Applying Plasicity Injection to Fine-Tune PPO Model on Atari Games

Bailey Cislowski
Department of Computer Science
University of Southern California
Los Angeles, CA 90007
bcislows@usc.edu

Elliott Kau
Department of Computer Science
University of Southern California
Los Angeles, CA 90007
kaue@usc.edu

Richard Liu
Department of Computer Science
University of Southern California
Los Angeles, CA 90007
rdliu@usc.edu

Eli Morris
Department of Computer Science
University of Southern California
Los Angeles, CA 90007
elimorri@usc.edu

Thomas Watson
Department of Computer Science
University of Southern California
Los Angeles, CA 90007
tfwatson@usc.edu

Abstract

Deep Reinforcement Learning is a unique yet increasingly important aspect of machine learning. Across multiple fields and problems, many algorithms and agent models rely on neural networks as the basis for learning and training, with reinforcement learning on games being a common topic. However, a common issue is once an agent has been trained, it often stagnates from an inability to learn new data. A plasticity injection is a procedure aiming to solve this issue by, put simply, learning a residual from a pre-trained agent, allowing for it to learn more about its environment than previously allowed. In this paper, we compare the efficacy of Stable Baselines 3 agents who have been altered by plasticity injection against their unmodified counterparts across multiple Atari games. Overall, we generally notice slight increases in performance, although the extent of long term benefits depends environment by environment. In the future, we hope to expand this work by applying it a wider range of games and RL algorithms to better understand plasticity injection as a whole.

1 Introduction

Reinforcement Learning (RL) is a type of machine learning where an agent learns to make decisions by interacting with an environment to achieve a goal. Unlike traditional methods such as supervised learning, where models are trained on a fixed set of data labels, RL operates on a system of rewards and penalties. The agent develops a policy to take actions that maximize cumulative rewards. RL's dynamic learning model allows it to adapt its strategies based on new experiences, making it particularly effective for complex problems requiring real-time decision making and adaptability, such as robotics, autonomous vehicles, and video-game playing; in recent times, the rise of large language models has spurred a greater interest in this field due to the use of reinforcement learning on human feedback (RLHF for short) to properly tune LLMs such as ChatGPT.

However, fine-tuning RL agents poses significant challenges. As models evolve, they often exhibit a reduced capacity to integrate new crucial information, often leading to performance plateaus or even degradation when tasked with new environments. This stagnation primarily arises from the inherent limitations in their learning algorithms, which, once optimized for a set of tasks, become less responsive to subsequent modifications or learning signals.

In response to these challenges, we integrate a novel approach, termed plasticity injection. Plasticity injection is a technique designed to rejuvenate the adaptability of RL agents by integrating additional learning layers into their existing architectures. This method aims to facilitate continual learning and adaptability and allows agents to remain receptive to new information even after extensive training phases. Our research focuses on the application of plasticity injection within the framework of Proximal Policy Optimization (PPO) Actor-Critic agents. Specifically, we evaluate the efficacy of modified agents in the context of playing Atari games via the Gymnasium library and compare their performance against their unmodified counterparts to assess improvements in learning efficacy and behavioral flexibility. This paper details the experimental results of plasticity injection and offers insights into its potential to enhance the adaptive capabilities of RL systems.

2 Related Work

To get a stronger background in fine tuning we read several papers related to fine tuning in reinforcement learning. One paper we found the most helpful was "Beyond Fine-Tuning: Transferring Behavior in Reinforcement Learning". In this paper the authors discussed a technique they used called Behavior Transfer. In this technique, the authors separated the training of behavior and weights. By using a pre-trained policy that was used to aid with exploration, they were able to accelerate the learning process as the agent would be exposed to useful experiences earlier in the learning process. The agent was exposed to pre-trained behavior in two ways, as an extra exploratory step that could be randomly activated, and as an additional pseudo action where the agent can defer its action to the pre-trained policy instead of choosing an action itself. The authors found that pre-training led to better results than without pre-training, and combining this with standard pre-training led to even more improvements [Campos et al., 2021].

The benefits of using a pre-trained model along with fine tuning were further backed up in the paper, "Never Stop Learning: The Effectiveness of Fine-Tuning in Robotic Reinforcement Learning". In this paper the authors again used fine tuning via off policy reinforcement learning, this time in regards to robotic behavior. Similarly to the first paper, this one also found that fine tuning pre-trained policies led to substantial performance gains, and that it was essential to utilize reinforcement learning in this fine tuning process. Furthermore, this also required less than 0.2% of the data required in learning a task from scratch [Julian et al., 2020].

