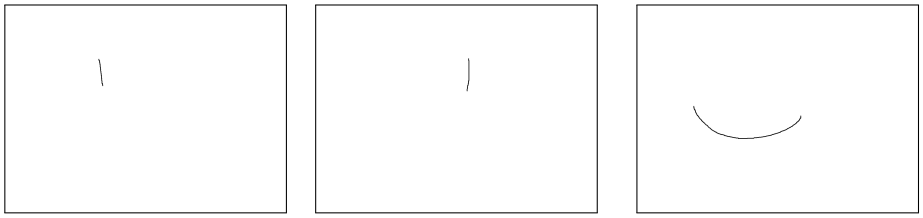
2.1.1 Drawing definitions

As there is no clear definition on what is art [4], it is also hard to define what making a drawing better means. A drawing assistant might come up with something that one person thinks is good, but someone else might think the same output is bad. However since the task of the drawing assistant is to improve the drawing in the subjective opinion of the one drawing, that is what we will focus on. We define D^* as the hypothetical drawing the artist would like to draw. We also define D_t to be a drawing after t number of lines have been drawn, as exemplified in figure 6 and d_t to be the line drawn at step t, as shown in figure 7. Note that d_0 is the blank page before any lines has been drawn.



(a) D_1 (b) D_2 (c) D_3

Figure 6: Drawing progression from left to right.



(a) d_1 (b) d_2 (c) d_3

Figure 7: The line of step $t = 1, 2, 3$ of the lines drawn in figure 6.

2.1.2 Images as data

The digital canvas that we draw on is represented as 3 2-dimensional arrays, one for each of the 3 color channels red, green and blue, RGB.

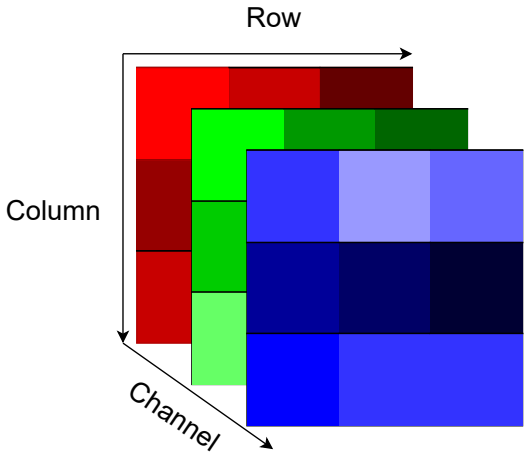
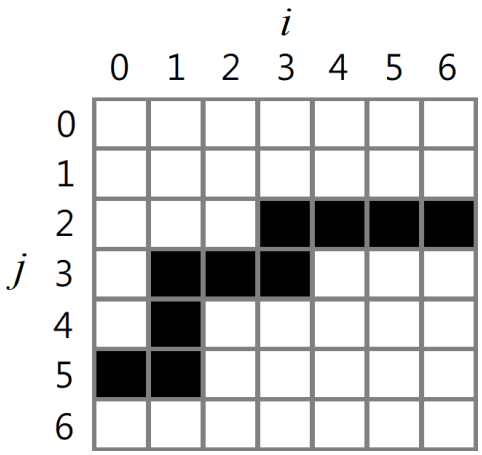


Figure 8: Representation of an RGB digital image.

This is more detail than is necessary to achieve the goals of this thesis, as the drawing assistant we create is not going to depend on colors, only what is drawn at each step. What we do is that we reduce the number of channels to one by taking the sum of the three colors in each pixel and set all pixels that has value greater than 0 to white and all else to black, to create a black and white representation of what is drawn.

Another approach is that instead of using the image, we only use the list of pixel locations that is accessed when a line is drawn. This approach lets us break down a drawn line to a time-series of coordinates, as shown in the list in figure 9. Form this we can easily look in the list and see where the line starts and where it stops, and what sequence the line was drawn in.



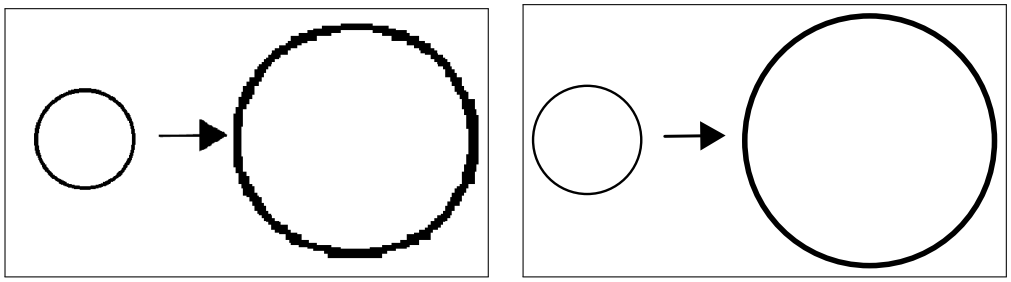
Entry number	i	j
0	0	5
1	1	5
2	1	4
3	1	3
4	2	3
5	3	3
6	3	2
7	4	2
8	5	2
9	6	2

Figure 9: A pixelated line and its list of accessed pixels.

2.1.3 Vector graphics

Vector graphics is a form of computer graphics where the image is created from geometric shapes such as lines, curves and polygons. A benefit of using vector graphics over a 2-dimensional pixel bitmap, is that since vector graphics is defined by mathematical equations, it is possible to scale an object arbitrarily, without loss of sharpness or fidelity. A good example of vector graphics is text on computers; it is possible to display computer text at any size, one letter per page if one wish, and every letter will still look sharp and clear. The same cannot be said for the images in this thesis that are not in vector form; they will end up blurry and the pixels clearly visible as seen with the circle figure 10.

Another benefit is that we can define an object with just a few parameters, for example a circle can be defined only by the radius of the circle and its location in space and (relative) line thickness.



(a) Bitmap (b) Vector

Figure 10: Sub-figures (a) and (b) show the difference between bitmap and vector graphics when scaling the figures.

2.2 Machine learning

Machine learning is a field in the broader artificial intelligence field where algorithms learn patterns and structures from data. Depending on the data we are working with and what the goal of the learning is, we traditionally divide machine learning in to three broad categories:

- Supervised learning
- Unsupervised learning [1]
- Reinforcement learning

In this thesis we will mainly interested in supervised learning and reinforcement learning, since one always has a notion of what the correct answer should be, or if the output of machine learning algorithms is good or bad.

[1] Unsupervised learning is used to find patterns and correlations in unlabeled data. This is useful when we want to find a way to group data that does not have a clear grouping.

Even though these are the three main categories, the border between them can become blurry, for example, a reinforcement learning problem can sometimes be solved by methods that are in the supervised learning category.

2.3 Supervised learning

Supervised learning is a machine learning task where algorithms learn to associate some input with some output. This is done by changing the parameters of an algorithm so that a set of examples inputs x have the corresponding outputs y. Supervised learning can be separate in to two types of problem:

- Classification
- Regression

Classification is when we use learning algorithms to assign specific discrete classes to data. An example of a Classification task is to train a machine learning algorithm to recognize and categorize the numbers in the MNIST dataset [5] of handwritten digits of a number from 0 to 9. The task is to map an image x to the corresponding digit label y.

Regression is a type of machine learning task where the goal is to estimate the relationship between feature variables, and continuous output variables. The result is a model that lets us predict the continues result of an unknown feature variables. An example of this is traffic modeling, where we use regression to create a model of how many cars are passing every hour, using recorded traffic data, and use the model to predict the traffic ahead of time. Here x could be the traffic at 10, while y is the traffic at 11.

2.4 Reinforcement learning

Reinforcement learning is the process of learning what actions to do in an environment, and to over time accumulate as much reward as possible. In contrast to supervised learning, the learner is not told what the correct action

to take is, but must discover through trial and error which action yields the most reward. In some situations, the actions of the learner do not only affect the immediate reward, but also the rewards downstream from that action. A good example of this is chess, where every move that is taken might lowers or increase the chance of winning, but since the eventual goal of the learner is to win the game, it does not matter if we loose the queen if that eventually lead to victory.

Reinforcement learning separates from supervised learning in the way that supervised learning is learning by example, where the correct answer for a situation is known. Supervised learning is often impractical in situation where the correct answer is difficult to obtain. For example, one can train the path-finding (an algorithm to create a map of where it is safe/possible to travel) systems of a Mars rover to give it a good model of how to navigate in a Mars like situations, but this training will be limited to what we know about Mars and what we can simulate on Earth. In this situation one could use reinforcement learning to continuously learn and improve how to navigate when on Mars, letting the rover adapt to its environment.

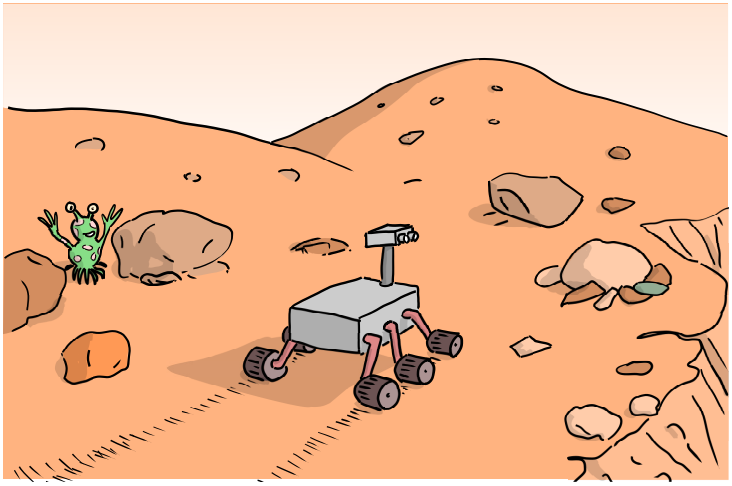


Figure 11: Rover on Mars. *Artist: Troels Arnfred Bojesen*

One of the biggest challenges exclusive to reinforcement learning is the trade-off between exploration, trying new actions, and exploitation, using what we already know to get the highest expected return. The dilemma is that one cannot follow one of them exclusively as this will lead to no learning being done. Let us go back to the rover example mentioned above: how comfortable are we with the rower exploring, when we know that it might lead to failure of the entire mission. But at the same time, if the rower only exploit what is know, we might end up failing if the rover ends up in a situation where it has no experience from.

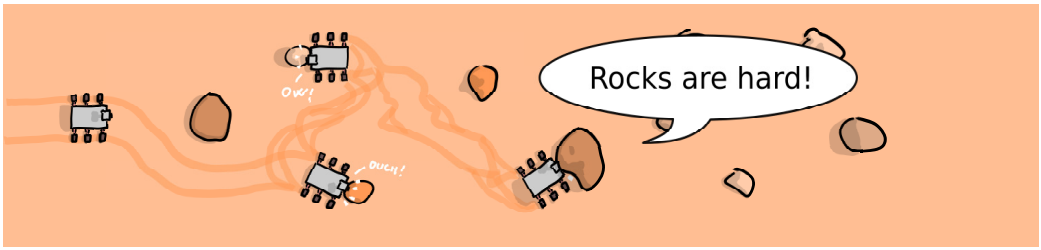


Figure 12: Rover continuously crashing in to rocks since it has not been trained to recognize Mars rocks. *Artist: Troels Arnfred Bojesen*

2.4.1 Reinforcement learning basics

In reinforcement learning we have the *agent*, the one doing the *actions*, and the *environment* with which the agents interacts with (despite uncertainty in how the actions of the agent might impact the future of the environment). The agent does not necessarily have to be one of the more obvious examples like a Mars rover, it can for example be smaller parts like the heating system that control the temperature of the rover components. For the agent to figure out what action to do, it needs to know what *state* it is in. The state is built up of features in the observations of the environment. For example, is there an obstacle in the path where I am driving?

