Augmenting Deep Reinforcement Learning with Clustering

By

Hemanth Machavaram

Thesis
Submitted to the Faculty of the
Graduate School of Vanderbilt University
in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE

in

Computer Science
May 8, 2020
Nashville, Tennessee

Approved:

Douglas H. Fisher, Ph.D.
Yuankai Huo, Ph.D.

ACKNOWLEDGEMENTS

First and foremost, I am grateful to my advisor Dr. Fisher who has shown me unending patience as I stumbled through failed experiment after failed experiment. He has always had my best interests at heart and guided me towards my goals even when I couldn't see where I was going myself.

Secondly, I am grateful to Dr. Huo, who introduced me to the research frontier that is Deep Learning. He has always encouraged my curiosity and experimentation with different, sometimes a little wacky, Neural Network Architectures.

Thirdly, I am grateful for all my peers and friends, as it was only through discussions with them that my ignorance was laid bare.

Finally, last but certainly not the least, I would like to thank my family, for their love as well as for always believing in me, my aspirations, and my ability to achieve them.

Contents

Page

List of Figures **vi**

List of Tables **vii**

Chapter

1 Introduction **1**

2 Related Deep RL Algorithms **5**
2.1 Model-Free Algorithms 6
2.1.1 Advantage Actor Critic Model 7
2.2 Clustering in Deep RL 11

3 Environment and Task **14**
3.1 Environment 14
3.2 Task 14
3.2.1 Intuition for the Optimal Policy 16
3.3 Computing Environment 16
3.3.1 Hardware Specification 16
3.3.2 Software Specification 17

4 Baseline Method **18**
4.1 Variational Autoencoder 18
4.2 PPO-GRU 20
4.2.1 Network Architecture and Loss Function 21

5 Cluster Augmentation **23**
5.1 Cluster Formation 23
5.1.1 Informing Value Prediction 26
5.1.2 Informing Policy Prediction 26
5.2 Augmented Loss Function 28

6 Results **30**
6.1 Episodic Return 30
6.2 Cluster Formation 33

7 Conclusion and Future Work **38**

Bibliography **42**

A Appendix **46**
A.1 Hyperparameters for β-Variational Autoencoder: 46
A.2 Hyperparameters for the Baseline: 46
A.2.1 PPO specific hyperparameters: 46
A.2.2 Neural Network hyperparameters 47
A.3 Hyperparameters for Augmented PPO: 47
A.3.1 Augmented PPO specific hyperparameters: 47
A.3.2 Neural Network hyperparameters 48
A.4 Supplemental Files 49

List of Figures

1.1 Sample Inefficiency caused by using Neural Networks 3

3.1 Cart-pole Environment 15

4.1 Neural Network for Baseline 22

5.1 Neural Network with Cluster Formation 25

6.1 Avg. Return Comparison 30
6.2 Frames from rollout of a good policy 32
6.3 Good Clustering 34
6.4 Somewhat Good Clustering 34
6.5 Bad Clustering 35

List of Tables

6.1 Baseline and Augmented PPO Comparison 31

Chapter 1

Introduction

Reinforcement Learning is a branch of Artificial Intelligence that deals with learning over a delayed feedback. This means that at each given moment, an agent does not attempt to maximize its immediate reward, but the sum of total rewards it expects to get in the future. This is akin to how a chess master only cares about winning the game, and may sacrifice pieces in the short-term to gain a long-term advantage. RL algorithms learn the long-term effects of their actions by averaging the results of multiple simulations of the environment. The action chosen by an agent at any given moment is modeled as a random variable, where the underlying probability distribution is exactly what the agent sets out to learn. This probability distribution is called the policy and is usually denoted by $\pi(s)$, where s is the state. The action at a state is then sampled from the policy: $a \sim \pi(s)$. For complex environments and tasks (which comprise most of the real world) the policy function may become very complex, and so the method used to model this function must allow for arbitrarily complex functions. For this reason, in recent years, Neural Networks have come into use to model the policy function. This spawned the field of Deep Reinforcement Learning.

While there are multiple methods to train a Neural Network, such as the Backpropagation Algorithm, or various Evolutionary Algorithms, in general the former is used since the computational costs of the others are prohibitory. However, there is a fundamental incongruity between how the Backpropagation Algorithm works, and how RL agents learn. A Neural Network learning through the Backpropagation Algorithm requires immediate feedback for each input, whereas in RL the feedback at each moment should encompass all the rewards gained in the future as well. To work around this, the method of Experience Replay is used, wherein the agent initially generates a trajectory of an episode before going back and annotating each moment in the trajectory with the now available future reward information. The Neural Network is then trained on the annotated trajectory. This process of generating, annotating, and training trajectories (which will loosely be called an iteration of the Deep RL algorithm in this thesis) is then repeated multiple times until the optimal policy has been learned.

While Deep Reinforcement Learning is powerful in the sense that it can learn to solve relatively complex tasks, it suffers from large inefficiencies. One such inefficiency, known as Sample Inefficiency, is the focus of this research. Sample Inefficiency refers to the large number of trajectories an RL agent needs to learn over before it converges to an optimal policy. Part of this is due to the very nature of Reinforcement Learning, which was created to average over multiple trajectories so that the variance in learning caused by the inherent variance of visited states and received rewards in any individual trajectory is smoothed over. There is, however, another inefficiency introduced by the use of Neural Networks.

Although Neural Networks and the Backpropagation Algorithm are very powerful, and can learn any arbitrary real-valued[1] function, other than for specified inputs it is difficult to

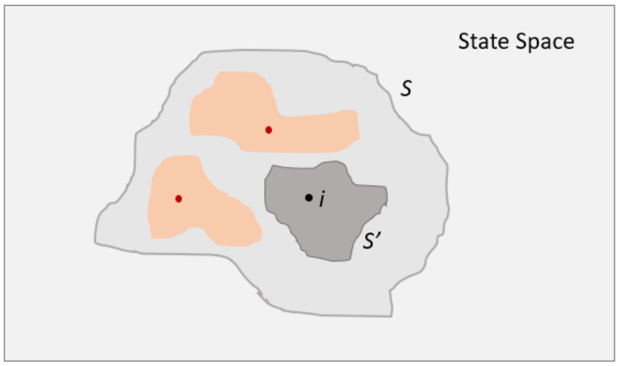

Figure 1.1: Sample Inefficiency caused by using Neural Networks

know what exactly is being learned. Moreover, for each given training input i, it is difficult to tell what region of the input space around i is being benefited by training on i. This means that if the region around i that should benefit from learning on i (say S) is larger than the region that actually benefits (say S'), then the algorithm would need to see multiple training samples in S to learn what could have been learned from just one training sample. This is pictorially represented in Figure 1.1 where multiple other inputs (red dots) are required to spread the learning throughout S.

To overcome this contributor to Sample Inefficiency, in this work the RL agent will be encouraged to generate a minimal set of clusters over the input space (here the state space of the RL task) such that whatever is learned for a training sample in a cluster is more widely learned for all points in the cluster. In this way, the number of trajectories required to reach the optimal policy should be decreased.

During the course of this research, a baseline method (Proximal Policy Optimization with

GRU-policy) was implemented before being augmented by a method to generate and learn over clusters. The PPO with clustering was shown to produce a policy with higher return per episode than the baseline. For the entirety of this thesis, return is defined as the sum of all the rewards obtained by an agent in an episode. Upon further investigation of the clusters being formed, it was found that the cluster formation itself was not very reliable, in that sometimes the Neural Network fell into local minima corresponding to degenerate clusterings. This will be discussed in more detail in section 6.2.

