Honors Thesis

USING DEEP LEARNING TECHNIQUES TO FIND THE 4D SLICE GENUS OF A KNOT

by

Dylan Skinner

Submitted to Brigham Young University in partial fulfillment of graduation requirements for University Honors

Mathematics Department

Brigham Young University

November 2023

Advisor: Mark Clifford Hughes

Honors Coordinator: Davi Obata

Abstract

USING DEEP LEARNING TECHNIQUES TO FIND THE 4D SLICE GENUS OF A KNOT

Dylan M. Skinner
Mathematics Department
Bachelors of Science

Deep reinforcement learning (DRL) has proven to be exceptionally effective in addressing challenges related to pattern recognition and problem-solving, particularly in domains where human intuition faces limitations. Within the field of knot theory, a significant obstacle lies in the construction of minimal-genus slice surfaces for knots of varying complexity. This thesis presents a new approach harnessing the capabilities of DRL to address this challenging problem. By employing braid representations of knots, our methodology involves training reinforcement learning agents to generate minimal-genus slice surfaces. The agents achieve this by identifying optimal sequences of braid transformations with respect to a defined objective function.

Contents

Title i

Abstract ii

Table of Contents iii

1 Introduction 1

2 Knot Theory 1
2.1 Introduction to Knot Theory . . . 1
2.2 Braids, Surfaces, and Braided Surfaces . . . 4
2.3 Slice Surfaces and Slice Genus . . . 13

3 Reinforcement Learning 17
3.1 Introduction to Reinforcement Learning . . . 17
3.2 Proximal Policy Optimization . . . 19

4 Our Problem and Approach 21
4.1 Using PPO . . . 24

5 Results 27

6 Future Work 33

7 Appendix 36
7.1 Model Architecture . . . 36

1 Introduction

In this thesis, we tackle the challenging problem in knot theory of determining the minimal genus of slice surface bounded by a given knot (section 2.3). To address this problem, we employ cutting-edge reinforcement learning algorithms, specifically leveraging the Proximal Policy Optimization (PPO) algorithm, to interact with a custom OpenAI Gym environment (sections 3.1, 3.2).

In order to introduce the problem and our approach, we build foundational knowledge in a few key areas. This includes an exploration of fundamental concepts in knot theory (section 2.1); braids, and Seifert surfaces (section 2.2). We also introduce slice surfaces, a critical aspect of the minimal slice genus problem (section 2.3). Additionally, we provide an overview of reinforcement learning, including Markov decision processes (section 3.1), and an exploration of the PPO algorithm (section 3.2).

2 Knot Theory

2.1 Introduction to Knot Theory

When a non-mathematician thinks of a knot, they typically think about taking a piece of string and tying the two ends together in some specific way. In mathematics, knots are thought of differently. Instead of taking a piece of string and tying the two ends together, a mathematical *knot* can be thought of by taking the two ends of a necklace, tying a knot in the middle of the necklace, and then clasping the two ends together. This results in a knotted loop where the only way to untangle the knot in the middle of the necklace is to cut the necklace and remove the knot. Put another way, a knot is a knotted loop of string except that we think of the string as having no thickness, and its cross-section being a single point [1]. Similarly, a *link* is a collection of multiple knots linked together.

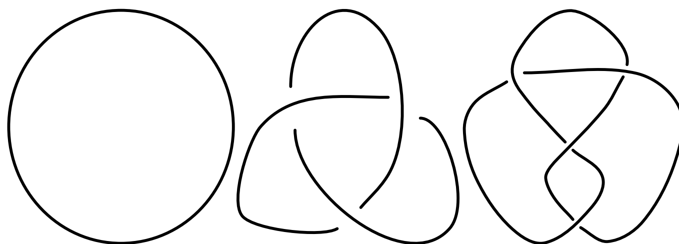


Figure 1: Three of the most basic knots that are used. On the left we have the unknot, the middle is 3_1 (the trefoil knot), and the right is 4_1 (the figure 8 knot).

An important idea in knot theory is that there is no distinction between any particular configuration of a knot. In other words, if you take a knot and deform it in some way without passing it through itself or by cutting the string, the resulting knot is thought of as being the same as the original one. But, one problem arises from this idea. How can we tell when two different-looking knots are equivalent?

Before looking at the mathematical equivalence of knots, it is important to understand a few key terms. The first is *projection*. Knots live in three-dimensional space. However, it is often convenient to describe them using two-dimensional pictures. These two-dimensional pictures are obtained by projecting the three-dimensional knot onto a two-dimensional plane. If you were to take two 'different' projections of the same knot to a plane and create a model of one of those projections out of string, you should be able to rearrange the string into the second projection without cutting the string. This idea of rearranging the string in 3-dimensional space is something knot theorists call an *ambient isotopy*. *Ambient* refers to the fact that the string is being deformed in three-dimensional space, and *isotopy* the deformation of the string. When deforming a knot projection, knot theorists use the term *planar isotopy*, with *planar* referring to the fact that the knot is only being deformed within the projection plane.

This idea of isotopy is important in determining the equivalence of knots. Although two knots are considered equivalent if you can change one knot into the other by stretching it and moving it around without tearing it or causing the knot to inter-

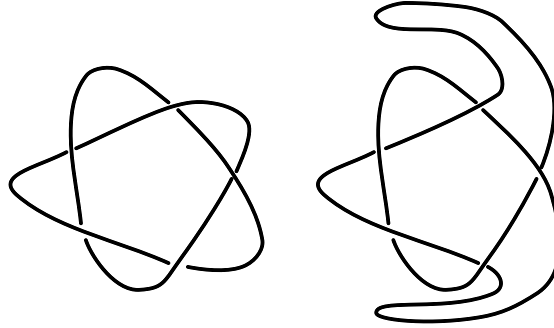


Figure 2: Here we have the knot 5_1 in two different configurations. While these knots have been stretched in different ways, they are clearly ambient isotopic and their projections are planar isotopic.

sect with itself, it is not immediately clear how this idea of 3-dimensional equivalence translates to 2-dimensional projections of the knot. In fact, it is a theorem that two knots are equivalent if and only if the projection of one knot can be transformed into that of another knot through a finite sequence of Reidemeister moves, as defined in Reidemeister's theorem below.

Reidemeister's Theorem (Chapter 1.4 [1])

Two links are ambient isotopic if and only if their diagrams can be joined by a sequence consisting of planar isotopies and the following three Reidemeister moves (see Figure 3):

I. Twist and untwist in either direction.

II. Move one strand over another.

III. Move a strand completely over or under a crossing.

Reidemeister moves represent three different changes that can be made to a knot projection which do not change the equivalence class of the knot itself. The first Rei-

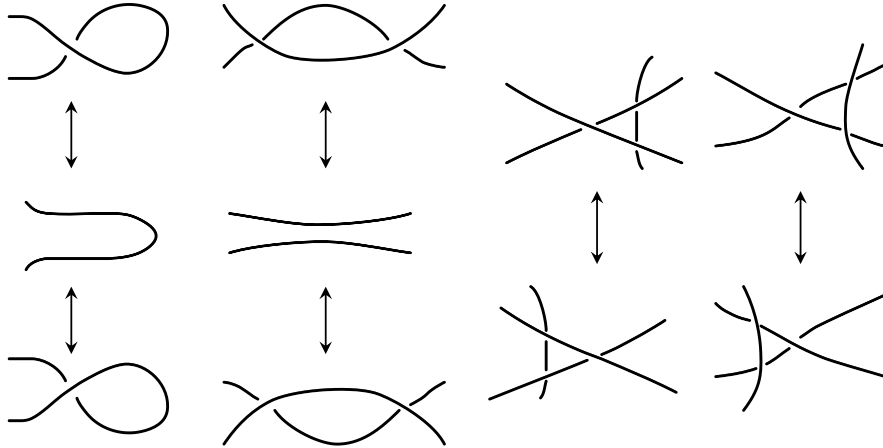


Figure 3: Move I (Left), Move II (Middle), Move III (Right)

demeister move (or Reidemeister move I) is done by twisting or untwisting the knot. This twist will create another crossing but does not change the knot. Reidemeister move II is done by pushing one strand above or below another strand. This push can be used to create two new crossings, or remove two existing crossings. Reidemeister move III involves sliding a strand from one side of a crossing to the other side of that same crossing. This will not change the number of crossings present.

These Reidemeister moves are often referred to as twisting (move I), poking (move II), or sliding (move III) a knot, and represent the only three ways to change the projection of a knot, together with planar isotopy, without changing the knot itself. These moves are named after Kurt Reidemeister, a German mathematician, who proved in 1926 that if there are two distinct projections of the same knot, one can be transformed into the other by a sequence of these three moves and planar isotopy.

2.2 Braids, Surfaces, and Braided Surfaces

In knot theory, knots can be represented in many different ways besides using the planar projections described above. One of these ways is using *braid* closures. Braids are particularly helpful because every knot can be described as the closure of a braid. A braid is a set of n strings that are attached to a horizontal bar at the top and the

bottom and which travel monotonically downwards (see Figure 4). These strings can cross over or underneath each other, but they cannot loop back up. Another way of putting this is that each string can cross any horizontal plane only one time. We let B_n denote the set of all braids with n strands.

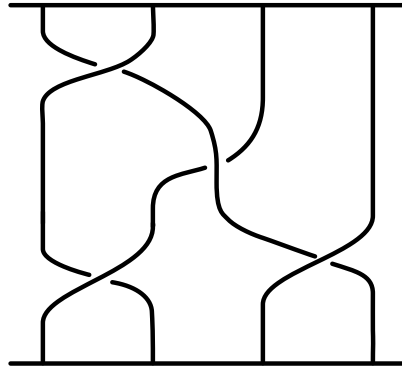


Figure 4: An example of a braid. Braids can have any number of strings and any number of crossings.

Similar to knots, there is a notion of equivalency for braids. In order to see that two braids are equivalent, we must be able to rearrange the strings of the braid without removing the strings from the top or bottom bar, and without allowing the strings to pass through each other or themselves.

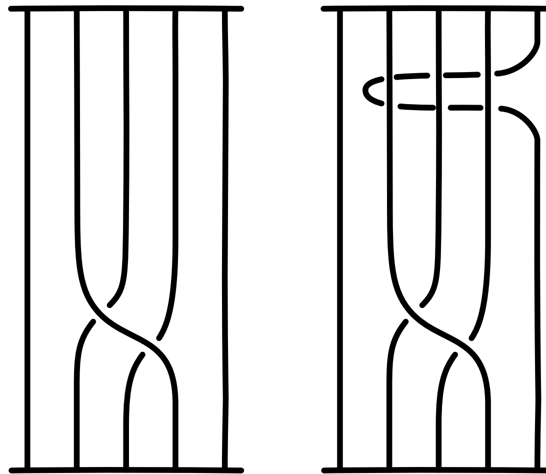


Figure 5: Even though these braids are not completely the same, since you can apply a series of Reidemeister moves to one and get the other, they are equivalent.