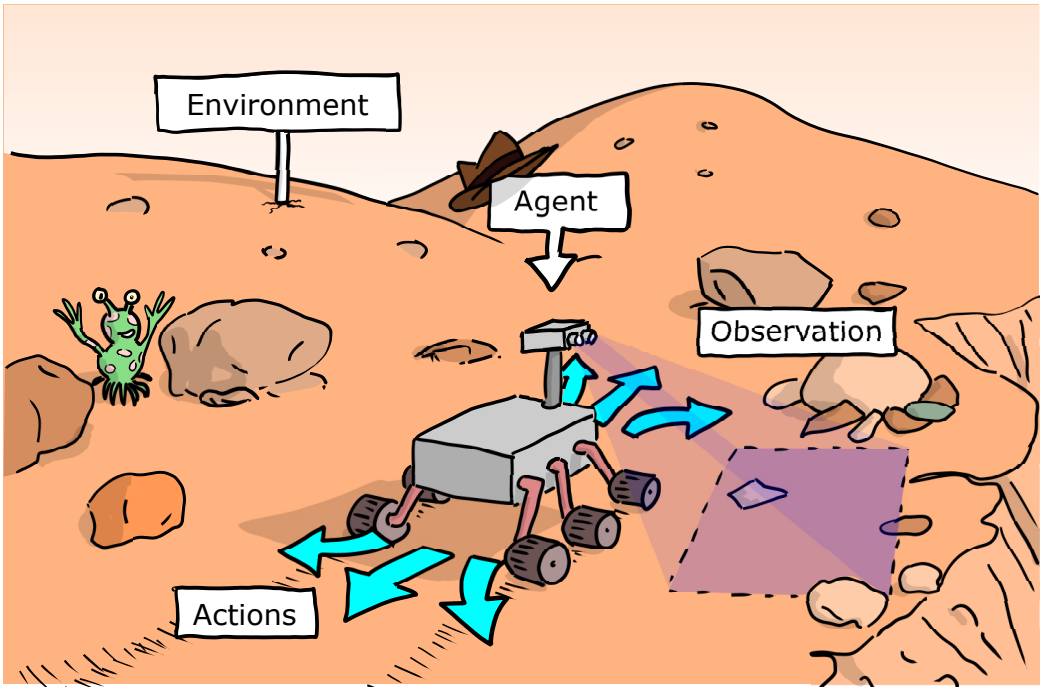


Figure 13: An agent observing the environment to take an action.
Artist: Troels Arnfred Bojesen

For an agent to know what to do in its current state it needs a *policy* on how to act in that state. At first glance, a policy might look like Supervised learning in the way a policy maps from state to action, and in supervised learning we associate some input with some output, but a policy can have many ways it selects the next action, some as simple as just choosing the best action A that the agent knows about, like in the greedy policy (1),

$$A = \underset{a}{\operatorname{argmax}}\,(a) \tag{1}$$

and others more complicated that are using search algorithms to look further ahead than just one action. We might not get the same action for the same state depending on if the policy is stochastic or not.

To get the agent to learn, it needs to know what actions are good or bad,

and to do that we *reward* each action with a corresponding reward based on how good the outcome of an action is. Just like how humans learn what is pleasing and what hurts in their environment, the agent learns what gives a high reward and what gives a low reward in their environment. And like humans, the agent wants to maximize the expected return, where return, denoted G_t, is the sum of reward after time step t.

$$G_t = \sum_{t=0}^{\infty} r(s_{t+1}, a_{t+1}) \tag{2}$$

We let a_t be an action at time step t, the state in t be s_t, and the reward from taking an action in a state be $r(s_t, a_t)$. What we want is to choose the action that results in the largest sum of rewards.

To do that, the agent uses the *reward signal* that is received from the environment after taking an action to change the policy to increase the reward it gets in the future. Thus, the reward signal defines the goal of the agent.

While a reward signal only indicates what is good or bad for the current action, a *value function* v indicates what is good or bad in the long run. A value function indicates the value of a state, that is how much reward the agent can expect to get in the future from being in the state. The value of a state when following the policy π then becomes:

$$v_\pi(s) = \mathbb{E}_\pi[G_t|s] \tag{3}$$

Where $\mathbb{E}_\pi[\cdot]$ denotes the expected value of a state following the policy π. A state can have a high value in the value function, despite having a low immediate reward signal in that state, when states following it leads to a high return.

Furthermore, we have *action value function* Q which instead of being the value of a state, is the value of the actions in a state, and how much return the agent expect to end up with when taking that action. Similarly to value

function, we define the value of taking an action a in state s when following the policy π, denoted $Q_\pi(s, a)$ as the expected return when in s taking action a and following a policy π:

$$Q_\pi(s, a) = \mathbb{E}_\pi[G_t|s, a] \tag{4}$$

Reinforcement learning tasks that only have actions, without states are called a *Multi-armed Bandits* problem. The actions in this type of problems have some unknown property to them, such as the wining chance of a slot machines, the agent have to find through exploration which action maximizes the expected return. In this thesis, the reinforcement learning problem we have is an *contextual bandits*. This is a type of bandit problem where we have a state that changes the outcome of an action. An example of this is if there is an slot machine that lights up in different colors, where the color determine the wining probability, we would have to take this in to account when selecting an action.

2.5 Deep learning

In this section we will go through deep learning, the definitions and math related to it and why it is relevant for this thesis.

Deep learning is a part of the broader machine learning field and revolves around the use of artificial neural networks as function approximators. The term "deep" comes form the fact that deep learning uses networks with multiple, stacked layers.

2.5.1 Neural network

Neural networks is a class of models that is originally inspired by neuroscience and the way a brain works in biology [6]. Similarly, to neurons in the brain, a neural network is built up by connected artificial neurons, illustrated in figure 14, that can transmit a signal to other neurons they are connected to.

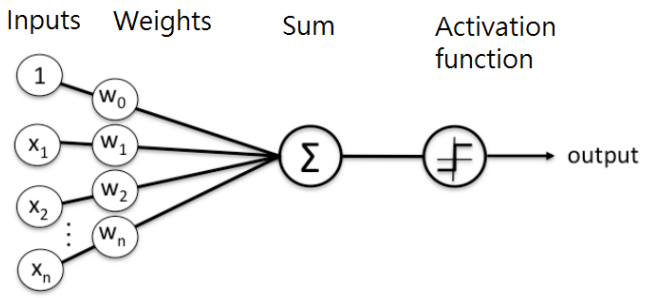


Figure 14: A schematic illustration of an artificial neuron.

An artificial neuron can be represented as the two subsequent operations, an inner product between the input signals and the weights,

$$\sum_{i=0}^{n} w_i x_i = z \tag{5}$$

followed by an activation function:

$$g(z) = y \tag{6}$$

An activation function can take many forms, both linear and non-linear, the simplest one being the identity function, shown in figure 15, where the output is the same as the input.

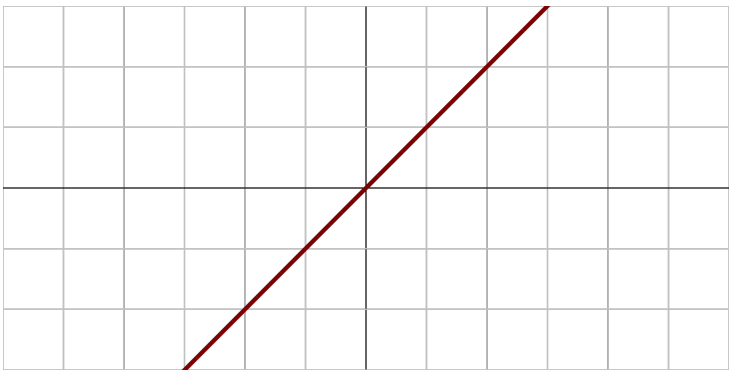


Figure 15: The graph of the Identity function.

The identity function is rarely used in neural networks as it limits the network to represent only linear transformations, and combining multiple linear transformation results in linear transformation. The problem is that we want the neural network to be able to approximate any function, not just linear ones.

There are two main activation function we will use, the first one being Rectified Linear Units (ReLU), which take the element wise maximum between 0 and z, with the graph of the activation shown in figure 16 and equation (7)

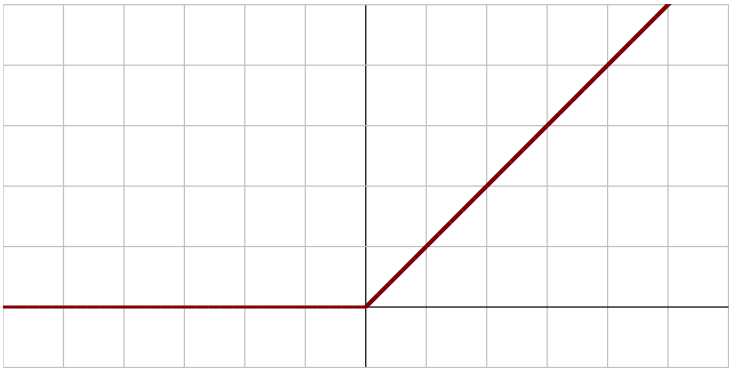


Figure 16: The graph of the ReLU function

$$g(z) = \max\{0, z\} \tag{7}$$

The second main activation function we will use is SoftMax (8). SoftMax is different from ReLU and the identity function, as it is not a function for a single input z, but vector of K values.

$$\sigma(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \quad \text{for } i = 1, 2, \ldots, K \tag{8}$$

The output vector of the SoftMax function consist of values in the interval (0,1) and the sum of all the values in the vector is 1. This results in the SoftMax output having the properties of a probability distribution. (To get

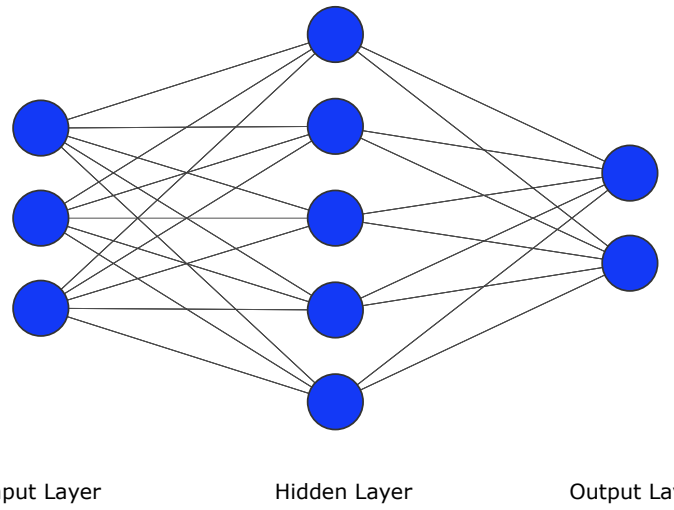


Figure 17: Illustration of a simple neuron network. The blue balls being the neurons in the network, and the connections goes from the neurons on the left to the right, and is how each neuron in the previous layer influence the next layer.

the input vector that SoftMax needs we need the output of multiple artificial neurons in parallel to build up a neural network layer.)

A simple neural network consists of an input layer, a hidden layer, and an output layer, as illustrated in figure 17

The input layer is the first layer of the network, and this is where we place the data we want to pass on to the rest of the network. The hidden layers are used to perform transformations on the input, to achieve the desired output. By using hidden layers the network can learn features in the data, for example, complicated shapes in images processing. The output layer is the last layer of the network, and is responsible for producing the output of the network.

Using a multi-layered neural network, we can create an universal function approximation [7]. This feature of neural networks, allows them to solve many tasks.

