Policy Gradient Reinforcement Learning in JAX

Sachin Pullil

May 2022

Abstract

In supervised machine learning, we need to offer a model a set of correct output values for the input training data. However, in many real world task-based applications, we may not know what the correct output value should be. We only know what task the model must accomplish. To apply concepts of machine learning to task-based problems, researchers invented reinforcement learning, the science of making decisions from experience. In this project, a deep reinforcement learning agent that employs a policy gradient algorithm was created using JAX. It was then deployed on the OpenAI FetchSlide environment to achieve the built-in task. The results of the model are analyzed and suggestions for improvement are proposed.

1 Introduction

Supervised machine learning models are excellent tools to solve problems where we can amass data on what the correct outputs are for a given set of inputs [1]. However, extending this supervised approach to real world task-based problems is not straightforward. In a task-based problem, the model will be given the initial state of the environment, and its objective will be to execute the correct tasks in sequence such that the environment reaches a predefined goal state. A supervised model does not work here because we cannot offer it the correct tasks it must execute at a given set of input environment states. This is because we either do not know which tasks must be executed at any given states to reach the goal state, or because we only know sub-optimal tasks and wish for a model that learns optimal ones.

To solve real world task-based problems using concepts of machine learning, researchers invented reinforcement learning. Reinforcement learning (RL) is the science of making optimal decisions using experiences [2]. Here, the goal is to build a model (referred to in RL as an "agent") that learns how to complete a certain objective. To achieve this, we let an untrained agent execute thousands of random actions until it completes the objective. Obviously, it will fail most of the time, but a couple dozen times, it may stumble upon sequences of actions that allow it to achieve the objective. We then reward the agent for successful completion and it learns that those sequences are the "correct" steps it must follow to achieve the goal.

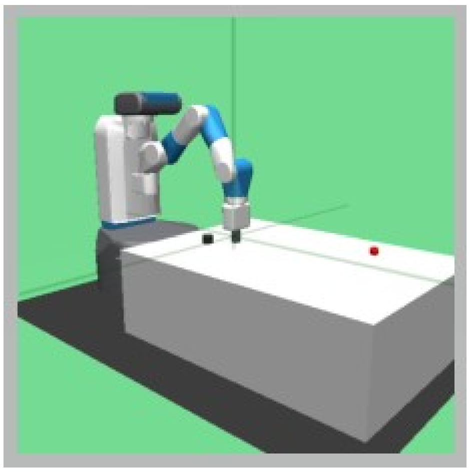


Figure 1: FetchSlide environment from OpenAI Gym.

With this experiential knowledge interpreted as training data, the agent tries to complete the objective again. We design the model such that it is more likely to follow the sequences of actions that allowed it to achieve success earlier, but still allow for randomness in task selection so that more optimal sequences can be discovered. Theoretically, this allows us to eventually train an agent that can achieve the goal every single time.

One main issue with reinforcement learning has been its data inefficiency. Typically, millions of samples are necessary for the agent to learn an effective policy. Researchers have been attempting to overcome this problem through various methods. In one example, an agent was designed that ignores its own policy when coming across states in the environment that it has observed before [10]. This is because policies are only slowly updated with each episode. However, if the model already has experience with the best outcome from a previous episode, it can automatically select that action to get the best outcome, and then update the policy accordingly after the current episode.

In another case, researchers went beyond model-based RL (where the agent knows what outcome its actions will have on the environment) and created a predictive model using video prediction techniques to train an agent [11]. Their agent was able to attain results with much greater sample efficiency than model-free or existing model-based approaches. In yet another example, researchers were able to train an agent to achieve complex, multi-dimensional tasks through the implementation of an hierarchical RL model - where a higher level model dictates a goal state for a lower level model to achieve [12]. For example, a lower level model may be in-charge of locomotion and grasping, while the higher level model will dictate the goals for the lower level model, which may be to move to a certain location, pick up an object, and drop it off at a different location. Using this method, researchers were able to create RL models that could solve complex tasks with improved data efficiency.

In this project, I implement a policy-gradient [5] deep reinforcement learning model using JAX [9] to achieve a defined objective. Specifically, my model interacts with the OpenAI Gym FetchSlide environment, depicted in figure 1. The environment contains a controllable robotic arm, a table, a grey puck that slides on the table, and a goal position on the table (shown in red). The agent has control over the robot arm and its objective is to slide the puck towards the goal position. Simulating this environment and the agent's actions, my goal is to allow the agent to learn an effective policy, i.e. a description of the "right" action given any state of the environment. Once it learns an effective policy, the agent will be able to make the right decision at any state and move closer to the completion of the objective.