Throughout the rest of this thesis, first, an overview of related Deep RL algorithms will be given. Next, the environment and task will be defined, before the baseline algorithm is discussed, followed by an explanation of the method of cluster augmentation. Although the concept of cluster augmentation can be applied generally to all Deep RL algorithms and tasks, due to time constraints, this thesis only experiments with one type of algorithm, and one task. After the cluster augmentation method is explained, the results will be shown and discussed. At the end, a conclusion will summarize the work and briefly go over future directions for research.

Chapter 2

Related Deep RL Algorithms

Reinforcement Learning algorithms can loosely be broken into two categories: model-based and model-free. Model-based algorithms attempt to learn the dynamics of the environment, that is, they try to understand how states change as different actions are taken. Model-free algorithms on the other hand rely completely on sampled trajectories of the environment to associate a value (expected sum of future rewards from that state) with every state or state-action pair (q-value). An advantage model-based algorithms have over model-free algorithms is that once they've seen enough trajectories of the actual environment, they can build an internal model of the environment, and then train exclusively on this internal simulation. This means that after the internal model has been learned, they no longer need to sample trajectories from the actual environment. Training over trajectories sampled from an internal simulation is known as Hallucinating, and the sampled trajectories are called Hallucinations[2]. This drastically improves sample efficiency. However, if the learned internal model is incorrect, then the subsequent training will suffer and it will become more difficult, if not impossible, to find an optimal policy. Since the goal of this thesis is to improve sample efficiency, the focus will be on model-free algorithms, since they suffer from it the worst. Throughout the rest of this chapter, various model-free Deep RL algorithms will be discussed and the use of the Proximal Policy Optimization algorithm for the baseline will be motivated. Deep RL algorithms that incorporate clustering, whether explicitly or implicitly, will also be briefly discussed.

2.1 Model-Free Algorithms

One cannot talk about model-free RL algorithms without bringing up Q-Learning[3]. Though it was not a Deep RL algorithm, it inspired the Deep Q Network[4] (DQN), which sparked a flurry of activity in the Deep RL community. Q-learning attempts to find the optimal policy by associating each state-action pair with a value (called the q-value). This q-value is found by - as is the norm for model-free algorithms - running through the environment multiple times, making somewhat random actions - and then averaging the sum of future rewards for each state-action pair across all the sampled trajectories. Once the q-values have been learned, the optimal policy is constructed as choosing the action with the highest q-value at every state. A point to note here is that Q-learning was developed for environments with a discrete state space and discrete action space. This was then adapted to work for continuous (specifically image, but in general any) state space in the DQN. The DQN algorithm works by trying to minimize an objective on the q-value function: the difference between the network predicted q-values and the actual q-values. An alternative way to do Deep Reinforcement Learning is to maximize an objective on the actual policy function. Such methods are aptly called Policy Gradient methods. The actual objective to be maximized is derived from a maximization objective on the expected return of an episode, and is known to be an unbiased estimate of this expected return. It can be found in the seminal textbook

"Reinforcement Learning: an Introduction" by Sutton and Barto[5]. In this book, they also describe REINFORCE, one of the first policy gradient reinforcement algorithms. This is a very simple algorithm and amounts to using Monte-Carlo methods to sample trajectories and learn by trying to maximize the policy gradient objective. Since then, many policy-gradient algorithms have been proposed. Some of the more recent ones are the SAC[6], IMPALA[7], D4PG[8], MADDPG[9] (for Multi-Agent systems), SVPG[10], ACER[11] and many more. Among the large category of policy gradient methods, is the class of actor-critic algorithms that aim to both minimize the value objective as well as maximize the policy gradient objective. In doing so, these actor-critic algorithms tend to perform better than the algorithms which learn the value function or the policy function exclusively.

2.1.1 Advantage Actor Critic Model

The actor-critic model was first published as the Asynchronous Advantage Actor Critic (A3C) algorithm[12]. After which it was found that similar results are obtained with and without synchronicity[13]. This lead to the development of the A2C model: Advantage Actor Critic. Actor-critic models aim to reduce noise in the values calculated from trajectories by subtracting a baseline from the calculated values. This baseline is quite familiar and is just the predicted value for that state, which is calculated by the critic. This new quantity (after subtracting the predicted value from the actual value) is called the Advantage and can be interpreted as a measure of how good the chosen action was compared to the average goodness of that state. This makes sense because a positive advantage implies that the chosen action gave the state a better value than the predicted value of the state. Similarly a negative advantage implies that the chosen action resulted in a worse value than the average value of that state. The critic function learns to minimize the value objective, so that it accurately predicts the value function. Replacing the actual value along the trajectory in the policy gradient objective with an advantage formulated in this manner is known to keep the modified policy gradient objective unbiased[14]. The actor portion of actor-critic models corresponds to the part of the algorithm that aims to maximize this modified policy gradient objective. The realization of the actor and critic models is through Neural Networks, and can even be done with the same Neural Network splitting into two branches at the very end to predict the policy and value respectively.

There is an issue with the way Deep RL algorithms are trained though. As was laid out in the Introduction, one iteration of an algorithm consists of sampling a trajectory, annotating it, then training the Neural Network for a few epochs. However, it is possible that during this training, the network overfits and the learned policy is destabilized. Though this can be somewhat mitigated by a careful choice of how many epochs the neural network is trained for, this can take a lot of experimenting to find, and sometimes a single choice might not be suitable throughout the entirety of the training. To overcome this, Trust Region Policy Optimization [15] (TRPO) was introduced.

2.1.1.1 Trust Region Methods

Trust Region methods attempt to train a neural network while ensuring that during learning, the neural network doesn't learn to predict a policy that is outside of some region around the original policy which generated the trajectory.

To do this, TRPO introduced a new policy gradient objective, again based on maximizing the return of an episode, using Importance Sampling Theory from Monte Carlo methods that solve Reinforcement Learning tasks[5]. A similar formulation is done in the DPG algorithm[16]. Effectively, this replaces the log of the policy in the modified policy gradient objective with a ratio of the probabilities of picking the action in the trajectory according to the current policy, to picking it with the policy that generated the trajectory (this will hereby be referred to as just ratio or ratio of probabilities). The objective function to be maximized for each state and action pair is now just the product of this ratio with the advantage of the action at that state. A point to note here is that the gradients for the purpose of backpropagation are only generated for the current policy, and both the policy that generated the trajectory and the advantage are considered constants.

Crucial in this formulation of the objective function in TRPO is that it must be maximized while under the constraint that the new learned policy is within some Kullback–Leibler Divergence (KL Divergence) of the policy which generated the trajectory that is currently being trained on. The importance sampling estimation is used as a scaling factor to attempt to ensure that even though the policy function has changed after training a few epochs, training at that current epoch approximates what the Advantage of an action would be if it had been sampled with the policy function at that epoch. The KL Divergence constraint ensures that the network predicts a policy close to the original policy that generated the trajectory, which is a roundabout way of trying to keep overfitting down. However, in practice, updating a neural network under a hard constraint like KL Divergence can become computationally intensive, and so sometimes the constraint is relaxed somewhat by tagged a KL Divergence term to the Neural Network loss function. This can be interpreted as attempting to minimize the average KL Divergence throughout training, as opposed to ensuring that the maximum KL Divergence remains below a threshold. The calculation of the KL Divergence and its gradients can still be computationally intensive though, and it does not guarantee that the policy during learning remains within the Trust Region. This is where the Proximal Policy Optimization comes in.