3 Plasticity Injection

Recently, a paper called “Deep Reinforcement Learning with Plasticity Injection” by DeepMind’s Evgenii Nikishin et al. proposed a simple yet effective approach on improving the performance of reinforcement learning models [2023]. The paper states that over time, RL models do not learn as well on new data, likening the model’s “plasticity loss” to the loss of plasticity in human brains with age. The author proposes plasticity injection both as a diagnostic for detecting plasticity loss in models as well as remedying the issue. This method is useful since it increases the plasticity of the model while crucially not changing the number of learned parameters or the initial predictions after performing the injection.

To describe the plasticity injection method, let h_θ be the original model and its parameters. We will freeze h_θ and create two copies of $h_{\theta'}$, which has but same architecture but randomly initialized parameters. $h_{\theta'_1}$ and $h_{\theta'_2}$ are thus identical; we freeze $h_{\theta'_2}$ and will train $h_{\theta'_1}$ We use $h_\theta + h_{\theta'_1} - h_{\theta'_2}$ as the output prediction, meaning $h_{\theta'_1}$ is training by learning the residual of h_θ with noise.

The paper then goes on to provide further details and address certain concerns. For example, it is mentioned that the injection is performed on only the later layers in the model such that the model does not need to relearn the encoder representations. A diagram illustrating the plasticity injection architecture is shown below 1. Also, while the number of trainable parameters stays the same, the

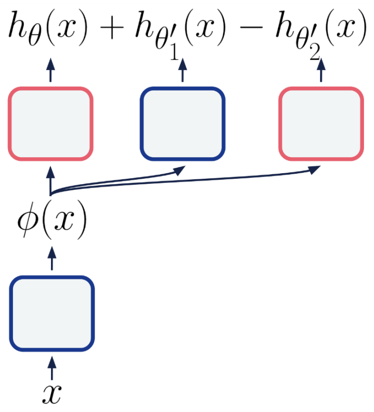


Figure 1: Diagram of the plasicity injection method; red boxes are parts that are frozen, while blue are still trainable.

total number of parameters is increase, which would increase training time. The author counters this by showing comparing the performances of two models: one large model, and a smaller model such that by performing plasticity injection on it would result in both having the same total number of parameters. The smaller model with plasticity injection performs better while being faster to train, proving plasticity injection to be both an effective and efficient way to fine-tune reinforcement learning models.

Our paper builds upon Nikishin's work by applying plasticity injection across a different range of games and algorithms. More specifically, while the initial paper performs injections on DQN and SAC, we apply it on algorithms that rely on the actor-critic model, such as Proximal Policy Optimization.

4 Methodology

4.1 Environment Setup

The main environment in which we wrote and executed our code was in Google Colab. There were some key advantages to using Colab for our project. One of the primary advantages was the ease at which we could share an environment across the entire team. The entire team could easily view and run the code in the same environment in the cloud. We could also take advantage of Google's accelerated hardware for Colab, enabling us to execute our code on Nvidia T4 GPUs. Another key factor was not having to strain our own hardware for multiple hours as we were training our deep learning agents. Using Colab also made it relatively simple to execute Python code from within a Jupyter Notebook ipynb file. We could also readily execute shell commands to set up the dependencies for the runtime environment that we were using. As part of that process, we were able to use git to manage the version control of our code, and interact with remote repositories on GitHub. An additional advantage to using Colab was the ability to directly mount our Google drive. This allowed us to be able to upload files directly onto our Google drive and access them within our runtime environment. Additionally, we could save our data onto Google drive when the runtime environment finished its execution. This allowed our data to persist even after the runtime environment was deallocated. This was essential to be able to later analyze the results of training our agents after multiple hours of background runtime execution.