When we have a braid, we can turn that braid into a knot by connecting the top and bottom bars together. The resulting form is a knot or a link, and this is called the *closure of the braid* (see Figure 6).

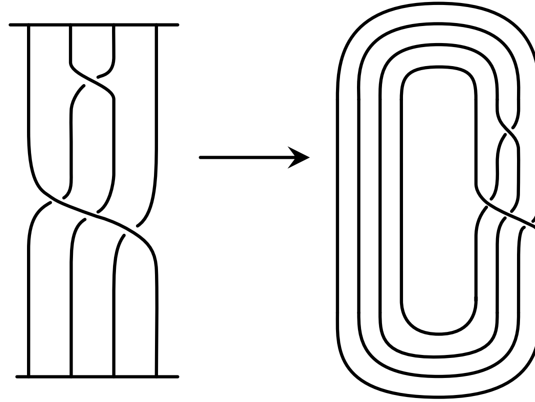


Figure 6: On the left, we have a braid. On the right, we have the closure of this braid, which is a knot.

As mentioned previously, one nice feature of braids is that every knot can be represented as the closure of a braid. This helpful fact was proven by J.W. Alexander in 1923 and is known as Alexander's Theorem.

Alexander's Theorem (Chapter 5.4 [1]

Every knot or link can be expressed as the closure of a braid.)

Simple as this theorem may be, it is incredibly helpful for working with knots. Since every knot can be represented as a braid closure, we have another useful way of studying and classifying knots. In the same vein, one useful quantity to consider when thinking of knots as braid closures is the *braid index*.

The braid index of a knot is defined to be the fewest number of strings in a braid whose closure is the knot of interest [1]. For example, the unknot (which is simply a circle) has a braid index of 1 since it can be expressed as the closure of a braid with a single strand (and no crossings).

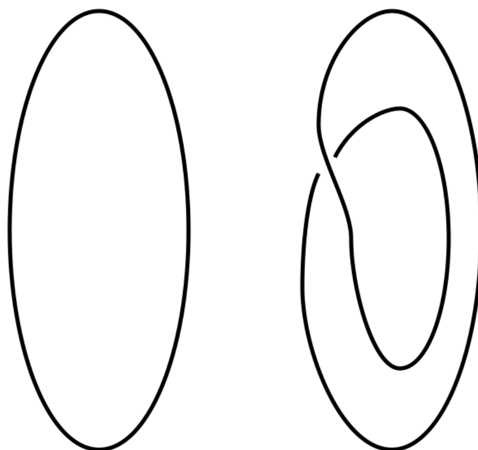


Figure 7: On the left, we have the simplest diagram of the unknot. On the right, we still have the unknot, but as the closure of a two strand braid with a single crossing. This simple example shows that different braids can have the same closure.

While calculating the braid index seems simple, it can actually be quite tricky. If we represent a knot in braid form and then count the strings of the braid, it does not guarantee that we have achieved the least number of strings possible for that knot. Counting the strings of a braid representative can certainly give us an upper bound on the braid index, but finding the actual minimal braid index requires more work.

In order to fully describe a braid, we look first at its projection. Once we have the projection of the braid—and ensure that no two crossings occur at the same height—we describe the braid by listing the strings that cross over and under other strings as we move toward the bottom bar. We always label the crossings from left to right. When the first string crosses under the second string, we call this crossing a σ_1 crossing. On the other hand, if the first string crosses over the second string, we call this crossing a σ_1^{-1} crossing. If the second string crosses under the third, it is a σ_2 crossing, and if the second string crosses over the third, it is a σ_2^{-1} crossing. This pattern continues, with a crossing of the jth strand passing under the $(j+1)^{th}$

strand being labeled as σ_j, while σ_j^{-1} denotes the jth strand passing over the $(j+1)^{th}$ strand. To describe a braid then we simply start at the top of the braid, and list off the crossings we encounter as we travel from top to bottom. We call the resulting sequence of crossings a *braid word*.

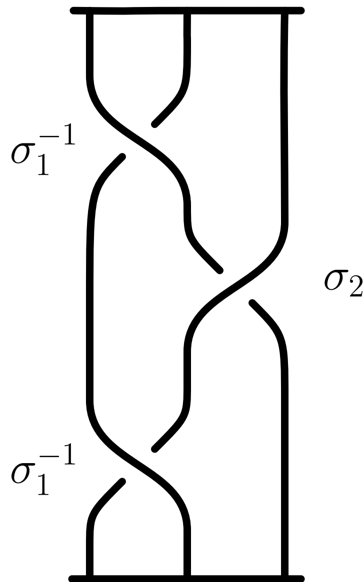


Figure 8: The braid word for this braid is $\sigma_1^{-1}\sigma_2\sigma_1^{-1}$

If we look at the braid in Figure 8, we can see that by listing the crossings from top to bottom, we obtain the braid word $\sigma_1^{-1}\sigma_2\sigma_1^{-1}$.

Along with giving a convenient way to describe the braid, there are other advantages to using braid representations of knots. One advantage is identifying which Reidemeister moves can be applied to simplify the braid. For example, if a braid word contains $\sigma_k\sigma_k^{-1}$, we know that the kth string goes under the $(k+1)^{th}$, and then immediately passes back underneath of it, returning to its original position. If we apply a simple Reidemester II move to this pair of crossings, the strings will straighten out and we are left with an equivalent braid (see Figure 9).

For a more complicated example, say we have the braid word $\sigma_1\sigma_3\sigma_2\sigma_2^{-1}\sigma_3^{-1}\sigma_4\sigma_3$. Through a series of Reidemester II moves, we can take this word and simplify it to

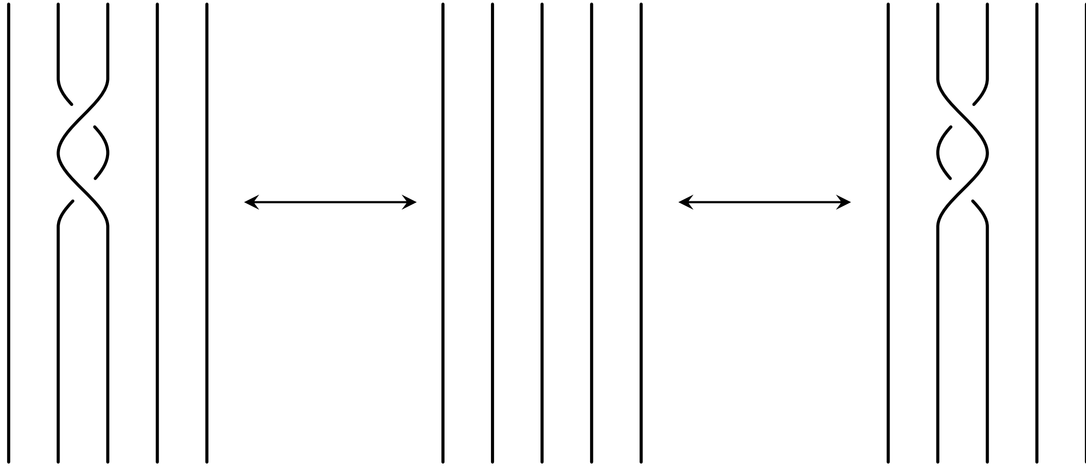


Figure 9: A Reidemester II move applied to a braid.

$\sigma_1\sigma_3\sigma_3^{-1}\sigma_4\sigma_3$, and then further to $\sigma_1\sigma_4\sigma_3$. This leaves us with a much simpler braid word which represents a braid that is equivalent to the original braid.

Another modification we can apply to braid projections is the Reidemeister III move. If we are given a braid projection and wish to move a strand over or under a crossing we are allowed to do this since a string does not need to be cut in the process. In general if you braid word contains $\sigma_i\sigma_{i+1}\sigma_i$, for $1 \leq i \leq n-2$, then it can be replaced using the substitution $\sigma_i\sigma_{i+1}\sigma_i = \sigma_{i+1}\sigma_i\sigma_{i+1}$ (see Figure 10, where the equivalent substitution $\sigma_i^{-1}\sigma_{i+1}^{-1}\sigma_i^{-1} = \sigma_{i+1}^{-1}\sigma_i^{-1}\sigma_{i+1}^{-1}$ is illustrated).

The final move that we can apply to braid projections is not a Reidemeister move; instead, it is a switch. If our braid word contains $\sigma_i\sigma_j$, where $|i-j| > 1$, then we can switch the order of σ_i and σ_j. So $\sigma_i\sigma_j$ becomes $\sigma_j\sigma_i$ (see Figure 11).

Using these three moves allows us to determine when two braids b_1 and b_2 are equivalent. This ideas leads to another very important theorem for working with braids: Markov's theorem.

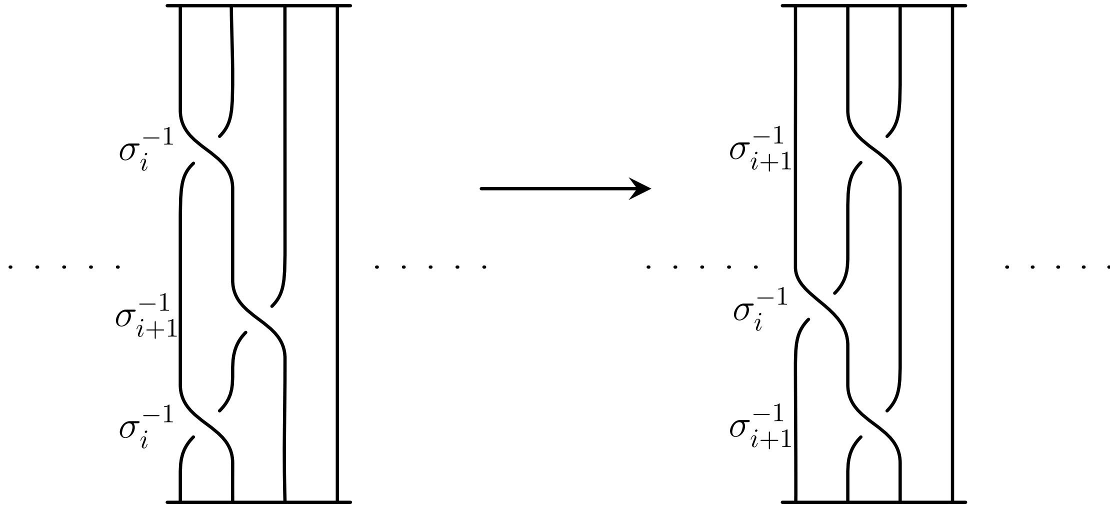


Figure 10: An example of a Reidemeister III move being applied to a general braid.

