An Automated and Scalable Approach to Hint Generation Using Deep Reinforcement Learning

Ethan Gumabay

Charlottesville, Virginia

M.S., University of Virginia (2022)

A Thesis Presented to the Graduate Faculty

of the University of Virginia in Candidacy for the Degree of

Masters of Science

Department of Computer Science

University of Virginia

Dec, 2022

Abstract

The rise of technology in recent decades has led to an increased interest in the field of Computer Science particularly from young students. This has caused educators to include more technology-based courses in their curriculum. In some counties across the United States, programming courses are now included in the standard offering in primary, secondary, and high schools. One recurring problem for educators, particularly those teaching programming courses, is the process of guiding students in the right direction (i.e towards a solution to a problem) [34]. The faculty to student ratio in most classrooms makes it challenging for educators to constantly be present for each student and personally guide them through problems, thus creating a need for an automatic guidance system, or hint generation framework.

One of the most common methods for solving problems such as hint generation which has a defined space in which there is a discrete set of valid actions, and the solution(s) are known ahead of time is known as Reinforcement Learning [39]. In this work I aim to prove that one block-based programming framework, *Tunescope* developed at the University of Virginia [8] can be modeled as a Reinforcement Learning problem, and therefore prove programming frameworks themselves can be modeled in such a way they can be solved programmatically. I continue to solve this problem using a particular type of Reinforcement Learning called Deep Reinforcement Learning (DRL). By solving the problem using DRL, I demonstrate (1) the hint generation problem can be solved programmatically and (2) the solution itself is scalable enough to be applied to other more complex problems.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The exponential development of technology in the 21st century has given rise to a technical revolution which has completely transformed modern society. The advent of the internet, smart phones, laptops, and the like have rendered the modern world entirely unrecognizable from what it was just a quarter century ago. At the epicenter of this technological revolution sits a group of experts in STEM (Science, Technology, Engineering, and Math). The continued growth of technology, however, lies primarily in the hands of the current generation of experts, and perhaps more importantly the current generation of educators worldwide. It is up to them to excite and educate their pupils about the technological revolution, and the opportunities within in to ensure its continued prosperity.

In furtherance of these aims, researchers from the University of California Berkeley have developed a programming language called *Snap!* designed to make the field of Computer Science more accessible to students [30]. *Snap!* is a block-based programming language designed to largely black-box the syntactic subtleties associated with software development thus making it easier for novices to focus on other important programming principles like semantics and logic. Recently, a research team at the University of Virginia has adapted the *Snap!* programming language in a project called Tunescope [8] which allows elementary-aged students to use *Snap!* to create art and music.

One particularly novel element of the Tunescope framework is the hint generation

feature which students can use to help them work through problems. Hint generation allows students to receive autonomous feedback that can help them reach a correct implementation for a given problem. Presently, this is done via a reinforcement learning framework known as Q-learning developed in 1989 by Christopher Watkins [41]. Q-learning stores a large table (often referred to as a Q-table) which represents the most optimal action from each state in the state-space (i.e the best possible move from each position). The Q-function then takes the current state as input and checks the Q-table for the corresponding optimal action.

In the Tunescope framework, the state-space can be understood to be the set of all possible combinations of code blocks, and the action-space can be understood to be all possible actions a student can take (i.e adding/removing a block, changing a parameter, etc). Currently, Q-learning has proven to be sufficient in generating hints to assist students in drawing a square, though its effectiveness does not reach beyond a single use-case. This is because it was added to the Tunescope system as a proof-of-concept with no intention of full-scale autonomous hint generation.

1.1 Problem Statement

In an environment where the state and action spaces are rather small, standard Q-learning is sufficient to predict the next best action for an agent to take. In particular, when the state and action spaces can reasonably be enumerated, Q-learning is very effective. For this reason, generating hints to solve basic problems in Tunescope can reasonably leverage a standard Q-learning approach [22]. One of the weaknesses of Q-learning however, comes from its lack of scalability. A table based representation of the state and action spaces quickly becomes infeasible as the number of possible states and actions increase, thus rendering the Q-function nearly impossible to compute for large state or action spaces. One solution proposed to combat this problem is known as Deep Q-learning, which employs a neural network known as Deep Q-Network to approximate the Q-function rather than the table based representation employed by standard Q-learning [10].

Since the Tunescope framework theoretically has both an infinite state-space as well as an infinite action-space, Q-learning is entirely insufficient to compute the Q-function for all possible inputs from students. This shortcoming of Q-learning presents a need for a more robust solution to the hint generation problem in Tunescope. The discovery of such an approach would allow for the deprecation of the standard Q-learning approach currently used in favor of a more complex and scalable solution.

1.2 Research Objective

In this work I aim to prove two concepts. The first is that the Tunescope framework itself can be adapted and modeled to fit into a reinforcement learning space. Although this has already been proven for the current hint generation Q-learning approach, the environment needs to be represented in a more robust way to allow for the application of a more complex algorithm like Deep Reinforcement Learning.

The second and more important concept I aim to prove is that Deep Reinforcement Learning as a framework is a sufficient solution to the hint generation problem. Though I will constrain the problem space in a few key ways, proving Deep Reinforcement Learning can be applicable to this problem is important in laying the groundwork for later removing these constraints. In this work I focus primarily on small, simple tasks which can be accomplished in Tunescope, but argue that Deep Reinforcement Learning can be still be applied in for solving more complex problems in the future without significant refactoring; this is a property inherited from the scalability of neural networks. This is an important property of Deep Reinforcement Learning and a significant advantage over the standard Q-learning approach to hint generation.

1.3 Thesis Structure

My work is organized as follows. Chapter 2 details relevant background information for understanding my work, specifically outlining Artificial Neural Networks

and Reinforcement Learning. In chapter 3 I survey related work in the fields on Hint Generation and Deep Reinforcement Learning. Chapters 4 and 5 describe the method I employ for setting up and solving the problem as well as the results attained from applying the algorithm to solve the hint generation problem. I conclude with chapter 6 in which I describe the findings from my work and suggest some potential future explorations.

Chapter 2

Background

In this section I outline the outline the relationship between reinforcement learning and deep learning. I will first introduce the fundamental principles of Artificial Neural Networks and how they can be extended to create powerful deep neural networks. I then introduce reinforcement learning with an emphasis on a popular model-free algorithm known as Q-learning. Finally, I introduce the concept of Deep Reinforcement Learning which leverages deep neural networks to improve the performance of Q-learning over large state-action spaces.

2.1 Artificial Neural Networks

Artificial Neural Networks (ANNs), the foundation of deep learning, were originally inspired by the structure of the human brain (i.e biological neural networks). The human brain contains over 100 billion neurons. Each neuron receives a set of inputs from other neurons, it then processes the set of inputs and produces a single output (this is generally called "firing") which it sends to a set of neurons it is connected to. The artificial neuron behaves in a very similar way; receiving input from a set of other artificial neurons, processing the input, and forwarding it on to a set of other artificial neurons which it is connected to.

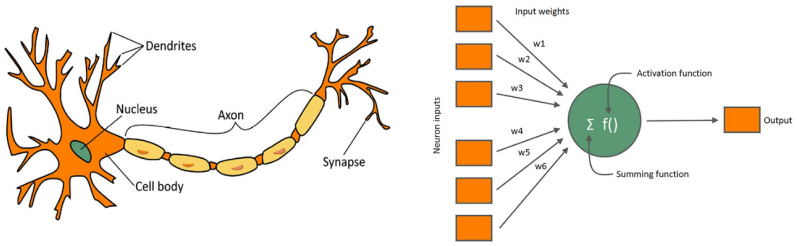

Figure 2-1: Biological vs Artificial Neuron

Figure 2-2 illustrates the similarities between a biological and an artificial neuron used in deep learning. In a biological neural network, neurons are connected via synapses which pass the signal from one neuron onto the rest of the network, whereas in artificial neural networks, these signals are passed as inputs, generally a multi-dimensional vector. In biological neural networks, neurons can receive millions of different signals and produce a wide range of signals to send to neighboring neurons in the network, each leading to indistinguishable micro-decisions. Conversely, in artificial neural networks, the processing done by each neuron can be expressed concretely as a weighted sum with some bias term passed through a nonlinear activation function. The output of an artificial neuron can thus be written in the following way.

$$y_i = f(b + \sum^{n} \theta_i x_i) \tag{2.1}$$

2.1.1 Training Process

In equation 2.1, the weight parameters θ are found through an iterative training process (the learning process) where small updates to the weights help to better approximate $\hat{y}$. The process of finding the optimal values of θ to approximate $\hat{y}$ such that it is close to y (the ground truth vector) is what is referred as the training process for neural networks.