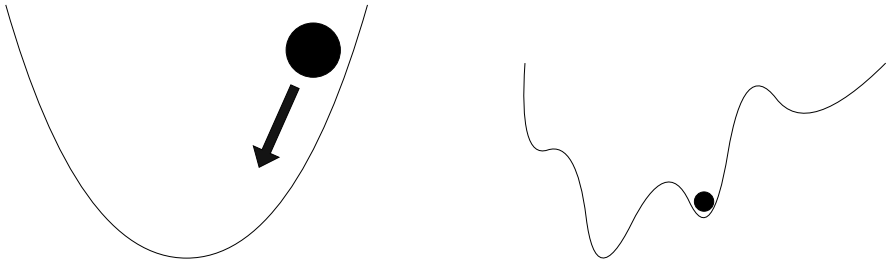
In practice, neural networks are implemented using on matrix multiplications. As an example, the network in figure 17, with identity activation function, looks like:

$$\begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix} \begin{bmatrix} w_{11}^1 & w_{12}^1 & w_{13}^1 & w_{14}^1 & w_{15}^1 \\ w_{21}^1 & w_{22}^1 & w_{23}^1 & w_{24}^1 & w_{25}^1 \\ w_{31}^1 & w_{32}^1 & w_{33}^1 & w_{34}^1 & w_{35}^1 \end{bmatrix} \begin{bmatrix} w_{11}^2 & w_{12}^2 \\ w_{21}^2 & w_{22}^2 \\ w_{31}^2 & w_{32}^2 \\ w_{41}^2 & w_{42}^2 \\ w_{51}^2 & w_{52}^2 \end{bmatrix} = \begin{bmatrix} y_1 & y_2 \end{bmatrix} \tag{9}$$

When not using identity activation, we pass the result of each matrix multiplications, starting from the left, through an activation function before continuing the multiplication.

To train the network, we need a way to adjust the weights in these matrices. This is done with an optimization algorithm that tells the weights how to adjust to minimize a loss function (other names: cost function, error function). The loss function $L(\theta, y, \hat{y})$ is how we represent the difference between the estimated $\hat{y} = f(\theta, x)$, where f is the neural network, with θ representing the trainable parameters of the neural network, such as weights, and the desired value y.

The optimization algorithms we are using are based on gradient decent. Gradient decent uses the gradient of the loss function with respect to the parameters of the network. The network starts at some point defined by the initial parameters of the network and uses the gradient to attempt to change the parameters such that the loss is reduced. We do this in an attempt to reach the global minimum, illustrated in figure 18a. In a more realistic situation, the loss function is typically not convex and more often the parameters end up in a local minimum (18b) and are trapped there. In cases where we have many parameters, we might not reach a minimum at all.



(a) Global minimum (b) Local minimum

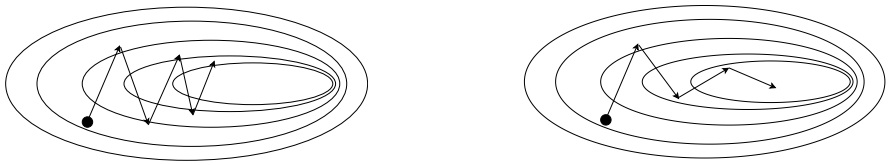
Figure 18: Gradient decent moving the black ball, representing the parameters, towards a global minimum (a), and the local minimum(b) the ball most likely will end up in.

The standard gradient descent formula is given by:

$$\theta \leftarrow \theta - \eta \nabla_\theta L(\theta, y, \hat{y}) \tag{10}$$

Gradient decent works by calculating the gradient of the loss function with regards to the parameters θ. We then update the parameters in the opposite direction of the gradient. How far we step is determined by the learning rate η.

We will use a version of stochastic gradient decent named Adam [8], as it has overall a good performance [9] due to its adaptive learning rate, in which the step length gets longer if learning is too slow, or get smaller if we are starting to jump over the minimum, and momentum, which acts like we have added weight to a ball that is traveling down hill, smoothing out the learning trajectory of the parameters.



(a) Without (b) With

Figure 19: Here we see the difference between gradient decent with (b) and without (a) momentum, and how (b) require fewer steps to get closer to the minimum.

2.5.2 Convolutional neural networks

Convolutional neural network is a class of neural networks that is most used in the field of computer vision [10]. Similarly to how neural networks are modeled after the neurons in the brain, convolutional networks are inspired by the visual cortex that is found in animals [11]. The convolutional network used in this thesis are built up by 3 modules, the convolutional layers, the pooling layers and fully connected layers (also called dense layers), which are like the neural networks explained in the previous section. In figure 20 , we can see an example of a convolutional neural network.

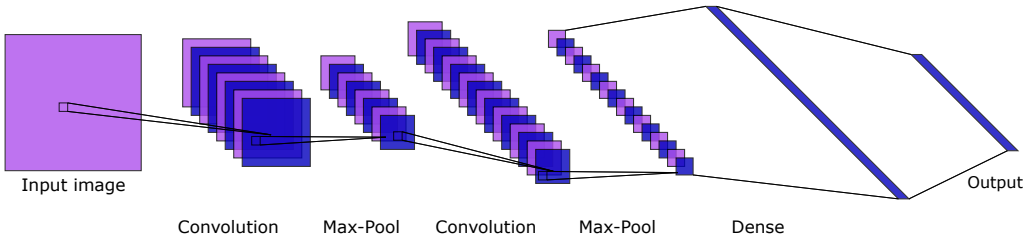


Figure 20: Example of a convolutional neural network, with a 2-dimensional image as input, then couple of convolutional and Max-Pool layers, ending in two fully connected layers, with the last being the output layer.

In this thesis, the input layer of the convolutonal network will be a 2-dimensional image. After the input layer, we have a filter, which is an 2-dimensional array of weights, same as a neural network, but instead of having a separate weight for each part of the image, we move the filter across the image and apply the dot product between the part of the image the filter is on and the weights in the filter. This process is called convolution.

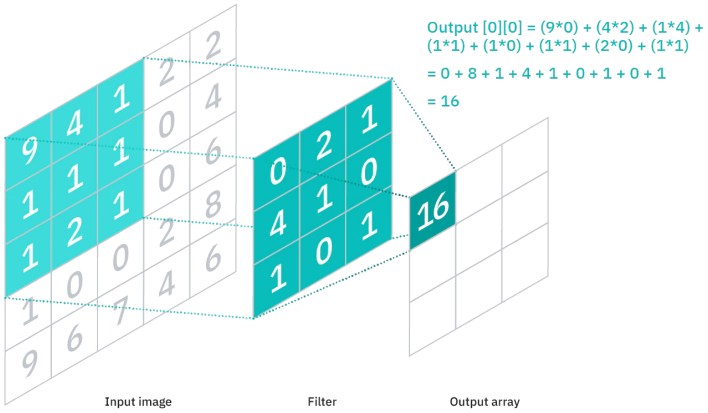


Figure 21: The first step of a convolution *Source: IBM [12]*

In figure 21 we can see convolution with a filter of size 3 by 3 applied to an image. How far the filter moves at each step is called stride. In figure 21 we have a stride of 1, this movement is illustrated in figure 22. If we want to get the same size image out as we started with, we can use padding. Padding puts zeros around the image so that we can move the filter to the edges of the image.

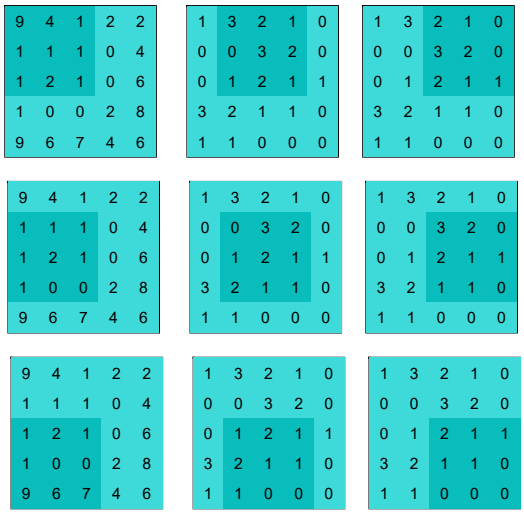


Figure 22: Shadow of the filter moving over the input image, starting in the top left of the image.

Pooling layers are used to reduce the dimensionality of their input. Similarly, to convolutional layers, it applies a filter over the input, but instead of having weights, it applies an aggregation function to the values it passes over. In this thesis we will use max pooling, in which the maximum of the area the filter is applied to is returned as the output, this is illustrated in figure 23. We do this because we want to extract the most important features in the image, for example what is drawn while ignoring everything else. Additionally max pooling helps reduce the impact of distortions in the image.

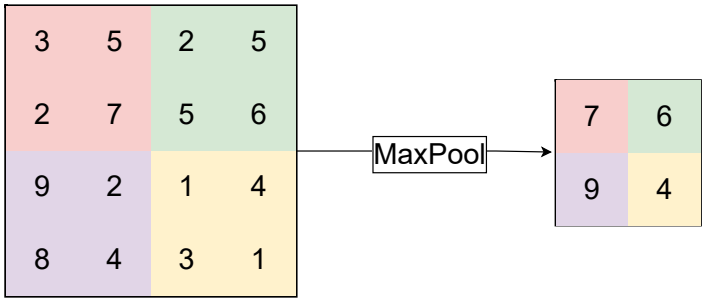


Figure 23: MaxPool transforming a 4 by 4 input down to 2 by 2 by taking the maximum value in each color patch.

2.6 Change of basis

In vector space we have a set of vectors that form the basis of the space. The standard basis in two-dimensional Cartesian space has the basis vectors $I = [1, 0]$ and $J = [0, 1]$, which are the unit vectors of the i and j dimension. These are shown in figure 24

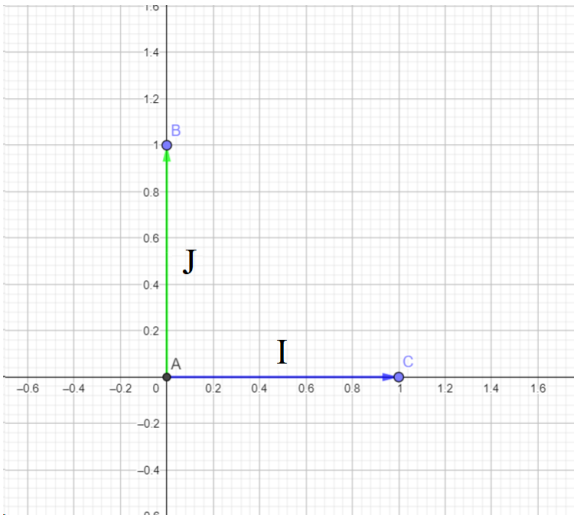


Figure 24: The basis vectors of Cartesian space.

Any coordinate in the 2D space is given by how many of each of these vectors you need to add to get to that coordinate. For example, to get to the coordinates [3, 4], you need 3 V vectors and 4 U vectors.

Change of basis is a technique that lets us change how a vector or a point is represented in one basis over to another basis. This allows us to transform the space by rotating and scaling everything the same way around the origin.

This is best explained by going through an example. Let's take the point [3, 4] and define a new space where it is in [1, 1] instead. We can do this by defining the new first dimension vector as $L = [3, 0]$ and the second dimension vector as $M = [0, 4]$. We now see that to get to [3, 4] we need one L vector and one M vector. What we want is a "translation tool" that allows us to go between the standard basis and the new basis. To get from the new basis to the standard basis we simply multiply the coordinate represented in the new basis, with the matrix containing the two basis vectors of the new basis vectors:

$$\begin{bmatrix} 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 3 & 0 \\ 0 & 4 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} \tag{11}$$

The general formula for change of basis is given as:

$$x_{\text{old}} = Ax_{\text{new}} \tag{12}$$

A is called the change-of-basis matrix, and is the "translation tool" that let us transform between the two basis. If we want to go in the other direction, to get from [3, 4] to [1, 1], we simply move A over to the other side of the equation using the rules of linear algebra:

$$A^{-1}x_{\text{old}} = x_{\text{new}} \tag{13}$$

$$\begin{bmatrix} 3 & 0 \\ 0 & 4 \end{bmatrix}^{-1} \begin{bmatrix} 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \tag{14}$$

$$\begin{bmatrix} 1/3 & 0 \\ 0 & 1/4 \end{bmatrix} \begin{bmatrix} 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \tag{15}$$

This way we have an easy way to transform expressions of coordinates relative to one basis, into another expression of coordinates relative to another basis.

