Optimizing Lap Times around a Circuit using Policy Gradient Methods

Dhruv Chauhan

Northeastern University

chauhan.dhr@northeastern.edu

Abstract

The project investigates how to optimize lap times in a simulated 2D racing environment. These methods and environments could be adapted to inform Formula 1 qualifying session strategies. The primary approach used is Proximal Policy Optimization (PPO), a policy gradient method that learns the parameters of a beta distribution to model the policy. To enhance the training process, a custom environment wrapper is employed, stacking consecutive frames as input to the state representation. This approach enables the model to infer the agent's velocity and direction of movement more effectively. The project evaluates the agent's ability to minimize lap times, highlights the benefits of the environment wrapper on training performance, and assesses the effectiveness of PPO in this continuous control task.

Introduction

Reinforcement Learning (RL) has achieved remarkable success in solving complex continuous control problems, thanks to advancements in deep learning. In this project, I aim to apply RL techniques to a simulated 2D racing environment to optimize lap times. The task involves navigating the track as quickly as possible while maintaining control and staying within track boundaries.

The primary algorithm used in this project is Proximal Policy Optimization (PPO), one of the most widely used policy-gradient methods in RL today. Unlike value-based methods, which focus on learning the value of states (how good or bad a state is), policy-gradient methods directly learn the policy—determining the best action to take given a state. These methods do so by learning the parameters of a probability distribution function. While many standard environments use Gaussian distributions (for continuous actions) or softmax distributions (for discrete actions), this project leverages a beta distribution, as it is well-suited for bounded action spaces.

This project goes beyond simply optimizing lap times; it also investigates the effects of modifying the environment and reward structure on the agent's learning process. This is achieved by introducing a custom wrapper around the environment, which adjusts the state space and reward function. To evaluate the impact of these changes, four models are trained and assessed, as summarized in the table below:

Model	Self-coded	Changed reward function	Changed state representation
Model 1	❌	❌	❌
Model 2	❌	❌	✅
Model 3	✅	❌	✅
Model 4	✅	✅	✅

The comparison between Models 3 and 4 provides insights into how well the self-coded algorithm performs relative to a library-based implementation when trained and evaluated on the same environment. Furthermore, by analyzing Models 1, 3, and 4, the project highlights the benefits of modifying the state representation and reward function in accelerating the agent's training and improving its performance.

This approach not only deepens the understanding of PPO, but also shows the importance of environment and reward structuring while trying to solve complex RL problems.

Background

Markov Decision Processes Most RL problems are formulated as Markov Decision Processes (MDP), which provide a mathematical framework for decision-making in environments where the agent's actions influence future outcomes. More formally, an MDP is a 5-tuple (S, A, T, R, γ) where S is the state space, A is the action space, T is the transition function (which specifies the probability distribution over next states, given the current state and action), R is the reward function, and γ is the discount factor. An important feature of MDPs is their Markovian property, meaning that the outcome at any given time depends solely on the current state and action, without any reliance on past states or actions. Therefore, when defining a state in an MDP, it is important to include all relevant information necessary for decision-making at that time.

Proximal Policy Optimization (PPO) The primary algorithm used in the project is Proximal Policy Optimization (PPO). It is a policy-gradient method, which means that it aims to learn the policy directly. It does so by trying to learn the parameters of a probability distribution, which essentially is the distribution of what action should be taken given a state. PPO uses an actor-critic structure, where it uses two networks to model an actor (the policy) and a critic (the value function). The critic informs the actor by providing feedback on how good a particular action is relative to the value of the state, and this is normally called the advantage. The advantage is then used to train the actor to perform actions with higher advantages (better than average returns). Another reason why PPO performs very well is because of the clipped-objective function, which makes PPO a very stable algorithm, helping work well in complex environments. The clipped objective function looks like

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

where

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$$

Essentially, the clipped function ensures that if the new policy deviates significantly from the old policy (i.e., the ratio is high), and this deviation leads to a high reward, the update is "clipped" within a specified range. This prevents the policy from changing too drastically, ensuring that the learning process remains stable and the agent does not stray too far from the previous policy. By doing so, PPO avoids making overly large updates that could potentially lead to unstable learning or the adoption of a completely new, suboptimal policy.