4.2 Pre-trained Agents

A crucial component of our project required the use of pre-trained agents. Firstly, they served as a baseline with which we could compare to. We could evaluate how the unmodified pre-trained agents performed during training compared to those modified with plasticity injection. We did not want to completely have to train these agents from scratch, so to save time and runtime resources, we imported the agents from Hugging Face [4]. We did not use the models uploaded directly from

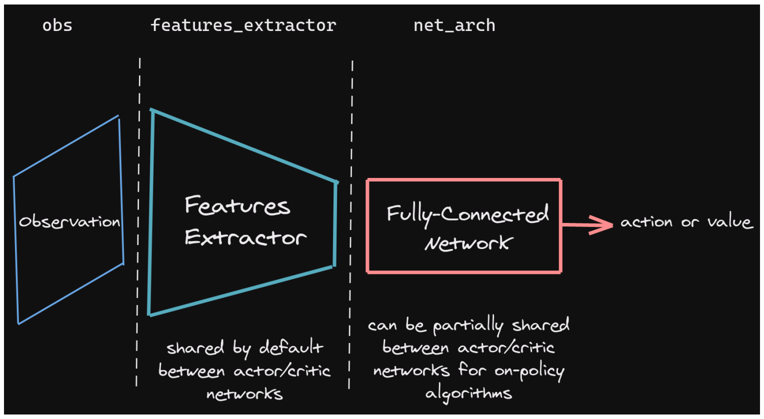

obs
features_extractor
net_arch
Observation
Features Extractor
Fully-Connected Network
action or value
shared by default between actor/critic networks
can be partially shared between actor/critic networks for on-policy algorithms

Figure 2: This is an overview diagram of the actor critic policy network courtesy of the Stable Baselines 3 documentation [5].

Stable Baselines 3 [5], which was the Python library that we were using. Rather, we used agents that were based on those initially provided by Stable Baselines, but were trained for longer periods of time. They still relied on the same interface and code base as Stable Baselines 3, so we could directly use the Hugging Face code provided by the library for importing pre-trained agents. This process involved directly retrieving a compressed zip file containing the execution state of the agent. It contained variables such as the hyperparameters used, the weights of the neural networks of the agent, the weights of the optimizer used, as well as other runtime variables. This enabled us to essentially directly continue training from where the pre-trained agent stopped. This also mimics how plasticity injection might be used in other contexts where an agent is paused during training and continues to train after plasticity injection is applied.

4.3 Actor Critic Policy

The main component of the agent that we needed to update for the implementation of plasticity injection was the actor critic policy. The actor critic policy network can be broken down into three major components as follows. For a brief visual overview, please see the diagram in Figure 2. First, observations are taken from the gymnasium environment. In the context of the Atari games with which we are working, these observations are 2-dimensional images of the environment. Then, these images must be pre-processed (via methods such as normalization) before they can be passed on in the network. After the image observations are processed, they are fed into the features extractor module. The features extractor module learns the key features from the observations. More details about the features extractor module will be explained below. Note that in the Atari environment context, the bulk of the learning of the agent takes place in the features extractor. This makes it an ideal target for the application of a plasticity injection. After the observations are passed through the features extractor, the linear output of the features extractor is passed to a fully-connected network module. The output of the fully-connected network is essentially the actor action distribution which must then be normalized and processed as well as the critic's value. There must be at least some independent layers here between the actor and the critic. In the case of the Atari agents, the fully-connected network consists of only a single output layer: one for the actor's actions and another for value of the critic.

4.4 Features Extractor Explained

The features extractor is responsible for learning the features from the observation input (i.e. an image) in our case. Thus, it is composed of a CNN submodule as well as a linear submodule that takes in the flattened output of the CNN layers. In order to appropriately apply plasticity injection, we must correctly freeze some of these layers and essentially learn a residual from noise. In our implementation, we decide to apply plasticity injection to the Conv2d layers and Linear layer as shown below. Note that we are only applying plasticity injection to the features extractor, not the mlp extractor (just an identity module in our case) nor the final output layers.

```
  ActorCriticCnnPolicy(
  (features_extractor): NatureCNN(
    (cnn): Sequential(
      (0): Conv2d(4, 32, kernel_size=(8, 8), stride=(4, 4))
      (1): ReLU()
      (2): Conv2d(32, 64, kernel_size=(4, 4), stride=(2, 2))
      (3): ReLU()
      (4): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1))
      (5): ReLU()
      (6): Flatten(start_dim=1, end_dim=-1)
    )
    (linear): Sequential(
      (0): Linear(in_features=3136, out_features=512, bias=True)
      (1): ReLU()
    )
  )
  (mlp_extractor): MlpExtractor(
    (policy_net): Sequential()
    (value_net): Sequential()
  )
  (action_net): Linear(in_features=512, out_features=6, bias=True)
  (value_net): Linear(in_features=512, out_features=1, bias=True)
)
```