3 Experimental studies

In this section we will explore the different ideas and ways to we can create an artificial drawing assistant.

Python is the coding language used to explore these ideas. Notable python libraries used are:

- Tkinter [13], a user interface library that we use to draw on, and a general purposes user interface.
- PyTorch [14], a deep learning library. PyTorch is combined with CUDA [15] from NVIDIA, that lets us use GPU acceleration
- NumPy [16] , a numerical computation library

3.1 Regression method

While we have an idea for how we want our drawing assistant to work, how we are going to create it is not. Let us begin whit a simplest question: is it possible to teach an AI drawing assistant to interpret a digital drawing? In this section we will look in to this question.

Let's explore the use of neural networks, and see if they can be used in the context of a pixel-based drawing application. A good way to do this is to test if a neural network can reasonably predict the continuation of a line that is drawn. This will let us see if it is reasonable to expect a neural network to learn the drawing space and if it manages to learn some human drawing behavior, such as, continuing to draw a curve. This is illustrated in figure 25 where the black line is what the drawer have drawn, and the red line being how we want the drawing assistant to continue to draw. To achieve this, the drawing assistant will use regression to determine how to subjectively best continue the drawers drawing.

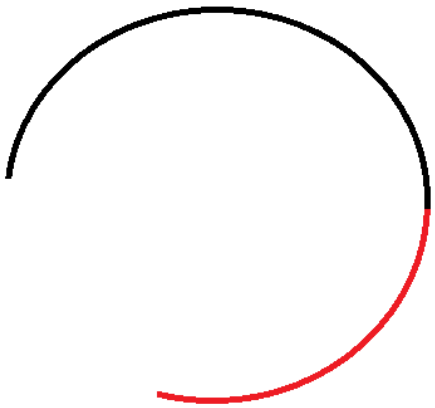


Figure 25: Desired outcome: the drawer draws a half circle, illustrated in black, and the drawing assistant continues to draw in red.

3.1.1 Data

The data we will use is the list of pixel coordinates visited when a line is drawn, starting with the first pixel in the list corresponding with where the drawer started to draw, and end with the last pixel in the list corresponding with where the drawer stopped drawing the line. The list will be two dimensional with each row containing the i and j coordinates of the pixel that was accessed, and the subsequently row will be pixels accessed after it. This data structure is illustrated in figure 9.

The size of the drawing canvas is 500 by 500 pixel, and therefore i and j are in the same range. To speed up the learning process, and to not end up with extremely large values in the neural network, which might lead to exploding gradients and unstable learning, we will divide the input by 500, have the output multiplied by 500 (rounding to the closest integer), as this is the simplest transform.

3.1.2 Machine learning

We choose the input of the neural network to be the three previous rows to what we want as the output. The reason we use three points is that it is the bare minimum information that is required to tell us if the line is curving or not. With two there is only enough information to predict a line through them. We could use more, but we want to keep it simple. In figure 26 we can se how we want the regression to work. It illustrate the first three pairs of x and y. The numbers in the illustrations correspond to the row number from the pixel coordinates access list. With x being the rows with arrows going out and x being the rows with arrows going in.

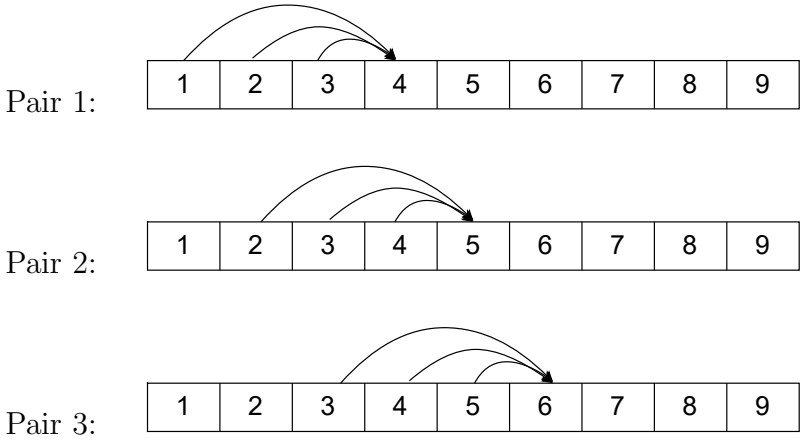


Figure 26: A illustration of the regression structure we use.

To learn from the data, we will use a fully connected neural network with the layout shown in table 1. Since we have a 2 dimensional space, we get 6 values from the 3 pixels as the input in the vector x, and the output y will be a vector of size 2 consisting of the i and j coordinates of the next pixel.

After trying out different learning rates, we find that a learning rate of 0.001 is the highest learning rate we can have before the network becomes unstable, and we want to use as high as possible learning rate, so we do not

Table 1: The neural network architecture we use for regression. We use 3 hidden layers as this should make the network able to learn more abstract structure, while not having a lot of parameters. The number of neurons are chosen as they are a power of 2, since this fits nicely in to how computers works, as they also work with powers of 2.

Layer	Input vector	Output vector
Dense	6	16
ReLU	16	16
Dense	16	32
ReLU	32	32
Dense	32	16
ReLU	16	16
Dense	16	2

need as much training data to get closer to a minimum. The loss function we will use is mean square error, which measure the average squared difference between the true values Y that we expected from the network, and the value we got from the network $\hat{Y}$. The formula for the mean square error is

$$\frac{\mathbf{1}}{\boldsymbol{n}} \sum_{\boldsymbol{t=0}}^{\boldsymbol{n}} (\boldsymbol{Y_t} - \hat{\boldsymbol{Y}}_{\boldsymbol{t}})^{\mathbf{2}} \tag{16}$$

We will use Adam with default parameters as they are in its PyTorch implementation.

Each time we draw a line in on the canvas, we run one training iteration, where the line we have drawn gets broken down as we described in the data part in this method, and used as one batch, as each line gets broken down to a varied batch size, depending on the length of the line. Then we train the network on that batch and update the parameters of the network, based on the loss. After the training step, the network starts with the last three pixel coordinates accessed as input and tries to predict the next point, but this time there is no true answer to what the output of the network should be, then we take this new point and add it to the list as the latest entry

and repeat the prediction, but this time one of the inputs to the network is the prediction we just made. We chose to do this 50 times to get a good visualization of how the drawing assistant behaves. In the end we end up with a prediction based on a on a prediction based on a prediction, and so on, back to the line we drew.

3.1.3 Learning

We train the network by drawing a variety of lines and curves on the canvas. In the beginning of training, the network predicts points goes across the drawing board, as shown in figure 27.

Figure 27: With little to no training the red continuation predictions starts at i and j in the top left where the coordinates in the canvas is zero, and increase at the same rate across the canvas.

The loss function over time when training is shown in figure 28. It is shown on an logarithmic scale as the loss in the beginning are in the 10^5 range while at the end is on an average 231.

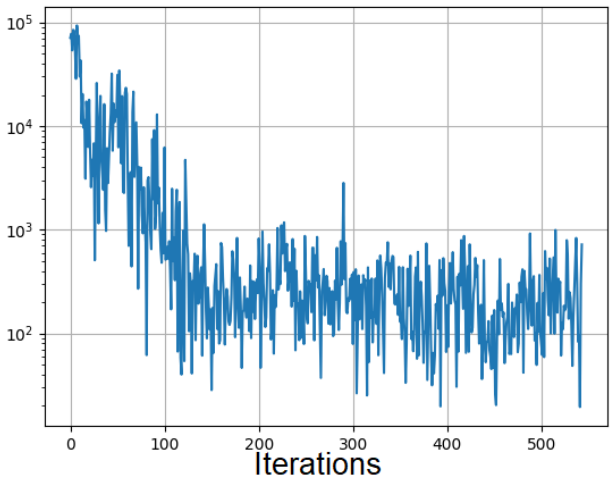


Figure 28: Result of the loss function over time on a logarithmic scale is on the vertical line. The horizontal line is number of lines trained on.

After drawing around 100 drawn lines we see the first mayor drop in the loss function, the predictions starts to get close to where the end of the line we drew is. An example of this is shown in figure 29.

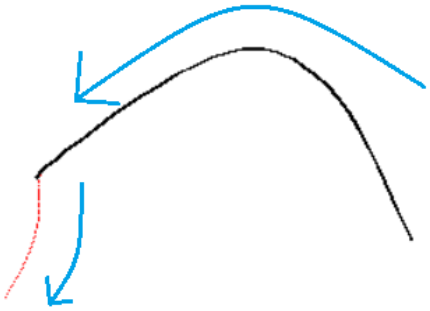


Figure 29: Network prediction in red after training on around 100 examples. The blue arrow being the direction of the drawing.

After 500 drawn lines, the average loss does not get any significantly lower. It seems as if the network does manage to learn and predict the general structure of a line and how each point is supposed to be after another in the same general direction, as we do not get any sharp turns in the predictions itself, but it fails to continue in the same manner as the line we drew, and sometimes end up going in the opposite direction. This might be caused by the network being given initial coordinates that makes it predict a 180° change in direction, this can be seen in figure 30, or it might be that it is a bias in the way we train. This might be solvable by taking in more coordinates, instead of only 3. This way we can give the drawing assistant more context for what it is drawing.

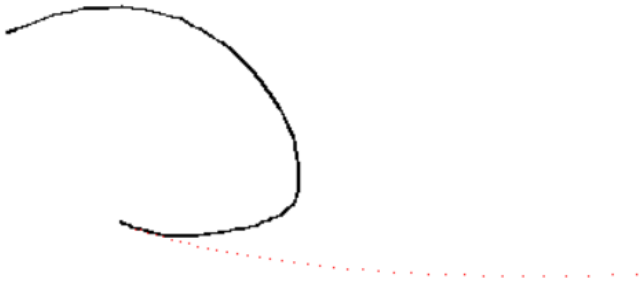


Figure 30: The red dots of the prediction is going in the opposite direction of the curve in black.

3.2 Convolutional neural network method

To give the AI drawing assistant more context, we will give it the same tools as humans, the ability to "see" the drawing as a whole. We will do this by using a convolutional neural network, that lets the drawing assistant learn shapes and get an overview of the entire drawing. In this method we want to make the drawing assistant that learns to mimic how the drawer draws. The idea is that if the drawing assistant learns what the drawer thinks is well drawn by learning directly from the what they draw, it can use this knowledge to help the drawer by using suggesting a line similar to what they drew, but based on the drawing assistants knowledge of what the drawer earlier thought were good lines.

3.2.1 Data

The data we will use is the image of what the newest drawn line d_t on the canvas, as in figure 7. The image from the canvas have the shape of a 3-dimensional array, that we will preprocess before using it as the input to the first convolutional layer of the network. The preprocessing we do is turning the image in to a black and white image by using the method dicused in the Images as data 2.1.2 section, the result is a 2-dimensional array with a size of 500 by 500 pixels. We also make use of the list of accessed pixels evaluate the result of convolutional neural network.

3.2.2 Machine learning

With the data transformed, we can use it as input to the network, in the shape of a 2-dimensional array. All the convolutional layers have the same filter size of 3 by 3. The reason we use a 3 by 3 filters, is that it uses less parameters compared to larger filters that is symmetric around a pixel. This lets us stack more convolutional layers while keeping the number of parameters low, resulting in the network being able to learn more complex

features. We will use a stride of 1 and padding as 1. This results in the output of the convolutional layer being the same shape as the input, the only difference being the number of filters that is used at each layer. In between the convolutional layers we have ReLU activation and Maxpool layers with a filter size of 2 by 2 and a stride of 2, resulting in a halving of size, while keeping the number of filters the same. Using MaxPool might lead to problems as it reduces the amount of information of the input, but we will use it since the input image is large. At the end we flatten the array, which has a shape of 15 by 15 by 256, down to a vector of size 57600 and use two dense layers to reduce the input down to 2048 as the final output of the network. This gives us a total of 1024 coordinates, which gives us some head room for how long of a line we can create. The network structure is shown in tabel 2.