2 Methodology

2.1 The environment

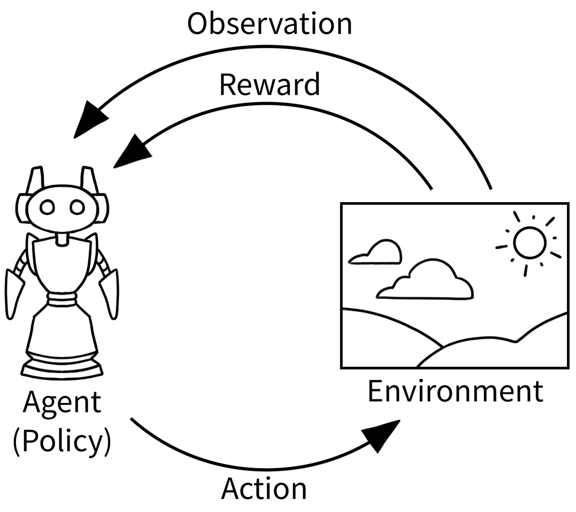
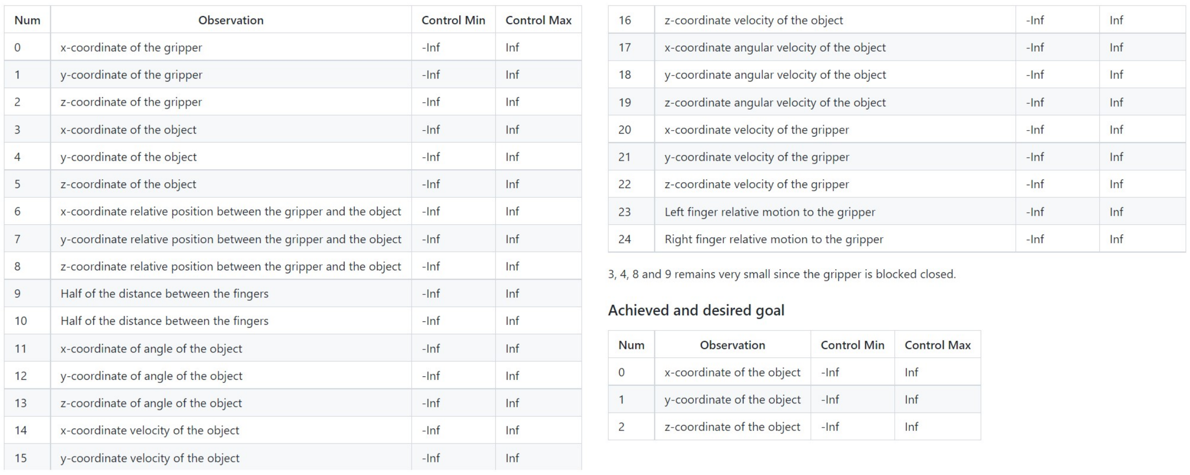


Figure 2: Agent-environment loop.

In reinforcement learning, the objective is to build an agent that performs the correct sequence of tasks to take the environment from an initial state to a goal state. The agent receives information from the environment about its current state (referred to as an "observation") and the possible actions available to the agent. Once the agent selects an action through its policy, the environment will process that action, and return the new state that came about as a result of that action. It will also return a value corresponding to whether or not the agent was able to achieve the goal. This is known as the Reward. The entire agent-environment interaction is shown in figure 2, which is referenced from the OpenAI Gym documentation [7].

In OpenAI's FetchSlide, the state of the environment consists of three elements: a set of 25 *float32* values corresponding to the observed values within the environment, a set of 3 *float32* values corresponding to the desired goal of the agent, and a set of 3 *float32* values corresponding to the achieved goal of the agent. These are detailed in figure 3.



Num	Observation	Control Min	Control Max
0	x-coordinate of the gripper	-Inf	Inf
1	y-coordinate of the gripper	-Inf	Inf
2	z-coordinate of the gripper	-Inf	Inf
3	x-coordinate of the object	-Inf	Inf
4	y-coordinate of the object	-Inf	Inf
5	z-coordinate of the object	-Inf	Inf
6	x-coordinate relative position between the gripper and the object	-Inf	Inf
7	y-coordinate relative position between the gripper and the object	-Inf	Inf
8	z-coordinate relative position between the gripper and the object	-Inf	Inf
9	Half of the distance between the fingers	-Inf	Inf
10	Half of the distance between the fingers	-Inf	Inf
11	x-coordinate of angle of the object	-Inf	Inf
12	y-coordinate of angle of the object	-Inf	Inf
13	z-coordinate of angle of the object	-Inf	Inf
14	x-coordinate velocity of the object	-Inf	Inf
15	y-coordinate velocity of the object	-Inf	Inf
16	z-coordinate velocity of the object	-Inf	Inf
17	x-coordinate angular velocity of the object	-Inf	Inf
18	y-coordinate angular velocity of the object	-Inf	Inf
19	z-coordinate angular velocity of the object	-Inf	Inf
20	x-coordinate velocity of the gripper	-Inf	Inf
21	y-coordinate velocity of the gripper	-Inf	Inf
22	z-coordinate velocity of the gripper	-Inf	Inf
23	Left finger relative motion to the gripper	-Inf	Inf
24	Right finger relative motion to the gripper	-Inf	Inf

3, 4, 8 and 9 remains very small since the gripper is blocked closed.