The most basic neural network is called a multi-layer-perceptron (MLP) and consists of one of more layers of fully connected neurons. The network is said to be fully

connected since each neuron (or node) in layer i is connected to each neuron in layers $i-1$ and $i+1$. Fully connected networks will also contain an input layer which accept a set of inputs, and an output layer which returns the approximation of the known output y. Each layer which is not the input or output layer is called a hidden layer; a neural network with more than one hidden layer is said to be a *deep neural network*.

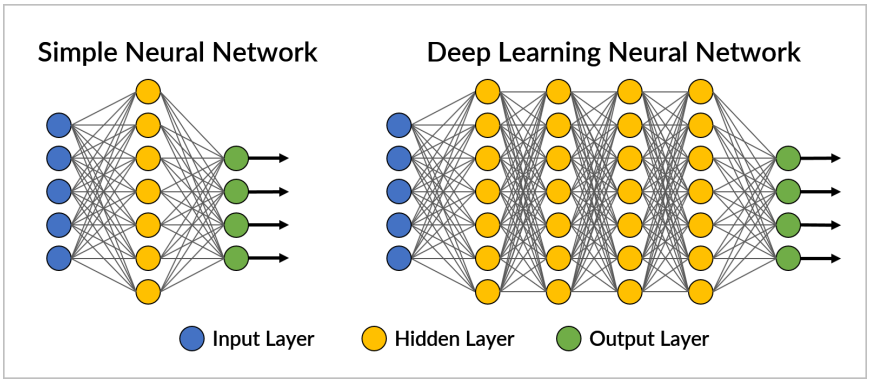

Figure 2-2: Fully Connected Neural Networks

Training a neural network can be broken down into four steps (1) Feed forward (2) Loss calculation (3) Back propagation (4) Parameter update. In the first step (Feed forward), a signal is passed from the input layer through each hidden layer and the network produces some prediction based on the signal(s) of the output layer.

In the Loss calculation step, immediately after the feed-forward step, $L(\theta)$ is calculated by comparing the predicted value $\hat{y}$ to the known value y. Though there are a variety of loss functions, I have used the Mean-Squared-Error (MSE) in my work to compute $L(\theta)$, or the vector distance between $\hat{y}$ and y. The objective is to minimize $L(\theta)$ such that it is close to 0.

$$L(\theta) = MSE(\hat{y}, y) = \frac{\sum_{i=1}^{n}(\hat{y}^T - y)^2}{n} \tag{2.2}$$

The third step, and perhaps the most pivotal step in the training of a neural network is known as the back-propagation step. The back propagation algorithm was introduced in 1995 by Rumelhart et al [32] and leverages the chain rule to perform

gradient descent which effectively computes the error contributions coming from each node in the previous layer. This can be understood to the be the partial derivative of the loss L with respect to θ_i, $\frac{\partial L}{\partial \theta_i}$. This value is then passed backwards through all hidden layers based on the error computed from the loss function immediately proceeding the feed forward step.

The final step of the training process is the parameter update of the weight matrix θ. The update process considers the gradients from the previous step and attempts to adjust each weight in θ accordingly in the opposite direction of the gradients in order to minimize the loss produced by any given node in the network. The parameter α controls the learning rate, or the amount to change the weights in θ based on the loss. A larger value of α will thus result in a more aggressive weight update, whereas a smaller update will have the inverse effect. A larger value for α may cause the weights to approach the local optimum for the loss faster, but risks overshooting this inflection point. Conversely, a smaller value for α may guarantee that the local optimum is reached, but could take an infinite amount of time. The weight matrix at each layer can be written as follows.

$$\theta_{i+1} = \theta_i + \alpha \nabla_{\theta_i} L(\theta_i) \tag{2.3}$$

Activation Functions

Activation functions are ultimately what allow neural networks to approximate nonlinear functions because they allow each node at each layer to learn non-linear transformations. Without them, the output of each layer would simply be an addition and a dot product and would prevent the network from learning any nonlinear function. This is because stacking any number of linear expressions will always result in a linear expression. There are a few different activation functions (i.e sigmoid, tanh, ReLU, softmax, etc), but for the purposes of this work I used the ReLU activation function detailed here.

$$relu(x) = max(0, x) \tag{2.4}$$

The ReLU activation function eliminates the exploding/vanishing gradient problems [27] by forcing the derivatives computed during the back propagation step to be large and constant when they are greater than 0, leading to a more meaningful weight update of θ. Figure 2-4 illustrates the ReLU activation function.

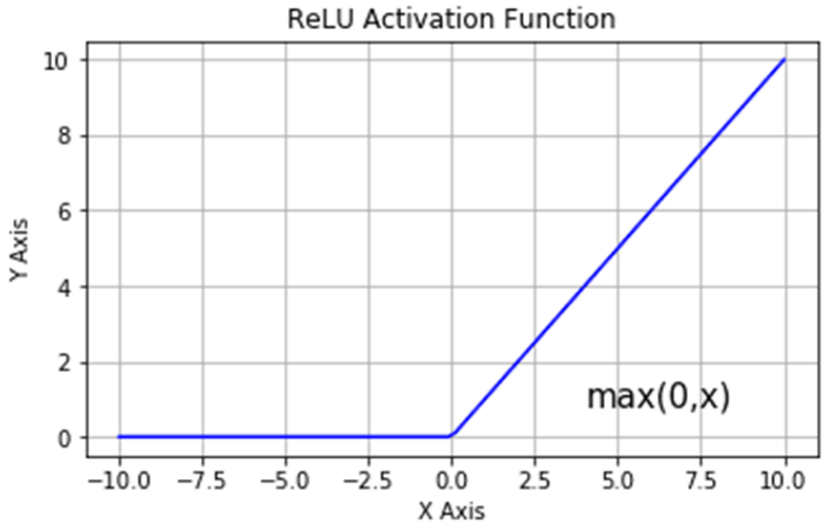

Figure 2-3: ReLU Activation Function

2.2 Reinforcement Learning

Reinforcement learning is a method used to teach an agent the optimal decision to make in an unknown environment in order to maximize some numerical reward. The objective of the agent is to make a series of decisions which alter the current state of the environment in order to solve a task. Reinforcement Learning differs from typical machine learning in that it learns from its own actions (and reward signals) rather than the analysis of data. Since the objective is to find an optimal policy, reinforcement learning is often modeled as a Markov Decision Process (MDP).

