Recreating Baselines for the MineRL Competition

Paul Estano Ethan Gordon Max Rudolph

Abstract

MineRL is a NeurIPS competition track aimed at developing robust solutions for heiracrhical deep reinforcement learning (Deep RL). Reinforcement learning is massively constricted when it is applied to complex, non obvious tasks with sparse feedback. Additionally, it is also difficult to define an effective reward structure when applying reinforcement learning to real-world tasks. Thus, with these numerous hindrances to effective learning, it is difficult to make advances. NeurIPS proposes the use of Minecraft, a mature sandbox environment video game, to make advances in hierarchical learning. MineRL is an international competition aimed at training an agent to obtain diamonds in the Minecraft game with little human priors.

1. Introduction

In this project, we attempt to use reinforcement learning to compete in the MineRL competition. MineRL is a competition to train an agent with little to no human priors to accomplish various tasks in the sandbox game Minecraft [4]. The MineRL competition started at NeurIPS in 2019 and provided baseline reinforcement algorithms (deep q-learning, proximal policy optimization, generative adversarial imitation learning, etc...) which have since been deprecated. We implement deep q-learning, proximal policy optimization, and behavioral cloning to solve tasks in Minecraft using the MineRL environment.

Minecraft and MineRL Environments

In Minecraft, the agent can move in all planar directions, jump, and use its weapon to attack. Figure 1 shows an example of a Minecraft world and the screen the player would see. The player can interact with each block within a vicinity by attacking it. The MineRL environment packages all of Minecraft into an easy-to-use python library that inherits from OpenAI's reinforcement learning framework, Gym.

The MineRL library provides several different environments with which we can test our algorithms. Each environment looks the same but differs in its reward, state-space, and action space. For instance, in the most basic environment, MineRL-TreeChop-v0, the state space is the 64x64 RGB screen image that the player sees and the reward is how many trees the agent chops down in a with a given action. All other MineRL environments have the same RGB image type but differing action spaces and reward calculations.

Figure 1. Player's Screen in Minecraft

2. Approach

In this project, we attempted to apply several common reinforcement learning algorithms to the MineRL environment. Namely, we implemented PPO, DQN, and behavioral cloning. Additionally, because MineRL is a sparse reward environment, we tried to incorporate the expert data provided by MineRL. Some MineRL environments avoid the sparse reward structure by articifically imposing a dense reward using information that the agent cannot see.

2.1. Action Space

We discretize and reduce the action space into combinations of 7 different actions as shown in Table 2. We intentionally don't use backward, left and right actions as they are not used often by players. For some experiments we also tried to remove pitch actions as we noticed our agents often ended up staring at the sky.

2.2. Deep Q-Learning

Deep Q-Learning is a state of the art value-based method based on the Q-Learning algorithm [3]. A neural network

actions	a_0	a_1	a_2	a_3	a_4	a_5	a_6
pitch +5	+						
pitch -5		+					
yaw +5			+				
yaw -5				+			
forward					+	+	
jump						+	+
attack	+	+	+	+	+	+	+

Table 1. Action Table

is used to approximate the Q function: $Q(S_t, A_t; \theta)$, where θ represents the network weights. This Q-function outputs the value of a given state-action pair.

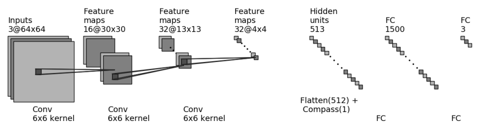

Figure 2. The CNN used to approximate the Q-function

We use a convolutional neural network (CNN) to create a 512-dimensional feature vector to represent the Minecraft screen. We then concatenate the compass angle to this feature vector to produce a 513-dimensional input to a fully-connected network. We use a 3-layer fully connected network to produce the final state-action value function. We train the network off-policy using epsilon-greedy action selection. This means we usually choose the network output, but with probability ϵ we choose a random action to help increase exploration while interacting with the environment. At test time, we always select the action with maximum Q-value.

During training, we use the one-step discounted return as the update target. We use a target network which is updated once every 10 iterations as described in [3]. We use RMSProp for optimization with a constant learning rate of 0.001.

The update target for $Q(s_t, a_t)$ is:

$$R_{t+1} + \gamma Q(s_{t+1}, a_{t+1}; \theta) \tag{1}$$

2.3. Proximal Policy Optimization

Proximal Policy Optimization (PPO) [8] is a state of the art policy gradient method. The main goals of PPO are to find a balance between ease of implementation, sample efficiency and ease of tuning. Unlike other Q-learning methods which can learn from stored off-line data, PPO learns directly from what the agent encountered in its environment.

The objective function commonly used by Policy Gradient methods is the following:

$$L^{PG} = \mathbb{E}_t[\log \pi_\theta(a_t|s_t)\hat{A}_t] \tag{2}$$

Where π_θ is the policy function that outputs the probability of an action given a specific state, $\hat{A}_t$ is the advantage function

$$\hat{A}_t = \sum_{k=t-T}^{t} \gamma^k R_k - V(s_t) \tag{3}$$

The value function function V is computed using a neural network just as one would tackle a supervised learning problem.

One may want to run multiple optimization steps on (1) but this would in practice hurt the policy and value functions. TRPO [7] fixes this problem by ensuring the new policy obtained after an update never gets too far from the old policy. Indeed, TRPO adds a KL constraint to its objective function to ensure the updated policy is parametrically similar to the previous policy.

The problem with the additional TRPO constraint is that it adds an overhead to the computation and it sometimes leads to undesirable training behaviors. Therefore, PPO directly integrates this constraint in its objective function:

$$L^{CLIP}(\theta) = \mathbb{E}[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)], \tag{4}$$

where $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$ is the probability ratio.