The Proximal Policy Optimization Algorithm[17] (PPO) was developed to cut back on the computational load of the KL Divergence. This is done in a two step process. In the first step, they introduce what they call a surrogate loss function which sets gradients to 0 if the ratio of probabilities is outside some small region around 1 (called the PPO clipping hyperparameter η, and usually set to 0.2). The actual objective for maximization is now formulated as the minimum of this surrogate objective and the original objective. Through this, the authors ensured that if the advantage is positive, learning only occurs when the ratio is less than $1 + \eta$, and if the advantage is negative, the learning only occurs when the ratio is greater than $1 - \eta$. This ensures that learning only occurs if the policy hasn't already moved too far away from original trajectory generating policy. There is criticism however that this algorithm doesn't ensure the learned policy stays close to the original, just that if it is already too far away, no further learning will take place. Nevertheless, this algorithm is shown to do well on many complex RL tasks. A major drawback of this objective formulation is that there may be many wasted training epochs, dependent on what part of the task is currently being learned and for how many epochs the network is trained on per trajectory. In other words, this algorithm is known to be heavily sample inefficient. This is why the PPO was chosen as the baseline for this thesis. It is known to be a good algorithm, as well as be highly sample inefficient.

2.2 Clustering in Deep RL

Clustering has rarely been explicitly learned and used by a Neural Network in Deep RL. Usually, if clustering is present, it is done implicitly through the architecture of the Neural Network. One highly successful method for doing this is by learning a hierarchy of actions. Algorithms dedicated to doing this are called Hierarchical Deep Reinforcement Learning Algorithms.

Consider the task of operating the legs of an ant to navigate it through a maze. This task can loosely be broken into a hierarchy where the lower level involves learning how to move the ant in a certain direction, and the higher level learns the sequence of movements of the ant that solves the maze. This task was solved by creating an algorithm that learns to chain together low level actions (moving each individual limb of the ant) to learn high level behaviors (moving left, right, forward, and backward)[18]. The success of this algorithm is attributed to two key components: a hierarchical structure in the Neural Network, and training each level of this hierarchy at different time scales. The hierarchical structure allows for the higher level (the initial part of the Neural Network) to select which branch of the lower level (each of which corresponds to a behavior) should be activated at each time step. All the branches are implemented as a Neural Network with the same architecture, they just end up learning different policies. The differing time scales allows the higher level to focus on long term objectives, while the lower level attempts to perfect its individual, more granular policy (such as moving left in the task described above). A significant contribution of this paper was in showing how after the low level policies have been learned for a task, the agent can be trained to solve a different task in the same environment much more easily. That is, the learning could be transferred. In the context of the task described above, this means that after the agent has learned to solve one maze, at can solve another much more easily since it already knows how to move, it just needs to learn the higher policy which tells the agent in which direction to move. A similar hierarchical method uses differing time scales and an intrinsic reward to learn a high-level goal function[19]. This goal and the intrinsic reward are then used to train the low-level policy.

Another subfield of Deep RL where clustering has been used is in Imitation Learning. Imitation Learning (IL) refers to learning a policy through observing the actions taken by another policy that is assumed to be good. An IL algorithm that has experienced much success is the ILPO[20], and it does something akin to k-means clustering. The algorithm itself consists of two parts: relating state transitions to actions, and learning the good policy over these actions. In the first part, a Forward Dynamics model is trained, which given a state and action, predicts the next state. For each input state, this is done for all possible actions, and the network is trained on whichever action best predicted the next state. This is implicitly doing k-means clustering by first assigning an action to the closest cluster, and then moving the cluster closer to that action.

In this thesis, the clustering will be done more explicitly. An initial part of the network will be used (with its own recurrent structure) to classify every state into a cluster. The recurrent structure ensures that the clustering mechanism incorporates temporal information, so that the clustering is actually over the state space, instead of just the per time step observation space. This clustering will then be concatenated to the input observation and passed through the rest of the network (the usual A2C architecture) to do policy and value prediction. A shortcut connection will also be made between the clustering and the value prediction (explained in more detail in Section 5.1.1). This shortcut connection explicitly tells the neural network how to use the clustering to inform the value prediction.

Chapter 3

Environment and Task

3.1 Environment

Although the methods that will be described in Chapter 5 should work for any environment and task, in this thesis, the experimentation will be done on the Cart-pole Swing-up task. The environment and task themselves are available in the DeepMind Control Suite[21], which is a repository of continuous action tasks with a standardized return calculation and Python API, coded on top of the MuJoCo physics engine[22]. In all these environments, each episode consists of 1000 transitions, where each transition refers to the process of the agent supplying an action to the environment, the environment updating its state according to the laws of physics, and returning the observations and reward to the agent. The observations for this task are the angular velcoity of the pole and the velocity of the cart. However, in this thesis, instead of using these observations, the direct image of the environment will be passed to the agent. This compounds the difficulty of the problem while bringing it closer to a real-world task.

3.2 Task

The Cart-pole Swing-up task is a continuous control task wherein the agent is allowed to apply a force (between -1 and 1) to the cart along the x-axis in order to swing the pole

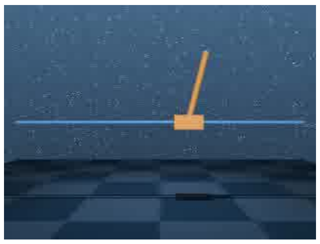

Figure 3.1: Cart-pole Environment

hinged to the center of the cart up until it is vertically balanced above the cart. The agent must then move the cart to ensure that the pole remains balanced above the cart. An image of the task is shown in Figure 3.1. The task begins with the pole vertically below the cart.

In this task, the reward provided at each state continuously varies from 0, when the pole is vertically below the cart, to 1 when the pole is vertically above the cart. Specifically, if θ is the angle made by the pole with the upward vertical, then the reward is calculated as $\frac{1}{2}(cos(\theta) + 1)$.

An important note to make here is that since the observation sent to the agent at each time step is just a single image of the environment, the agent would not be able to obtain any velocity or angular velocity information. However, obviously this information is required for the agent to solve the task. For example, knowing if the pole is moving up or down requires images from previous time steps, and is vital for the agent to decide which way the cart needs to be moved. This means a method to encode temporal information across time steps of the environment is required. In this thesis, this will be done through the use of an RNN structure, specifically, the Gated Recurrent Unit as proposed in [23].

3.2.1 Intuition for the Optimal Policy

The goal of the Cart-pole Swing-up task is to swing the pole above the cart, and then balance it for as long as possible. To accomplish the swing-up, the cart must be given a strong impulse in one direction, say the right. This would cause the pole to swing clockwise. Once the pole is above the cart, the agent must provide an impulse in the opposite direction, to further swing the pole towards the vertical, and bring the cart to rest. Once the pole is near the vertical, the agent must provide small impulses to keep the pole balanced. For example, if the pole is at the vertical, but seems to be tipping towards the left, the cart must be moved slightly to the left to swing the pole back to the vertical. The task eventually ends after 1000 transitions have been made, and the sum of the rewards across all transitions is considered the return of the episode.

3.3 Computing Environment

Since this work utilizes Neural Networks and the GPU through CUDA APIs, it is important to note the exact configuration of the hardware and software set up. This will be laid out throughout the rest of this section, and this same configuration will be used for implementing both the baseline and clustering augmented algorithm.

3.3.1 Hardware Specification

The processor on the system is the Intel Core i7-8750H @2.2GHz (9M cache, up to 4.10 GHz), the Graphics card is the NVIDIA GeForce 1070 (8GB GDDR5), and the RAM available is 16GB DDR4 (2400MHz).

3.3.2 Software Specification

The operating system on the machine is Windows 10 home addition. The Deep Learning framework that was used is PyTorch (Python) version 1.1.0, while the CUDA version is 10.0

Chapter 4

Baseline Method

In this chapter, the baseline algorithm will be described, which consists of an initial β-Variational Autoencoder[24], followed by the actor-critic network for the Proximal Policy Optimization algorithm[17].