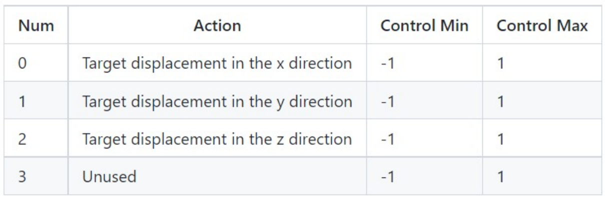
Achieved and desired goal

Num	Observation	Control Min	Control Max
0	x-coordinate of the object	-Inf	Inf
1	y-coordinate of the object	-Inf	Inf
2	z-coordinate of the object	-Inf	Inf

Figure 3: Observation space of the OpenAI FetchSlide environment referenced from the GitHub repository [6].

The achieved goal refers to the location of the puck that the agent was able to achieve with the actions it has taken so far. This information is used in advanced reinforcement learning algorithms like Hindsight Replay Experience (HER) [4] to facilitate learning by ”remembering” action sequences that led to previously achieved goals. For my policy gradient implementation of RL, this information is not necessary and is therefore discarded. Thus, the observation my agent receives from the environment can be seen as a set of 28 *float32* values.

The actions available to the agent can be represented as a set of 4 *float32* values, as shown in figure 4.



Num	Action	Control Min	Control Max
0	Target displacement in the x direction	-1	1
1	Target displacement in the y direction	-1	1
2	Target displacement in the z direction	-1	1
3	Unused	-1	1

Figure 4: Action space of the OpenAI FetchSlide environment referenced from the GitHub repository [6].

Every time the environment is reset, the location of the puck, robot, and the goal is changed. Hence, the agent must learn to achieve the objective for any initial condition. When an agent has taken an action, we say that it has taken a ”step” in the environment, and influenced the environment's current state. For each step that the agent takes without achieving the goal, it receives a reward of -1. If it achieves the goal, it receives a reward of 0. For example, if the agent clears the objective after 150 steps, it will receive a reward of -149. Its goal is to maximize its rewards by sliding the puck to the goal as fast as possible.

2.2 Policy gradient

In this project, the reinforcement learning algorithm called policy gradient is implemented. The objective of this algorithm is to learn a statistical distribution that maximizes the probability of selecting the correct action at the given state which will allow the agent to maximize its rewards. Its formal definition follows.

Let θ be the set of parameters that the policy depends on. In this case, θ is the weights and biases of the neural network. At timestep (same as "step" discussed in section 2.1) t, let the state of the environment be s_t, i.e. the 28 observed values, and the action that the agent chooses to take be a_t. Then, let the policy that depends on these parameters be represented as π_θ and be defined as:

$$a_t = \pi_\theta(s_t)$$

At time $t = 0$, the state of the environment will be s_0 and the first action the agent takes, a_0, will bring the environment to a new state, s_1. Similarly, as the agent takes further actions, the state of the environment will continue to change. At time $t = T$, the agent will have taken many actions, and the next action it takes, a_T, will bring the environment to state s_{T+1}. We define a Trajectory, τ, to be the set of states and actions that have been observed and taken, respectively, until a certain point in time.

$$\tau = s_0, a_0, s_1, a_1, s_2, ...a_T, s_{T+1}$$

The reward at the end of following a trajectory is defined as $R(\tau)$, which is simply the sum of the rewards at each time step, r_t. In the policy gradient algorithm, we strive to maximize the expected reward for following a trajectory, τ, under a given policy, π, with respect to the parameters θ. That is, we attempt to maximize:

$$J(\pi_\theta) = E_{\tau\sim\pi_\theta}[R(\tau)]$$

To do so, we utilize the conditional probability that we get a certain trajectory given the set of parameters, i.e. $P(\tau|\theta)$.

$$\begin{aligned}
\nabla_\theta J(\pi_\theta) &= \nabla_\theta E_{\tau\sim\pi_\theta}[R(\tau)] \\
&= \nabla_\theta \int R(\tau)P(\tau|\theta)d\tau \\
&= \int R(\tau)(\nabla_\theta P(\tau|\theta))d\tau \\
&= \int R(\tau)P(\tau|\theta)(\nabla_\theta log[P(\tau|\theta)])d\tau \\
&= E_{\tau\sim\pi_\theta}[R(\tau)(\nabla_\theta log[P(\tau|\theta)])]
\end{aligned}$$

To differentiate the conditional log probability of the trajectory given the parameters, we break down what this probability is exactly.

$$P(\tau|\theta) = \rho_0(s_0)\prod_{t=0}^{T} P(s_{t+1}|a_t, s_t)\pi_\theta(a_t|s_t)$$

In the equation above, $\rho_0(s_0)$ represents the probability that the environment starts at state s_0 and $P(s_{t+1}|a_t, s_t)$ represents the probability that the state transitions to s_{t+1} given that it started from s_t and the action a_t was taken. Note that both of these probabilities depend only on the environment and not on the parameters θ. Finally, π_θ is our policy. The product from $t = 0$ to T represents the fact that the probability of the trajectory τ is simply the product of the individual probabilities of the environment transitioning to the states described by the trajectory at each time step and the probabilities of selecting the correlated action. Now,

$$
\begin{aligned}
P(\tau|\theta) &= \rho_0(s_0)\prod_{t=0}^{T} P(s_{t+1}|a_t, s_t)\pi_\theta(a_t|s_t) \\
logP(\tau|\theta) &= log\rho_0(s_0) + \sum_{t=0}^{T}[logP(s_{t+1}|a_t, s_t) + log\pi_\theta(a_t|s_t)] \\
\nabla_\theta logP(\tau|\theta) &= \nabla_\theta log\rho_0(s_0) + \sum_{t=0}^{T}[\nabla_\theta logP(s_{t+1}|a_t, s_t) + \nabla_\theta log\pi_\theta(a_t|s_t)] \\
&= \sum_{t=0}^{T} \nabla_\theta log\pi_\theta(a_t|s_t)
\end{aligned}
$$