Beta Distribution A beta distribution is a parameterized continuous probability distribution function over the interval of [0, 1]. This makes it useful to model variables that are constrained to this (or a certain) range. The two parameters that shape the distribution are α, β. The beta distribution is highly flexible and can represent a multitude of behaviors from uniform distributions (when α = β = 1) to highly skewed distributions (when α and β are not equal). The versatile nature of this distribution over a fixed interval makes it particularly useful to model probability distributions over bounded action spaces in reinforcement learning.

Project Structure

This section will be used to describe the environments and models used in the project. I will then talk about how these environments and models are relevant to the goals of the project - creating a PPO agent and showing the importance of environment design.

Environment description The environment used for this project is OpenAI gym's box2d Car Racing environment. The MDP is defined as follows. The state space of this environment is a 96x96x3 RGB image of a car on the track. The action space is a 3-tuple of (steering, gas, break), where steering is a number in the interval [-1, 1], and the gas and break are numbers in the [0, 1] interval. The environment is purely deterministic in nature, and so taking a particular action at a state will always yield in a fixed next state. The basic reward function is a combination of time and the number of tiles visited on the track, where we get a small negative reward for each time step and a positive reward for each unique track tile visited. The environment also truncates after 1000 frames, so that if the agent is stuck it can reset. The reward threshold for the environment (a reward to "successfully" beat the environment) is 900.

Two key modifications are made to the environment in this project. First, the reward function is adjusted to penalize going off-track by giving a negative reward for driving on the grass. This discourages the agent from straying off-track, as the default reward function does not strongly penalize such behavior. Second, a frame-stacking mechanism is added, where the last four frames are stacked together and passed as the state. This modification provides the model with a sense of velocity and movement direction, which is not captured in the original environment that uses a single RGB frame.

Model descriptions There are four models used in this project. The first model is a PPO agent trained using the stable_baselines3 CNN-policy PPO agent on the original car racing environment. The second model is trained using the stable_baselines3 PPO agent as well, however, this time using a VecFrameStack wrapper on the car racing environment to stack the previous 4 frames on top of each other. This is done to give the velocity of the agent as information to the model, however, this environment still uses the original reward function. The third model is a similar frame-stacked environment, but this time the PPO agent is self-coded. Finally, the fourth model uses both a changed reward function - gives a negative reward for going off track - as well as stacks frames to give the model information about the agent's velocity.

PPO agent To validate the self-coded PPO agent, we compare the performance of Model 2 (stable_baselines3 PPO) and Model 3 (self-coded PPO) trained on the same environment. Evaluation results from these models will help assess the accuracy and effectiveness of the self-coded implementation. The architecture of the self-coded PPO agent comprises of 2 neural networks - one for the policy and the other for the value function. Both networks are convolutional neural networks each with 7 fully-connected hidden layers. The policy network learns the parameters for a beta distribution since our action space is bounded (we scale the distribution to be between [-1, 1] for the steering and use the value as is for the gas and break).

Environment design To understand how changes in the environments impact the training of the agent, we can compare models 1, 3, and 4. We can train them on their respective environments - model 1 with the original reward function and state representation, model 3 with the original reward function but changed state representation, and model 4 with the changed reward function and state representation. Then, evaluating them under the same conditions will reveal the impact of these changes. We can also look at how these agents interact with the environment by rendering the environment and observing general behavior.

Experiment Results (PPO agents)

Model 2 As mentioned earlier, model 2 is a stable-baseline based PPO agent trained on the frame-stacked environment. The agent was trained using the CNN Policy (better suited for images) and was able to give decent results.

Model 3 Model 3 is a self-coded PPO agent trained on the same environment. It uses two CNNs - one for the policy parameters and the other for the value function. The pseudocode in Algorithm 1 is what is used to train the PPO agent. The model can train steadily as shown below (figure 1). The sudden spike in the training curve is most probably because of the location of a "sharp turn" some frames into the environment. It struggles a lot to make that turn and thus loses on rewards until it makes that turn. However, once it learns how to make that turn, it learns relatively fast how to complete the lap.