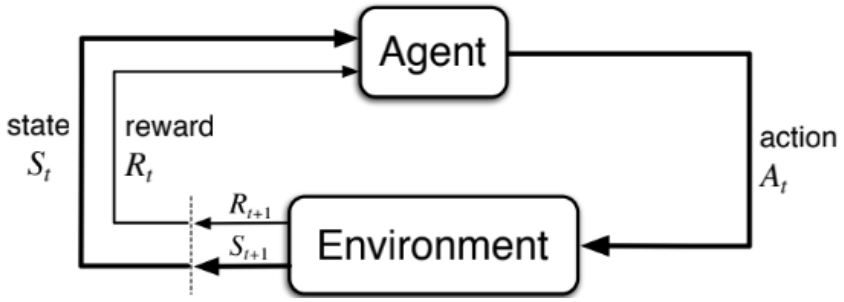

Figure 2-4: Markov Decision Process

2.2.1 Markov Decision Process

The Markov Decision Process as illustrated above is a stochastic mathematical model which can be used in the decision making process for a given state-action space. In the Markov Decision Process, an agent is given the current state S of the environment, chooses some action A_t based on its current policy moving it to the next state s_{t+1}, and receives some reward R_t for doing so. Markov Decision Processes are defined by 5 elements: $< S, A, \pi, R, \gamma >$, where

- $s_t \in S$ is the current state of the set of all states
- $a_t \in A(s_t)$ is the action chosen from all actions available from state s_t
- $P_a(s_t, s_{t+1})$ is the probability of selecting a given action a_t from the current state s_t which results in transitioning to the next state, s_{t+1}
- R_t is the expected reward from choosing action a_t from state s_t and transitioning to state s_{t+1}
- γ is the discount factor which changes the value of the reward based on importance

The goal of the Markov Decision Process is to determine the optimal set of actions (generally called the optimal policy) which maximizes the sum of (discounted) rewards from any state $s_t \in S$. An optimal policy can then be defined as $a_t = \pi(s_t)$ such that the agent receives the maximum possible discounted reward G. This can be expressed by the following equation.

$$G_t = \sum_{i=t}^{\infty} \gamma^i * R(s_i, a_i, s_{i+1}), a_i = \pi(s_i) \tag{2.5}$$

2.2.2 Exploration vs Exploitation

During the learning process, the agent can use one of two strategies at each step:

- Exploration - The agent can choose an action at random, allowing it to visit new states and potentially discover a more optimal policy.
- Exploitation - The agent can use its existing knowledge (the known policy at the current step) and choose its action accordingly.

The optimal behavior for the agent is to explore when it is not certain that its policy is optimal (i.e early on during the learning process), and exploit when it is confident that the known policy is close to the optimal policy for every state $s_t \in S$.

If the agent were to exclusively explore, it may never achieve a high reward because it may never improve its policy; conversely if the agent were to exclusively exploit its known policy, it may get stuck in its current policy and never reach the optimal one. The optimal behavior then, is some balance between exploration in exploitation where the agent sometimes chooses to explore the environment at random, and sometimes chooses to exploit its known policy.

The most common method of striking this balance is by following an epsilon greedy (ϵ-greedy) policy. In an ϵ-greedy policy, a parameter ϵ is set ($0 \leq \epsilon \leq 1$) which represents the probability with which the agent will explore its environment (i.e make a random choice). For each time the agent chooses an action, there is a probability of ϵ that its action will be random (exploratory), and a probability of $1 - \epsilon$ that it will follow its known policy (exploitative).

$$a_t = \begin{cases} \text{rand}(\text{a}_t) \text{ rand}(0,1) \leq \epsilon \\ \pi(s_t) \text{ otherwise} \end{cases}$$

Higher values of ϵ will result in the agent exploring its environment more (thus

a lower probability in acting optimally), while lower values will result in the agent following its known policy more, thereby reducing the chance the agent will explore new (potentially more optimal) policies. Generally, ϵ is chosen as a higher value at the start of the learning process (i.e $\epsilon = 0.99$), and is decreased over time; $t \to \infty, \epsilon \to 0$. The rate at which this value is decreased is called the Epsilon decay rate. This approach results in the agent exploring more at the beginning, when its policy is far from optimal, and exploiting its known policy with a higher probability as it learns more about its environment.

2.2.3 Value-Based Learning

Value-based learning is a method of learning directly from values associated with taking a particular action a_t at a given state s_t in order to maximize the reward which can be achieved from that state G_t. A value function, then $V^{\pi}(s_t)$ can then be understood to be the expected reward G_t under policy π from state s_t on-wards following the policy π. This can be defined by the Bellman equation.

$$V^{\pi}(s) = \sum_{a \in A} \pi(a|s) \sum_{s_{t+1} \in S} P(s_{t+1}|a) * [R(s, a, s_{t+1}) * \gamma(V^{\pi}(s_{t+1}))] \tag{2.6}$$

It follows, then, that the value of choosing action a_t from state s_t, or $Q(S_t, a_t)$ can be expressed by the Bellman equation as the return G_t from choosing action a_t from state s_t.

$$Q^{\pi}(s, a) = \sum_{s_{t+1} \in S} P(s_{t+1}|s, a)[R(s, a, s_{t+1})] + \gamma[\sum_{a_{t+1} \in A} \pi(a_{t+1}|s_{t+1}) Q^{\pi}(s_{t+1}|a_{t+1})] \tag{2.7}$$

The objective of value-based learning, then is to learn the value of all states $s_t \in S$ for actions $a_t \in A$, thus allowing us to derive an optimal policy π^* from the Q function such that the value of state s_t is greater than or equal to the value of the same state s_t under any other policy π' for all states in S. Thus the objective of value-based learning, $\forall s \in S, \pi'$ is to find $V^{\pi *}(s)$ such that

$$V^{\pi *}(s) = V^{*}(s) \geq V^{\pi \prime}(s) \tag{2.8}$$

2.2.4 Q-Learning

Q-learning, also known as the Temporal Difference (TD) method is value-based algorithm originally introduced in 1989 by Christopher Watkins [41] which iteratively updates its value function based on the bellman equation to improve the behavior of an agent until its policy converges to a local optimum. Q-learning is particularly useful for finding an optimal policy in discrete action spaces such as gridworlds. Q-values are state action pairs Q(s,a) which represent the value of taking a particular action a_t from a state s_t stored in a structure called a Q-table. As an agent moves from state to state by taking actions, the state action pairs are updated according to the following equation.

$$Q(s,a) = Q(s,a) + \alpha(R + \gamma_{\underset{a}{max}}(Q(s',a') - Q(s,a)) \tag{2.9}$$

In equation 1, R represents the reward for taking an action a from state s, α represents the learning rate, γ represents the discount factor, and $max(Q(s_{t+1}, a_{t+1}))$ represents the action a_{t+1} which maximizes the reward from the next state s'.

The objective of Q-learning is to iteratively update the Q function parameters θ based on experience such that the mean squared error (δ) between the estimated Q-value and the actual Q-value approaches 0. The Q-table is then updated at every step with equation 2.5 until the the difference between the estimated Q-value and the real Q-value is sufficiently close to 0. The optimization process for the Q function parameters can then be understood to be as follows.

$$\theta_{i+1} = \theta_i + \theta_{\Delta i} \tag{2.10}$$

The standard Q-Learning algorithm is shown below.

Algorithm 1 Q-learning

Input:Agent Q-table Q**, n-step update** n**, exploration rate,** ϵ**, learning rate** α**, discount factor** γ
Initialize Q(s,a) to 0 $\forall$ a $\in$ A
for *each episode* **do**
  $s \leftarrow s_0$
  while s not in terminal state **do**
  Choose action a, receive reward r, move to state s'
  $a'^* \leftarrow max_a(Q(s', a))$
  $\delta \leftarrow Q(s, a) - r - Q(s', a'^*)$
  $Q \leftarrow Q - \alpha * \delta$
  $s \leftarrow s'$
 end while
end for

2.2.5 Deep Reinforcement Learning

The Q-learning algorithm described in section 2.2.4, though effective has several significant disadvantages. Namely, values need to be stored in some sort of table or data structure. This is particularly disadvantageous in terms of its applicability in the real world. Most real-world problems have vast state and action spaces which cannot be enumerated. For example, a self-driving car determining what speed to drive, what direction to swerve at to avoid an accident, etc. cannot be neatly encoded into a table.

The most popular way to overcome such drawbacks is to replace the table with a function which can approximate the best action to choose based on the current state. Though this may not be as accurate and deterministic as using a table-lookup, this is much more scalable to larger state and action spaces. Deep neural networks (as described in section 2.1) are particularly good at approximating non-linear functions to reveal hidden relationships between relevant features and optimal actions. Replacing the Q-table with a neural network to approximate optimal actions based on

the perceived environment is known as Deep Reinforcement Learning, and the neural network used to approximate the value function is called a Deep Q-Network.