The above follows because only the policy depends on the parameters θ. Finally we get,

$$
\begin{aligned}
\nabla_\theta J(\pi_\theta) &= E_{\tau\sim\pi_\theta}[R(\tau)(\nabla_\theta log[P(\tau|\theta)]) \\
&= E_{\tau\sim\pi_\theta}[R(\tau)(\sum_{t=0}^{T} \nabla_\theta log\pi_\theta(a_t|s_t))
\end{aligned}
$$

Since this is an expectation, we can compute it using the Monte-Carlo approximation. Thus, for a set of D trajectories:

$$\nabla_\theta J(\pi_\theta) \approx \frac{1}{D}\sum_{d=1}^{D} R(\tau_d)\sum_{t=0}^{T_d} \nabla_\theta log\pi_\theta(a_t|s_t)$$

Now that we have the gradient, we simply apply gradient descent to maximize the expected returns.

2.3 Vanilla policy gradient algorithm

Policy gradient is a foundational algorithm in reinforcement learning which has been modified many times into more advanced algorithms. To avoid confusion, the implementation of the original policy gradient algorithm is often referred to as vanilla policy gradient. The algorithm is shown below. For simplified understanding, it is nested into two.

Algorithm 1 Vanilla policy gradient

```
Initialize parameters $\theta_0$, max iterations $N$, max trajectories per iteration $D$
for $k = 1, 2, ..., N$ do
    Simulate batch of $D$ trajectories according to Algorithm 2.
    Collect all the steps and actions taken in each trajectory in the batch, $S_D$ and $A_D$.
    Collect all the rewards for each trajectory in the batch, $R_D$.
    Calculate the top 30th percentile reward value in the batch, $r_{30}$.
    Filter trajectories to find ones with reward higher than $r_{30}$. Call this "elite batch".
    Estimate policy gradient $\nabla_\theta J(\pi_\theta) \approx \frac{1}{D}\sum_{d=1}^{D} R(\tau_d)\sum_{t=0}^{T_d} \nabla_\theta log\pi_\theta(a_t|s_t)$ using elite batch.
    Compute policy update using Adam optimizer to get $\theta_{k+1}$.
end for
```

In algorithm 1, we first simulate D trajectories and collect all the states, actions, and rewards associated with each trajectory. Then, we filter these trajectories to find the ones with the top 30% of rewards. This is because we will quickly have too many trajectories to compute. Instead of calculating gradients across all of them, we stick to the best ones. We refer to these top trajectories as the "elite batch", and its state-action pairs become the training data for our neural network. Next, we evaluate the policy gradient with respect to the neural network parameters based on the expected returns in the elite batch. Then, we update the parameters using an Adam optimizer. Finally, we repeat all the steps for N iterations. The value of N is discussed further in section 2.5.

Algorithm 2 Simulation of D trajectories

```
for $d = 1, 2, ..., D$ do
    Initialize the state as $s_0$
    for $time = 1, 2, ..., T$ do
        if $k <= 50$ then                                   ▷ $k$ is from Algorithm 1
            $a_t$ = Randomly sampled from action space
        else
            $a_t = \pi_\theta(s_t) + noise$
        end if
        $s_{t+1}, r_{t+1} = env.step(a_t)$
    end for
    $S_d = \text{list}(s_0, s_1, s_2, ..., s_{T+1})$
    $A_d = \text{list}(a_1, a_2, a_3, ..., a_T)$
    $R_d = \sum_{i=1}^{T+1} r_i$
end for
$S_D = \text{list}(S_1, S_2, S_3, ..., S_D)$
$A_D = \text{list}(A_1, A_2, A_3, ..., A_D)$
$R_D = \text{list}(R_1, R_2, R_3, ..., R_D)$
```

To simulate D trajectories as shown in algorithm 2, we first initialize the environment to a random state s_0. Then, if the iteration count is less than 50, we randomly sample an action from the action space. This is because at the beginning, when the policy is young, it will be biased towards a set of actions for a given state. Thus, we will not be able to attain the necessary random actions needed to achieve initial success. Once the policy has been trained for an arbitrary number of iterations (empirically defined as 50), we begin to sample from the policy. We also add noise to the sample to allow the agent to explore new actions that might achieve even better results.

Once we sample an action, we take a step in the environment, and store the next state and reward achieved. We repeat these steps until the maximum number of time steps allowed per trajectory (empirically defined as 200, as explained in section 2.5). Then, we collect all the states and actions observed in the trajectory. We also calculate the overall reward at the end of the trajectory. We repeat these steps for D trajectories. Finally, we collect the steps, actions, and rewards for each trajectory and use them in algorithm 1.