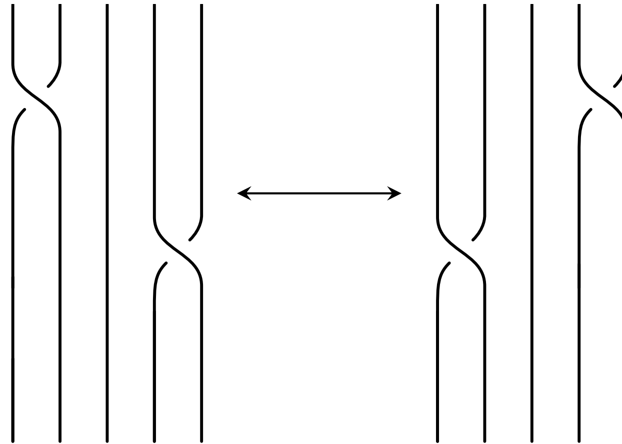


Figure 11: On the left we have $\sigma_1^{-1}\sigma_4^{-1}$. Since $|1-4| > 1$, we can switch the order to $\sigma_4^{-1}\sigma_1^{-1}$.

Markov's Theorem [2]

Given two braid words $\beta_n \in B_n$, $\beta'_m \in B_m$ with n and m strands respectively, their closures are equivalent links if and only if β'_m can be obtained from β_n by applying a sequence of the following operations:

1. conjugating β_n in B_n;
2. replacing β_n by $\beta_n \sigma_n^{\pm 1} \in B_{n+1}$;
3. the inverse of the previous operation (if $\beta_n = \beta_{n-1}\sigma_n^{\pm 1}$ with $\beta_{n-1} \in B_{n-1}$, replace β_n with β_{n-1}).

In Markov's theorem, we learn about two new moves that can be applied to braids to obtain different braids with equivalent closures. The first comes in part 1: conjugation. Conjugation is an operation applied to the braid word where the beginning of the word is multiplied by σ_j, and the end of the word by σ_j^{-1}, or vice-versa. In the closure we are only adding a Reidemeister II move, so we are not changing the resulting knot type (see Figure 12).

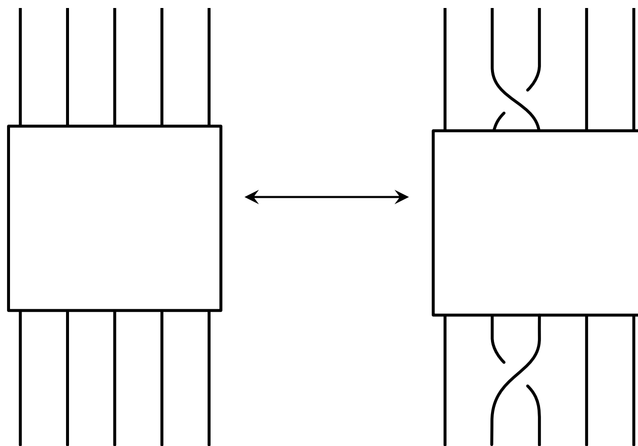


Figure 12: If we were to connect this braid into a knot and move the bottom crossing near the top crossing, we would see that it is simply a pair of crossings that can be removed by a Reidemeister II move.

The second move comes in part 2: stabilization. Stabilization involves adding a new strand and a single crossing to a braid as illustrated in Figure 13. This is performed by taking the braid word w that corresponds to an n-string braid, adding a strand to make it an $(n+1)$-string braid, and then adding σ_n or σ_n^{-1} to the beginning or end of the word w. Destabilization is the opposite of stabilization: simply remove a string and a crossing from a braid as shown in Figure 14 below.

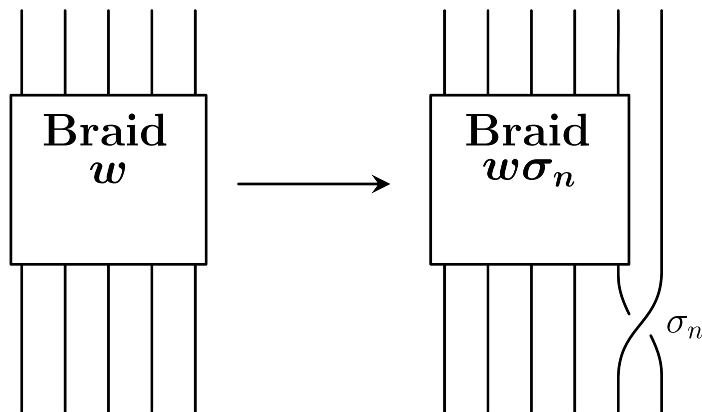


Figure 13: Through adding a single strand and crossing $\sigma_n^{\pm 1}$ at the end of the braid representation, we obtain a different braid with equivalent closure.

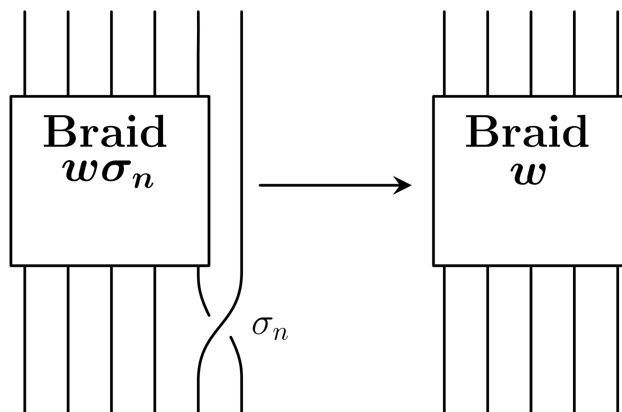


Figure 14: Through removing a single strand and crossing $\sigma_n^{\pm 1}$ at the end of the braid representation, we obtain a different braid with equivalent closure.

2.3 Slice Surfaces and Slice Genus

In the results to follow, one important idea to understand is that of a *topological surface* (or simply a *surface*). A topological surface is a two-dimensional manifold, intuitively representing a flat, rubbery sheet that can be stretched, bent, and manipulated without tearing or gluing. Surfaces can be *orientable*, which simply means you can distinguish between the 'front' and 'back' of the surface.

An important piece of information about an orientable surface is its *genus*. The genus of a surface is a fundamental topological invariant describing the shape and structure of the surface. Intuitively, it can be thought of as the number of "handles" or "holes" that a surface possesses (see Figure 15). Every orientable surface is topologically equivalent some surface with specified genus.

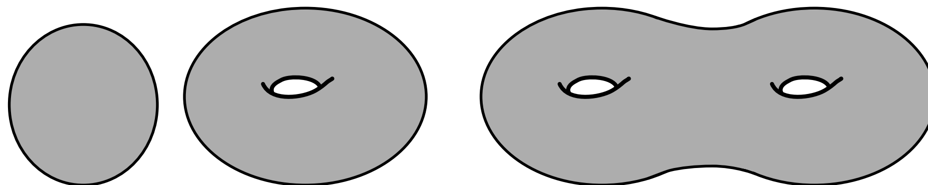


Figure 15: The left most surface has genus 0 (no holes), the middle surface has genus 1 (one hole), and the right most surface has genus 2 (two holes).

If we take an orientable surface and cut a hole in it, we then get a surface with boundary. Given a knot K, we can always find a surface with boundary whose boundary is the knot K [5]. Such a surface is called a *Seifert surface for K*. More precisely, a Seifert surface is an oriented surface associated with a knot or link on its boundary in three-dimensional space (see Figure 16).

One of the ultimate goals of representing knots in this way is to find the simplest surface possible to represent a given knot, which is not always obvious. For example, the minimal genus of a Seifert surface bounded by a certain knot may be 3, but an explicit minimal genus surface might be difficult to find. Thankfully, the Seifert genus

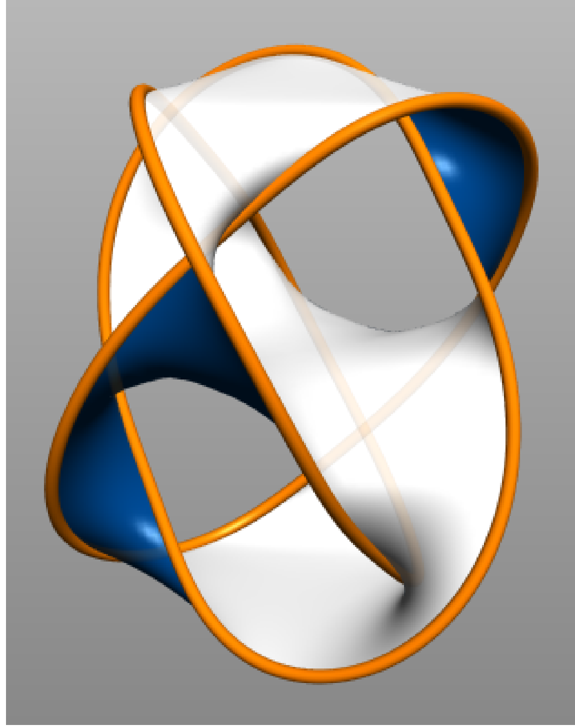


Figure 16: This is a Seifert surface. The white and blue represents the surface. You can see it is orientable because there is a clear distinction between the front and back of the surface (shown by the two colors). The orange is the knot that is on the boundary of the surface.

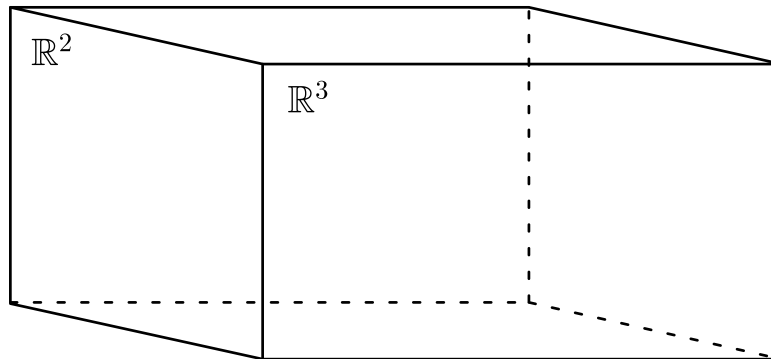


Figure 17: $\mathbb{R}^2$ represents the boundary of $\mathbb{R}^3$, which in this example is a cube.