4.1 Variational Autoencoder

The observation available to the RL agent at every time step is the image of the environment at that time step. These images are high-dimensional and highly-correlated. Due to the high-dimensionality of images, a Neural Network attempting to learn a policy with an image as input will take a very large amount of training data. This is because the effective feedback given to the neural network for each input image is the value of the state represented by the image and the goodness of the chosen action; both of which have high variance since they are calculated from a single trajectory. This makes it difficult for the neural network to learn the feature extractors it needs to decode the high-dimensional image, which in turn contributes to sample inefficiency. However, if there were a method available to quickly learn feature extractors that preserve all the semantic information in the image, and feed just this low-dimensional semantic vector to the RL agent, then we can feel comfortable that all the relevant information is being passed to the agent, and the agent does not need to spend a great number of trajectories learning with high-variance feedback to create the required feature extractors.

The Autoencoder is just such a method. Through the network architecture design, the autoencoder can be forced to condense a high-dimensional highly-correlated input into a low-dimensional weakly-correlated semantic vector, which is then once more forced to expand into the high-dimensional highly-correlated input. The expansion process is used to enforce that the semantic vector encodes all the information in the input, since otherwise, the input could not be reproduced.

Now, if an autoencoder is trained to encode the image observations, and the encoded semantic vector is used as the input to the RL agent network, then the RL agent will not need to learn image feature extractors at all. This is exactly what was done in [2]. In this thesis, a similar approach is followed, and a β-Variational Autoencoder[24] is used to encode the image observations. The network architecture is the same as in [2] and the hyperparameters are available in the appendix. The β-Variational Autoencoder was initially trained on a training set of 10,000 images generated by making random movements on the task environment. The network was trained for 50 epochs, though it was found that 10 were usually enough. The encoded semantic vector had a dimension of 32. The same trained β-Variational Autoencoder network was used for both the baseline and augmented methods. Furthermore, since it is entirely possible that as the RL agent learns it sees new parts of the observation space that the β-VAE did not initially train on, during each iteration of the PPO algorithm, the β-VAE is first trained on the sampled trajectories for 1 epoch. The other hyperparameters are the same as for when it was initially trained.

4.2 PPO-GRU

The baseline Deep RL algorithm that is used in this thesis is the Proximal Policy Optimization Algorithm[17]. This algorithm is an improvement over the Trust Region Policy Optimization Algorithm[15], and is based on the A2C algorithm model[13], which in turn is based on the A3C algorithm[12].

The Proximal Policy Optimization algorithm was chosen because it is considered competitive and one of the best Deep RL algorithms out there today. Furthermore, Sample Inefficiency is a known drawback of the PPO. Even for this specific task, PPO finds some difficulty since the agent must first luckily stumble upon the exact sequence of actions it must take (move the cart one way to swing the pole above the horizontal, and then move the other way to swing the pole further from the horizontal to the vertical) over multiple different episodes, to be able to learn the optimal policy. This is unlike the D4PG Algorithm[8] which through a prioritized experience replay buffer, is able to train over good sequences of actions across multiple episodes.

As mentioned in Chapter 2, a GRU will be used to encode temporal information into the state. In addition to the usual PPO algorithm, in the baseline, the advantages are first calculated through the Generalized Advantage Estimation[25] (GAE) and are then normalized across all the episodes in each iteration of the algorithm so as to stabilize training.

4.2.1 Network Architecture and Loss Function

The network architecture is shown in Figure 4.1. Each arrow corresponds to a layer. Every linear layer is followed by a ReLU nonlinearity. The initial shaded rectangle represents the image input (depth for 3-channels), and all subsequent rectangles represent some tensor. Below each rectangle is a number specifying the dimensions of the tensor. If there is a letter or word above the dimension number, this specifies the name of the tensor. As can be seen from the image, there are three output tensors from the network. The first two correspond to the actor portion of the A2C which is supposed to predict a probability distribution. As is common practice for continuous valued actions, the probability distribution is modeled as a Gaussian, with the parameters μ and σ^2 obtained from the network as $\mu =$ mean, $2log(\sigma) =$ logvariance. The log of the variance is predicted instead of the variance itself to improve numerical stability. The last output tensor corresponds to the critic portion of A2C, and predicts the value of the current state.

The loss function for the actor $L^{CLIP}(\theta)$ is exactly the same as the one in the original PPO paper[17], where θ represents the network parameters. The critic loss L^{critic} is formulated as the L^2 loss between the value predicted by the network, and the exponentially-weighted sum of rewards returned by the GAE. Finally, an entropy regularization loss L^{entr}[6] is also used. The total loss function is thus:

$$L^{baseline} = L^{CLIP} + (critic\_coef)(L^{critic}) + (entr\_coef)(L^{entr}) \tag{1}$$

where $critic\_coef$ and $entr\_coef$ are hyperparameters.

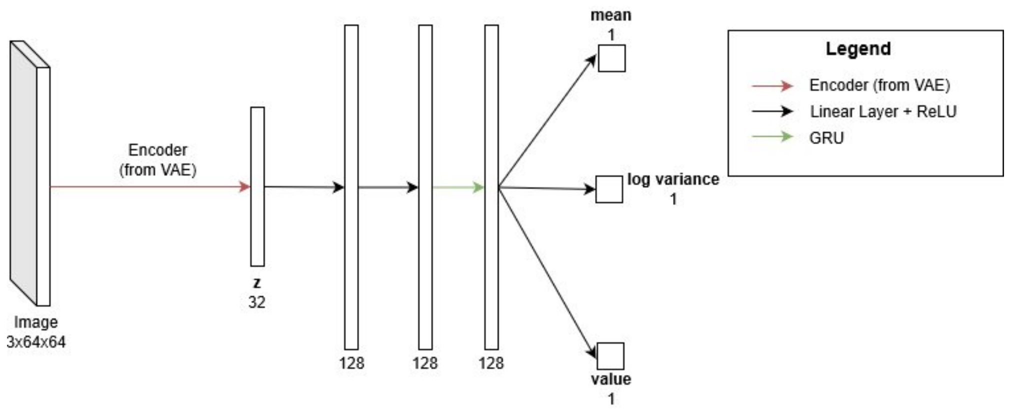

Figure 4.1: Neural Network for Baseline

All the hyperparameters used for this baseline PPO implementation are listed in the appendix.

Chapter 5

Cluster Augmentation

5.1 Cluster Formation

The hypothesis behind this research is that if the RL agent can be encouraged to create a minimal set of clusters over the state space, and more generously distribute knowledge it learns for any single input across the entire cluster to which it belongs, then the RL agent should be able to learn the optimal policy with lesser training data. As briefly discussed in Section 2.2, the main components of the algorithm are an initial clustering done with a recurrent neural network over the per time step observations, followed by concatenation of this clustering to the observations before passing it onto the rest of the network (the usual A2C architecture). A shortcut connection is also made between the clustering and the value prediction to more explicitly enforce the relationship between the clustering and the value of a state. Throughout the rest of this chapter, the exact mechanisms to form and use the clustering will be motivated and explained.

The first step in this process is obviously the cluster formation itself. How exactly do we want the clusters to be formed? In the A2C model, we would want our clusters to inform the rest of the agent about which regions in the state space require similar actions to be taken, and should have similar values. This means that we are carving up not the input space, but the state space into clusters. The difference between the two being that the input to the Neural Network is just a single image, whereas the state space also contains temporal information. From this, we can conclude that whatever method is used to generate the clusters, it must also consider temporal information.