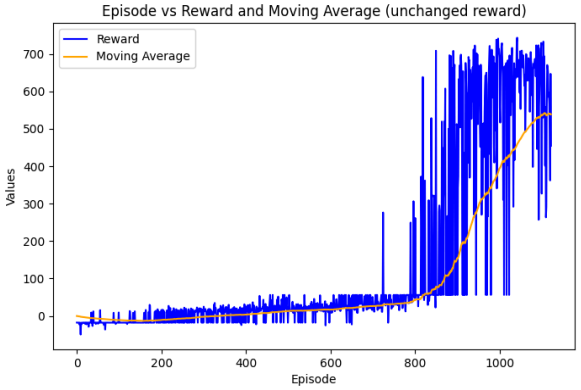


Figure 1: Training curve plot for self-coded PPO Agent (unchanged reward function)

Algorithm 1 PPO-Clip Algorithm

Require: Initial policy parameters θ_0, initial value function parameters ϕ_0

1: **for** $k = 0, 1, 2, \ldots$ **do**

2: Collect set of trajectories $\mathcal{D}_k = \{(s_i, a_i, r_i, s'_i)\}$ of $buffer\_size$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.

3: Compute the target value ($G_t = r_t + \gamma \hat{V}_{\phi_k}$) and current state value $\hat{V}_{\phi_k}$

4: Compute advantage estimates $\hat{A}_t$ (based on the current value function and the target function)

5: Get α, β from $\hat{P}_{\theta_k}$ and get log probabilities of actions using this new distribution $Beta(\alpha, \beta)$ as new_log_prob

6: Get $ratio = \exp(new\_log\_prob - old\_log\_prob)$

7: Update θ_k using loss

$$loss = clip(ratio * \hat{A}_t, 1 - \epsilon, 1 + \epsilon)$$

stochastic gradient ascent with Adam.

8: Update ϕ_k using loss

$$loss = \Sigma \hat{A}_t$$

via gradient descent algorithm.

9: **end for**

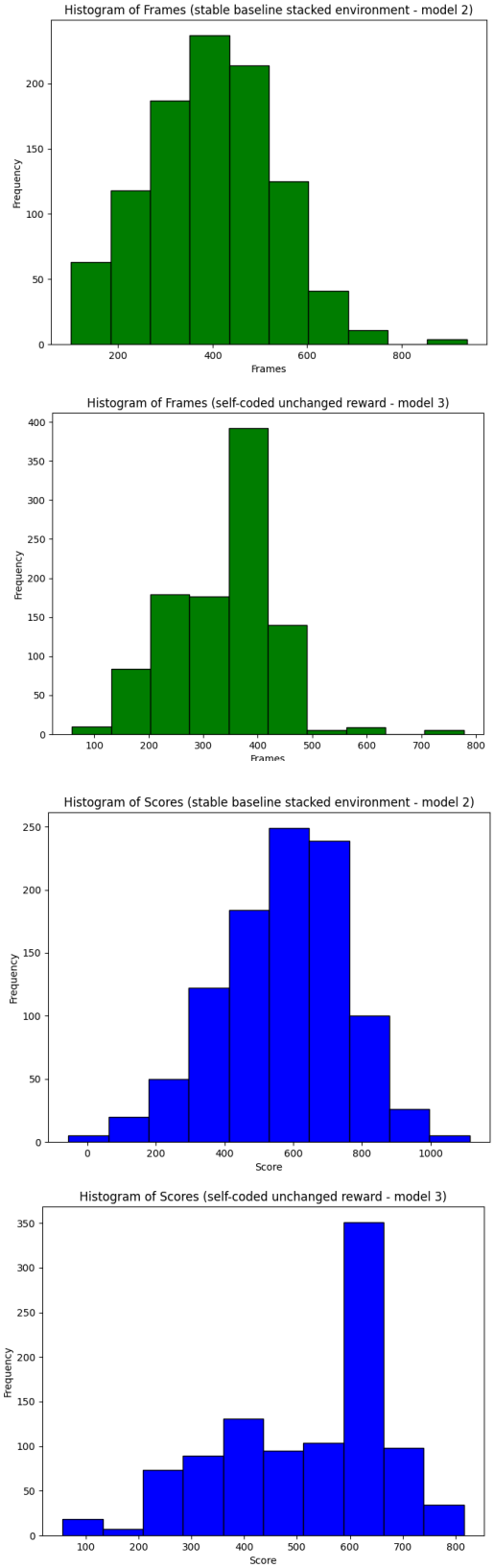


Figure 2: Comparison between model 2 (stable baselines) and model 3 (self-coded) models' rewards and frames used.