(which is defined as the minimal genus of a Seifert surface bounded by the knot) is relatively easy to calculate.

Now suppose that you have knot that bounds a surface in $\mathbb{R}^3$ of genus 2, but would like to construct a surface that it bounds with genus 1. One possible solution is that

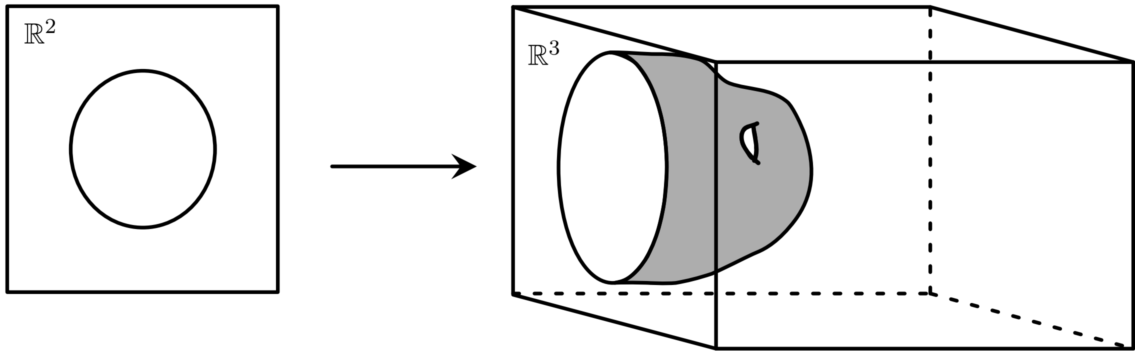


Figure 18: Here we have a knot in $\mathbb{R}^2$, with a surface in $\mathbb{R}^3_+$. Since the knot is in $\mathbb{R}^2$, there are no crossings; it's just a circle.

instead of requiring the surface to live entirely inside $\mathbb{R}^3$, we can allow it to dip into $\mathbb{R}^4$. Just as we can think of $\mathbb{R}^2$ being the boundary of $\mathbb{R}^3_+ = \{(x, y, z) \in \mathbb{R}^3 \,|\, z \geq 0\}$ (see Figure 17), we can think of $\mathbb{R}^3$ as being the boundary of $\mathbb{R}^4_+ = \{(x, y, z, t) \in \mathbb{R}^3 \,|\, t \geq 0\}$. Given a knot K in $\mathbb{R}^3$ we can therefore consider surfaces in $\mathbb{R}^4_+$ that are bounded by K. These surfaces that dip into $\mathbb{R}^4_+$ are called *slice surfaces*, and the *slice genus* of a knot is the minimal genus of any slice surface you can find for that knot. Unfortunately, the slice genus is much more difficult to calculate than the Seifert genus.

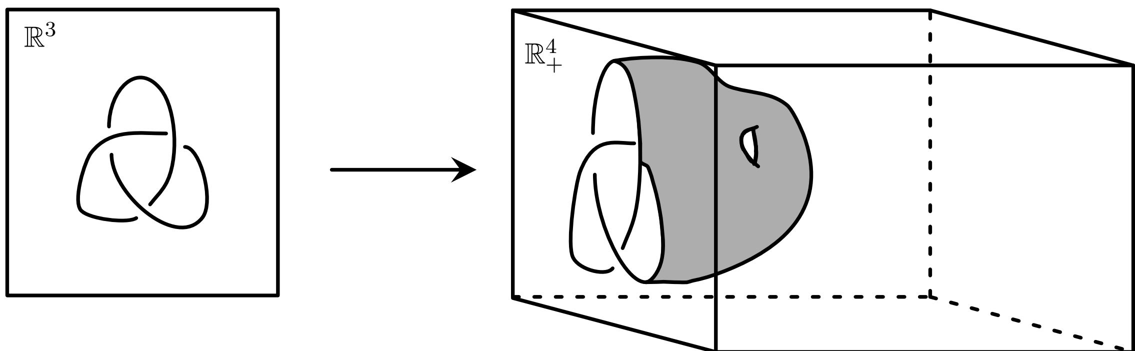


Figure 19: Here is the trefoil knot in $\mathbb{R}^3$, and a slice surface it bounds in $\mathbb{R}^4_+$

Since visualizing $\mathbb{R}^4$ is quite difficult, representing slice surfaces can be a bit of a problem. One way to overcome this is by looking at level sets. Level sets are a way of representing a slice surface in $\mathbb{R}^4_+$ by taking a 'slice' out of the surface at various

levels and seeing what the surface looks like at the location of the slice. One dimension lower, taking level sets of a surface in $\mathbb{R}^3$ yields a sequence of two-dimensional planes each containing a single slice of the surface. If you take enough of these level sets you will be able to get the full picture of the surface (see Figure 20).

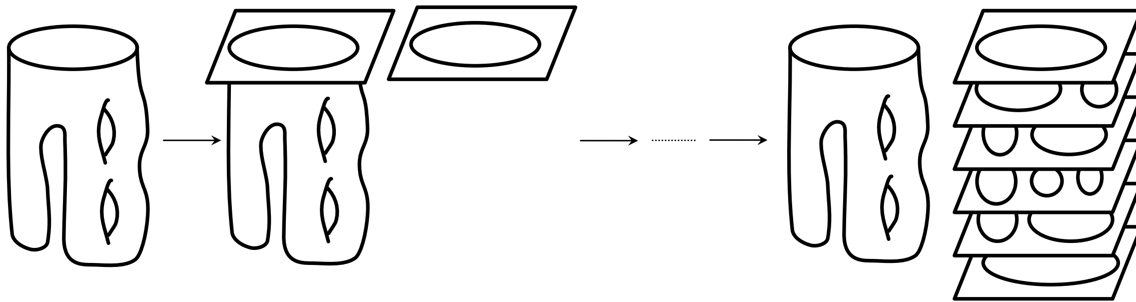


Figure 20: The left most part of this figure is the surface inside $\mathbb{R}^3$. By taking level sets of this surface, we can obtain an accurate representation of the surface.

To take level sets in $\mathbb{R}^4$ we do the exact same thing as in $\mathbb{R}^3$. The only difference is now the level sets are each 3-dimensional slices with knots inside, instead of 2-dimensional planes with planar curves.

When studying the level sets of a slice surface in $\mathbb{R}^4_+$ there are a finite number of ways the level sets may change from one picture to another. If the surface has a *saddle point* then passing the saddle point will change the level sets by bringing two nearby strands together and merging them (see the change that happens between the fourth and fifth diagrams in Figure 21 below). In this context, we define a saddle point to be a point on the surface where the curvature takes on both positive and negative values along different directions. If the surface has a local minimal point, then passing the mimimal point will result in a single circle being deleted from the level set (in Figure 21, the two circles in the final level set sit just above a pair of local maxima, and passing these maximum points results in the two circles being deleted from the level set). Likewise, if the surface has a local maximum point, then passing the maximum point results in a single circle being added to the level set.

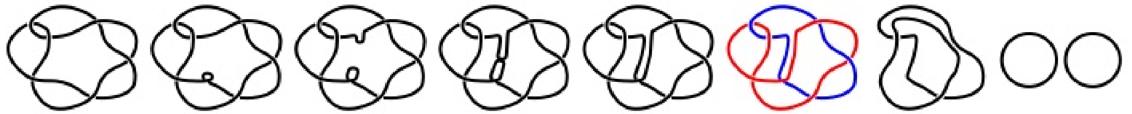


Figure 21: Here is an example of a surface being represented by cross-sections.

3 Reinforcement Learning

3.1 Introduction to Reinforcement Learning

Reinforcement learning (RL) is a branch of machine learning that allows an agent to interact with an environment that it is placed in and to learn from the results of its interactions. When the agent is placed into an environment it is given a set of actions that it is allowed to take, which is how it interacts with the environment. The goal of reinforcement learning is to train the agent in such a way that it learns to select actions that yield optimal results given whatever situation it is in. As a simple example, consider the Atari game *Breakout*. The goal of the game is to break all the bricks in the level, which is done by using the paddle to hit a ball at the bricks. Tackling this problem using reinforcement learning, the agent controls the paddle and learns to move it in the most efficient way to break the bricks.

When an agent begins training, it is passed a starting state s_0 from the environment. The agent looks at this state, thinks about what it knows (which in the beginning is nothing), and selects an action a_0. This action is then sent back to the environment, analyzed, and assigned a reward r_0 based on the effect of the action on the environment. The environment then creates the next state s_1, and sends it and the reward r_0 back to the agent. This cyclical pattern occurs until the agent achieves the goal, fails, or some pre-determined number of time steps is reached.

A useful way to model this situation is as a *Markov Decision Process* (MDP), defined as $(S, A, R, \mathbb{P}, \gamma)$, where S is the set of states, A is the set of actions, R is the distribution of rewards, $\mathbb{P}$ is the transition probabilities, and γ is the discount factor.

Figure 22: The Atari 2600 version of Breakout.

Reinforcement learning agents must learn decision making strategies not only in situations where actions create immediate rewards, but actions which impact rewards far into the future. In an MDP the current state s_t tells us everything we need to know about the environment we are working in (this is called the *Markov property*). This is beneficial because there is no risk of filling up memory, but can be detrimental because all information about the past is essentially forgotten.

The goal of the agent utilizing the MDP is to pick an action to maximize the reward. To do this, the agent uses a policy π which is a function that maps S to A, represented as $\pi : S \rightarrow A$ (in some cases it is more useful to think of A as a probability distribution across all actions, conditioned on the current state s_t). This function will pick an action based on the state the agent is in. Our hope is to find the best policy π^* that will maximize the cumulative possible reward for the agent $\sum_{t=0}^{T} \gamma^t r_t$ (here the discount factor γ is included so future rewards are not considered

as heavily as current rewards). But this is a difficult task.

Through the work of researchers many different algorithms—both classical and ones that rely on deep learning—have been created to aid in finding π^* in the most efficient way possible. These algorithms are either policy or value-based, with current research showing that policy-based methods often learn faster than value-based approaches.

There have been several successful policy-based methods presented by researchers, but one algorithm currently at the forefront of research and application which we will utilize here is Proximal Policy Optimization (PPO).