Chapter 3

Related Work

My work can be best understood as the cross section between two fields; Hint Generation, and Deep Reinforcement Learning. My work will focus specifically on the Deep Reinforcement Learning aspect of this cross section, though it will inherently enhance hint generation method for the Tunescope platform. My literature is thus divided into two sections in order to understand the current state-of-the-artt hint generation techniques and deep reinforcement learning frameworks.

3.1 Hint Generation

The autonomous generation of hints to assist struggling students is not a necessary part of programming environments given that not all students will struggle with any given task. Some studies have even argued that help-seeking environments can potentially hinder students' abilities to learn [1]. Others, however, have shown that meaningful assistance can be an extremely effective learning strategy [34]. I argue that the integration of a hint generation system has the potential to greatly enhance the learning process, particularly for struggling students on whom we'd ordinarily like to focus most of our attention, and defer to education experts as to whether or not to include the hint generation in learning platforms.

Though significant work has been done in data-driven autonomous hint generation [19, 29], none of these methods employ deep reinforcement learning. Instead, most

of the work in the field of hint generation relies largely on graphically representing students' actions over time [23, 28, 38]. One common problem across almost all hint generation frameworks is the timing of hints [31]; specifically should the students be offered a hint proactively, or should the system wait until the students asks for a hint.

To address the problem presented by the both the timing of hint generation (which can be understood to be a continuous action space) as well as the innumerable state and action spaces of the Tunescope framework, I argue that Deep Reinforcement Learning is an ideal strategy for autonomous hint generation.

The innumerable state and action spaces of Tunescope and programming as a whole require a powerful hint generation approach such as Deep Reinforcement Learning. Although I will defer to future work to solve the timing of hint generation, a Deep Neural Network is a powerful tool which has been shown to have the capability of solving problems with continuous action spaces [20]; such findings could be adapted to solve the hint generation timing problem in the future.

The use of Deep Reinforcement Learning in conjunction with autonomous hint generation has not yet been explored based on the current literature, thus making the cross-section as a a part of a learning framework such as Tunescope an entirely unexplored field.

3.2 Deep Reinforcement Learning

Reinforcement learning can be loosely defined as the training of an agent in a specified in environment to take certain actions based on its current state [39]. Reinforcement learning has been proven to be successful in elementary tasks such as navigating finite action spaces such as mazes, basic card games like blackjack, and simple tasks like controlling traffic lights [3, 15, 16]. Standard reinforcement learning, however, has proven to be ineffective in environments with large or continuous state/action spaces [15]. This shortcoming in conjunction with the recent success of neural networks has resulted in the rise of a technique known as deep reinforcement learning which employs a neural network in place of the Q-table in order to combat

these issues [4].

Deep reinforcement learning has proven its ability to outperform humans at basic tasks such as simply defined board games and action spaces with high-dimension inputs [4, 24, 25], as well as its proficiency in continuous action spaces such as vehicular control and the operation of robots [2, 40]. Deep reinforcement learning is therefore a reasonable solution to the problem faced by the current implementation of hint generation in the Tunescope framework.

Since students are able to choose any configuration of code blocks, where each configuration is considered a separate state, Tunescope can be understood as an environment with an infinite state space. Similarly, a student can choose any code block to be their next state where the choice of each different code block could be considered a distinct action, creating an infinite action space. By leveraging a Deep Q-Network, the hint generation problem (or predicting the next best move for a student to make) is theoretically possible because of their ability to handle such environments.

Deep reinforcement learning has not been employed in the process of hint generation for programming environments largely because block-based software development is fairly uncommon. Most Integrated Development Environments (IDEs) instead act similar to text editors in which programmers type individual characters thus relying on natural language processing techniques for tasks like hint generation. In these environments, it would be more useful to employ a Recurrent Neural Network (RNN) or perhaps a Generative Adversarial Network (GAN) to perform text generation which can understood to be semantically similar to hint generation [5, 6, 11, 13, 14, 18, 21, 33]. The volume of research into text generation has led to successful exploration into the idea of code generation [9, 17], though these systems have not been adapted to provide hints to students and are instead published as proof-concepts.

One of the most prominent works done in the field of deep reinforcement learning was published in 2016 by DeepMind which discusses mastering the game of Go [35] using an engine called AlphaGo. The problem space explored in this paper has several similarities to the problem space I explore here. In particular, the game of Go features

complex state and action spaces which, though finite, are so vast they are regarded as infinite and thus impossible to enumerate in a Q-table. The AlphaGo framework employs a deep neural network and tree search algorithm to predict what the next best move is for a player to make that maximizes their probability of winning the game. In 2016, AlphaGo played against the best human Go player in the world and was able to win 4 of the 5 matches, marking the first time an engine was able to surpass human proficiency in the game of Go.

The core principles of this introduced by the AlphaGo are likely to serve as the foundation for the future of artificial intelligence, however the implementation of the engine is proprietary and cannot be directly applied to my work here. Moreover, the engine itself was designed to master the game of go, and adapting or re-purposing the engine for the hint generation problem would be very complex and out of the scope of this work. Though the complexity and depth of the implementation of AlphaGo renders it out of scope for this work, the fundamental principles that it introduces can be useful to us in solving the hint generation problem.

The Tunescope framework, unlike most IDEs is a block-based programming environment and thus does not rely on the same principles as natural language text generation techniques. As a result, there are no published techniques which detail hint generation for block-based programming environments. Since code blocks simplify the state and action spaces (relative to natural language spaces), the use of deep reinforcement learning will be sufficient to understand the current state of the environment and predict the best action to take given the current state. This process can be understood as provide meaningful feedback (hints) to students, ultimately enhancing the overall learning process for novice programmers.

Chapter 4

Method

In this section, I begin by describing the ways I have constrained the problem to make it feasible for a proof of concept in section 4.1. I then continue by describing my approach to solving the problem using deep reinforcement learning in section 4.2, and finally introduce the environment I use to represent the Tunescope framework as a reinforcement learning problem in section 4.3.

4.1 Constraints

To demonstrate the effectiveness of my solution, I have constrained the problem in several ways. I have constrained it purely to prove that the the employment of deep reinforcement learning is sufficient to solve the hint generation problem, and note that removing these constraints to scale the solution would not be tremendously difficult. The constraints I've set to frame the problem are enumerated below.

1. The goal states will be a finite and static set of solutions; I have chosen 4 different solutions and encoded them concretely them as goal states.

2. The problem space will contain only 5 different blocks; these blocks are pen-up, pen-down, move (always forward), turn (cardinal directions), and a loop block.

3. When a student inserts a code block, the parameter associated with that block (i.e how many degrees to turn) is restricted to a binary choice (either correct

or incorrect)

4. The agent is only given a finite number of steps; I have chosen to give the agent 100 steps to reach the goal state.

My rationale for each of these constraints is as follows. The first constraint on the goal states allows there to be a particular goal state or set of goal states the agent is approaching. Since I are simulating an environment in which students can write code, it is possible to imagine an infinite number of possible goal states that could still result in a correct solution. One approach could have been to make an API call to Tunescope on every iteration to determine correctness, however that would have significantly increased the training time required as well as the applicability of the solution at scale. Additionally, the agent can approach this goal state using a standard reward function (i.e -1 per step and +20 for reaching the goal state). The alternative to this would have been to use some sort of distance function to calculate the distance the current position is from the goal state. Though this is possible, and perhaps would result in faster convergence, I demonstrate that a naïve approach of concretely encoded goal states still results in convergence.

The second constraint I choose allows for a significantly reduced action space while maintaining a relatively complex state space. Although there are only five distinct code blocks, it is possible to achieve many different goal states. Although I have chosen to make the set of goal states a correctly drawn square, it is trivial to draw the parallels between solving this problem and other similar problems. This reduces the problem of adapting the solution I have devised here to solve other problems to the redefinition of a finite list of goal states and potentially increasing the number of actions the agent can choose from.

The third constraint I have placed on this problem is on the correctness of the parameter associated with any given block. In Tunescope, block parameters are a continuous space, this means students are able to choose any real number in the parameter space for any given block - for example turn 4000 degrees, or loop for $1,000,000$ iterations. I have removed this for simplicity sake and instead only allowed

students to choose between correct (in the goal state) and incorrect (not in the goal state) for their parameter value. The parameter is considered correct if it appears anywhere in the goal state with the associated code block.