Since we must include temporal information, the immediate conclusion we might draw is that the output of the GRU in the baseline might be used as the input to the cluster generation method. However, this would not work for our ultimate goal since we want to augment the current observation with the cluster information to generate our policy and value. We could not do this if the cluster itself is formed at the end of processing the observation input. This creates somewhat of a chicken and egg problem, wherein we want the GRU output to generate the clusters, and we want the clusters to inform the GRU in generating the output. To avoid this, we can have the cluster formation occur through a second path branching from the input, and passing though its own GRU. The output of this clustering GRU would be a probability vector wherein each index of the vector corresponds to a cluster, and the value at that index is the probability that the current state belongs to the cluster of that index. The reason for returning a probability vector instead of a hard clustering is so that eventually the loss can be backpropagated through the probability vector.

The actual probability vector is formed by taking the output of the GRU, passing it though a linear layer and Sigmoid nonlinearity, followed by dividing each element of this vector by the sum of all elements of the vector. This enforces that the vector represents a valid categorical probability distribution. As an implementation note, to avoid dividing

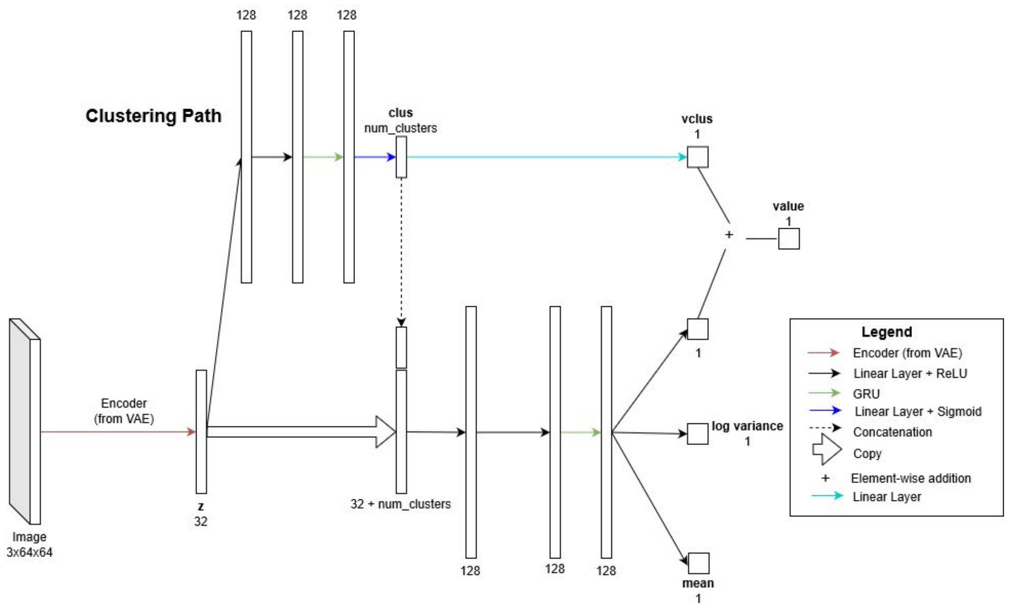

Figure 5.1: Neural Network with Cluster Formation

by 0, 0.000001 was first added to each element of the clustering GRU output. In this implementation, the number of clusters is a hyperparameter.

Now that we have generated our clustering, the next step is to utilize this clustering to inform the rest of the network in its prediction of a policy and value.

A point to note here, is that although the cluster formation and utilization is being developed for the A2C architecture, the method of forming clusters is generic to most Deep RL architectures. It is only the method in which the clusters are used to inform the policy that might need to be adapted.

5.1.1 Informing Value Prediction

In the context of value prediction, our original interpretation of clustering was that it would split the state space into regions such that the value of every point in a region is roughly the same. One way to model this is by assuming an average value for each cluster, and then predicting the offset from this average for each state in the cluster:

$$V(s) = V(c) + V(s|c) \tag{2}$$

where $V(s)$ is the value at state s, $V(c)$ is the average value of a cluster c, and $V(s|c)$ is the offset of the value of the state from the average value of the cluster

This then becomes very easy for us to incorporate into a neural network by generating $V(c)$ through the addition of a single linear layer after the probability vector. This layer would then predict a single number (the average value or vclus in Figure 5.1) and could be trained by using an L^2 loss between vclus and the actual sum of rewards returned by the GAE. Let this loss be called L^{clus_value}. As an implementation detail, the network seemed to learn better if the gradients generated from L^{clus_value} were not allowed to flow into the cluster formation network.

5.1.2 Informing Policy Prediction

Using clustering to inform policy prediction isn't as cut-and-dry as it was to inform value prediction. This is because the policy is itself a probability distribution, and so an analogous interpretation of average probability distribution and offset probability distribution might not make much sense. In such an average-offset formulation, the final policy distribution would be the sum of two Gaussian distributions. While this is known to be Gaussian itself, the variance of the summed distribution would be the sum of the variances of the individual distributions. In Deep RL, the variance of the predicted policy is an important scaling factor during learning, and can be interpreted as the confidence the algorithm has in its policy prediction. Thus, saying that the variance of the summed policy is larger than the variances of the individual distributions from which it was composed is equivalent to saying that the model had more confidence in predicting the average cluster policy and the offset distribution than the final predicted policy. This clearly doesn't make much sense. If anything, the offset distribution should function to improve the confidence the model has on the average cluster policy.

An alternative at this point is to use the cluster not to generate an average probability distribution, but a shifting parameter. This shifting parameter can then be added to the policy predicted in the usual way and can be interpreted as shifting the mean of the policy distribution. However, when this was tried, the results didn't change in any significant way.

In the end, the method adopted in this thesis is to take the probability vector, and concatenate it to the latent observation vector z before passing it onto the normal A2C network. This can be seen in Figure 5.1 as a dashed arrow. This way, the policy and value predicted by the A2C network can be thought of as the policy and value conditioned on the cluster. This architecture also allows for the gradients from the PPO Loss L^{CLIP} and L^{critic} to inform the cluster formation itself. Roughly speaking, this would ensure that the network forms clusters in such a way as to improve its overall performance. This might not be the best way to inform the cluster formation however, and an explicit loss function for cluster formation might provide better results. Nevertheless, this was the implementation used in this thesis.

5.2 Augmented Loss Function

Since the augmented Neural Network was formed on top of the baseline A2C network, the loss function is also built on top of the baseline loss function $L^{baseline}$. As was discussed in Section 5.1.2, the gradients for cluster formation are generated from L^{CLIP} and L^{critic} themselves, so nothing new must be added for that. In Section 5.1.1, we stated that the loss needed to learn vclus was L^{clus_value}.

During the course of training, it was found that using a regularization loss for the cluster formation helped in keeping the network from falling into a local minimum. The regularization loss is constructed by first squaring every term of the probability vector, normalizing this squared vector so that the sum of all elements is 1, and then using this normalized squared vector as the target of an L^2 loss between the original probability vector and the normalized squared vector. It was found during training that multiplying this L^2 loss with the advantage of that state further helped in avoiding local minima. Let the regularization loss constructed in this way be called L^{clus}.

Finally, the loss function used to train the Augmented PPO network is:

$$L^{aug} = L^{baseline} + (cval\_coef)(L^{clus\_value}) + (clus\_coef)(L^{clus}) \tag{3}$$

where $cval\_coef$ and $clus\_coef$ are hyperparameters

All the hyperparameters that were used are included in the appendix.

Chapter 6

Results

Both the baseline and the augmented method were implemented as described in Chapters 4 and 5 respectively. The computational environment was described in Section 3.3 and the other hyperparameters that were used have been listed in the appendix.