2.4 Model architecture

Figure 5 shows the architecture of the neural network used in the deep reinforcement learning model to model the policy. The entire neural network was implemented in JAX. The input layer consists of 28 values that comprise the observations of the environment. There are three hidden layers of 256 neurons each, with a ReLU layer between them. It might have been possible to use a smaller neural network to learn the policy, but I decided to use this network so that the code can be adapted to more complex reinforcement learning algorithms, if necessary. Hyperbolic tangent is used as the final activation function so that the values in the output layer are constrained between -1 and 1, as required by the environment. However, between hidden layers, ReLU is used as the activation function to prevent vanishing gradients in the initial layers that might occur if a hyperbolic tangent activation was used instead.

Figure 6 shows the architecture of the overall program used in this project. The flow is as follows. The trajectory simulator within the main program queries the environment for its current state. It then passes this state to the policy to generate an action. It adds noise to this action and feeds it to the environment, which returns the next state and the reward as a result of executing that action. This continues until the trajectory ends and enough batches of trajectories are collected. Then, the simulator obtains the elite batch of trajectories and passes the states and actions of that batch to the gradient calculator. Here, the gradient of the policy is calculated with respect to the inputs, and then it is passed to the Adam optimizer (implemented in JAX) within the deep RL model class. Finally, the optimizer then updates the policy, which then becomes more likely to predict the correct action for a given state. This constitutes one iteration and the process is repeated until the maximum number of iterations.

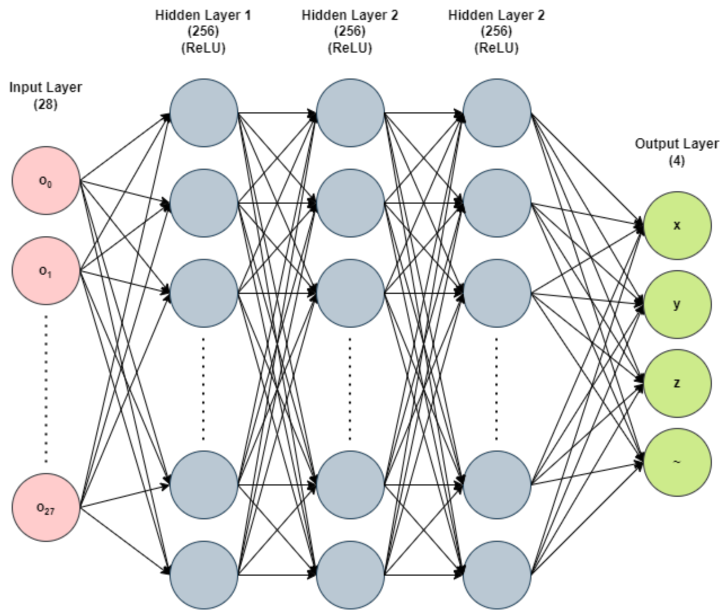


Figure 5: Neural network architecture used in the deep reinforcement learning model. Diagram created using [8].

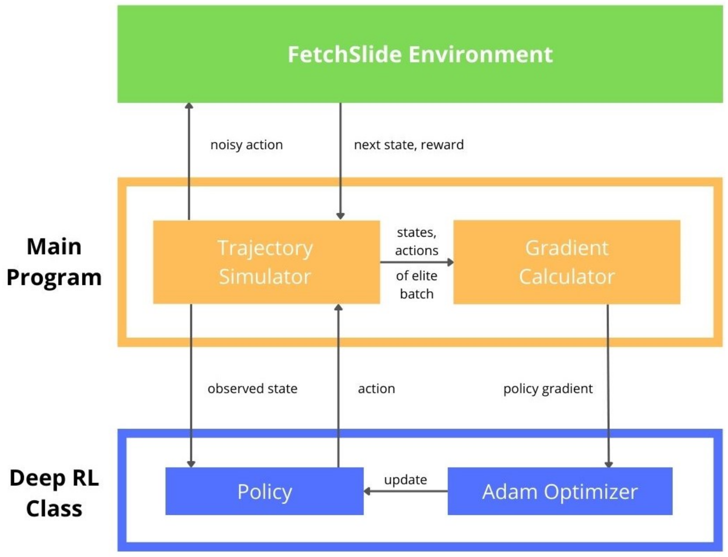


Figure 6: Architecture of the overall program.

2.5 Training

Each trajectory d in algorithm 2 consisted of $T = 200$ time steps. This value was determined empirically to be enough for an untrained agent to have a success rate of approximately 1%. The total number of trajectories per batch, D, was determined to be 100. This was so that, on average, at least one successful trajectory could be obtained per batch when the agent was young and took random actions. If there were no successes in the batch, the size of the elite batch would be zero, and the agent could not be trained in that iteration.

Given the small success rate of the early agent, each iteration would only have 1-3 successful trajectories that could be used for training. Therefore, instead of discarding the successes after each iteration, I stored them for use in the next iteration. For example, if the agent was successful 2 times in the first iteration, and 1 time in the second iteration, I would train it based on all the successful trajectories so far, i.e. 3. Following this method, I stored a maximum of 100 successes in the buffer, and deleted older ones as new successes were achieved.