The learning rate we will use is 0.001, as this is what worked in the regression method, and we will also use the same loss function we used in the regression method, mean square error loss, which is given in formula 16.

To train the network, we draw a line on the canvas, then the image d_t is used as the input x to the network. The output coordinates of the network gets placed on the canvas. Then, depending on what the one drawing think of the result, they can chose which line is better, or nether of them. If they chose their own line, the network will use that as the target y. If the line from the drawing assistant $\hat{y}$ is chosen, then the target of the network becomes $\hat{y}$. If nether is a option is satisfying, they can simply remove both their line and the drawing assistant suggestion and draw again with no impact on the drawing assistant.

Table 2: The architecture of the convolutional neural network we use. In this architecture the shape is mostly determend by the MaxPool layers, while we increase the number of filter by a power of 2, each time we use convolution.

Layer	Input shape	Output shape	Filters
Conv2d	500, 500	500, 500	16
ReLU	500, 500	500, 500	16
MaxPool	500, 500	250, 250	16
Conv2d	250, 250	250, 250	32
ReLU	250, 250	250, 250	32
MaxPool	250, 250	125, 125	32
Conv2d	125, 125	125, 125	64
ReLU	125, 125	125, 125	64
MaxPool	125, 125	62, 62	64
Conv2d	62, 62	62, 62	128
ReLU	62, 62	62, 62	128
MaxPool	62, 62	31, 31	128
Conv2d	31, 31	31, 31	256
ReLU	31, 31	31, 31	256
MaxPool	31, 31	15, 15	256
Flatten	15, 15	57600	
Dense	57600	4096	
ReLU	4096	4096	
Dense	4096	2048	

3.2.3 Learning

The first which is noticed when we try to teach the network how to act, is that the time it takes the network to process the ends up being 1-2 seconds with GPU acceleration, and 4-5 seconds without. This is just for the drawing assistant to come with a suggestion, and results in a bad user experience, which is one of the more important things we are trying to improve, not worsen, in this thesis. Another thing is that the network does not seem to learn what we want it to learn. This is illustrated in figure 31, where the red dots do not match what is drawn after 300 iterations.

Figure 31: Suggestion in red after training for 300 drawn lines.

Training up to 500 drawn lines only change the output slightly, but not closer to the output we want. Additionally it seams that two different inputs results in close to same output, this is seen in figure 32.

(a) (b)

Figure 32: Two different inputs in black, result in the same suggestion in red from the drawing assistant.

It looks like the drawing assistant need a lot more data to learn the behavior we want it to have. The amount that is seemed to be required, based on what we have done so far, is beyond the scope of this thesis. This is mainly due to the large number of parameter in the network we are using,

244, 716, 672 total trainable parameters, compered to the network we used in the regression method, shown in table 1, which only have 1, 218 trainable parameters. As mentioned, using MaxPool layers results in the lose of information and it seams as it has made the network unable to distinguish between objects in the image, which is most likely why we gets results like in figure 32, so we might need to create a different architecture without MaxPool. This is difficult because of the size of the initial image we have, so a simpler option would be to greatly reduce the size of the canvas, and this way needing an smaller architecture.

What we can do, is to try an approach that does not require as much data and the use of complex neural networks. We can insted use predetermed shapes that the training assistant has to distinguish between, this way the drawing assistanat do not have close to unlimited options as it has now, and it becomes easyer to determen what the "correct" action of the drawing assistant is.

3.3 Classification method

We have now explored two methods of creating a drawing assistant, both trying to place all the pixels needed to draw a line. What if instead we rely on geometry parametrization, and let the drawing assistant come up with the parameters best fitting the situation. Taking this one step further, we can reduce the complexity down to just choosing a geometric shape that the drawing assistant classifies as the closest to what we have drawn. The difference between what we are proposing in this thesis, and what is done in AutoDraw [3], is that in this thesis we want these shapes to be as little intrusive to the one drawing as possible. By that we mean that the shapes should be simple and seen as only a part of the larger picture, such as a straight line from one point to another, or a half circle, or a 90-degree angle.

These three figures are selected as we can combine into many different, more complex shapes. Such as a square with two angels or a circle with

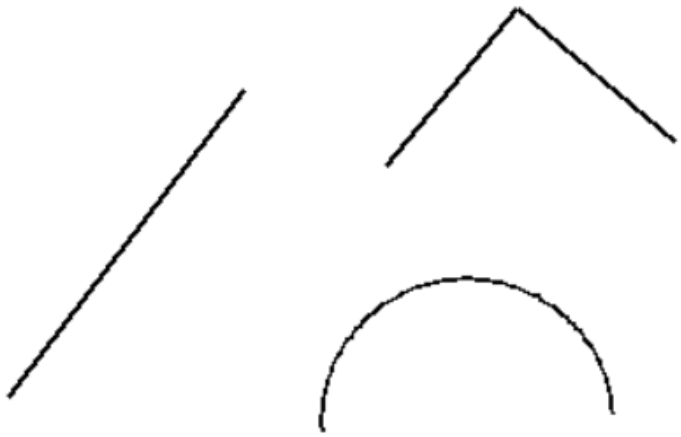


Figure 33: The three shapes we that has been selected to use, with the option to add more in the future

two half circles. In addition, these shapes are not limited to what sizes we want to make them in since they are created using vector graphics in the background.

3.3.1 Data

The data we will use is a sample of the list of accessed pixels (figure 9) from the most recent drawn line. The sample will consist of 9 pixel coordinates with equal index interval in the list from the start to the end of the line. This reduces the amount of data needed while the shape of what is drawn, we can see this in figure 34. The reason we use 9 is that we get a reasonable resolution, and since we have an odd number, we have one point as the middle value in the sequence.

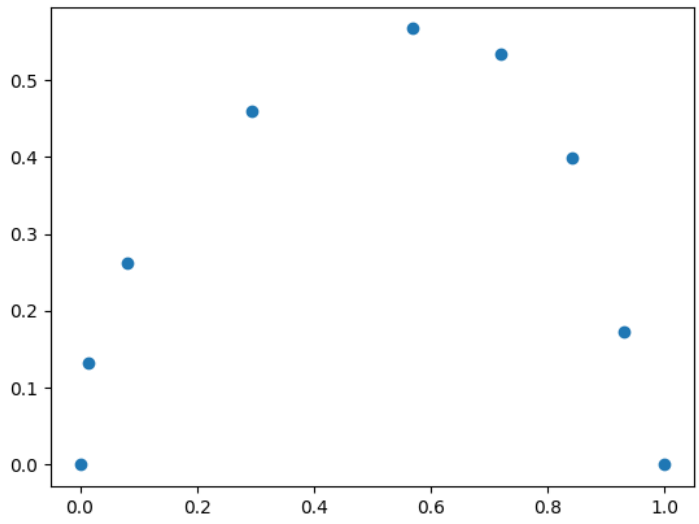
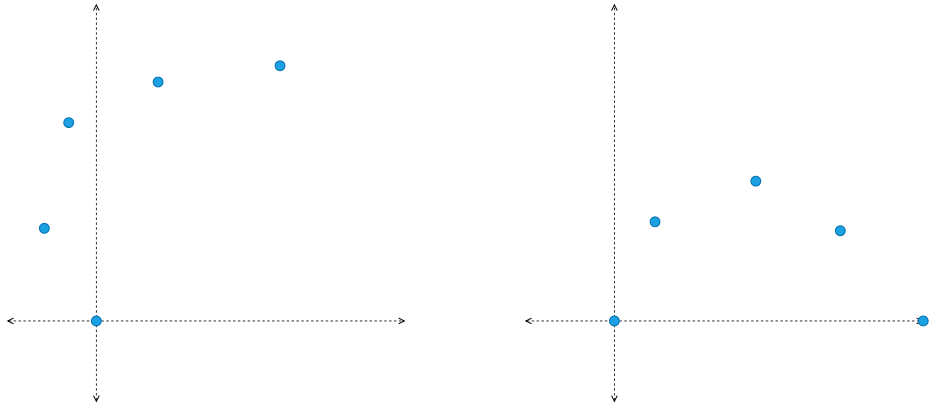


Figure 34: A scatter plot of 9 pixel coordinates.

In addition to limiting the number of points to 9, we also transform them. The transformations we do are two-fold, first we subtract the location of the first coordinate from the list from all the coordinates in the list, this way we always start with the first coordinate being in the location $[0, 0]$. The reason we do this is because we want the same shape to have the same coordinate inputs to the network, no matter if it was originally drawn in the upper left or lower right corner of the canvas.

The second transform is a change of basis. This transformation allows us to both rotate and scale the points at the same time, so that the last point is always in the location $[1, 0]$, and the rest of the points are scaled so that they are at the same relative position in regards to the start and end points.



(a) Before change (b) After change

Figure 35: Changing the orientation of pixel coordinates to get the first and last point in the sequence on the i axis.

The transformation matrix A is made by using the vector from the origin to the last point in the list as the first basis vector, and we use the same vector but tilted 90 degrees as the second basis vector. We also need to take the inverse of the matrix since we want to transform into the new basis. The result is the transformation matrix that changes the locations of all the accessed pixel coordinates so that it the orientation and scale of what is drawn the same:

$$A = \begin{bmatrix} x_{\text{end}} & y_{\text{end}} \\ y_{\text{end}} & -x_{\text{end}} \end{bmatrix}^{-1} \tag{17}$$

3.3.2 Machine learning

With the input data to the network transformed we can pass it to the network as one vector containing all the 9 points, a total of 18 input values to the network. The neural network we will use is a fully connected network containing 3 layers. The first two layers have a ReLU activation function. The output layer ends up having 5 output classes, as we have to account for the fact that each shape has a mirrored version of itself, except for the straight line, that we have to be able to distinguish between. We will use softMax as the output, since it makes the output values sum to 1, and can be used as a probability distribution and can be used in cross entropy loss. At the end, we select the output with the largest value from SoftMax, as the choice of the network. This choice then gets used to create the shape that was classified, starting in the first pixel of the list of accessed pixels, and scaled and rotated so that it ends in the last pixel in the list of accessed pixels.

Table 3: The neural network architecture we will use for classifying the shapes. We use the same network design idea we used in the for the regression method, but with one less network layer to reduce the number of parameters.

Layer	Input vector	Output vector
Dense	18	32
ReLU	32	32
Dense	32	16
ReLU	16	16
Dense	16	5
SoftMax	5	5

After drawing a line on the canvas we have the option to select which one of the 5 shapes we have draw. When selected, the network will use cross entropy as the loss function, which increase the output of the network corresponding with that shape and decrease the output for the other shapes.

Since we have more than two classes, cross entropy is given as

$$-\sum_{c=0}^{K} b_c \ln(f(\theta, x)_c) \tag{18}$$

where b is a binary indicator if the shape number c is correct and $f(\theta, x)_c$ is the network output value for shape c.

The optimizer we use is Adam with standard parameters from PyTorch. We found that a learning rate of 0.01 is the largest learning rate we can have.

3.3.3 Learning

The training process ends up requiring lots of iterations. In addition, the networks tend to get stuck choosing one over the other two other, sometimes getting hard stuck and having to reinitialize the weights of the network and start over. After around 250 drawn lines, the network starts to be able to choose the straight line over the curve and angled shapes, which the network has a hard time with, because of their similarities.

In figure 36 we can see how the loss stabilize after around 100 drawn lines, and at this point it starts to predict the correct shapes with output of the SoftMax being above 0.85, which can be interpreted as the network gives an above 85% probability of the input data being that shape. But we also gets some mayor spikes every time the network predicts wrong between the angle and the curve.