The fourth and final constraint I have placed on the problem is on the number of steps the agent can take in a given episode. This constraint is placed on all reinforcement learning problems since allowing the agent to take an infinite number of steps will necessarily result in convergence. Proving that an agent can converge on an optimal policy with a constrained number of steps per episode is sufficient for a proof of concept of convergence. I have chosen to constrain the agent to 100 steps for two reasons. The first is that the goal states are not overly complex, so the optimal policy should be found rather quickly. The second is that the agent is meant to simulate a student as they write code in the Tunescope framework. Typically, we would expect a student to solve this problem in fewer than 100 steps; it follows, then, that the agent should be constrained by a similar number of steps.

4.2 Approach

Ultimately the problem of hint generation can be reduced to the problem of predicting the next most optimal move from the current position. This is very similar to the way a GPS works to direct users to their destination. Based on the current position of the user, the GPS calculates the most optimal route a user could take in order to reach their destination. This parallel can be drawn by interpreting the destination of the GPS problem to be the goal state of the hint generation problem, and the optimal route of the GPS problem to be the order in which a student places blocks into the environment. In essence, the next turn of a driver using a GPS can be understoof to be the next hint to generate for students.

The GPS problem is extremely similar to a problem designed and solved by OpenAI using one of their gym environments. In particular, they propose the Taxi problem in which the taxi (agent) is taxed with locating a passenger in a gridworld, picking them up, driving to the passenger's desired destination, and dropping them off [26].

Since this problem has a relatively small state-space (the agent is constrained to an nxn gridworld) and a very small action-space (the agent is only able to move up, down, left, and right), standard Q-learning has proven to be a sufficient approach to solving the problem [12].

Adapting the taxi problem to support a more complex state-space, in particular one which is not constrained by an nxn gridworld, but rather a d-dimensional plane, creates a need to employ a Deep Reinforcement Learning framework to solve the problem. Solving the problem of arriving at a particular state in a dxd plane given a discrete set of actions is roughly the same problems as a places code blocks in a given order to reach a particular goal state in Tunescope since students are given a discrete set of actions (code blocks to place), and a goal state on a dxd plane (configuration of those code blocks which represents a the solution).

4.3 Environment

In this section, I introduce the environment I use to prove the effectiveness of using Deep Reinforcement Learning for hint generation in Tunescope. The environment I describe here is a modification of the taxi environment developed by OpenAI as a part of Gym [7]. For a full description of the taxi environment from OpenAI see Appendix A.

The taxi environment, as previously described, is an nxn plane, where a taxi (agent) is able to perform 6 actions (4 cardinal direction moves, pick up, drop off). In the environment I designed here, I instead give the agent a greater number of actions. The actions are enumerated in Appendix A. Note the parameter column is always 1 (correct) for actions which do not have parameters associated with them such as the Pen-up/Pen-down actions. Note that the action space cannot be concretely enumerated since the possibilities created by swapping two blocks will grow as the number of occupied spaces increase. The five blocks have been encoded with numerical values 0-4 corresponding to pen-up, pen-down, move, rotate, and loop respectively.

The neural network architecture I found to be most optimal for maximizing the

reward an agent sees is an input layer with 24 nodes, followed by 8 hidden layers 40, 50, 75, 75, 50, 40, 24, 16, and a final output layer of size *actions*, which for the purposes of these experiments is 8. After the two hidden layers with the greatest number of nodes I also included a dropout layer with a value of 0.2 to prevent overfitting [37]. As described in section 2.1.1, I chose a ReLU action function at each layer. The network architecture is shown in the figures below.

Layer (type)	Output Shape	Param #
dense_10 (Dense)	(None, 50, 24)	72
dense_11 (Dense)	(None, 50, 40)	1000
dense_12 (Dense)	(None, 50, 50)	2050
dense_13 (Dense)	(None, 50, 75)	3825
dropout_2 (Dropout)	(None, 50, 75)	0
dense_14 (Dense)	(None, 50, 75)	5700
dropout_3 (Dropout)	(None, 50, 75)	0
dense_15 (Dense)	(None, 50, 50)	3800
dense_16 (Dense)	(None, 50, 40)	2040
dense_17 (Dense)	(None, 50, 24)	984
dense_18 (Dense)	(None, 50, 16)	400
dense_19 (Dense)	(None, 50, 8)	136

Total params: 20,007
Trainable params: 20,007
Non-trainable params: 0

Figure 4-1: Model Summary

An episode begins with a blank board, which I have chosen to represent as a vector of dimension $dx2$ initialized to -1 to denote available/unused spots the student can place code blocks in. In practice, $0 \leq d \leq \infty$, since students could in theory use an

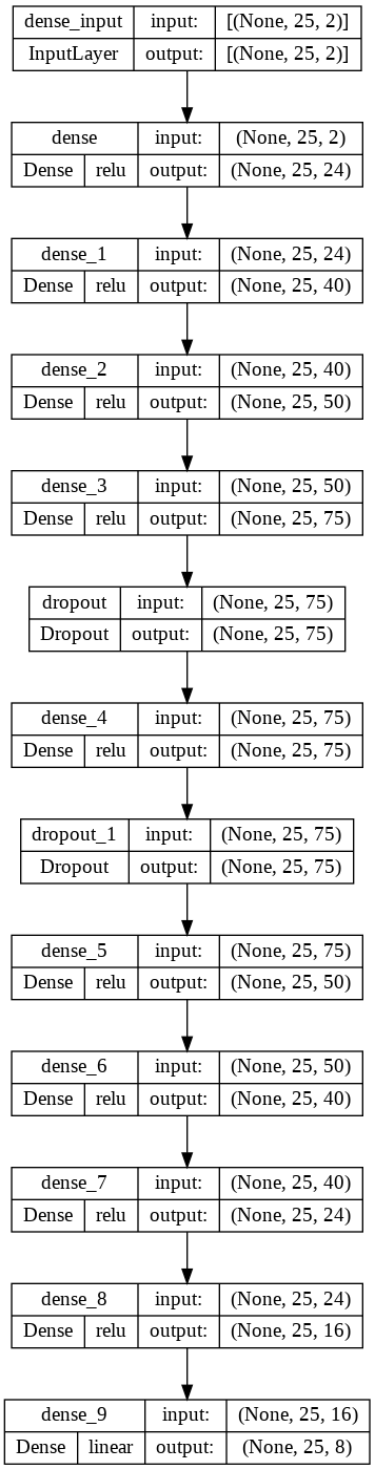

Figure 4-2: Network Architecture

infinite number of code blocks to reach the goal state, though I have constrained this number to 50 distinct code blocks for the purposes of my experiments.

On each iteration, the agent chooses an action between 0 and 8 which represents a single step in an episode. The action an agent is able to choose at each step is dependent on the state that it is currently in. This prevents the agent from doing things like attempting to remove a code block from an empty board, or attempting to insert a pen-up block when the pen is already up. After each step, a reward is given by calculating the euclidean distance from each block to the corresponding block in each of the goal states. Each block is assigned a score on every step based on which of the goal states the agent-inserted block is closest to. The algorithm for this is detailed below. For a list of actions possible from each state see Appendix A.

Algorithm 2 Tunescope Environment - Training

1: **Input:Current State** s_i**, exploration rate,** ϵ**, learning rate** α**, Deep Q-Network** Q**, goal states** G
2: Initialize all weights in $Q^*(S)$ to random values
3: **for** *each step* **do**
4: $s_i \leftarrow s_0$
5: **while** s not in goal states **do**
6: determine valid state transitions $A(s_i)$ from s_i
7: choose p at random s.t. $0 \leq p \leq 1$
8: **if** $p \leq \epsilon$ **then**
9: choose action a_i from $A(s_i)$ at random
10: **else**
11: choose action a_i from $max_a Q^*(\phi(s_i), a_i\theta)$
12: **end if**
13: take action a_i, move to state s'
14: receive reward r s.t. $\|\mathbf{r}\| = (\mathbf{s}', \mathbf{G}) * -1$
15: Update Q with a_i, s', α, r
16: $\epsilon = \epsilon * 0.95$
17: $s \leftarrow s'$
18: **end while**
19: **end for**

The above algorithm differs from the standard Deep Q-Network algorithm in two main ways. The first is on line 6; this is the line in which the valid state transitions are identified based on the current state. Although there is always some determination of valid state transitions, the particular method I employ here is designed specifically

for the Tunescope environment as seen in Appendix A. In particular, I determine the current state the agent is in by encoding the state to be one of 7 possible states. Once the state is determined, the set of actions is considered alongside the output of the model or random action chosen depending on the value of epsilon to determine which action the agent will choose at each step.