Ideally, the *for* loop in algorithm 1 must be continued until the reward collected by the agent exceeds a threshold value. In this project, I chose to stop the iterations at $N = 100$. The Gym FetchSlide environment was slow in returning the next state given an action, so the model took time to train. Due to the fact that I utilized Google Colaboratory for the implementation of this project, I could not train the agent for more than a few hours before the session timed out. Hence, I stopped the training at 100 iterations and stored the parameters of the neural network in a *.npy* file. Then, I relaunched the Colab session, initialized the model with the stored parameters in the *.npy* file, and ran the session for another 100 iterations. Essentially, I introduced a training reset. I repeated this four times to obtain a model that was trained for 400 iterations.

3 Results and Analysis

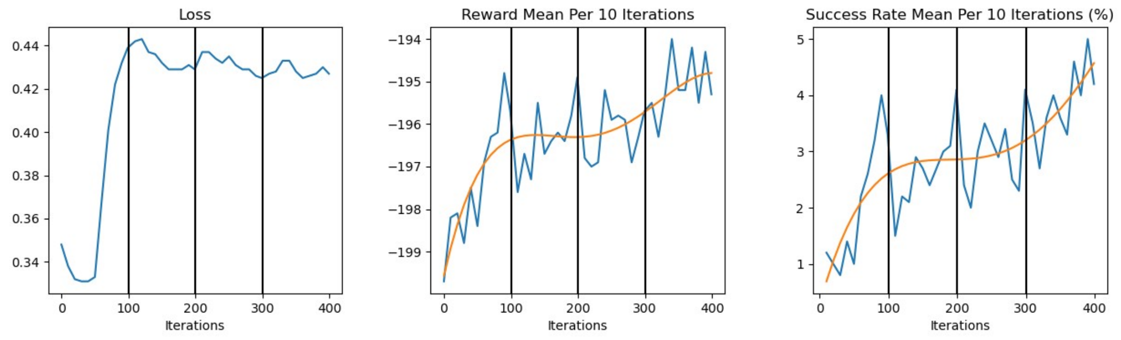


Figure 7: Plots of various performance indicators over iterations. Orange lines represent 4-degree best-fit polynomials.

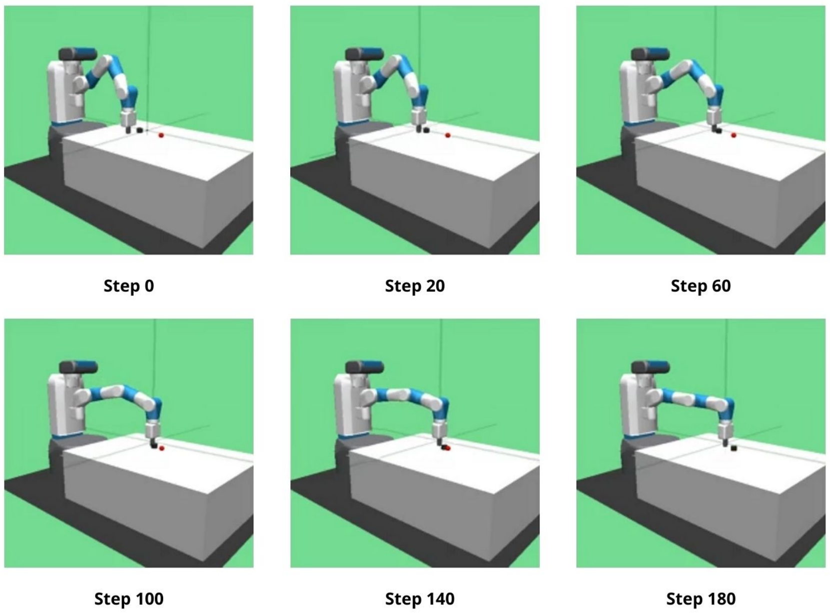


Figure 8: Reinforcement learning agent trained for 400 iterations. Example of a successful episode where the puck reaches the goal position.

Figure 7 shows the results at the end of training over various performance indicators. The vertical black lines represent when the training was stopped and restarted, according to section 2.5. Figure 8 shows a successful episode where the agent is able to achieve the given objective. The sections below discuss specific insights and analyses of these results.

3.1 Loss

In supervised machine learning, the loss function measures how divergent the outputs of the model are from the true output labels/values. While the same is true in reinforcement learning, loss becomes less of a strict measure of performance because the "true output labels/values" are not always optimal. Consider the scenario depicted in figure 9.

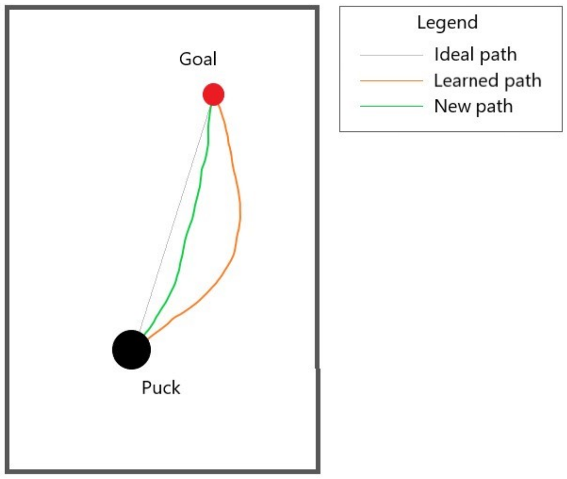


Figure 9: Paths taken by the robot to push the puck to the goal location.