Experiment Results Model 2 and Model 3 behave similarly. Model 2 uses around 500 episodes to train, compared to the 1100 episodes used by Model 3. The inefficiencies in the self-coded version may cause this, however, we can see that given enough time, they can both perform equally well.

The two pairs of histograms compare the rewards (blue histograms) and number of frames used (green histograms) by the two models. As we can notice, they both use around 200-400 frames to complete the lap, which is decent for a trained model, however, the average reward is somewhere around the 600 mark.

This is because the car is currently not incentivized enough to stay on track, and goes onto the grass and loses some reward potential. This would also not be a great agent to solve the problem, as in racing the car is required to stay on track throughout the race. As seen in the images (Figure 5) on page 5, the agent frequently goes off track. The following section discusses how changing this reward function might help the agent.

Experiment Results (Environment Design)

Model 1 Model 1 is the most basic agent trained for this project. It uses the standard (non-stacked/unchanged reward function) environment to train a stable-baselines PPO agent. This agent was trained for 2000 episodes.

Model 3 Self-coded model trained on stacked-frames environment but original reward function

Model 4 Model 4 is the agent that uses all of the changes combined. It is a self-coded PPO agent that uses a stacked-frame environment with a changed reward function. The reward function in particular penalizes the agent for going off track. The model trains steadily as shown below. Again, like model 3, the spike in the learning curve is attributed to the sharp turn, and the model learns relatively fast once it figures that turn out. The spikes in the blue curve are because PPO still explores, but at this point, it has learned the policy well.

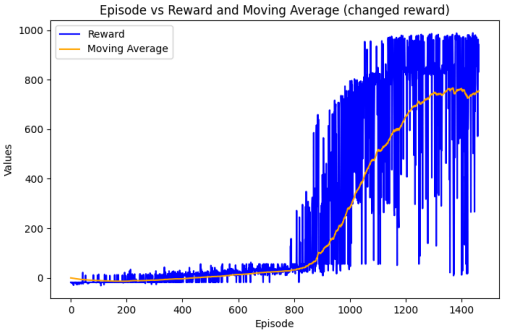


Figure 3: Training curve plot for self-coded PPO Agent (changed reward function)

Experiment Results As we can see in the graphs on this page (figure 4), model 1 is unable to cross the 500 reward mark (staying around 300) and almost always uses all frames. This is because as shown in the image on the next page (figure 5), it gets stuck while trying to make sharp turns. The environment starts with a bunch of wide turns which the agent is able to learn how to deal with, however, once it encounters a sharp turn it is no longer able to handle it. It then just goes off the track and starts spinning.

Contrasting this with the performance of model 3, where the average reward sits around the 600 mark and the frames sit around the 300-400 mark, we can see that the agent is able to complete laps. Giving the model a sense of velocity by stacking the previous 4 frames helps the model understand that it should slow down as the turn is sharper than the turns taken previously. However, as indicated by the 500 reward, which is far below the environment threshold, the agent is not best trained yet. There are two main problems - the agent is too slow and goes off track too often. The second problem can be seen by the image shown on the next page (Figure 5) where it goes off track and thus loses out on potential reward.

To fix the above problems, we can look at how model 4 works. This model has an average reward of around 800-900, which seems good enough given the environment's reward threshold. It is able to maintain a decent amount of speed throughout the track while also staying off the green area. This model being able to stay on track can be attributed to giving a penalty for going off track. We can contrast the speed using the average frames used, which sits around 150-200, as compared to the higher number of frames used by model 3. Also, compared to how model 1 goes completely off track, we can see in the image on the right how model 4 can take a sharp turn while being fast as also indicated by the drift marks (Figure 5).