4.5 Plasticity Injection Modifications

In order to be able to apply a plasticity injection to the model, we had to create our own custom features extractor which inherited from the base features extractor class provided by Stable Baselines 3. We only need one features extractor, since for the Atari environments the features extractor is shared by both the actor and critic by default. Note that this is essentially a PyTorch neural network module. Thus, it had to override the `init` method to provide the sub-modules/layers used as well as the `forward` method to implement the forward pass of the module. Importantly, we required that a pre-existing policy module is passed as an input parameter to the `init` method of our custom features extractor. This was to ensure that we could copy the architecture and parameter weights used in the old policy's features extractor within our new features extractor as required by the plasticity injection.

4.5.1 Plasticity Injection in Custom Features Extractor Initialization

To implement the plasticity injection, we copied the old features extractor twice to have a total of three copies. We keep the same parameter weights for the first copy. The second copy is then reinitialized with new parameter weights as required by plasticity injection. We decided to use the same weight initialization method that the is used during the original initialization of the PPO agent, i.e. orthogonal initialization is applied to all of the parameterized layers with a gain of $\sqrt{2}$. The second copy (after it has been re-initialized) is then copied over again to the third copy. We must then freeze the parameters of the first, original copy and the third copy, so they remain unchanged throughout the training process. Note that in our code, as shown in the code section below, we implement this process separately for the cnn and linear sub-modules of the features extractors.

4.5.2 Plasticity Injection in Custom Features Extractor Forward Pass

To implement the forward pass of our custom features extractor class, we must combine the three copies as follows. First, we pass the input through each of the copies independently. Then we take the output of the first, original, frozen copy and add the output of the second, unfrozen copy. Lastly, we subtract the output of the third, frozen copy and return the result. Thus, the forward pass stays true to the concept of the plasticity injection: it essentially learns the residual of the original features extractor over time. Please see the code section below for the exact details.

4.5.3 Plasticity Injection Final Comments

After making the custom features extractor, we updated the old actor critic policy network with an instance of our new class. Additionally, we also had to update the optimizer for the actor critic policy network since we are training new parameters that have been freshly initialized.

4.5.4 Plasticity Injection Code

```python
class PlasticCNN(BaseFeaturesExtractor):
"""
:param observation_space: (gym.Space)
:param features_dim: (int) Number of features extracted.
    This corresponds to the number of unit for the last layer.
"""

def __init__(self, observation_space: spaces.Box,
             old_policy: ActorCriticCnnPolicy, features_dim: int = 512):
    super().__init__(observation_space, features_dim)

    # copying over the feature extractor's cnn for plasticity injection
    self.cnn1 = old_policy.features_extractor.cnn
    self.cnn2 = copy.deepcopy(self.cnn1)

    # re-initializing the weights
    self.cnn2.apply(partial(old_policy.init_weights, gain=np.sqrt(2)))

    self.cnn3 = copy.deepcopy(self.cnn2)

    # freezing the parameters in cnn1 and cnn3

    for param in self.cnn1.parameters():
      param.requires_grad = False

    for param in self.cnn3.parameters():
      param.requires_grad = False

    # copying over the feature extractor's linear for plasticity injection
    self.linear1 = old_policy.features_extractor.linear
    self.linear2 = copy.deepcopy(self.linear1)

    # re-initializing the weights
    self.linear2.apply(partial(old_policy.init_weights, gain=np.sqrt(2)))

    self.linear3 = copy.deepcopy(self.linear2)

    # freezing the parameters in linear1 and linear3
    for param in self.linear1.parameters():
      param.requires_grad = False

    for param in self.linear3.parameters():
      param.requires_grad = False

def forward(self, observations: th.Tensor) -> th.Tensor:
    h1 = self.linear1(self.cnn1(observations))
    h2 = self.linear2(self.cnn2(observations))
    h3 = self.linear3(self.cnn3(observations))
    return h1 + h2 - h3
```