6.1 Episodic Return

The goal of any RL agent is to maximize the sum of the rewards obtained throughout an episode. In Figure 6.1 we show a graph of the baseline (in red) and the augmented network (in blue) during training. The x-axis corresponds to the training episode number, and the

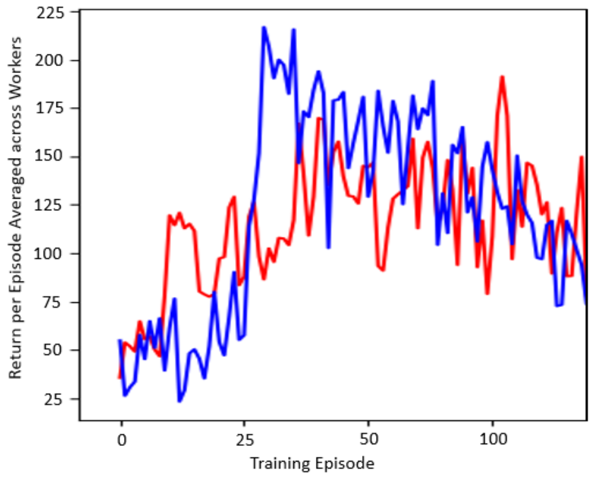

Figure 6.1: Avg. Return Comparison

y-axis corresponds to the sum of rewards obtained in that episode averaged across all the training workers. The best return for the baseline and the augmented method are tabulated, along with the corresponding training episode, in Table 6.1.

Algorithm	Largest Return	Episode Number
Baseline	191	77
Augmented PPO	217	30

Table 6.1: Baseline and Augmented PPO Comparison

From both the graph and the table, it can be seen that the clustering augmented Neural Network achieved a higher maximum return at an earlier training episode. Specifically, the clustering Neural Network had a peak 13.6% higher than the baseline, after seeing only 39% as many training episodes.

A video of the policy at the peak of the Augmented Network was generated and every 20th frame was extracted and placed in Figure 6.2. The number below each frame corresponds to the frame number in the video. From the figure, it can be seen that the agent has learned to swing the pole up above the cart - and does this multiple times - but is not able to balance it indefinitely. It does manage to balance the pole for a good portion of the video though, from frame 160 to 260 and 600 to 800. This is 300 frames out of 1000, and is considered as solving the Cart-pole Swing-up task. This is clearly not the optimal policy though, which was described in section 3.2.1.

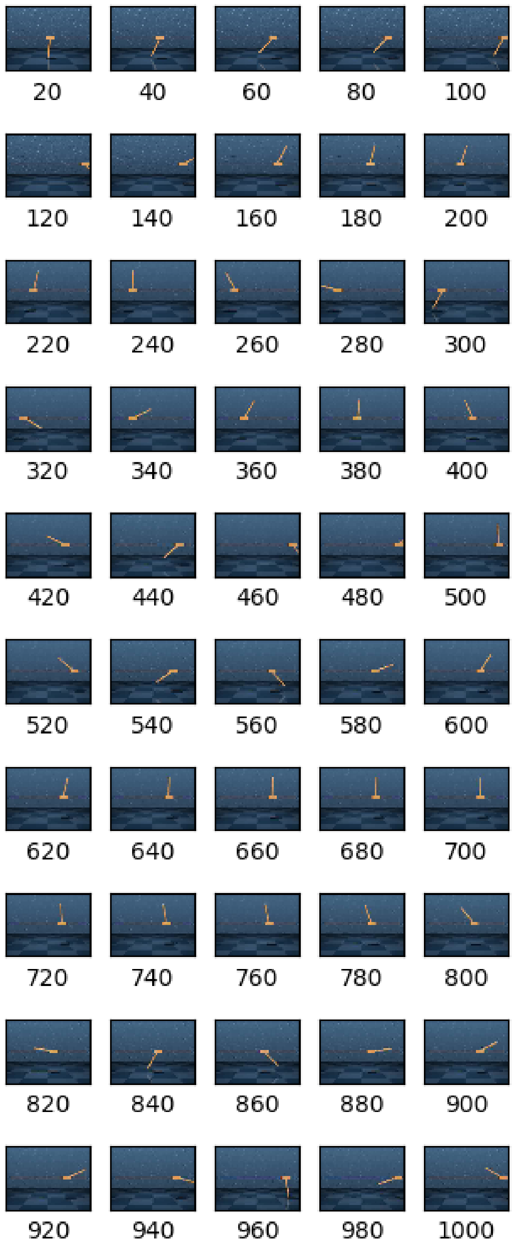

Figure 6.2: Frames from rollout of a good policy

6.2 Cluster Formation

The Augmented Network clearly got better results than the baseline, but does this mean that it is making diverse, informative clusters? To investigate this, extra information that the DeepMind Control Suite[21] provides about the state of the agent is used. Although in this thesis the actual image of the environment is used as the observation sent to the RL agent, the DeepMind Control Suite offers us alternate observations in the form of the velocity of the cart, and the angular velocity of the pole. In fact, these velocity and angular velocity observations contain all the information required for the agent to learn. This means we can use these alternate observations to verify the quality of the formed clusters.

As was explained in Chapter 5, the actual gradients used for learning the clustering neural network come through the final policy and value predictions, through the A2C network. This means that the cluster formation is driven by just the RL objective and a critic loss. Consequently, the network is encouraged to form clusters based on the loose objective that the formed clusters must be helpful in obtaining high return. Although this is the best description of what we would want the policy prediction of a generic RL algorithm to do, it may be too vague to drive the network to make good clusters. This subsection investigates whether the current method of informing cluster formation seems sufficient to consistently and reliably make good, informative clusters.

Figures 6.3, 6.4, and 6.5 give examples of a good clustering, a somewhat good clustering, and a bad clustering respectively. Each row corresponds to a cluster, the first column shows