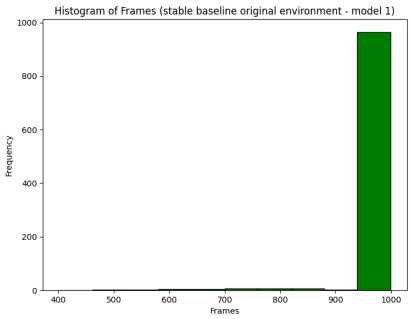
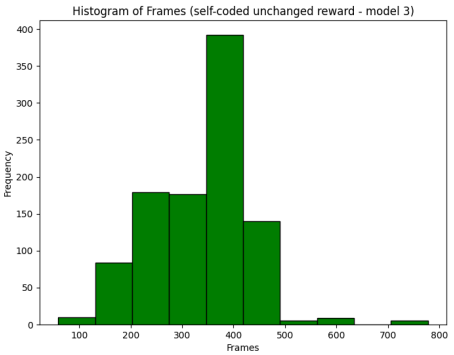
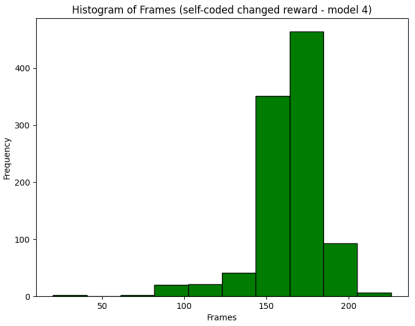
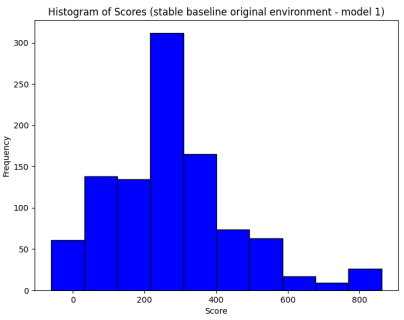
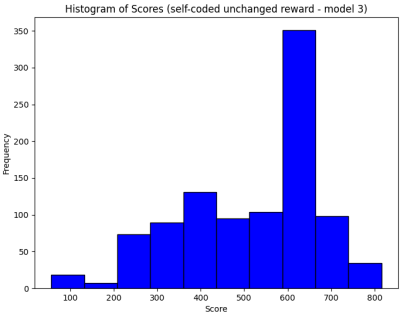


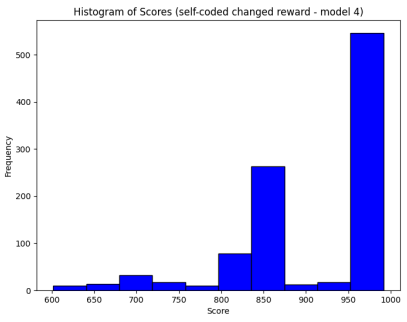
Figure 4: Plots of frames used (green) and rewards (blue) for model 1, model 3, and model 4 in order.











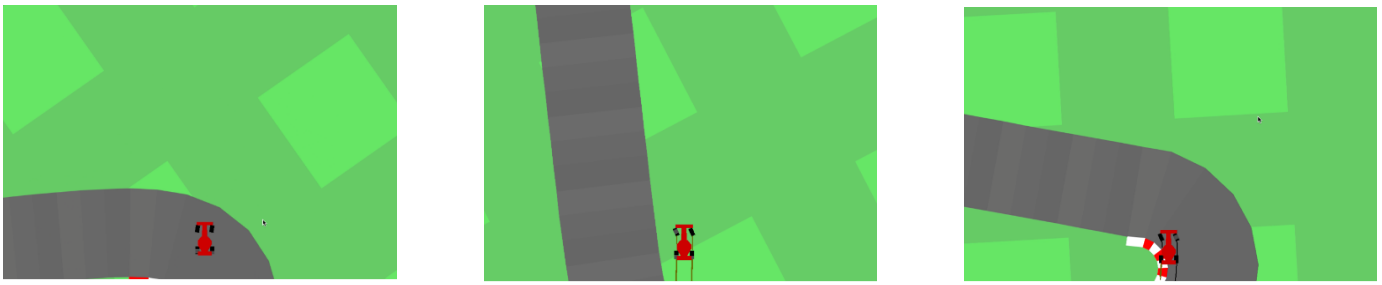


Figure 5: Policy behaviors from model 1, model 3, and model 4 (in order)

The images above are a good representation of the policy behaviors talked about in this section - the first shows model 1 not knowing how to take a sharp turn, the second shows model 3 going off track, and the third shows how model 4 has learned how to take the turn well.

Finally, to emphasize the performance difference between the three models and how important good environment design is, below is a scatter plot.