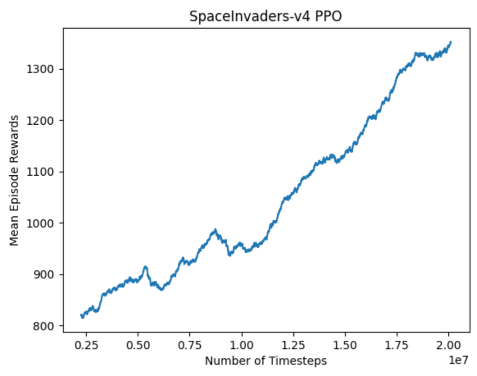
5 Experiment

5.1 Experimental Setup

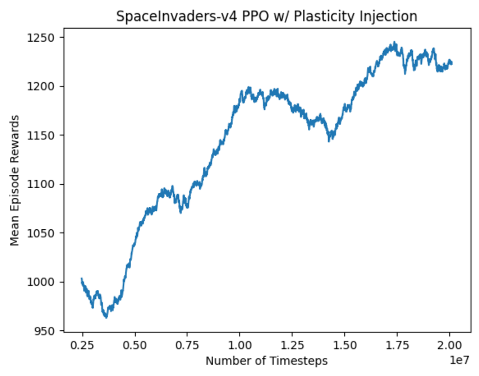
We imported three different pre-trained agents trained on three separate Atari environments: SpaceInvaders-v4, Enduro-v4, and Qbert-v4. We had two separate versions of each agent: one with the plasticity injection applied as discussed in the previous section and one without. We trained each agent for 20 million timesteps of the environment. Note that the length at which each pre-trained agent was initially trained varies across the environments, so it is difficult to directly compare the results of each Atari environment. However, we can make some general observations about the performance of plasticity injection by comparing the results of training with and without the plasticity injection. While training, we collected performance data: namely, the mean reward for each training episode across all of the parallel sub-environments used by the agent.

5.2 Result Graphs

In the following graphs, we plot the learning curve, i.e. the mean episode reward over the number of timesteps. We smooth the learning curve using a moving average. Note that for each environment we use a different window for the moving average, but the window remains consistent across each environment's two graphs.

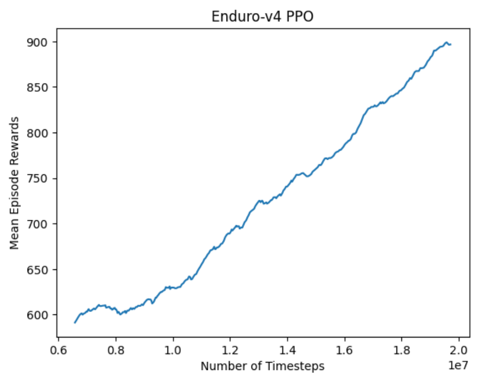


(a) without plasticity injection

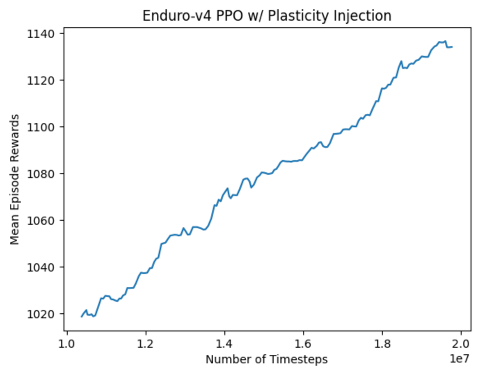


(b) with plasticity injection

Figure 3: The smoothed learning curves for SpaceInvaders-v4. Uses a window of 1000 episodes.

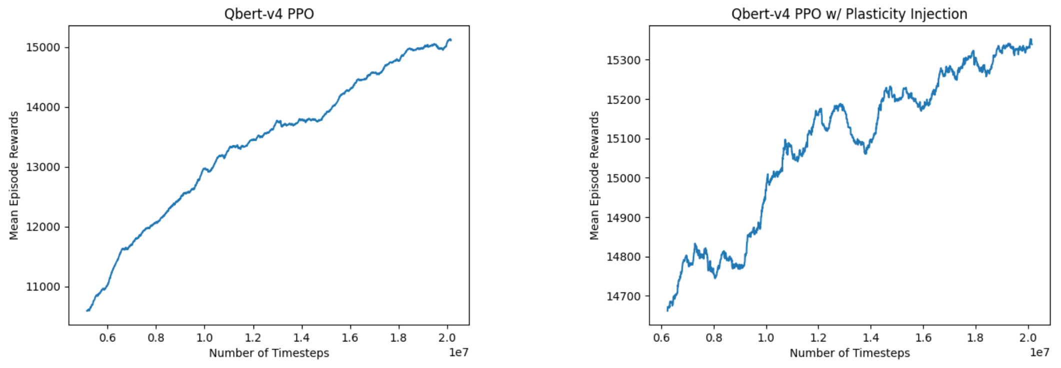


(a) without plasticity injection



(b) with plasticity injection

Figure 4: The smoothed learning curve for Enduro-v4. Uses a window of 500 episodes.