The second way the algorithm I have proposed here differs from standard Deep Q-Learning is in line 14 where the reward for each step is calculated. In my algorithm, the reward function is determined by the Euclidean distance between the next state s_{t+1} and the set of goal states G. Specifically, the minimum of the euclidean distances between the agent's current state and all goal states in order to encourage the agent to approach the nearest possible goal state. The reward is then multiplied by -1 to negatively reward the agent for moving further from the goal state(s).

Chapter 5

Experiments

In this section I enumerate four experiments I explore to determine the effectiveness of a Deep Q-Network to predict the next best code block for a student to insert in order to ultimately reach their goal of completing an assignment. Sections 5.1 and 5.2 serve as Proof-of-Concept experiments, whereas the following two sections describe two variations in which the agent is attempting to draw a square and approaching multiple goal states. These experiments demonstrates the effectiveness of leveraging a Deep Q-Network to effectively choose the next best code block to insert to eventually reach a goal. In particular, this experiment is meant to demonstrate the effectiveness of such a technique in practice in a classroom-like setting. Below are the parameters which remained constant throughout all four experiments.

Parameter	Value
Learning Rate (α)	0.001
Max Exploration Rate (ϵ_{max})	0.99
Min Exploration Rate (ϵ_{min})	0.05
Exploration Decay Rate	0.95
Training Episodes	50,000
Maximum Steps/Episode	100

Table 5.1: Experimental Constants

The parameters in the table above are further detailed here.

- Learning Rate - The rate at which the weight parameters of the neural network

are updated (see section 2.1.1)

- Max Exploration Rate - The maximum probability the agent will make a random decision
- Min Exploration Rate - The minimum probability the agent will make a random decision
- Exploration Decay Rate - The rate at which the agent decreases its probability to make a random decision (see Algorithm 2, line 16 and section 2.2.2)
- Training Episodes - The number of times the agent begins from an empty knowledge base and attempts to approach the goal state(s)
- Maximum Steps/Episode - The number of attempts the agent to reach the goal state in a given episode (see section 4.1, constraint No. 4)

I have selected the maximum/minimum exploration rate, exploration decay rate, and maximum steps/episode from widely accepted values that are commonly used in Reinforcement Learning problems. Common values for learning rate in deep learning are often factors of 10 [36]; I found the a learning rate of 0.001 to be more effective in terms of convergence speed and final configuration distance than either 0.01 or 0.0001. Lastly, I found the agent frequently converged on a distance of 0 after around 30,000 episodes, though sometimes found as high as 45,000 episodes were needed. For this reason I selected 50,000 training episodes to sufficiently train the agent past the point of convergence.

5.1 Reach Single Goal State

In this experiment, I encode a random (strictly valid) goal state for the agent to attempt to reach. The purpose of this experiment is purely a proof-of-concept to demonstrate that the Tunescope environment can be effectively modeled, and that the naïve approach to estimating the next best code block to insert is insufficient in

solving the problem. For the proof-of-concept experiment I encode a goal state which used all of the available blocks thus to encourage the agent to add blocks as much as possible rather than to converge on an empty board.

5.1.1 Results of Reaching Single Goal State

Below I have included a table with the results from training a naïve agent to converge on a single goal state.

Start Distance From Goal State	75.0
Naïve Distance Over Episode	7500
Aggregate Distance Over Episode	7500
Distance From Nearest Goal State on Final Configuration	75

Table 5.2: Reach Single Goal State

It is clear from the result of this experiment that the agent does not learn to converge on the goal state at all. In fact, the agent appears to learn that its starting state is the most optimal state, and thus converges on making no changes from where it began. The agent decides, on average that the best option for it to choose at every episode results in a distance of 75 from the goal state - the same distance from the goal state as its start state. Over the course of 100 episodes, the agent makes this same decision all 100 times. Thus this method is entirely insufficient to train an agent to reach the goal state and ultimately solving the problem of hint generation.

5.2 Reach Set of Goal States

In this experiment, I encode a set of random (strictly valid) goal states for the agent to attempt to reach. This experiment is theoretically more optimal than the experiment described in section 5.1 because the agent is now able to approach a set of goal states rather than a singular point in d-dimensional space. In this experiment I encode 2 goal states; one goal state is identical to the one described in section 5.1, and the other is a slightly more sparse goal state which is analogous to drawing 4 separate

triangles. Both goal states are designed to encourage the agent to add code blocks to the start state rather than remain close to an empty board in order to effectively observe the agent's learning patterns.

5.2.1 Results of Reaching Set of Goal States

Below I have included a table with the results from training an agent to converge on a set of goal states.

Start Distance From Goal States	75.0, 151.66
Naïve Distance Over Episode	7500
Aggregate Distance Over Episode	2317.0
Distance From Nearest Goal State on Final Configuration	0

Table 5.3: Reach Set of Goal States

From the results shown in table 5.3, it is clear that the agent does in fact choose a better action on average than the naïve approach of converging on an empty board. Here, the agent instead converges on a total distance of only 231.7 per episode, or distance of only 2.31 at every step from the goal state. The agent also converges on a distance of 0 from a goal state - this means the agent's state-space configuration at the end of the episode is exactly equal to that of a goal state. This is significantly better than the results observed in section 5.1 where the agent is on average 75 units from the nearest goal state. The results from this experiment confirm that leveraging a Deep Q-Network to predict the next code block to be inserted is an effective approach and thus sufficiently solves the hint generation problem.

5.3 Drawing Multiple Types of Squares

In this experiment, the agent is meant to approach three potential goal states. The first goal state represents the optimal way to draw a square in Tunescope. This entails inserting a loop block, followed by a pen-down block, then move/rotate blocks (with correct parameters), and a pen-up block. The second potential goal state is

the same as the first goal state, though it has 4 "no-op"s in it in which insert a pen-down and pen-up block on back-to-back code blocks. Though this does not add any functionality to the code, it increases the amount of blocks in the goal state, further encouraging the agent to insert blocks rather than to converge on an empty board as the optimally close state to minimize penalty. The third and final goal state is the "hard-coded" approach to drawing a square in which the student uses no loop blocks, and only uses repeated move and rotate blocks in order to draw a square. Though this is not an optimal way to solve the problem, this goal state simulates one way novice programmers may choose to complete the task.

5.3.1 Results of Reaching Multiple Goal States (Drawing Square)

Below I have included a table with the results from training an agent to converge on multiple goal states encoded to draw a square.

Start Distance From Goal States	16.46, 47.93, 78.76
Aggregate Distance Over Episode	784.7
Distance From Nearest Goal State on Final Configuration	0

Table 5.4: Reach Set of Goal States (Squares)

From table 5.4, the agent concludes each episode with an average distance of 78.47 from the nearest goal state. The agent also converges on a distance of 0 from a goal state - this means the agent's state-space configuration at the end of the episode is exactly equal to that of a goal state. The results of this experiment suggest that leveraging a Deep Q-Network to approach multiple realistic goal states results in the agent reaching a state which is very close to a goal state. These results further illustrate the effectiveness of the algorithm to generate hints for students learning to insert code blocks in Tunescope.

5.4 Drawing Square With Adjusted State Space

In this experiment, I use the same three goal states as described in section 5.3.1 but instead change the size of the state space. In particular, I use a state space in which the student is only able to insert a maximum of 25 blocks, and another state space where the student can use up to 100 blocks instead of the 50 blocks in the previous three experiments. This experiment is designed to demonstrate the effectiveness of the algorithm at difference size state spaces to understand the complexity of Tunescope problems which this approach can be applied to.