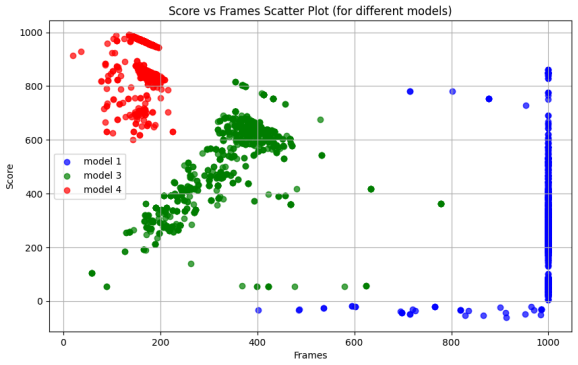


Figure 6: A scatter plot that has frames on the x-axis and the score (reward) on the y-axis for all 3 models (model 1 blue, model 3 green, and model 4 red).

The scatter plot shows how many frames are used by a model and how much score they get using those frames. There are 3 clear classes - blue, green, and red - each indicating a different model (1, 3, and 4 respectively). The blue class (model 1 - rightmost) uses all the frames always but is able to get a variety of scores (mostly in the middle). This means it is not able to complete most of the time. The green class is a cluster in the middle of the plot, showing that it uses an average amount of frames and gets an average reward. It is able to do better but as also shown in figure 5, goes off track quite often. Finally, the red class, being in the top-right section, is able to get a high reward using a low number of frames. This shows that it has trained well to go around the lap in a fast way. Thus, we have seen how a combination of changing state representation and environment design helps in solving this environment.

Further Work

This project was a success for the most part - I was able to code a PPO agent to go around a lap in the fastest way possible while also demonstrating the impact of good environment design.

The real-life applications would be training such an agent to find optimal racing lines around tracks by Formula 1 teams. There are, however, a lot of things to consider when trying to use this for real-life racing.

Firstly, the environment is not complex enough to simulate the impact of different physics concepts such as aerodynamics and friction. Depending on the tire type and circuit type, these things vary a lot, and so we must use a more complex environment to simulate these changes.

There are also a lot of hyperparameters that could have been changed while training the agent. One of the biggest choices made was what distribution to model for the policy network. Some parameterized functions such as a squashed Gaussian could be used to model a bounded action space, and these could potentially be better than a beta distribution.

There were also ways to make training faster, although a bit complex and out of the scope of this project. One of them was to randomize the starting position of the agent. This way, it is possible that it comes across the sharp turn earlier and can learn faster.

Finally, even though there is only one track provided by the gym environment, it would be interesting to see how the agent performs in randomized environments. I would assume that the agent is able to complete a lap without crashing in a random environment, but would need some training to be able to optimize the lap time.

Conclusion

This project successfully demonstrated the application of Proximal Policy Optimization (PPO) for optimizing lap times in a simulated 2D racing environment. By coding up my own PPO agent and tweaking the environment to suit the needs of the problem on hand, the project demonstrates the impact of good environment design. While the project focused on a simplified 2D racing environment, its principles are directly applicable to real-world scenarios, such as optimizing racing lines for Formula 1 teams. Ultimately, this project contributes to the understanding of how policy gradient methods, particularly PPO, can be leveraged to solve complex continuous control problems, with significant potential for real-world applications.

References

Beta Distributions for a Given Mean, Median or Mode. Wolfram Demonstrations Project. (n.d.). https://demonstrations.wolfram.com/BetaDistributionsForAGivenMeanMedianOrMode/

Gymnasium documentation. Car Racing - Gymnasium Documentation. (n.d.). https://gymnasium.farama.org/environments/box2d/car_racing/

PPO. PPO - Stable Baselines3 2.5.0a0 documentation. (n.d.). https://stable-baselines3.readthedocs.io/en/master/modules/ppo.html

Proximal policy optimization. Proximal Policy Optimization - Spinning Up documentation. (n.d.). https://spinningup.openai.com/en/latest/algorithms/ppo.html

Simonini, T. (n.d.). *Proximal Policy Optimization (PPO)*. Hugging Face – The AI community building the future. https://huggingface.co/blog/deep-rl-ppo

Tam, A. (2023, April 7). *Building a convolutional neural network in Pytorch*. MachineLearningMastery.com. https://machinelearningmastery.com/building-a-convolutional-neural-network-in-pytorch/