(a) without plasticity injection (b) with plasticity injection

Figure 5: The smoothed learning curve for Qbert-v4. Uses a window of 1000 episodes.

5.3 Result Analysis

We can make some general observations about the impact of plasticity injection based on the graphs in Subsection 5.2. In the SpaceInvaders environment, we see some marginal initial improvement in the mean episode reward; however, by the end of the training period, the agent without plasticity injection seems to catch up to the agent with the plasticity injection. However, in the Enduro environment, the plasticity injected agent realizes a much more significant initial improvement in the mean episode reward. Additionally, it maintains that gap in improvement to some extent throughout the duration of the training period. However, it is worth noting that the rate of increase in mean episode reward is higher for the baseline agent. It would be a worthwhile experiment to continue training to see if the baseline agent eventually catches up. Again in the Qbert agent, we see a significant initial improvement jump in the mean episode reward. And similar to the Enduro environment, it seems like the baseline agent rapidly catches up to the plasticity injected agent within the low 15000 mean reward range.

6 Conclusions

First of all, it seems that the effect of the plasticity injection depends significantly on the agent's environments and perhaps the duration of total training to an extent. In some environments, the plasticity injection can yield some significant initial improvement, but usually tends to level out relative to the baseline agent in terms of mean reward improvement rate. This may highlight a key limitation on the potential improvement afforded by plasticity injection to actor-critic based agents. Additionally, another factor to consider is the additional training time that plasticity injection incurs due to the increased number of total parameters, both frozen and unfrozen combined. In the SpaceInvaders, Enduro, and Qbert environments, the total training time increased by approximately 26%, 6%, and 8% respectively. Based on these conclusions, it may only make sense to apply plasticity injection to actor-critic agents that have already been trained for significant periods of time and have lost most of their plasticity. In such instances, plasticity injection could be used as a potential method of providing a quick boost to performance. Additionally, this may not work for all types of Atari environments, and agents trained in certain environments may benefit more than others.

6.1 Future Work

Our work thus far has shown promising results in utilizing plasticity injection to improve the performance of our models. In the future we plan to find ways to improve our performance even more. One possible way this could be done is by implementing the encoder-head idea and only perform plasticity injection on later layers, and see if that results in both better efficiency and performance. We also feel an obvious next step to this research would be expanding the scenarios we test on. Namely, we could evaluate how plasticity injection performs across other Atari environments as well as other actor-critic based algorithms such as A2C, DDPG, and TD3.

References

[1] V. Campos et al., “Beyond Fine-Tuning: Transferring Behavior in Reinforcement Learning,” arXiv (Cornell University), vol. 3, Jan. 2021, doi: https://doi.org/10.48550/arxiv.2102.13515.

[2] R. Julian, B. Swanson, G. S. Sukhatme, S. Levine, C. Finn, and K. Hausman, “Never Stop Learning: The Effectiveness of Fine-Tuning in Robotic Reinforcement Learning,” arXiv (Cornell University), vol. 2, Jan. 2020, doi: https://doi.org/10.48550/arxiv.2004.10190.

[3] E. Nikishin, J. Oh, G. Ostrovski, et al, “Deep Reinforcement Learning with Plasticity Injection,” 2023, arXiv:2305.15555.

[4] Q. Gallouédec, “PPO Agent playing SpaceInvadersNoFrameskip-v4,” huggingface.co, Feb. 27, 2023. https://huggingface.co/qgallouedec/ppo-SpaceInvadersNoFrameskip-v4-2978466957 (accessed May 04, 2024).

[5] A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus, and N. Dormann, “Stable-Baselines3: Reliable Reinforcement Learning Implementations,” Journal of Machine Learning Research, vol. 22, no. 268, pp. 1–8, 2021, Accessed: Jun. 28, 2023. [Online]. Available: https://jmlr.org/papers/v22/20-1364.html