3.2 Proximal Policy Optimization

The PPO algorithm, developed by OpenAI, seeks to strike a balance between ease of implementation, sample complexity, and ease of tuning; all of which posed challenges to earlier algorithms (see [4]). PPO does this by trying to compute an update at each step that both minimizes the cost function and only slightly deviates from the previous policy. This ensures that the agent does not take too big of steps and goes off track, but does not take steps that are too small which may lead to the agent going nowhere.

In order for this to work, the algorithm utilizes two separate policy neural networks—the current policy $\pi_\theta(a_t|s_t)$, and the older policy $\pi_{\theta_k}(a_t|s_t)$—and a rather unique objective function:

$$L^{CLIP}(\theta) = \hat{E}_t\Big[min\Big(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t\Big)\Big]$$

where

- θ is the policy parameter.

- $\hat{E}_t$ is the expected value (calculated by taking the average over a sequence of actions).

- $r_t(\theta)$ is the probability ratio, or the ratio of the current policy over the older policy, $\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)}$. If $r_t(\theta) > 1$, it indicates that the new policy has a higher probability of selecting a_t than the old policy. If it is less than 1, the new policy has a lower probability.

- $\hat{A}_t$ is the estimated advantage at time step t, calculated as $\hat{A}_t = R_t - V(s_t)$, where R_t is the reward from the most recent action, and $V(s_t)$ is the estimate of return starting from current state s_t, and

- ϵ is a hyperparameter setting the size of the epsilon-neighborhood for step size.

One thing that makes this loss function interesting are the components inside of the minimization function. The first part, $r_t(\theta)\hat{A}_t$, is simply the probability ratio times the advantage. This is done to determine how much to update the policy for a specific action in a specific state as it quantifies the advantage of the action a_t taken in state s_t and its relative likelihood under the new and old policies. The second part, $clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t$, is a little bit different. If our new policy has a much higher probability of selecting a_t than our old policy, then $r_t(\theta) \gg 1$. Similarly, if our new policy has a much lower probability of selecting a_t than our old policy, we have $r_t(\theta) \ll 1$. This can be an issue because it can cause our algorithm to take a step too far in the wrong direction, potentially ruining its learning. The *clip* function ensures all of our steps stay in a specified range. More specifically, if $r_t(\theta) < 1-\epsilon$ or $r_t(\theta) > 1+\epsilon$, the new value of $r_t(\theta)$ becomes $1-\epsilon$ or $1+\epsilon$, respectively.

Once $r_t(\theta)$ and $clip(r_t(\theta), 1-\epsilon, 1+\epsilon)$ are calculated, L^{CLIP} selects the minimum of the two, and then finds the expected value of that result, thus allowing us to find the optimal step size.

Algorithm 1 PPO

for iteration= $1, 2, \ldots$ **do**
  for actor= $1, 2, \ldots, N$ **do**
    Run policy π_{θ_k} in environment for T timesteps
    Compute advantage estimates $\hat{A}_1, \ldots, \hat{A}_T$
  end for
  Optimize L^{CLIP} with respect to θ, with K epochs and minibatch size $M \leq NT$
  $\theta_k \leftarrow \theta$
end for

When the PPO runs, it starts off by running the old policy π_{θ_k}, N times for T time steps. For each $n \in N$ iterations, the advantage $\hat{A}_t$ is calculated for each $t \in T$. Once the N iterations are over, L^{CLIP} is optimized with respect to θ, and then $\pi_{\theta_k} \leftarrow \pi_\theta$.

In addition to these two policy networks, PPO also utilizes a value network. The value network is another neural network model used to estimate the expected cumulative reward (value) associated with a given state. It is used to assess the quality of states and provide feedback for policy improvement. There is only one of this network–not two networks like with the policy network.

4 Our Problem and Approach

One difficult problem in knot theory is finding the minimal genus of slice surfaces bounded by a given knot. Fortunately, the problem of constructing a minimal genus slice surface can be formulated as an MDP. That is, through a series of prescribed actions, rewards, and penalties, we can try to train an RL agent to find the minimal genus slice surface of a knot. We created an OpenAI Gym environment to allow a deep reinforcement learning agent to try and tackle this minimal slice surface problem.

In our environment, knots are represented by their braid words. For example, the knot whose braid word is $\sigma_1\sigma_3\sigma_2^{-1}\sigma_1$ is represented as $[1, 3, -2, 1]$. A cursor is instantiated on this representation that allows that agent to focus on and interact with a specific part of the braid. The 'row' cursor allows the agent to focus on a

specified crossing, while the ‘column’ cursor allows the agent to focus on a specific strand.

```
Braid word:                [ 1  3 -2  1]
Component list:            [1 2 1 1 3 4 5]
Euler characteristics:     {1: 0, 2: 1, 3: 1, 4: 1, 5: 1}
Score:                     0
Cursor:                    [0 1]
Is Terminal:               False
Action list =  {}
  ↓
 | | | |
→ /  | |
 | |  /
 |  \  |
  /  | |
 | | | |
```

Figure 23: When calling env.render(), our environment produces the following figure. This is what the braid word $\sigma_1\sigma_3\sigma_2^{-1}\sigma_1$ looks like. The cursor starts off at the first possible location for a crossing even if there is no crossing at that spot.

Before going into the moves our agent can take, we briefly describe an important topological invariant of surfaces, namely the *Euler characteristic* of a surface. The Euler characteristic of an oriented surface with single boundary component is related to the genus of the surface by the formula $g = \frac{1}{2}(1 - \chi)$, where g is the genus and χ is the Euler characteristic. In our MDP it is more convenient to track the Euler characteristic instead of the genus directly. Our goal of minimizing the genus of the surface we create therefore implies that we want to maximize the resulting Euler characteristic. The Euler characteristic is a fixed value for every surface, with the disk having Euler characteristic 1 and punctured torus having Euler characteristic -1, just to name a few. It is important to note that most surfaces have a negative Euler characteristic.

In our environment there is a set of distinct moves available for manipulating the braid structure. These moves include: 1: “Move Down” and 2: “Move Up” to move

the cursor vertically within the braid; 3: "Move Left" and 4: "Move Right" for moving the cursor laterally; 5: "Cut Strand" cuts the entire braid at a point and moves the bottom half to the top; 6: "Add Positive R2 Move" and 7: "Add Negative R2 Move" introduce Reidemeister Type II (R2) crossings pairs; 8: "Remove R2 Move" removes such crossing pairs; 9: "Apply R3 Move" to implement Reidemeister Type III (R3) transformations; 10: "Far-Commuting Move" for changing the order of distant crossings in the braid word; 11: "Add Positive Crossing" and 12: "Add Negative Crossing" to add new crossings; and 0: "Remove Crossing" to eliminate crossings. The only moves that change the Euler characteristic are adding or removing crossings (which change the Euler characteristic by -1), and any move that creates an unknotted, unlinked strand (which changes the Euler characteristic by $+1$) [3]. Thus, our goal is to create unknotted, unlinked strands in the braid, by adding and removing as few crossings as necessary.

Our state space has dimension 227. This is because all of the information about the knot is one-hot encoded. This includes the cursor location, Euler characteristics, and whether the knot has been solved or not (see Figure 23).

The reward in this environment is calculated based on several factors. First, it subtracts a small inaction penalty at each time step to provide incentive for the agent to take actions. As mentioned above, most surfaces we will obtain have a negative Euler characteristic, so the inaction penalty encourages our agent choose meaningful actions right away. The inaction penalty is a hyperparameter that we chose to be 0.1 throughout our experiments. For each action the environment computes the change in Euler characteristic of the knot component before and after the action and adds this difference to the reward function. This encourages the agent to simplify the knot, aiming for a higher Euler characteristic that indicates a more straightforward topological structure. Additionally, if the maximum number of allowed actions is reached (a hyperparameter that we changed depending on the complexity of the

knot), a final penalty (another hyperparameter, set to 350 throughout all experiments) is applied to motivate the player to reach a solution within a certain timeframe. This setup is designed to maximize the cumulative reward by maximizing the Euler characteristic while minimizing the number of actions taken.

4.1 Using PPO

The algorithm we decided to use for this problem was PPO. We made this choice for a few reasons. One major reason is the clipped surrogate objective that PPO employs to prevent large policy updates, resulting in greater stability and robustness. Another reason we decided to use PPO is because of how efficiently it can be parallelized. Throughout the training process, we trained our model on four GPUs at once. Since parallel training with PyTorch is a relatively easy task, parallelizing PPO across four GPU's enables training rates to be about four times faster than if we trained on only one GPU.

One important task in deep learning is finding the optimal hyperparameters for the model and situation at hand, and our situation is no different. For our hyperparameters, we chose

- Learning rate = 5e-3
- Epochs = 200
- Environment samples = 100
- $\gamma = 1$
- Batch size = 256
- $\varepsilon = 0.15$
- Policy epochs = 30

The learning rate of 5e-3 controls the step size for updating the policy. A smaller learning rate makes the updates more conservative, while a larger learning rate can lead to faster convergence but risks overshooting the optimal policy. Our learning rate of 5e-3 is a moderate learning rate that is a reasonable starting place. The number of training epochs (epochs) is set to 200. This is a long training time, but our task is a difficult one that requires a lot of training. Environment samples represent the number of episodes collected from the environment for each iteration, impacting policy stability.

The discount factor $\gamma = 1$ implies that we did not discount future rewards. We used this value for two reasons. One reason is that setting $\gamma = 1$ can be suitable for tasks requiring long-term planning. The second–and most import reason–is that when $\gamma < 1$ our agent learned to put off taking any actions that would result in a temporary negative reward until after those rewards had been discounted a sufficiently large amount. When we set $\gamma = 1$ the agent no longer made those unnecessary filler moves, resulting in more direct solutions.

A batch size of 256 was chosen to balance the accuracy and efficiency of policy updates. Larger batch sizes can improve the accuracy of the policy gradient estimate but may require more memory and computation. We found a batch size of 256 is a reasonable choice for balancing accuracy and efficiency. The clipping parameter $\varepsilon = 0.15$ controls policy clipping during a PPO update. As mentioned in Section 2.2, clipping helps prevent large policy updates that could destabilize training. We found that a value of $\varepsilon = 0.15$ resulted in moderate clipping, helping maintain stability during training. This differs from the standard $\varepsilon = 0.10$ value common for PPO. Finally, setting the policy update epochs to 30 allowed multiple policy updates per iteration, aiding in policy fine-tuning.