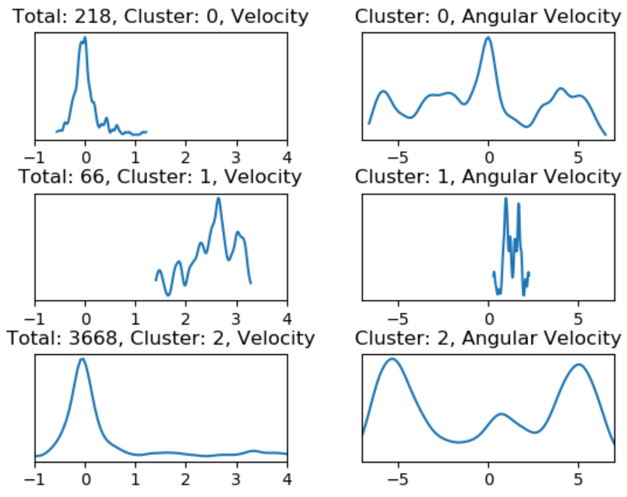

Figure 6.3: Good Clustering

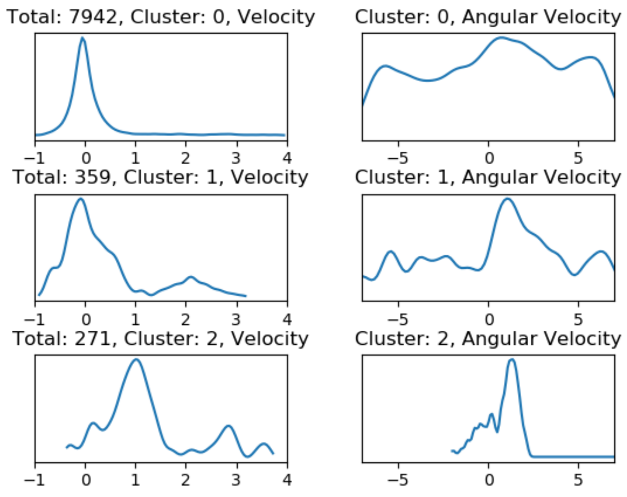

Figure 6.4: Somewhat Good Clustering

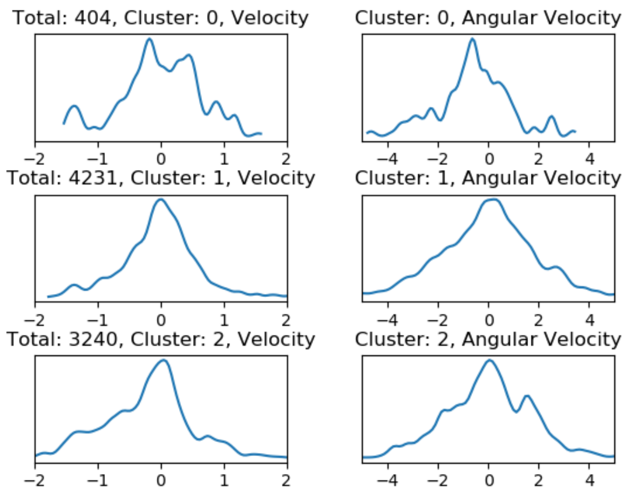

Figure 6.5: Bad Clustering

the distribution of cart velocities for all states in that cluster, and the second column does the same for the pole's angular velocity. The total number specifies how many states in the sampled trajectory were assigned to that cluster. A state was considered to belong to a cluster if the probability of that state belonging to that cluster was at least $\frac{1}{2}$. This threshold was chosen somewhat arbitrarily, with the loose justification that if a state has greater than $\frac{1}{2}$ probability of being in one cluster, then there is no way the probability of it being in another cluster is larger, irrespective of how many clusters were formed. This has the drawback that some states will not be assigned any cluster, unlike in an alternative assignment scheme wherein the state is assigned to the cluster for which it has the highest probability. There is an issue with this alternate scheme as well though. It occurs many times that the network predicts a state to exist in multiple clusters with about equal probability. In this situation, it doesn't make much sense to assign a state to one cluster if the probability of being in another cluster is just slightly lesser. For these reasons, $\frac{1}{2}$ was taken as a hard threshold. The number preceded by Total associated with each cluster refers to how many states were assigned to that cluster.

Figure 6.3 is considered a good clustering because the plots for all three clusters are so different. Just from inspection, one can intuit that the first cluster (cluster 0 in the figure) seems to represent the case when the pole is above the cart and the agent is attempting to balance it there. This is because the velocity of the cart is almost always zero, and the angular velocity of the pole is mostly zero, except for possibly when the pole is falling out of balance. The argument can be made that this cluster could also be representing the case when the pole is below the cart and the cart is trying to swing it up. However, knowing that the policy is unable to balance the pole above the cart for long, the total number of states assigned to this cluster seems to imply that the cluster represents the balancing case and not the swinging up case. The second cluster formed (cluster 1 in the figure) seems to correspond to the situation where the cart is moving and the pole is in the process of swinging up. The third cluster (cluster 2 in the figure) seems to represent the situation when the pole is rotating quickly in circles around the cart because even though the impulses the agent applied to the cart eventually caused the cart to come close to rest, they were so strong that the pole just kept rotating instead of slowing and stopping near the top.

Figure 6.4 on the other hand is a somewhat good clustering because although the third cluster (cluster 2 in the figure) seems to represent the situation when the cart is moving and the pole is slowly being swung up, the first two clusters (clusters 0 and 1 in the figure) are degenerate and seem to represent roughly the same scenario.

Finally, Figure 6.5 is completely degenerate since all the plots look the same. Clearly the Neural Network has fallen into the local minimum of predicting all the clusters are equivalent. This isn't the only type of degeneracy observed though. Sometimes the Neural Network predicts that all the states belong to a single cluster.

From these graphs, it can be seen that although the algorithm is capable of generating diverse clustering, it is still prone to destabilizing and collapsing into degenerate clusters.

Chapter 7

Conclusion and Future Work

Although Deep Reinforcement Learning is a very powerful technique for learning to solve tasks approaching Real-World complexity, due to the inherent nature of Reinforcement Learning and Neural Networks, Deep RL suffers from Sample Inefficiency. A major contributor to this sample inefficiency can be found in how when a Neural Network trains through the Backpropagation Algorithm, there is no guarantee that the entirety of the region of the input space that can benefit from a training example, actually benefits from training on that example. Due to this, the RL agent may need to repeat similar trajectories multiple times, thus exorbitantly increasing the training data and time.

In this thesis, clustering was suggested and implemented as a method to force the neural network to more equally distribute knowledge across the entire region which should benefit from training on an example. The PPO algorithm (described in Chapters 2 and 4) was chosen as a baseline since it is known to suffer from sample inefficiency, and it was augmented with clusters (described in Chapter 5) in the hopes of decreasing this inefficiency and improving the performance of the RL agent as a whole.

In Chapter 6, the baseline and the augmented algorithms were compared and it was shown that the augmented algorithm managed to produce an agent that reached a higher episodic return at an earlier episode than the baseline agent. This shows that the augmented algorithm used a lesser number of samples to learn a better policy than the baseline. This was however only shown for a single task, with a single baseline algorithm. In the future, work should be done to verify that these gains are not environment, task, or baseline algorithm specific.

Some immediate environments and tasks to test this algorithm on would be other tasks available in the DeepMind Control Suite[21]. Of these, tasks of particular interest would be the humanoid-walker or cheetah-runner tasks. These tasks attempt to simulate the real-world tasks of walking for bipedal and quadrupedal robots. Combined with grasping tasks, this would allow robots to replace humans in hazardous jobs, such as roofers, or any task requiring the need to get to high places in construction. They could also be used in dangerous situations, like in fossil fuel mining and extraction. Theoretically, these robots could replace warehouse workers as well. However, solving atomic tasks in a simulated environment is a far cry from being able to deal with tasks in the real world, and we are still at least a few years away from reaching that level of competency from robots, if not decades.

An important consideration to make here is also the mass unemployment that would be caused by indiscriminately replacing humans with robots in these jobs. Such unemployment could drive an already lower-middle class or poor family into destitution. This illustrates the need for careful regulations and reforms to be put in place to ensure that robots are used only in those circumstances that constitute a possibility of bodily harm or death. This is necessitated not by any failing in the robots themselves, but due to the economic structure of our society. On a more positive note, robots trained to do basic cleaning tasks could be a huge help for the elderly who are living on their own. These examples highlight the potential good that Deep RL algorithms can do in our society, as long as they are deployed responsibly to improve the quality of life of workers, and not to outright replace them.

Returning to the work done in this thesis, after comparing the results of the baseline and augmented algorithms, the actual clusters that were formed were investigated. It was found that sometimes, what appeared to be diverse cluster formation occurred, while other times, the clusters were all very similar to each other. This shows that the current cluster formation scheme is not sufficient to reliably form diverse clusters. In the future, more experimentation could be done in developing a loss function specifically to drive informative cluster formation. The development of this loss function could go hand-in-hand with the development of a better method to augment the actual Neural Network architecture.

Another possible avenue of future research could involve using the Gumbel-Softmax Reparameterization trick[26] to form a harder clustering while still being able to backpropagate gradients through it. While this might force the Neural Network to form more diverse clusters, it would also introduce another hyperparameter and thus complicate the training further.

Finally, interpreting the formed clusters as a semantic vector offers another direction of research. The versatility of human reasoning comes from the ability to abstract away low-level details and apply logic or heuristics to high-level semantic categories. If it is possible to devise an algorithm that interprets these clusters as abstracted state representations, learns high-level actions to move across these abstracted states, and learns what low-level actions and behaviors comprise those high-level actions, then this algorithm would be much better equipped to plan and solve tasks it never trained on. A very good environment to test this in would be the AnimalAI-Olympics[27], which provides tasks to test RL agents for basic animal cognitive functions.

Bibliography

[1] Moshe Leshno et al. "Multilayer feedforward networks with a nonpolynomial activation function can approximate any function". In: *Neural networks* 6.6 (1993), pp. 861–867.