5.4.1 Results of Reaching Multiple Goal States (Adjusted State Space)

Below I have included a table with the results from training an agent to converge on multiple goal states with different size state spaces. Note that I compare the agent trained with a 25 and 100-block state spaces to the agent trained with 50-block state spaces on the same goal states as found above in experiment 5.3.

Start Distance From Goal States - 25 blocks	164.6
Start Distance From Goal States - 50 blocks	164.6
Start Distance From Goal States - 100 blocks	164.6
Distance From Nearest Goal State on Final Configuration - 25 blocks	0
Distance From Nearest Goal State on Final Configuration - 50 blocks	0
Distance From Nearest Goal State on Final Configuration - 100 blocks	0

Table 5.5: Reach Set of Goal States (Adjusted State Space)

The results in table 5.5 indicate that the agent actually converges on the same distance from the goal state, and reaches a final distance of 0 from a goal state in its final configuration regardless of the number of blocks available to it in the state space. In fact, it appears that the agent converges on the same number as it did in the previous experiment. This is likely because changes the state space to include on 25 blocks or as many as 100 blocks actually does not impact the encoding of the goal optimal goal state. The length of the goal states themselves remained at $5, 10, 15$

respectively, thus the blocks between the greatest length goal state (15) and the total state space (25, 50, 100) actually remained constant (at 0). The results here suggest that the total amount of states available to the student does not impact convergence as long as it remains greater than the length of the goal states.

Chapter 6

Conclusion

This work has proven two important concepts. The first is that the Tunescope environment can effectively be modeled as a reinforcement learning problem. By constraining the environment in several key ways, the Tunescope block coding environment can indeed be encoded into a gridworld-like environment. Although this encoding serves no real purpose in a vacuum, proving that it is possible to create such an abstract a mapping between the Tunescope framework and a grid-world is vital to being able to apply powerful techniques such as Deep Reinforcement Learning to solve the complex problem of hint generation in the learning environment.

The second and more relevant concept is the effectiveness of Deep Reinforcement Learning to solve the problem of hint generation in the Tunescope Framework. The empirical observations demonstrate two key components that together prove the validity of the algorithm. The first is that the algorithm approaches a realistic goal state and converges on a state which is very close to one goal state even in d-dimensional space (section 5.4). The second is the effectiveness of the algorithm at different state spaces (section 5.5). The lack of degradation in the algorithm at different size state spaces indicates the Deep Q-Networks will be a sufficiently scalable solution to apply to the Tunescope framework evan as the state space is adjusted to suit more or less complex problems.

Together these two concepts have proven the effectiveness of Deep Reinforcement Learning in training an agent to reach a desired goal state in a Tunescope-like environ-

ment. Since the process of reaching the goal state can be equated to taking a series of actions (i.e inserting or removing blocks), it is reasonable to conclude that suggesting each step the agent takes to the student would optimally guide them towards a goal state, thus effectively generating hints.

6.1 Limitations

There are a few significant limitations in solving the problem of hint generation in Tunescope which I describe here. One limitation comes from the parameter extrapolation to the binary case. This becomes particularly clear when considering cases in which two different parameters may actually be identical. For example, the current configuration considers a rotate block to have a parameter either 1 or 0 (correct or incorrect); in practice this may mean rotating 90 degrees to the right, or it could also mean rotating 270 degrees to the left. In practice, a student could choose either of these and it would be correct, but the current environment would only have the capability of capturing one of them as correct, and the other as incorrect.

Another limitation of the current approach is the concretely encoded goal states. This significantly limits the agents ability to learn a wider range of solutions for a given problem. The agent is instead constrained to learn only how to generate hints to reach a set of solutions, and will not learn other (potentially correct) solutions, and therefore not be effective in guiding students closer to a goal state if they begin to approach such a state.

The last limitation of the current approach comes from the use of Deep Reinforcement Learning to solve the problem in practice. In particular, the agent's training distribution is different from the offline testing distribution. During the training process, the agent chooses the next best state based on what will generate the best long-term reward. This is different from what will be seen in practice because the agent is only able to suggest a single step (rather than a full path). The agent therefore assumes during training that given the step it takes, a very particular set of steps will be taken thereafter (and eventually generate the maximum reward), whereas in

practice since it is only able to suggest one step, the path the student takes on future steps will not necessarily generate the maximum reward. Therefore, the suggested hint will not necessarily always be optimal.

6.2 Feasibility in Tunescope

A few of the limitations described in the previous section impact the feasibility of adapting this approach in Tunescope. Since Tunescope is intended to be used as an out-of-the-box solution for novice programmers, it is difficult to know any information ahead of time. In particular, it is difficult to have a known solution set. Although a group of instructors may be able to devise a set of known solutions, it would be very hard (even impossible) to have an exhaustive list of all solutions to a given problem. This degrades the quality of the hint generation framework itself because it would have a difficult time guiding a student to a potentially nearby goal state if is had not been encoded ahead of time.

Another similar issue arises from the type of problems students often encounter when using Tunescope. Empirical data from students' use of Tunescope in the classroom has revealed that students actually have a harder time in Tunescope when it comes to its applications to the arts (i.e Music) rather than solving problems such as drawing a square. This is problematic because with such problems there is no concrete goal state; for example if students are asked to create any valid song. In this problem, since any valid song (i.e follows a particular set of rules) would be considered correct, the goal space is truly infinite, and encoding any set of goal states may actual hinder students' creativity by guiding them to a "correct" solution due to its specificity.

In order to effectively add this solution to Tunescope in practice, then, these problems would need to be solved. Specifically, there would need to be some mechanism which is able to define a valid goal state rather than encoding them concretely before training the Deep Q-Network. A "valid" goal space could be understood to be a definition of what is considered correct for a given problem, or a rubric. This would

overcome the primary hurdle preventing this approach to hint generation from being utilized in Tunescope because educators would be able to loosely define what is considered correct, and the agent would learn how to guide students towards a valid goal state which adheres to the rubric rather than guiding them to concretely encoded states.

6.3 Future Work

Although this work has demonstrated two concepts which illustrate the effectiveness of leveraging deep reinforcement learning to generate hints, I have left several aspects to future work. One such aspect is the exploration of larger action spaces. For all of the experiments I conducted I here I used a consistent number of blocks (or actions) for the agent to choose from. Increasing the size of the action space and exploring the interaction between its size and the size of the state space would be a tremendous addition to the work done here. I also constrained the agent to only use binary parameters, and removed the ability for the agent to choose parameters along a continuous axis (i.e rotate correct, or incorrect amount instead of rotate 1-359 degrees). This would also increase the action space tremendously, but fell outside the scope of this work. Both of these are fairly straightforward adaptations to make to the work done here, but may require some re-architecting of the neural network.

The final aspect which I have deferred to future work pertains to the goal state definitions. In my experiments, the goal states must be known ahead of time in order to help the agent approach some optimal position in d-dimensional space. Though this isn't optimal since all possible goal states cannot be encoded ahead of time. Ideally, the set of goal states is dynamically computed such that the set of reachable goal states is a function of the current configuration. This is a much more challenging adaption to make to the work done here, though would make a great topic for future exploration.

Appendix A

A.1 OpenAI Taxi Environment

The environment I built is based off of the OpenAI Taxi environment written in Python using the gym framework. The gym framework is an open source Python library designed for building and testing reinforcement learning algorithms. The taxi environment was built as a grid-world where the agent can only move in four cardinal directions and the state space is an nxn grid.