Recall that PPO uses two different neural networks: a policy and value network. For these networks, the number of parameters contained in each is 28877 and 28673,

respectively. While this is quite small compared to modern state-of-the-art deep learning models, we found it to be sufficient for our needs. Additionally, increasing the number of parameters in our models may result in running out of memory on the GPUs if we were not careful. We reduced the risk of that issue with our current parameter sizes.

In addition to finding optimal sizes for our models, we also needed to choose appropriate loss, activation, and optimization functions. Since we have two different models, we chose two different loss functions. We used mean squared error for our value function, and for our policy function we used the clipped surrogate function as described in Section 2.2. These are both typical loss functions when using PPO.

For our activation function, we used penalized tanh. Penalized tanh is an activation function mentioned in Xu et. al. [6]

$$f(x) = \begin{cases} \tanh(x) & x > 0 \\ 0.25\tanh(x) & x \leq 0. \end{cases} \tag{1}$$

This function is rather simple, but Xu et. al. found it to outperform ReLU and leaky ReLU on deep CNNs. While our model is obviously not a CNN, we found it effective in our situation.

Finally, for our optimizer, we used AdaBelief as put forth by Zhuang et. al. [7] AdaBelief is very similar to the popular Adam optimizer with a simple twist. Zhuang et. al. found that AdaBelief performed similarly to stochastic gradient descent (SGD) and Adam on tasks such as image classification and GAN training. While we are not doing image classification nor training a GAN, we found AdaBelief to contribute to the success of our model.

5 Results

Before addressing the results achieved by our agent we describe how the agent was trained. Starting off, the agent is given a range of crossing numbers to consider. The environment samples a random knot between those crossing numbers and presents it to the environment. The agent is given 500 chances to solve this knot. (Each epoch provides 100 chances for the agent, so the agent is given 5 epochs total.) If the agent solves the knot 20 times (allowing for the agent to optimize a solution it found for that knot) a new knot is sampled and the agent begins again. If the agent does not solve it 20 times in those 500 attempts, we sample and give the agent a new knot attempt. This process takes approximately 3-and-a-half hours on our GPU for each crossing number range.

We started off by training on knots with five crossing or less, then slowly increased the complexity. We found that allowing our algorithm to see and work on easier knots to solve while still being challenged by higher crossing knots aided in the success of the algorithm long term. In our experiments, our algorithm learned to construct minimal genus slice surfaces for (some, but not all) knots up to 13 crossings.

We separate our result figures below into two main parts. We have the raw training score data on the left panel. Each graph (and associated color) represents the model's progress on a different GPU. Since we trained on four GPUs at once, we have four different sets of training data. If the agent did not find a slice surface with the correct Euler characteristic for the knot it was given a reward of -350. Thus, when the agent is successful the graph contains a spike near 0, but when unsuccessful the reward does not climb above -350.

The right panel represents the exponential moving average (EMA) for the reward. The exponential moving average is a type of moving average that places more weight on recent data points, making it more responsive to recent changes in the data. The

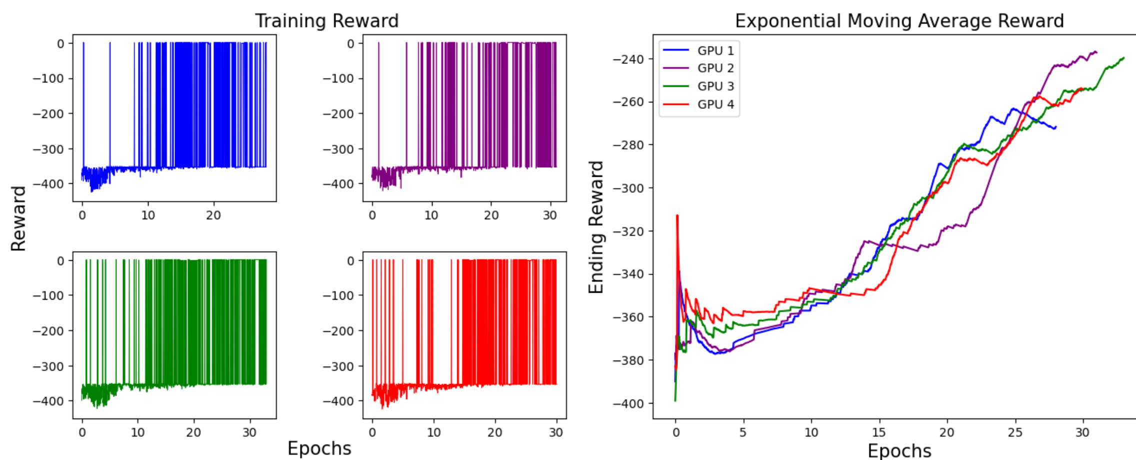
formula for the exponential moving average is calculated as:

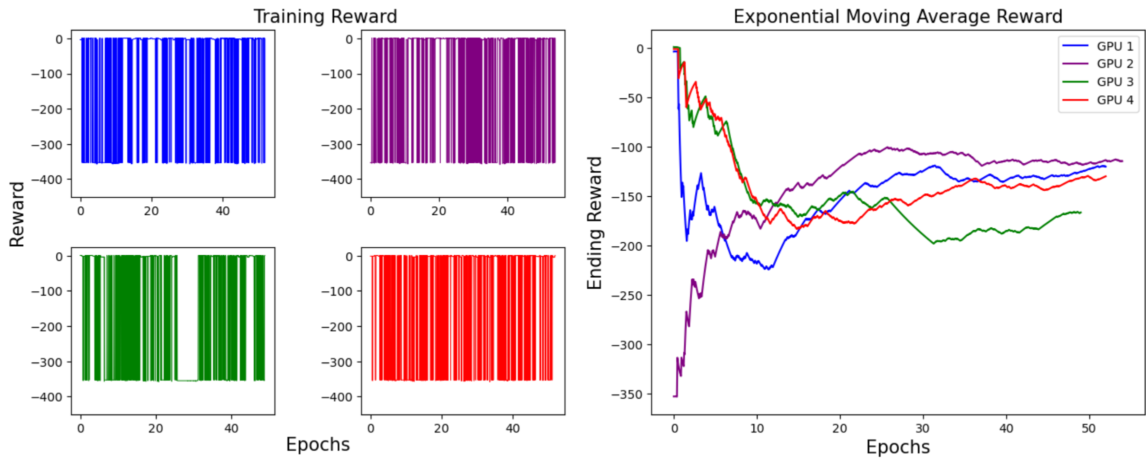
$$EMA_t = (1 - \alpha) \cdot EMA_{t-1} + \alpha \cdot X_t$$

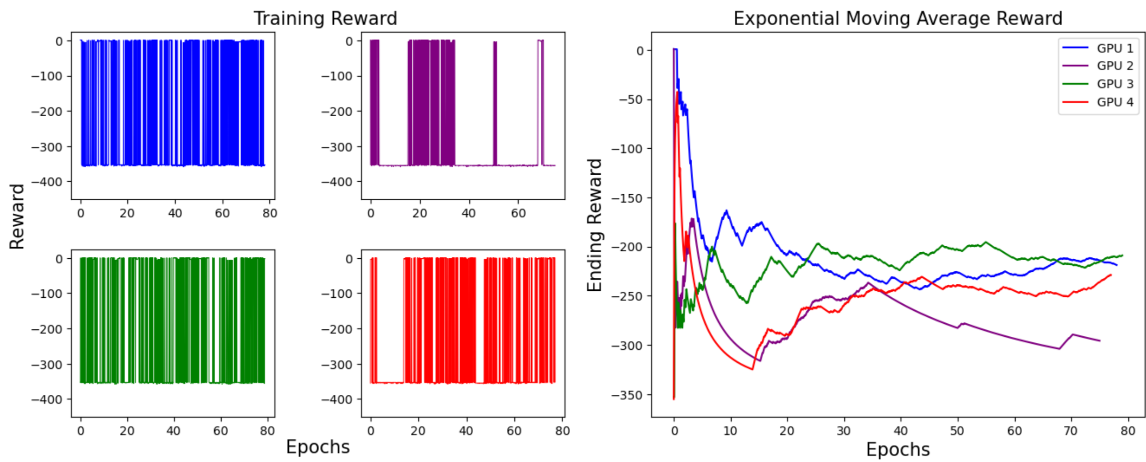
Where:

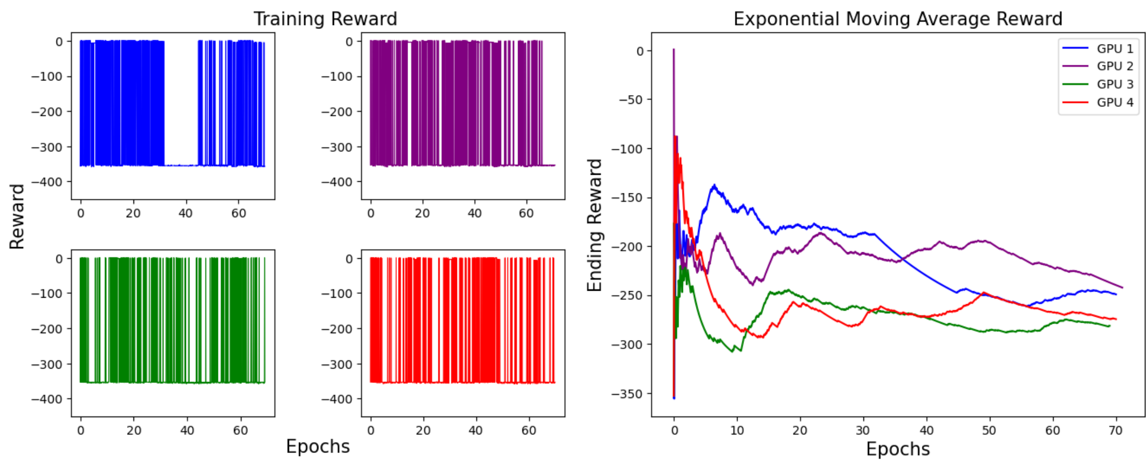
- EMA_t is the EMA at time t.
- EMA_{t-1} is the EMA at the previous time step.
- X_t is the value of the time series at time t.
- α is the smoothing factor or weight applied to the most recent data point. The smaller the α, the more weight given to older data.