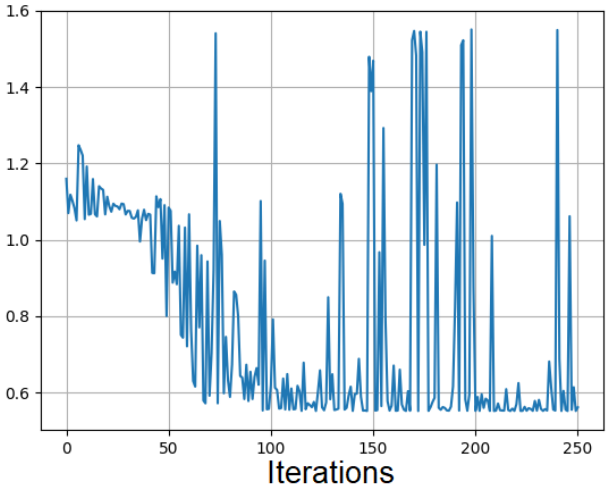


Figure 36: Loss function (vertical line) over training iterations (horizontal line).

To improve the performance and to reduce bias in the training, we could use a collection of training data. So instead of of training after each line we draw, we collect the labeled data in a list, and when we want to train, we can sample randomly selected entries in the list to get a more diverse training. With a list we also do not need to recreate the learning data each time we restart the program.

3.4 Off policy model based reinforcement learning

Continuing directly from the classification method, the process of always having to give the correct answer for what shape is correct, is a tedious process that takes away from the drawing process, and it would get worse if we add lots of new shapes that we have to deal with. What if we could instead just tell the drawing assistant if it chose the correct shape or not, simply by either continuing to draw or pressing the undo button. In this method we will use reinforcement learning, as it is well suited in this thesis as we do not have a lot of data to work with, and we want the AI drawing assistance to continually adapt and improve to the environment, which is the one drawing. What we have is an contextual bandit problem, where the drawing assistant choose the shape that maximizes the expected return, and the line that we draw as the state.

From the previous method we concluded that only using one example at the time resulted in the need for lots of examples, but by creating a model of the environment, that is, we remember what the input to the network was, what action the network took and what reward it received, we can use this to improve the network, without having to get new data from the user. To get a more diverse model, we will also use a different policy for when we are creating new training data for the model, then when we want to learn from the model. The policy for creating new data will be and Epsilon greedy policy, which results in us getting random actions from the network. This makes the network explore different actions which result in a better model of the environment. The one we use when learning is the greedy policy as we want to learn what the best action is for what is drawn. Using the model, we sample randomly entries and use that as training to improve the drawing assistant.

3.4.1 Data

We will use the same data structure as in the Classification method.

3.4.2 Machine learning

The network we will use is an altered version of the one found in the Classification method. The difference being the addition of an epsilon greedy layer, and 5 outputs of the third dense layer. If the network is set to generate new entry in for the model, we use the epsilon greedy policy, this layer takes in the probability distribution from SoftMax and returns either the largest action value or, with a 5% chance, a random action value. If it is not, the network always returns the larges action value. The network will also return what shape is selected, and we use this to replace what user drew, with the shape that was selected.

Table 4: The neural network used in the classification method, modified for reinforcement learning. We follow the same layout as in earlier methods, with the exception of the epsilon greedy-layer. We use SoftMax as it is needed in how we calculate the loss of the network.

Layer	Input shape	Output shape
Dense	18	32
ReLU	32	32
Dense	32	16
ReLU	16	16
Dense	16	5
SoftMax	5	5
Epsilon-Greedy	5	2

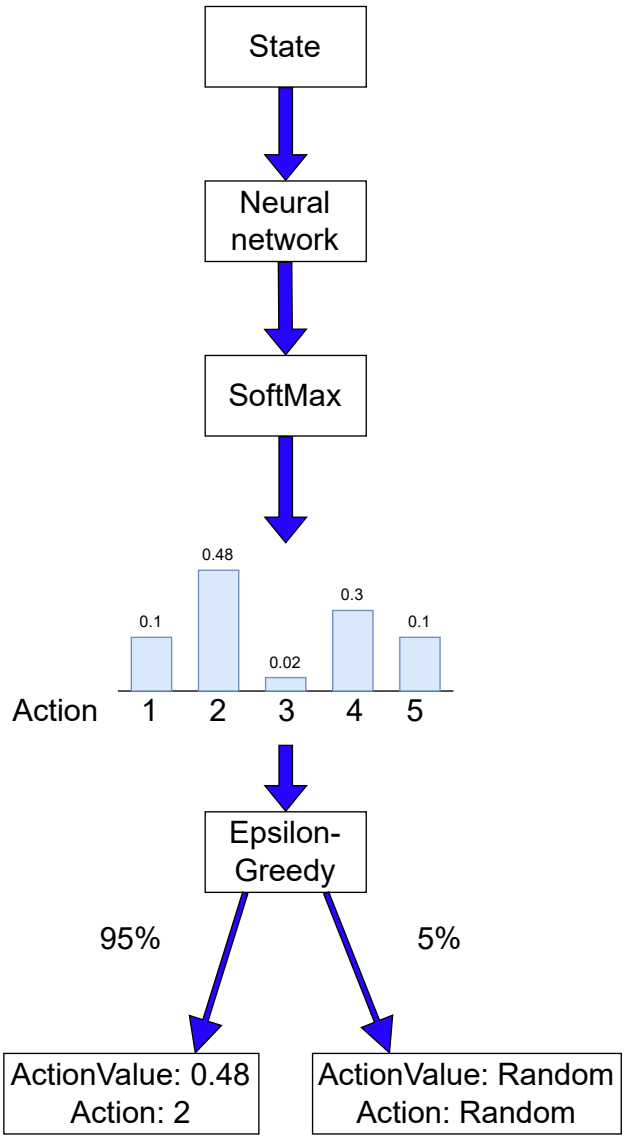


Figure 37: The action selection process of our drawing assistant. Starting with the state, which is the transformed line drawn, as the state, this then gets passed on to the network which produces a softmax output where we select either the largest value with a 95% probability, or an random value. The end result is the action value and what action it is.

After we have replaced what the user drew, the user can either do nothing and continue drawing, which results in the reward for the agent becoming 1, or the user can remove the change, and the reward to the agent becomes 0. The state, the action of the agent and the reward, then get stored in the model to be used when training. To updates the parameters of the network we will use a modified version of binary cross entropy loss, that takes the reward r and the action value $Q(a)$ into account:

$$L = -(r \cdot \ln(Q(a)) + (1 - r) \cdot \ln(1 - Q(a))) \tag{19}$$

This loss function makes the action that gets 0 reward go to 0 in action value, and action that gets 1 in reward go to 1 in action value. When training, we use the samples from the model which contain a state, action and reward, we make the network return the same action value as the one corresponding to the action in the model, and we use the loss function seen in (19) to update the weights.

Adam is used as an optimizer with standard parameters from PyTorch, with learning rate of 0.01, as this is the highest value we can have before the network becomes unstable. Since there is only one of the action values returned from the network, the changes of the SoftMax output-values when we apply the change as given from the loss function ends up being one of two types, either increase the action value we got out, and decrease the others action values, or decrease the action value we go out and increase the others.

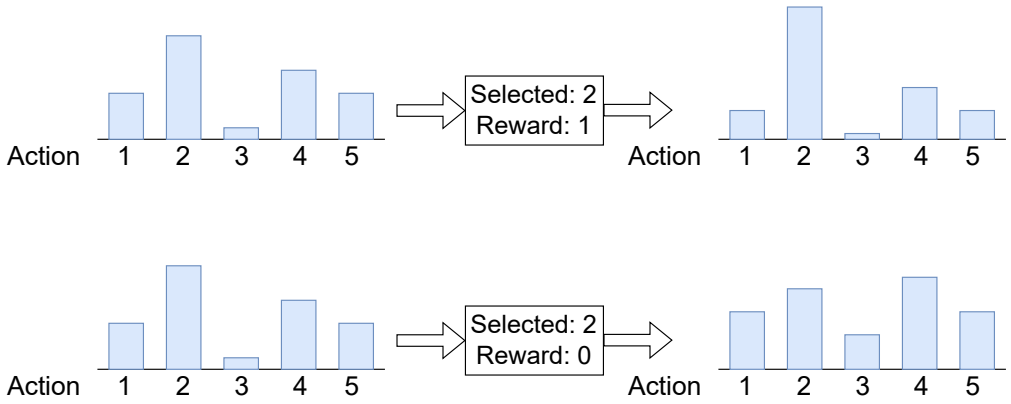


Figure 38: Illustrates how the output of the SoftMax layer changes, based on if the reward for the action 2 is 1 or 0.

3.4.3 Learning

After creating 700 entries in the model, and training the neural network on 1000 sets of 10 randomly sampled entries, we get a plot of the loss over the training in figure 39. We can see the same spikes in the loss function, same as in figure 36, which most likely is caused by the same reason, difficult in separating the curve and the angle.

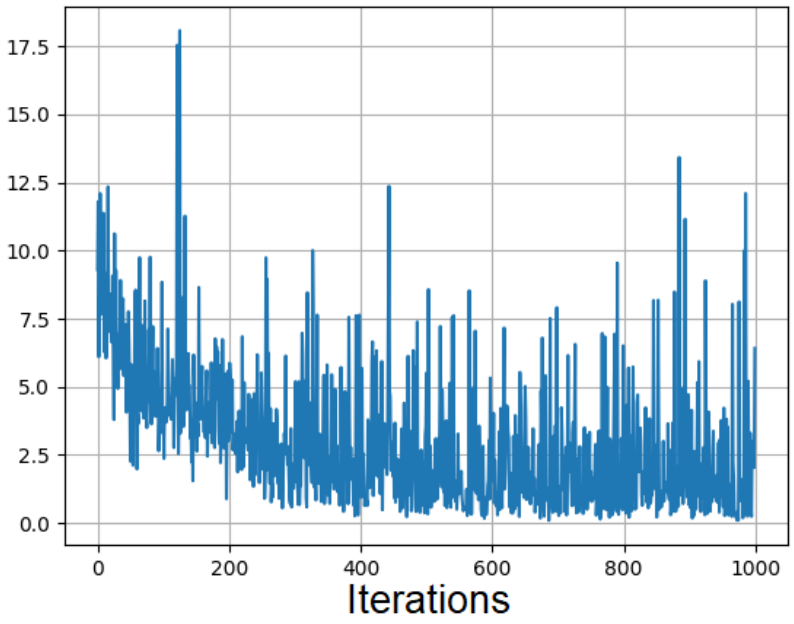


Figure 39: Result of the loss function from each of 1000 training iteration.

To see how accurate the network is we have made a measurement for accuracy that is defined by the formula:

$$\frac{1}{n}\sum_{t=0}^{n}(a_{\text{Model}_t} = a_{\text{Network}_t}) = R_{\text{Model}_t} \tag{20}$$

The formula uses logical equivalence to first check if the network and the model did the same action for a state, then compare that to the reward. The idea is that if the action of the model and the network is the same, and the reward is 1, the network chose the right action. If the network and the model was different and the reward was 0, the network have learned that it should not do that action for the given input state. The accuracy does not consider if the network chooses the right action for a given state, just how well it learns the entries in the model.