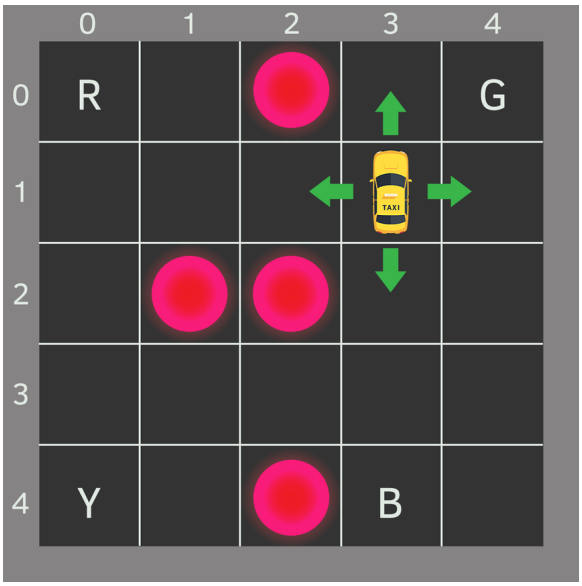

Figure A-1: Taxi Environment built with OpenAI Gym

A.2 Action Space

Encoding	Action	Parameters
1	Insert (Pen-up)	1
2	Insert (Pen-down)	1
3	Insert (Move)	1, 0
4	Insert (Rotate)	1, 0
5	Insert (Loop)	1, 0
6	Remove	block
7	Swap	block1, block2
8	Change Parameter	1

Table A.1: Action Space

A.3 Valid State Transitions

Board State	Possible Actions
Empty Board	1,2,3,4,5
1 block, no parameter	1,2,3,4,5,6
1 block, parameter	1,2,3,4,5,6,8
2 or more blocks, no parameters	1,2,3,4,5,6,7
2 or more blocks, parameter(s)	1,2,3,4,5,6,7,8
Full Board, no parameters	6,7
Full Board, parameter(s)	6,7,8

Table A.2: State Transitions

Bibliography

[1] Vincent Aleven, Bruce McLaren, Ido Roll, and Kenneth Koedinger. Toward tutoring help seeking. In *International Conference on Intelligent Tutoring Systems*, pages 227–239. Springer, 2004.

[2] Janaína R Amaral, Harald Gollinger, and Thiago A Fiorentin. Improvement of vehicle stability using reinforcement learning. In *Anais do XV Encontro Nacional de Inteligência Artificial e Computacional*, pages 240–251. SBC, 2018.

[3] Itamar Arel, Cong Liu, Tom Urbanik, and Airton G Kohls. Reinforcement learning-based multi-agent system for network traffic signal control. *IET Intelligent Transport Systems*, 4(2):128–135, 2010.

[4] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. Deep reinforcement learning: A brief survey. *IEEE Signal Processing Magazine*, 34(6):26–38, 2017.

[5] Mathias Berglund, Tapani Raiko, Mikko Honkala, Leo Kärkkäinen, Akos Vetek, and Juha T Karhunen. Bidirectional recurrent neural networks as generative models. In *Advances in Neural Information Processing Systems*, pages 856–864, 2015.

[6] Denny Britz. Understanding convolutional neural networks for nlp. *URL: http://www. wildml. com/2015/11/understanding-convolutional-neuralnetworks-for-nlp/(visited on 11/07/2015)*, 2015.

[7] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[8] Glen Bull, Joe Garofalo, and N Rich Nguyen. An introduction to computational thinking through art, music, and games. Society for Information Technology & Teacher Education, 2019.

[9] Juan Cruz-Benito, Sanjay Vishwakarma, Francisco Martin-Fernandez, and Ismael Faro. Automated source code generation and auto-completion using deep learning: Comparing and discussing current language model-related approaches. *AI*, 2(1):1–16, 2021.

[10] Jianqing Fan, Zhaoran Wang, Yuchen Xie, and Zhuoran Yang. A theoretical analysis of deep q-learning. In *Learning for Dynamics and Control*, pages 486–489. PMLR, 2020.

[11] I Goodfellow, J Pouget-Abadie, M Mirza, B Xu, D Warde-Farley, S Ozair, A Courville, and Y Bengio. Generative adversarial nets in: Advances in neural information processing systems (nips). 2014.

[12] Miyoung Han, Pierre Senellart, Stéphane Bressan, and Huayu Wu. Routing an autonomous taxi with reinforcement learning. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, pages 2421–2424, 2016.

[13] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[14] Touseef Iqbal and Shaima Qureshi. The survey: Text generation models in deep learning. *Journal of King Saud University-Computer and Information Sciences*, 2020.

[15] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996.

[16] Saqib A Kakvi. Reinforcement learning for blackjack. In *International Conference on Entertainment Computing*, pages 300–301. Springer, 2009.

[17] Duseok Kang, Euiseok Kim, Inpyo Bae, Bernhard Egger, and Soonhoi Ha. C-good: C-code generation framework for optimized on-device deep learning. In *Proceedings of the International Conference on Computer-Aided Design*, pages 1–8, 2018.

[18] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[19] Kenneth R Koedinger and Vincent Aleven. An interview reflection on "intelligent tutoring goes to school in the big city". *International Journal of Artificial Intelligence in Education*, 26(1):13–24, 2016.

[20] Kyowoon Lee, Sol-A Kim, Jaesik Choi, and Seong-Whan Lee. Deep reinforcement learning in continuous action spaces: a case study in the game of simulated curling. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 2937–2946. PMLR, 10–15 Jul 2018.

[21] Sidi Lu, Yaoming Zhu, Weinan Zhang, Jun Wang, and Yong Yu. Neural text generation: Past, present and beyond. *arXiv preprint arXiv:1803.07133*, 2018.

[22] Adam Marcus and Rich Nguyen. Development of a q-learning agent to guide student coding in snap! University of Virginia, 2021.

[23] Jessica McBroom, Irena Koprinska, and Kalina Yacef. A survey of automated programming hint generation–the hints framework. *arXiv preprint arXiv:1908.11566*, 2019.

[24] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[25] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[26] OpenAI. A toolkit for developing and comparing reinforcement learning algorithms.

[27] George Philipp, Dawn Song, and Jaime G. Carbonell. The exploding gradient problem demystified - definition, prevalence, impact, origin, tradeoffs, and solutions, 2017.

[28] Chris Piech, Mehran Sahami, Jonathan Huang, and Leonidas Guibas. Autonomously generating hints by inferring problem solving policies. In *Proceedings of the second (2015) acm conference on learning@ scale*, pages 195–204, 2015.

[29] Thomas W Price, Yihuan Dong, and Tiffany Barnes. Generating data-driven hints for open-ended programming. *International Educational Data Mining Society*, 2016.

[30] Thomas W Price, Yihuan Dong, and Dragan Lipovac. isnap: towards intelligent tutoring in novice programming environments. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on computer science education*, pages 483–488, 2017.

[31] Leena Razzaq and Neil T Heffernan. Hints: is it better to give or wait to be asked? In *International conference on intelligent tutoring systems*, pages 349–358. Springer, 2010.

[32] David E Rumelhart, Richard Durbin, Richard Golden, and Yves Chauvin. Backpropagation: The basic theory. *Backpropagation: Theory, architectures and applications*, pages 1–34, 1995.

[33] Alex Sherstinsky. Fundamentals of recurrent neural network (rnn) and long short-term memory (lstm) network. *Physica D: Nonlinear Phenomena*, 404:132306, 2020.

[34] Benjamin Shih, Kenneth R Koedinger, and Richard Scheines. A response time model for bottom-out hints as worked examples. *Handbook of educational data mining*, pages 201–212, 2011.

[35] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

[36] Leslie N. Smith. No more pesky learning rate guessing games. *CoRR*, abs/1506.01186, 2015.

[37] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958, 2014.

[38] John Stamper, Michael Eagle, Tiffany Barnes, and Marvin Croy. Experimental evaluation of automatic hint generation for a logic tutor. *International Journal of Artificial Intelligence in Education*, 22(1-2):3–17, 2013.

[39] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[40] Yoshihisa Tsurumine, Yunduan Cui, Eiji Uchibe, and Takamitsu Matsubara. Deep reinforcement learning with smooth policy update: Application to robotic cloth manipulation. *Robotics and Autonomous Systems*, 112:72–83, 2019.

[41] Christopher John Cornish Hellaby Watkins. Learning from delayed rewards. 1989.