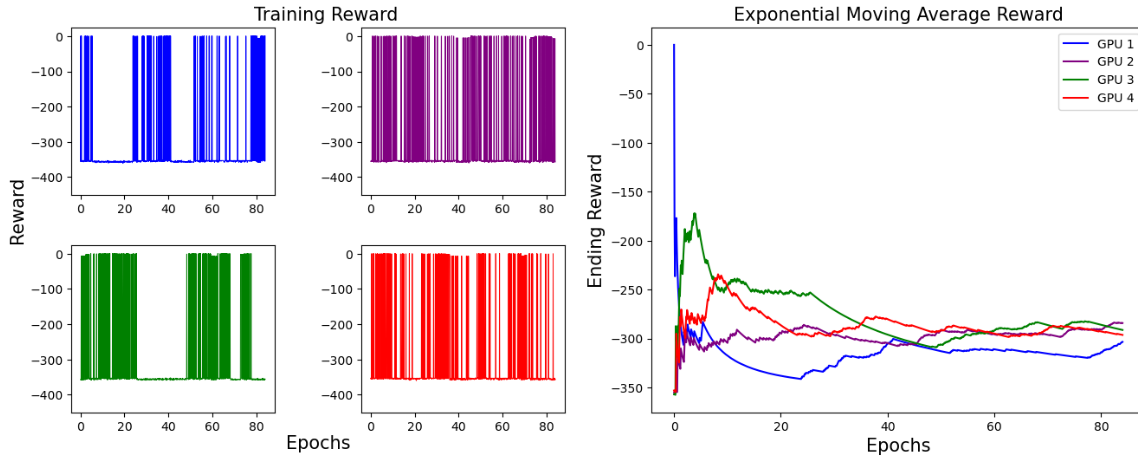
For these graphs, we used a smoothing factor of $\alpha = .01$. In the first few graphs, you can see that the EMA increases as training progresses, implying that the agent is becoming more and more successful at finding the surfaces with the correct Euler characteristic. However, as the complexity of the knots increases we see the agent struggle somewhat. This is not surprising because this simply means that the agent takes longer to solve more complicated knots.

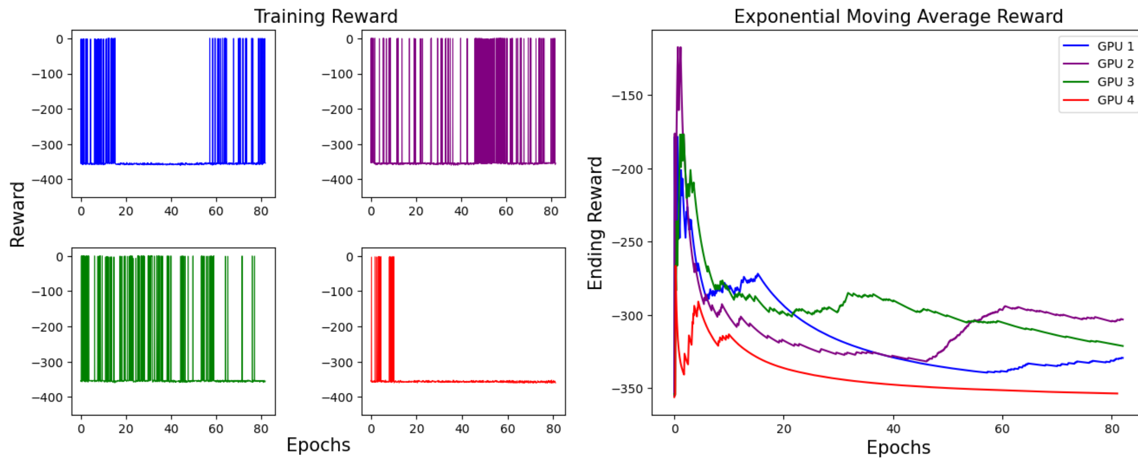


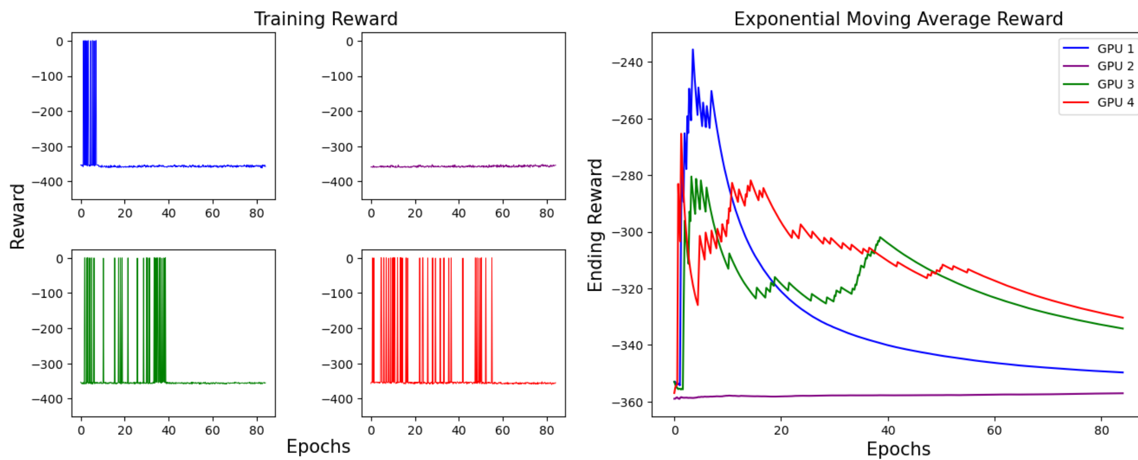

Knots Between 3 and 6 Crossings
Training Reward
Reward
Epochs
Exponential Moving Average Reward
Ending Reward
Epochs
GPU 1
GPU 2
GPU 3
GPU 4

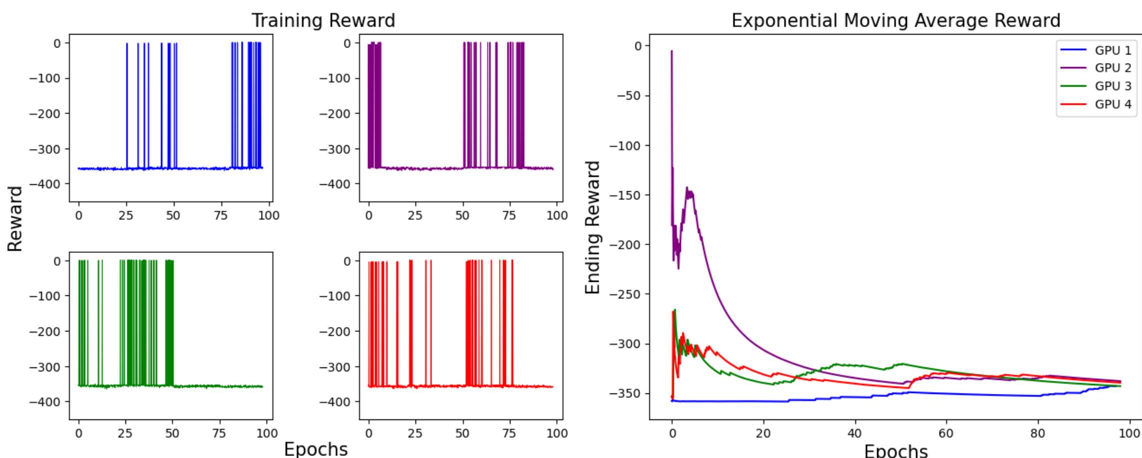

Knots Between 4 and 7 Crossings
Training Reward
Reward
Epochs
Exponential Moving Average Reward
Ending Reward
Epochs
GPU 1
GPU 2
GPU 3
GPU 4

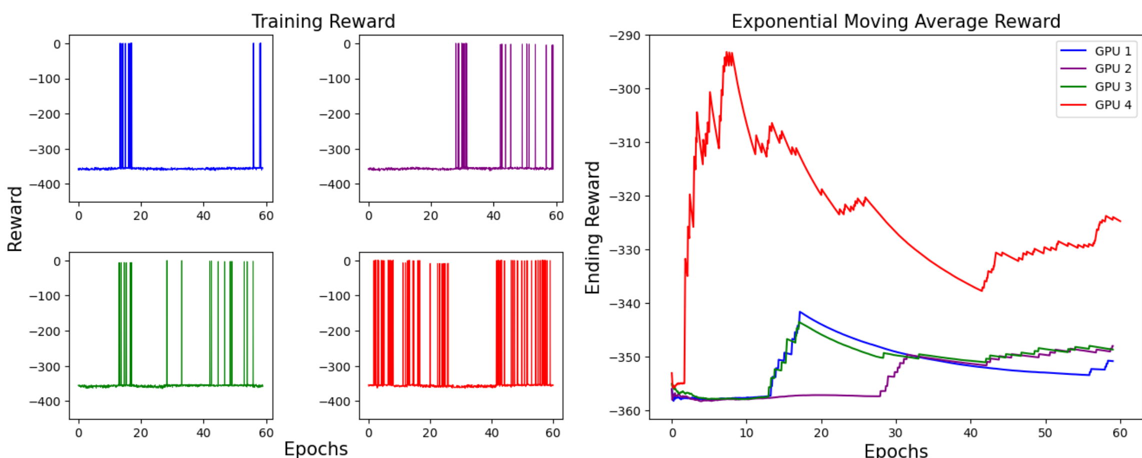

Knots Between 5 and 8 Crossings
Training Reward
Reward
Epochs
Exponential Moving Average Reward
Ending Reward
Epochs
GPU 1
GPU 2
GPU 3
GPU 4

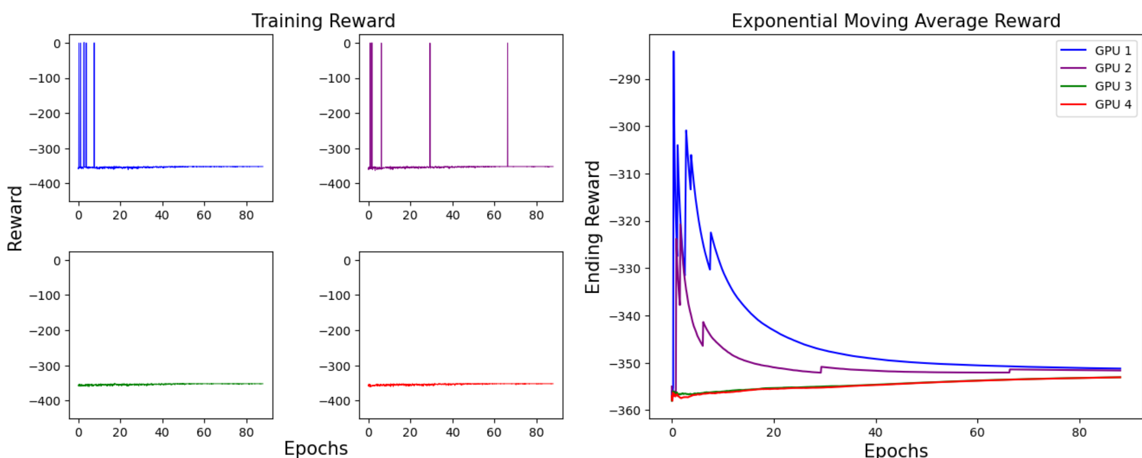

Knots Between 6 and 9 Crossings
Training Reward
Reward
Epochs
Exponential Moving Average Reward
Ending Reward
Epochs
GPU 1
GPU 2
GPU 3
GPU 4