Here, the ideal path for the robot to push the puck towards the goal is shown in grey. This path would likely take the least amount of time, assuming it's the most efficient path for the robot joints to trace. Assume that at a certain iteration n, the policy dictates that the robot follow the path shown in orange, AKA the learned path. This path allows the robot to complete the objective, but it is not optimal. Assume that, due to the noise introduced, the robot instead follows the path shown in green. The loss is then calculated between the learned path and the new path, and might be reported as substantial. However, the new path is more optimal and therefore will stay in the elite batch for future iterations. Thus, the model will learn to follow the more optimal path, even though the reported loss was high.

The graphs show that the model started with a low loss around 0.340 but quickly jumped to a high value around 0.440 after 100 iterations. This is likely due to the fact that, initially, the successful paths that arose due to random actions were similar to the paths predicted by the untrained agent. However, as we filtered the most optimal paths from the random actions, we discovered that these paths were very different from what the agent could predict, and therefore the loss continued to climb.

At around 100 iterations, the loss stabilized, and the agent became likely to predict somewhat optimal paths. After that, the loss was shown to decrease as iterations increased. However, the reduction was not smooth. The many times that the loss stayed stagnant or went up can be explained due to the earlier learned vs. new reasoning. It is a good sign that as the iterations go up, and as more optimal paths are added to the buffer, the loss continues to go down. This means that the agent is learning the more optimal paths to achieve success.

3.2 Reward mean and success rate mean

Experimental results showed high fluctuation of the reward mean and success rate mean per 10 iterations. However, there was a general rising trend in both numbers. Both these indicators represent different aspects of the performance of the agent.

The success rate mean per 10 iterations describes how successful the agent is in finding a solution, optimal or otherwise. Whereas, the reward mean per 10 iterations describes the ability of the agent to find more optimal paths. As the reward mean goes up, it indicates that the agent is able to push the puck to the goal position in fewer steps.

The rising trend in both these numbers show that the policy gradient method is effective in teaching a reinforcement learning agent to achieve a desired objective.

3.3 Effect of training resets

As described in section 2.5, after every 100 iterations, the training was stopped and the model parameters were saved. Then, training was restarted with the saved parameters for another 100 iterations. The graphs show the effect of this process. After each reset, the loss spikes initially before continuing a downward descent. The reward mean and success rate mean both take a negative dip before returning to their climb. There are two possible reasons for this behavior.

The first is that, after each reset, the buffer of elite batches is cleared. Towards the end of a set of 100 iterations, the buffer contains 80-100 "good" trajectories [1]. Each iteration trains the model on these trajectories. However, once the training is reset, the buffer is emptied and for the next couple of iterations, the number of trajectories in the buffer will be around 5-20. This negatively impacts the amount of training that can be carried out in those initial iterations.

The second reason could be that, when the model is saved after 100 iterations, the first and second running moments of the gradient in the Adam optimizer are not saved. Hence, when the model is trained for the first iteration after reset, the optimizer has "forgotten" the previous values of moments and starts fresh again.

3.4 Data inefficiency

As described in section 1, one of the biggest flaws of reinforcement learning is its data inefficiency. In this project, to attain a success rate of about 5%, I trained the agent for 400 iterations. Each iteration consisted of about approximately 80 trajectories, where each trajectory was made of around 195 state-action pairs. This amounts to 15,600 data points for training per iteration. Recall that these trajectories are not constant. Every time new elite trajectories are discovered, the older ones are deleted, meaning that the true number of total data points is likely in the hundreds of thousands and will only grow with each iteration. Compared to other forms of supervised machine learning, reinforcement learning consumes much more data to produce similar results.

4 Ideas for Improvement

4.1 Iteration count

After 400 iterations, the agent was able to achieve a success rate mean of approximately 5% and could clear the task in around 195 steps. The graphs over these two indicators show that they improved with each iteration. With further iterations, the agent could achieve better performance: complete the objective faster and more often.

4.2 Adam moment and elite batch buffer carryover

To remove the effects of the training resets, the obvious solution would be to train the model outside Google Colab and on an actual PC where it can be run for many hours. However, given the amount of time the RL model takes to train, training resets might still be necessary to evaluate the model from time to time. As described in section 3.3, it might be possible to decrease the effect of the training reset if the moments in the Adam optimizer and the elite trajectories in the buffer were carried over to the new training environment.

4.3 Elite batch buffer optimization

Currently, once the number of elite trajectories in the buffer exceeds 100, the first 20 trajectories are deleted to make room. The reasoning here is that the earliest trajectories in the buffer correspond to earlier trajectories where the policy wasn't as well trained. Hence, by making room for newer trajectories, we can train the agent on more optimal trajectories. However, this was not always the case.