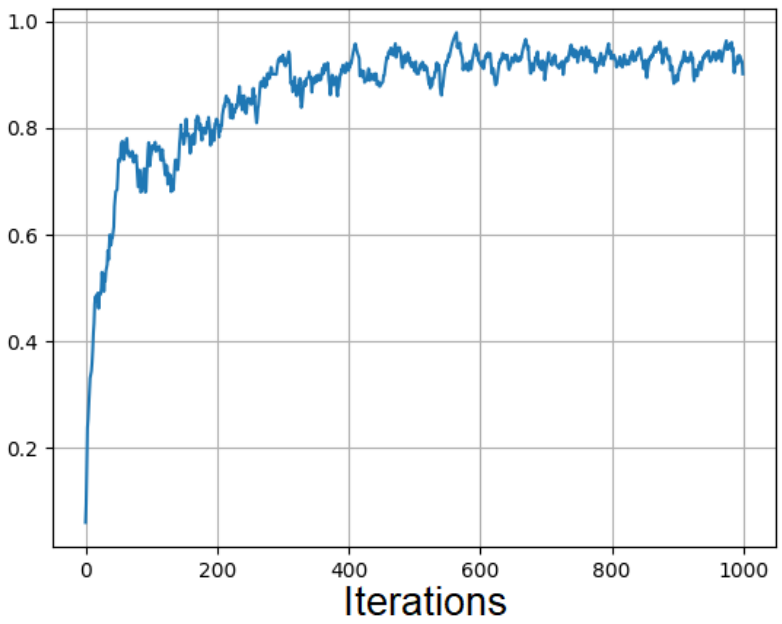


Figure 40: Moving average of the accuracy over the duration of the training.

The moving moving average plot of the accuracy can be seen in figure 40. The end value of the accuracy after training is 89%. We can see that after 200 iteration the accuracy passes 80% and it ends up on an average of 91% after 600 iterations.

Another measurement we have made, is a way to see how the average action value is compered to what it should be according to the reward. We call this the confidence, and is given by the formula:

$$\frac{1}{n}\sum_{i=0}^{n}\begin{cases}1-Q(a_t), & \text{if } r_t = 0\\ Q(a_t) & \text{if } r_t = 1\end{cases} \tag{21}$$

This formula gives values close to 1 if the action value is close to 0 for model entries with 0 in reward, also if the action value is close to 1 for

model entries with 1 in reward. Given the 75% confidence of the network, that means that for model entries it has on average action value of 0.75 for actions it should take, and an average action value of 0.25 for actions it should not take.

The moving moving average plot of the confidence can be seen in figure 41. The end value of the confidence after training is 86%. It takes 260 iterations before the confidence gets over 80%, and it stabilizes at around 85%.

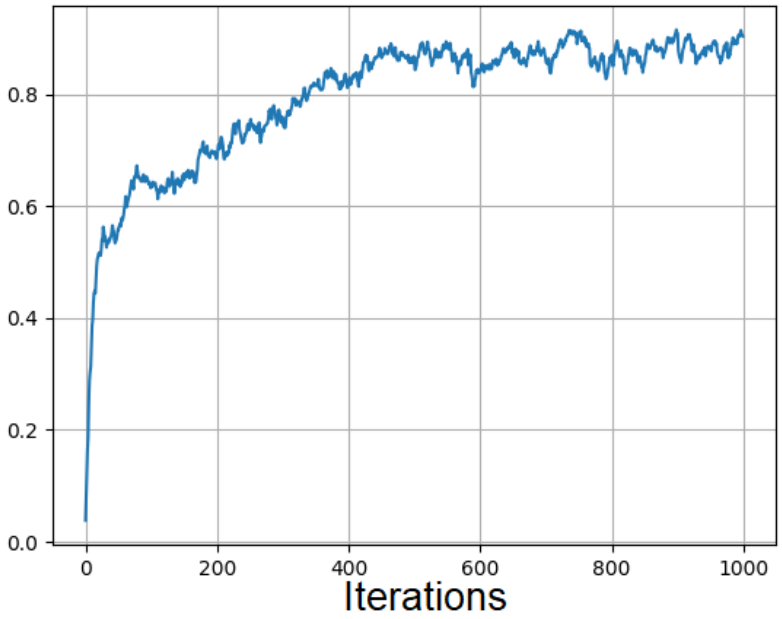


Figure 41: Moving average of the confidence over the duration of the training.

For this method we also created a memory module that can record something that is drawn and redraw it exact. This gives us a more visual representation for how well the network is doing. The correct drawing is seen in figure 42.

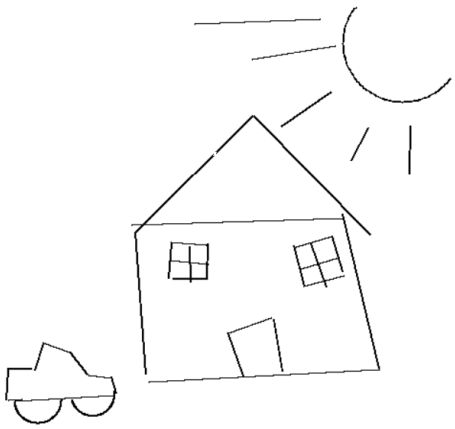


Figure 42: The correct drawing that we use as a visual test.

And here are the results from two different network initializations can be seen seen in figure 42.

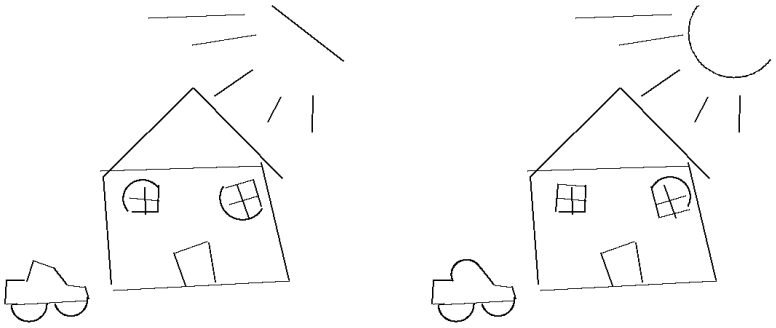


Figure 43: Different initialization and training of the neural network result in different results on the visual test. We can see how the drawing assistant have difficulty in separating the curve from the angle for example in the windows, and also mistaking the sun for a straight line on the left.

This method resulted in an AI drawing assistant that most of the times chose the right action and can continuously improve over time by gathering more entries to the model.

This method can be improved further. We can for example add more shapes, like the angels with 30, 60 and 120 degrees as shapes that can be learned, but given how hard it is for the drawing assistant to separate the half circle and the 90 degree angle, we most likely will be limited to shapes that are more distinct to each other for the network to be able to learn them. We can also add add more preprocessing and postprocessing that can increase the number of figures the drawing assistant can create, without having to learn new shapes. This can for example be to find a way to break down complex shapes that the drawing assistant is not trained on, to shapes it have trained to recognize.

When showing this version to 4 people, the first all of them did was trying to draw a full circle, a shape the drawing assistant is not trained to recognize. This could be solved by either breaking the circle up in two half circles, for which the drawing assistant know, or we can add the circle as a new shape.

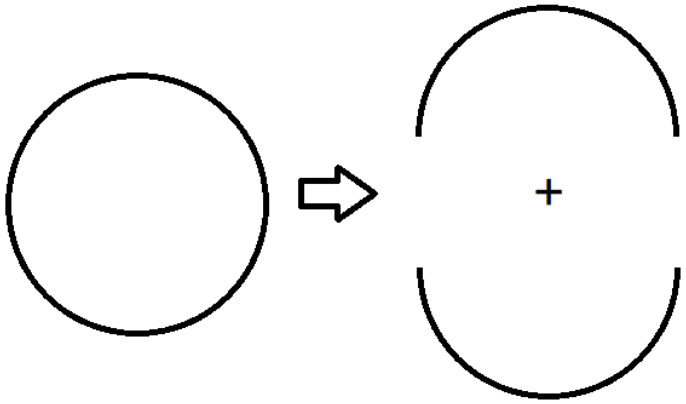


Figure 44: We can express more complex figures by using multiple shapes together.

We could also increase how much the drawing assistant know about what is already drawn to give it more spatial awareness to base the action values on, this could make the drawing able to learn what shapes often follow each other and increase the overall performance, but we also run the risk of teaching the drawing assistant in a way that makes it get stuck on guessing the same shape all the time.

3.4.4 Continuation

Now that we have a method that works reasonably well, let's try to improve it even more! Let's see if there is any benefit by giving the drawing assistants an awareness of previous shapes and its position. We do this by adding the previous action to the network input, this way we get information d_{t-1}, and not only d_t. Additionally, we will add a way for the network to tell the distance to the previous drawn shape. In total we add three inputs to the network, the previous action and the distance to the first and last point of the previous shape.

After creating 300 entries in a new model with the new network inputs, we get the plot of the loss function in figure 45. Here we can see that there is an decrees in height of the spikes compered to earlier methods, from this it seams like this method has less problems separating the shapes from each other.

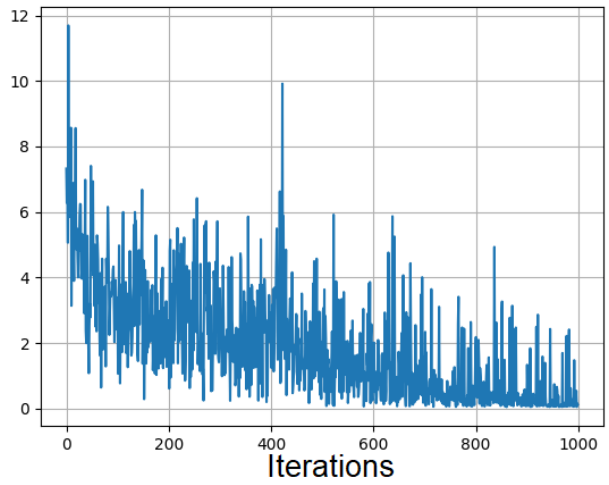


Figure 45: Plot of the loss function over the training iterations.

This is also evident in the fact that the average accuracy has increased up to 97%, with the moving average plot over the training in figure 46.

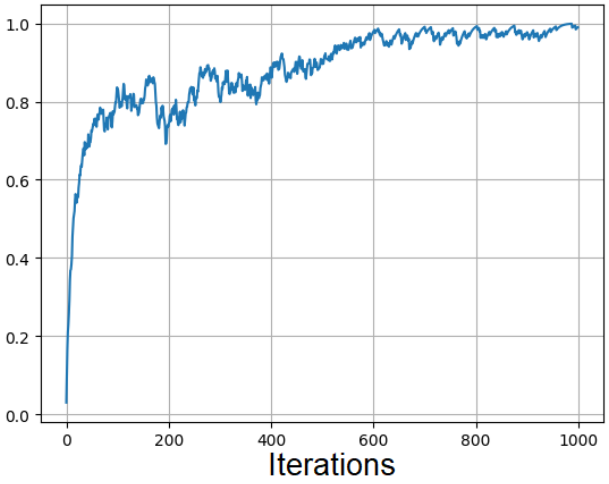


Figure 46: Moving average of the accuracy over the duration of the training.

It takes about the same number of iterations to reach 80% accuracy com-

pered to not using history data, but it continues to rise up to an average confidence of 98%

The confidence has also increased and ends at 95% which means that the drawing assistant is really certain in its shape selection. The plot of the moving average confidence can be seen in figure 47. Here we can see that we reach a confidence of over 80% before 200 iterations, and it seams like it stabilizes at around 96% confidence.

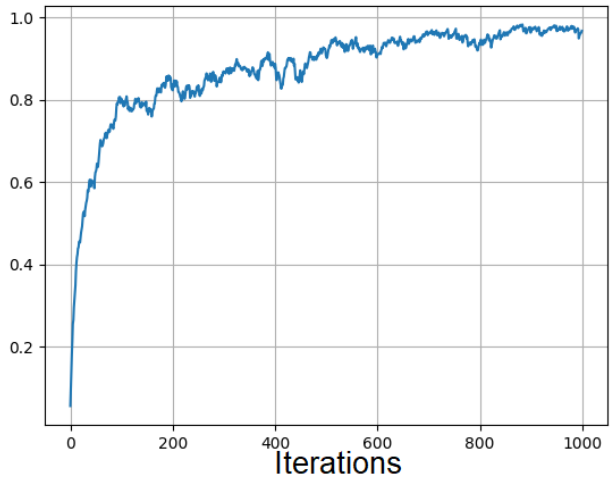


Figure 47: Moving average of the confidence over the duration of the training.