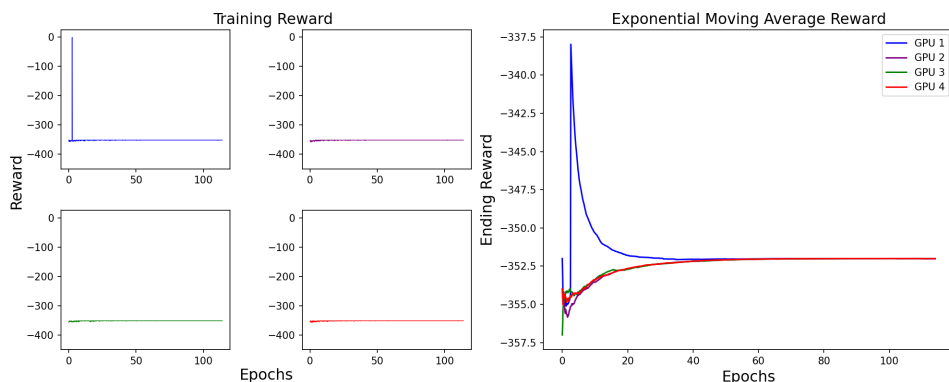

Knots Between 6 and 10 Crossings
Training Reward
Reward
Epochs
Exponential Moving Average Reward
Ending Reward
Epochs
GPU 1
GPU 2
GPU 3
GPU 4


Knots Between 7 and 11 Crossings
Training Reward
Reward
Epochs
Exponential Moving Average Reward
Ending Reward
Epochs
GPU 1
GPU 2
GPU 3
GPU 4

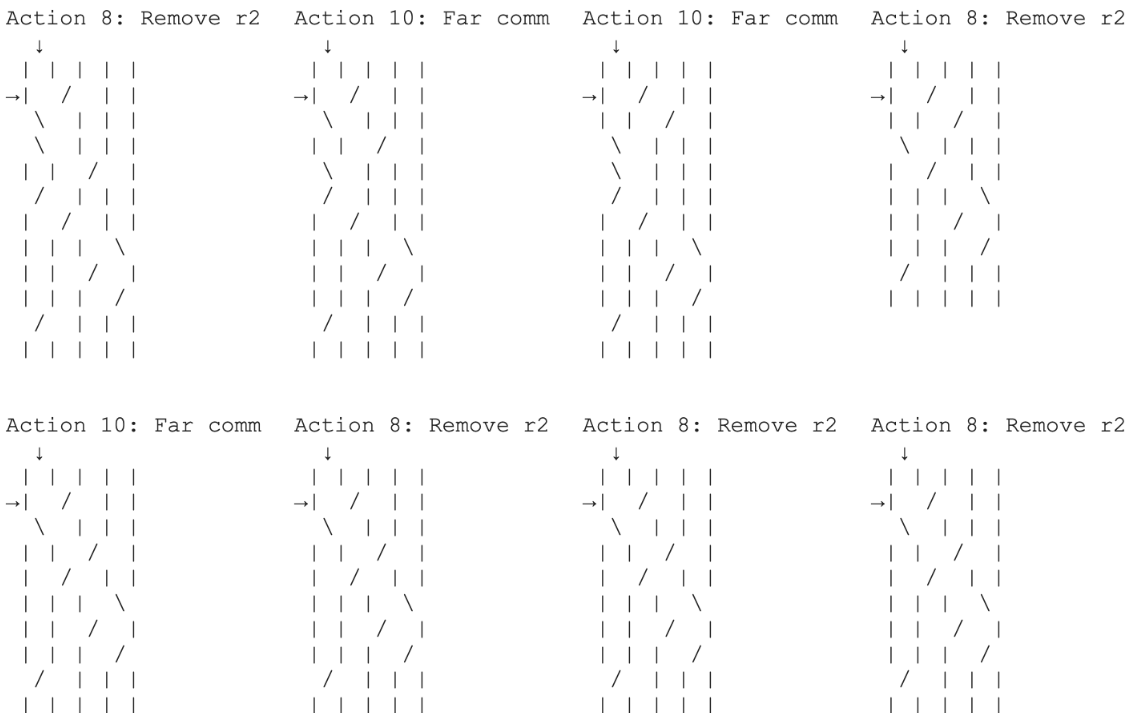

Knots Between 6 and 11 Crossings
Training Reward
Exponential Moving Average Reward
Reward
Ending Reward
Epochs
GPU 1
GPU 2
GPU 3
GPU 4


Knots Between 7 and 12 Crossings
Training Reward
Exponential Moving Average Reward
Reward
Ending Reward
Epochs
GPU 1
GPU 2
GPU 3
GPU 4


Knots Between 8 and 13 Crossings
Training Reward
Exponential Moving Average Reward
Reward
Ending Reward
Epochs
GPU 1
GPU 2
GPU 3
GPU 4



Below is an example of our agent finding a minimal genus slice surface for the 10 crossing knot with braid word $[2, -1, -1, 3, 1, 2, -4, 3, 4, 1]$. For this knot, our algorithm received a reward of 0.7 by performing the moves [8, 10, 10, 8, 10, 8, 8, 8, 0, 8, 0, 8, 8, 8, 9, 8, 8, 10, 0, 8, 8, 8, 9, 1, 8, 9, 8, 8, 8, 0]. Here is a visualization of these moves being applied to the prescribed knot.



6 Future Work

One improvement that could be made to this project in the future to improve performance is using GPUs with larger memory capacity. Our model size was ultimately

limited to the memory on the GPU, which could very well be what kept us from consistently finding minimal genus slice surfaces for knots with more than 13 crossings.

Additionally, we might not have been using the most effective way to represent knots. The use of alternative knot representation methods (other than braids) could yield more effective approaches, potentially enhancing the overall performance of our deep reinforcement learning model in uncovering the minimal slice genus of knots

With ongoing research in the Deep RL space new algorithms are being developed which outperform PPO. Exploring and incorporating these future developments will be instrumental in increasing the progress we have made for finding the minimal slice genus of knots.

References

[1] Collin C. Adams. *The Knot Book*. American Mathematical Society, 2004. ISBN: 978-0821836781.

[2] Joan S Birman. *Braids, links, and mapping class groups*. 82. Princeton University Press, 1974.

[3] Mark C Hughes. "Braiding link cobordisms and non-ribbon surfaces". In: *Algebraic & Geometric Topology* 15.6 (2016), pp. 3707–3729.

[4] John Schulman et al. "Proximal Policy Optimization Algorithms". In: *CoRR* abs/1707.06347 (2017). arXiv: 1707.06347. URL: http://arxiv.org/abs/1707.06347.

[5] Heinrich Seifert. "Über das Geschlecht von Knoten". In: *Mathematische Annalen* 110 (1935), pp. 571–592. DOI: 10.1007/BF01448044.

[6] Bing Xu, Ruitong Huang, and Mu Li. "Revise Saturated Activation Functions". In: *CoRR* abs/1602.05980 (2016). arXiv: 1602.05980. URL: http://arxiv.org/abs/1602.05980.

[7] Juntang Zhuang et al. "AdaBelief Optimizer: Adapting Stepsizes by the Belief in Observed Gradients". In: *CoRR* abs/2010.07468 (2020). arXiv: 2010.07468. URL: https://arxiv.org/abs/2010.07468.

7 Appendix

7.1 Model Architecture

As mentioned above, PPO utilizes two networks: the policy and value networks. For our work, we used simple PyTorch linear layers with our PenalizedTanH function separating them. These are fairly simple networks. The size of our model simply came down to making our networks as large as possible while still fitting them on the GPUs we were using.

Listing 1: Our Policy Network

```
class PolicyNetwork(nn.Module):
  def __init__(self, state_size, action_size):
    super().__init__()
    hidden_size = 8
    self.net = nn.Sequential(
                 nn.Linear(state_size, hidden_size*3),
                 PenalizedTanH(),
                 nn.Linear(hidden_size*3, hidden_size*4),
                 PenalizedTanH(),
                 nn.Linear(hidden_size*4, hidden_size*6),
                 PenalizedTanH(),
                 nn.Linear(hidden_size*6, hidden_size*8),
                 PenalizedTanH(),
                 nn.Linear(hidden_size*8, hidden_size*12),
                 PenalizedTanH(),
                 nn.Linear(hidden_size*12, hidden_size*8),
                 PenalizedTanH(),
                 nn.Linear(hidden_size*8, hidden_size*6),
                 PenalizedTanH(),
```

```
                nn.Linear(hidden_size*6, hidden_size*4),
                PenalizedTanH(),
                nn.Linear(hidden_size*4, hidden_size*2),
                PenalizedTanH(),
                nn.Linear(hidden_size*2, action_size),
                nn.Softmax(dim=1))

def forward(self, x):
  return self.net(x)
```

Listing 2: Our Value Network

```
class ValueNetwork(nn.Module):
  def __init__(self, state_size):
    super().__init__()
    hidden_size = 8
    self.net = nn.Sequential(
                nn.Linear(state_size, hidden_size*3),
                PenalizedTanH(),
                nn.Linear(hidden_size*3, hidden_size*4),
                PenalizedTanH(),
                nn.Linear(hidden_size*4, hidden_size*6),
                PenalizedTanH(),
                nn.Linear(hidden_size*6, hidden_size*8),
                PenalizedTanH(),
                nn.Linear(hidden_size*8, hidden_size*12),
                PenalizedTanH(),
                nn.Linear(hidden_size*12, hidden_size*8),
                PenalizedTanH(),
                nn.Linear(hidden_size*8, hidden_size*6),
                PenalizedTanH(),
                nn.Linear(hidden_size*6, hidden_size*4),
                PenalizedTanH(),
                nn.Linear(hidden_size*4, hidden_size*2),
                PenalizedTanH(),
                nn.Linear(hidden_size*2, 1))

  def forward(self, x):
    return self.net(x)
```