Due to the noisy actions taken by the agent, we often received new trajectories with lower reward means than the ones in the buffer. To fix this issue, we could sort the buffer based on reward mean. Then, when it is time to delete trajectories, we can delete the ones with the lowest reward means, and only add

[1] Recall that the buffer is designed to hold up to 100 trajectories for training. If the number of trajectories exceeds this number, the first 20 trajectories are deleted to make room for more.

new ones that have higher reward means than the ones in the buffer. Thus, we can optimize the buffer to hold only the best trajectories.

4.4 More advanced RL algorithms

Vanilla policy gradient is arguably the most fundamental algorithm in reinforcement learning. Due to its drawbacks in efficiency and low complexity of tasks, more advanced algorithms like Deep Deterministic Policy Gradients [13] and Soft Actor-Critic [14] were invented. These were avoided for this project because it was my first foray into reinforcement learning, so I decided to stick to the fundamentals. However, the usage of these algorithms would allow for better performance within the same number of iterations, as well as the ability to expand the task to higher complexities if necessary.

5 Conclusion

In this project, a deep reinforcement learning agent was created to function on the OpenAI Gym Fetch-Slide environment. JAX was used to build the Adam optimizer and the deep neural network that modeled the policy. The agent was trained using a policy gradient algorithm for 400 iterations and was able to achieve a success rate mean of approximately 5% and could achieve the objective in around 195 steps. The performance of the agent was analyzed and ideas for improving the model were proposed.

6 Acknowledgement

The author would like to thank Chris Hsu, Ph.D. candidate, University of Pennsylvania, for his help in improving the model performance and identifying a key result. Thanks also goes to Dr. Paris Perdikaris, Shyam Sankaran, and Jin Wu, University of Pennsylvania, for their tireless efforts in making ENM 531 a valuable and enjoyable class this semester.

7 References

[1] Tammy Jiang, Jaimie L. Gradus, Anthony J. Rosellini, *Supervised Machine Learning: A Brief Primer, Behavior Therapy*, Volume 51, Issue 5, 2020, Pages 675-687, ISSN 0005-7894, https://doi.org/10.1016/j.beth.2020.05.002.

[2] Richard S. Sutton, Andrew G. Barto., *Reinforcement Learning.* 2nd ed., MIT Press Ltd, 2018, p. 1.

[3] "Gym: A Toolkit For Developing And Comparing Reinforcement Learning Algorithms". Gym.Openai.Com, 2022, https://gym.openai.com/envs/FetchSlide-v0/.

[4] Andrychowicz, Marcin and Wolski, Filip and Ray, Alex and Schneider, Jonas and Fong, Rachel and Welinder, Peter and McGrew, Bob and Tobin, Josh and Abbeel, Pieter and Zaremba, Wojciech, *Hindsight Experience Replay*, arXiv, 2017, https://arxiv.org/abs/1707.01495

[5] Schulman, John, *Optimizing Expectations: From Deep Reinforcement Learning to Stochastic Computation Graphs*, University Of California, Berkeley, 2016. p. 12-17

[6] Gallouédec, Quentin. "Gym-Robotics/Fetchslide.Md At Main · Farama-Foundation/Gym-Robotics". Github, 2022, https://github.com/Farama-Foundation/Gym-Robotics/blob/main/docs/FetchSlide.md.

[7] "API - Gym Documentation". Gymlibrary.Ml, 2022, https://www.gymlibrary.ml/content/api/.

[8] "Diagrams.net." https://app.diagrams.net/

[9] "JAX Reference Documentation — JAX Documentation". Jax.Readthedocs.Io, 2022, https://jax.readthedocs.io/en/latest/index.html.

[10] Matthew Botvinick, Sam Ritter, Jane X. Wang, Zeb Kurth-Nelson, Charles Blundell, Demis Hassabis, *Reinforcement Learning, Fast and Slow*, Trends in Cognitive Sciences, Volume 23, Issue 5, 2019, Pages 408-422, ISSN 1364-6613, https://doi.org/10.1016/j.tics.2019.02.006.

[11] Kaiser, Lukasz and Babaeizadeh, Mohammad and Milos, Piotr and Osinski, Blazej and Campbell, Roy H and Czechowski, Konrad and Erhan, Dumitru and Finn, Chelsea and Kozakowski, Piotr and Levine, Sergey and Mohiuddin, Afroz and Sepassi, Ryan and Tucker, George and Michalewski, Henryk, *Model-Based Reinforcement Learning for Atari*, arXiv, 2019, https://doi.org/10.48550/arxiv.1903.00374.

[12] Ofir Nachum, Shixiang Gu, Honglak Lee, Sergey Levine, *Data-Efficient Hierarchical Reinforcement Learning*, 32nd Conference on Neural Information Processing Systems (NeurIPS 2018), Montréal, Canada, 2018

[13] Silver, David et al. *Deterministic Policy Gradient Algorithms.* JMLR: W&CP, Volume 32, 31 St International Conference On Machine Learning,, 2014, Accessed 10 May 2022.

[14] Haarnoja, Tuomas and Zhou, Aurick and Abbeel, Pieter and Levine, Sergey, *Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor*, arXiv, 2018, https://doi.org/10.48550/arxiv.1801.01290