And now the drawing assistant can with some of the initializations get a perfect score on the test drawing, and when it does not get a perfect score, its only one shape that is wrong. An overall improvement when including information from the previous shape drawn. These to different cases can be seen in figure 48.

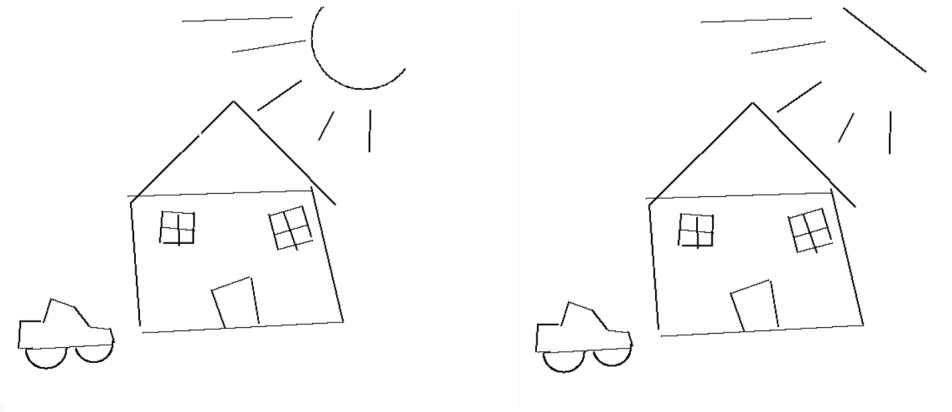


Figure 48: Two results on the test drawing with the continuation improvements. the left ending up perfect, while the one on the right only having 1 mistake.

4 Analysis and Discussion

The goal of this thesis was explored how to create an AI drawing assistant. The first question that comes to come to mind is, what even is an AI drawing assistant? It being an assistant suggests that the AI should do tasks that typically would be a chore for the one drawing, but the task still must be done. But it could also be that the assistant should help achieve a certain goal, such as drawing in a particular manner, by copying the drawing style of a of another, kind of like style transfer [17] or a deepfake [18] for drawing.

Another thing we had to think about was what is a drawing. How do we judge one drawing over another. Ask enough people and we will find that every drawing is both the best in the world and the worst in the world. This has been one of the bigger challenges in this thesis, as it is hard to put a number on how good or bad the results of one method over another. Are the results in the regression method better than the one in the convolutional method? No, since we need the experience from trying both methods to learn what we learned from them. In the regression method we learned that we need to use more information than just a few points to learn drawing

behavior. And in the convolutional method we learned that we can't just give the drawing assistant a visual cortex as animals has and create a complex convolutional neural network, as this ends up being detrimental to the speed of the network to make decisions, and it would be a bad experience if the one drawing must wait many seconds to get the prediction from the drawing assistant, each time a new line is drawn.

Ideally, we would have a perfect understanding of what the one drawing is thinking, and by analyzing the neural impulses in their brain, we can predict the shapes and forms that is in their mind. But this is at the current level of technology not possible to do, and even if possible, it would require more hardware than the average person have to use.

Not having any suitable premade datasets have been one of the obstacles in this thesis, as it does not give a way to train models on larger sets of data. To solve this, we have delve into the field of reinforcement learning, as it can learn only from gathering experience. It is also more suited for continuous learning over time as it gets more and more data to work with. We also have a thesis that is well suited for reinforcement learning, as we have a state, the input to the network, an action, what drawing assistant do with the information from the state, and the environment, which ends up being the user, who has an opinion of what is good and bad and can give reward accordingly.

The classification method borrows elements from the regression and the convolutional method. The data is the list of accessed coordinates from the latest drawn line, that was used in the regression method, but in contrast, we sample evenly spaced coordinates in the list, not just the last three, this way we can still "see" the drawn line in how we represent the data as the input to the neural network, similarly to the convolutional method.

This method ended up as being the core of the continuation of this thesis, as it had a good performance, and the problems can be solved by adding more elements to the drawing assistant, such as training on more than one example

at the time and adding a data set which we can train from. The general data structure of having a selection of shapes, which can be used to make more complex figures, ends up being the most efficient way to create an AI drawing assistant.

To try and get more performance we decide to change the way we learn. On one hand we could stick with the classification method and try to improve it, but this would make it harder to add new shapes or alter the data input or output in any way without having to recreate the data set. On the other hand, we can implement a reinforcement learning technique which require more complexity of the neural network we use and how data is stored and learned, but altering the number if shapes would not be a problem as the example only tells if a particular action is good or bad, not what action is good or bad, so we end up in this "pull and push" learning, where you can learn when to do an action and when to not do an action, independent of all other actions.

For our goal of expanding the number of shapes we can learn and the way we learn shapes independently of each other would make learning new shapes easier, since the network can still use what it already know of the old shape, to separate them from new shapes, the reinforcement learning approach ends up being the best alternative.

When implementing this new reinforcement learning method, we decide to make it use two different policies. The reason we do this, is to have one policy we use to generates example that we learn from, and to explore different actions using a epsilon greedy policy, and have a policy that we are trying to learn, which is the greedy policy, as we do not want any random actions when we are just using the network. We also make the reinforcement learning model based, so that we can store data in the model, and we do not have to acquire new training examples every time we reinitialize the network. Using a model also means that we can learn with a smaller amount of data from the environment by sampling experience from the model.

This AI drawing assistant ended up getting a good performance on both the accuracy and confidence. The test drawing was also mostly correct, but we can see that most of the places where there was a mistake, it was the half circle witch most often ended up being in the wrong place. This is most likely because it is in between the straight line and the angle, so if the curve of what I drawn is too small, the results become a straight line, but if it is too steep, the result become an angle. Same goes for the other two shapes just in a different sector.

In the continuation section we try to improve this performance by giving the drawing assistance knowledge about the past. For the drawing assistant to really understand what the logical next step is, it needs to know about the past. Not only what the past objects is but also where they are relative to the new objects the drawing assistant is trying to make. With only one step back in time the drawing assistant manages to reach an almost perfect score. This can be because past knowledge has such a big impact on the result, but we must not exclude over-fitting as a conclusion, as it might be that both the model and the test drawing is almost the same, and if we try to draw something completely new, the result might drop significantly.

5 Summary

In this thesis we wanted to create an AI drawing assistant that would make it easier for a person to draw. We experimentally explored ways we can use neural networks and machine learning techniques to succeed in this task. We have created a way to preprocess what is drawn into a standardized format where the placement and the orientation of what is drawn is normalized. The result is an AI drawing assistant that can accurately predict what shapes the user want to draw and create shape at the place where it is intended to be. It works as "support wheels" for the one drawing, creating perfect lines since the shapes are generated using vector graphics and it helps creating vector graphics faster than having to rely on selecting shapes in a menu. We have created a platform for which it is easy add new shapes and expand further with more features.

We gained an interesting insight in to how to approach a problem for which there is not concrete answer for what is correct and what is wrong, and how to deal with a neural network when the data we need to train is limited to what we can produce ourselves. We also got proficient at using reinforcement learning to get around the limitations and how we can use a combination of reinforcement learning and manipulation of how the gradients in the networks are calculated, to update the weights in the network the way we wanted.

6 Future work

As the AI drawing assistant is now, there is many opportunities for where we can develop it next. The most obvious one is to add more shapes that we can use. And we also have increased the number of steps back in history the drawing assistant can see. What would be interesting would be to see what happens with the drawing assistant when $t - n$ in d_{t-n} is approaching 0 and we get information about the entire drawing D_t. One could also find a way to make more complex structures by combining shapes that we have learned. We could for example add figures similarly to the one in Autodraw, where after drawing smaller shapes it at some point, the drawing assistant combine all the shapes into a larger figure as we see in figure 3.

Another approach is to drop the use of premade shapes and try to create an drawing assistant that tries to predict the parameter of a curve. In vector graphics we can create a curve that goes through two points and has a curving towards a third point, for example in a quadratic Bezier curve [19].

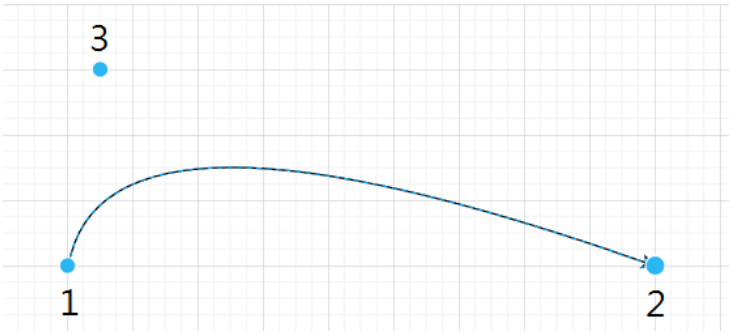


Figure 49: A quadratic Bezier curve given by a start and end point, 1 and 2, with a offset point 3, which the line curve towards.

Since we know where the curve start and ends from the list of accessed pixels, we only need to predict the point that is numbered 3 in figure 49, and to give us more flexibility and diversity we can use a probability distribution of where that point can be. This way we can get different results from drawing the same line multiple times. The hope is that we end up with a

drawing assistant that do not act as predictable, and machine like. Just like how humans have a have a hard time drawing the exact same line two times.

References

[1] David Ha and Douglas Eck. A neural representation of sketch drawings. *arXiv preprint arXiv:1704.03477*, 2017.

[2] Takashi Kawashima Jongmin Kim Nick Fox-Gieg Google Creative Lab Data Arts Team Jonas Jongejan, Henry Rowley. Quick, draw!, 2017. https://web.archive.org/web/20220407135948/https://experiments.withgoogle.com/quick-draw, urldate 2022-04-7.

[3] Google Creative Lab Dan Motzenbecker, Kyle Phillips. Autodraw, 2017. https://web.archive.org/web/20220326173444/https://experiments.withgoogle.com/autodraw, urldate 2022-03-26.

[4] Thomas Adajian. The Definition of Art. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, Spring 2022 edition, 2022.

[5] Christopher J.C. Burges Yann LeCun, Corinna Cortes. The mnist database of handwritten digits. http://yann.lecun.com/exdb/mnist/, urldate 2022-05-4.

[6] Adam H. Marblestone, Greg Wayne, and Konrad P. Kording. Toward an integration of deep learning and neuroscience. *Frontiers in Computational Neuroscience*, 10, sep 2016.

[7] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[8] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014.

[9] Sebastian Ruder. An overview of gradient descent optimization algorithms. 2016.

[10] Zewen Li, Wenjie Yang, Shouheng Peng, and Fan Liu. A survey of convolutional neural networks: Analysis, applications, and prospects. 2020.

[11] Fukushima Kunihiko. A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position, 1980.

[12] IBM. Convolutional neural networks. https://www.ibm.com/cloud/learn/convolutional-neural-networks.

[13] Guido van Rossum Steen Lumhol. Tkinter: Python standard interface kit. https://docs.python.org/3/library/tkinter.html.

[14] Meta AI. Pytorch: an open sorce machine learning framework. https://pytorch.org/.

[15] NVIDIA. Cuda: a parallel computing platform and programming model. https://developer.nvidia.com/cuda-zone.

[16] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[17] Leon A. Gatys, Alexander S. Ecker, and Matthias Bethge. A neural algorithm of artistic style, 2015.

[18] Thanh Thi Nguyen, Quoc Viet Hung Nguyen, Dung Tien Nguyen, Duc Thanh Nguyen, Thien Huynh-The, Saeid Nahavandi, Thanh Tam Nguyen, Quoc-Viet Pham, and Cuong M. Nguyen. Deep learning for deepfakes creation and detection: A survey, 2019.

[19] Michiel Hazewinkel. Encyclopaedia of mathematics: Supplement. page 119.