[2] David Ha and Jürgen Schmidhuber. "World Models". In: *CoRR* abs/1803.10122 (2018). arXiv: `1803.10122`. URL: `http://arxiv.org/abs/1803.10122`.

[3] Christopher JCH Watkins and Peter Dayan. "Q-learning". In: *Machine learning* 8.3-4 (1992), pp. 279–292.

[4] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518.7540 (2015), pp. 529–533.

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

[6] Tuomas Haarnoja et al. "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor". In: *arXiv preprint arXiv:1801.01290* (2018).

[7] Lasse Espeholt et al. "IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures". In: *CoRR* abs/1802.01561 (2018). arXiv: `1802.01561`. URL: `http://arxiv.org/abs/1802.01561`.

[8] Gabriel Barth-Maron et al. "Distributed Distributional Deterministic Policy Gradients". In: *CoRR* abs/1804.08617 (2018). arXiv: `1804.08617`. URL: `http://arxiv.org/abs/1804.08617`.

[9] Ryan Lowe et al. “Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments”. In: *CoRR* abs/1706.02275 (2017). arXiv: 1706.02275. URL: http://arxiv.org/abs/1706.02275.

[10] Yang Liu et al. “Stein Variational Policy Gradient”. In: *CoRR* abs/1704.02399 (2017). arXiv: 1704.02399. URL: http://arxiv.org/abs/1704.02399.

[11] Ziyu Wang et al. “Sample Efficient Actor-Critic with Experience Replay”. In: *CoRR* abs/1611.01224 (2016). arXiv: 1611.01224. URL: http://arxiv.org/abs/1611.01224.

[12] Volodymyr Mnih et al. “Asynchronous Methods for Deep Reinforcement Learning”. In: *CoRR* abs/1602.01783 (2016). arXiv: 1602.01783. URL: http://arxiv.org/abs/1602.01783.

[13] Yuhuai Wu et al. “Scalable trust-region method for deep reinforcement learning using Kronecker-factored approximation”. In: *CoRR* abs/1708.05144 (2017). arXiv: 1708.05144. URL: http://arxiv.org/abs/1708.05144.

[14] Ronald J Williams. “Simple statistical gradient-following algorithms for connectionist reinforcement learning”. In: *Machine learning* 8.3-4 (1992), pp. 229–256.

[15] John Schulman et al. “Trust Region Policy Optimization”. In: *CoRR* abs/1502.05477 (2015). arXiv: 1502.05477. URL: http://arxiv.org/abs/1502.05477.

[16] David Silver et al. “Deterministic policy gradient algorithms”. In: 2014.

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

[18] Kevin Frans et al. "Meta Learning Shared Hierarchies". In: *CoRR* abs/1710.09767 (2017). arXiv: 1710.09767. URL: http://arxiv.org/abs/1710.09767.

[19] Tejas D. Kulkarni et al. "Hierarchical Deep Reinforcement Learning: Integrating Temporal Abstraction and Intrinsic Motivation". In: *CoRR* abs/1604.06057 (2016). arXiv: 1604.06057. URL: http://arxiv.org/abs/1604.06057.

[20] Ashley D. Edwards et al. "Imitating Latent Policies from Observation". In: *CoRR* abs/1805.07914 (2018). arXiv: 1805.07914. URL: http://arxiv.org/abs/1805.07914.

[21] Yuval Tassa et al. "DeepMind Control Suite". In: *CoRR* abs/1801.00690 (2018). arXiv: 1801.00690. URL: http://arxiv.org/abs/1801.00690.

[22] Emanuel Todorov, Tom Erez, and Yuval Tassa. "MuJoCo: A physics engine for model-based control." In: *IROS*. IEEE, 2012, pp. 5026–5033. ISBN: 978-1-4673-1737-5. URL: http://dblp.uni-trier.de/db/conf/iros/iros2012.html#TodorovET12.

[23] KyungHyun Cho et al. "On the Properties of Neural Machine Translation: Encoder-Decoder Approaches". In: *CoRR* abs/1409.1259 (2014). arXiv: 1409.1259. URL: http://arxiv.org/abs/1409.1259.

[24] Irina Higgins et al. "beta-VAE: Learning Basic Visual Concepts with a Constrained Variational Framework". In: *ICLR*. 2017.

[25] John Schulman et al. “High-Dimensional Continuous Control Using Generalized Advantage Estimation”. In: *CoRR* abs/1506.02438 (2015).

[26] Eric Jang, Shixiang Gu, and Ben Poole. *Categorical Reparameterization with Gumbel-Softmax*. 2016. arXiv: `1611.01144 [stat.ML]`.

[27] Benjamin Beyret et al. “The Animal-AI Environment: Training and Testing Animal-Like Artificial Cognition”. In: 2019.

Appendix A

Appendix

A.1 Hyperparameters for β-Variational Autoencoder:

The encoded vector is of dimension 32.

Batch Size = 50

$\beta = 0.000001$

Optimizer: Adam

Learning Rate = 0.001

Adam Epsilon = 0.0001

All other Adam parameters are the PyTorch default

A.2 Hyperparameters for the Baseline:

A.2.1 PPO specific hyperparameters:

$\gamma = 0.99$

$\lambda = 0.95$

$\epsilon = 0.2$

No. of Workers = 6

No. of episodes per PPO learning iteration = 12 (2 per worker, 12000 total transitions)

Exploration Noise is drawn from a normal distribution = N(0, 0.09)

No. PPO epochs = 25

critic_coef = 0.5

entr_coef = 0.002

grad_clipping_norm = 1000

A.2.2 Neural Network hyperparameters

Batch Size = 300

Batch Size for GRU = 6

Sequence Length = 50

Learning Rate = 0.01

Optimizer: Adam

Adam Epsilon = 0.0001

All other Adam parameters are the PyTorch default.

A.3 Hyperparameters for Augmented PPO:

num_clusters = 3

A.3.1 Augmented PPO specific hyperparameters:

$\gamma = 0.99$

$\lambda = 0.95$

$\epsilon = 0.2$

No. of Workers = 6

No. of episodes per PPO learning iteration = 12 (2 per worker, 12000 total transitions)

Exploration Noise is drawn from a normal distribution = N(0, 0.09)

No. PPO epochs = 25

critic_coef = 0.5

entr_coef = 0.012

cval_coef = 0.9

clus_coef = 0.01

grad_clipping_norm = 1000

A.3.2 Neural Network hyperparameters

Batch Size = 300

Batch Size for GRU = 6

Sequence Length = 50

Learning Rate = 0.0004

Optimizer: Adam

Adam Epsilon = 0.0001

All other Adam parameters are the PyTorch default.

A.4 Supplemental Files

The code that was used to train and test both the baseline and the augmented method was included in a supplemental file that was submitted along with this thesis. The file itself is a zipped archive, composed of four python scripts: train.py, utils.py, env_wrapper.py, and models.py

train.py contains the main training code, and the baseline and augmented models can be toggled in the main() function at the bottom of the script. To do this, the appropriate model must be loaded, and the last argument to the ppo() function must be toggled between True and False. True refers to the augmented algorithm, while False refers to the baseline algorithm.

utils.py contains helper functions for neural network weight initialization, as well as for training the Variational Autoencoder.

env_wrapper.py provides an API over the DeepMind Control Suite to train on multiple instances of an environment at once.

Finally, models.py contains all the network architectures. The first one is the Variational Autoencoder architecture, called VAE. The second and fourth models refer to non-RNN architectures that were used initially to test the code itself, but were not used to generate any of the results in this thesis. The third (A2Cgru) and fifth (A2Cmgru) models represent the baseline and augmented networks respectively.