Clipping the probability ratio allows us to limit the effect of the gradient update. Taking the point-wise minimum between the regular objective function and the clipped objective function allows us to take into account points where the action taken was bad ($\hat{A} < 0$) but its probability increased

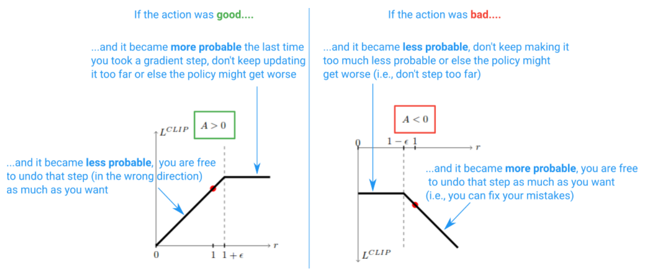

Figure 3. Plots showing one term (i.e., a single timestep) of the surrogate function L CLIP as a function of the probability ratio r, for positive advantages (left) and negative advantages (right). The red circle on each plot shows the starting point for the optimization (from [8])

The PPO objective function is similar to the TRPO objective function but its implementation is simpler and does not require KL divergence computation.

2.4. Behavioral Cloning

Behavioral cloning is a sector of machine learning where an autonomous agent learns a control policy by observing an expert act in an environment. Given that we have access to ample expert data, behavioral cloning/imitation learning is a natural solution to the MineRL problem. As explored in [10], behavioral cloning (also known as apprenticeship learning) can be reduced to a classification problem of state-action pairs. Behavioral cloning aims to replicate the policy of the expert by training a classifier to classify the given state into the expert's supposed action; however, this creates an unstable solution because, if the agent deviates from the expert state trajectory at all, there is no guarantee that the learned policy will drive the agent back to the expert trajectory. In the context of MineRL, we used a convolutional neural network to classify the state observation into the movement actions described in Table 2. Implementation will be discussed in the Experimentation and Results section.

3. Experimentation and Results

3.1. Deep Q-Learning

Based on code from [6] we implemented our Q-network using PyTorch. Due to the lack of convergence in our initial tests, we further restricted the action space as follows to help decrease the amount of time needed for optimization. This was acceptable because our task of navigation doesn't require the pitch actions. The agent can adequately be controlled using just the yaw and forward actions.

actions	a_0	a_1	a_2
yaw +1			+
yaw -1		+	
forward	+		
jump	+		
attack	+	+	+

Table 2. Action Table

With these modifications the agent still fails to complete the task, but appears to show some degree of improvement over time. While deep Q-learning is often regarded as the simplest deep model for reinforcement learning, it performed better than all of our other methods.

3.2. Proximal Policy Optimization

We adapted the code from [1] to the MineRL environment. In particular, we adapted the neural networks used in PPO by concatenating the batch values of the navigate-Dense environment compass angle to the feature maps just before the linear layers of the networks. Then, we tried several combinations of hyperparameters and actions. First, we trained the agent using an action space similar to 2.1. for the

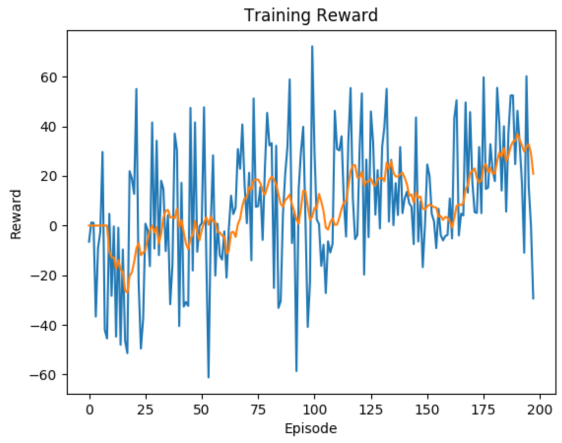

Figure 4. Cumulative reward for every episode using DQN

usual hyperparameters such as learning rate and discount rate we used values similar to [5]. We chose a maximum number of steps per episode of 5000. This first experiment resulted in an average cumulative reward close 0 over a 100 episodes of testing. After debugging we figured that the agent was not moving. We supposed the agent learned that it could not get a negative reward if it did not move. In the NavigateDense environment, the agent accumulated positive reward for nearing its target and negative reward for distancing itself from the target.

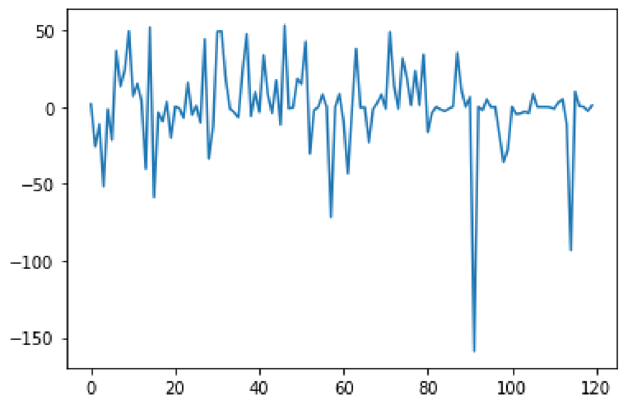

Figure 5. Cumulative reward for every episode during our first training of PPO

Therefore, we decided to force the agent to move at every step by setting the forward action to one for every step. Furthermore, we also saw that our agent ended up stuck after a few thousands of steps in almost every episode. Therefore, we restricted the maximum length of every episode to 5000. Unfortunately, the agent did not perform better than for the first training: we still got a result close to 0. This time the agent would dig a hole next to the starting point and would fall into it, preventing it from moving any farther.

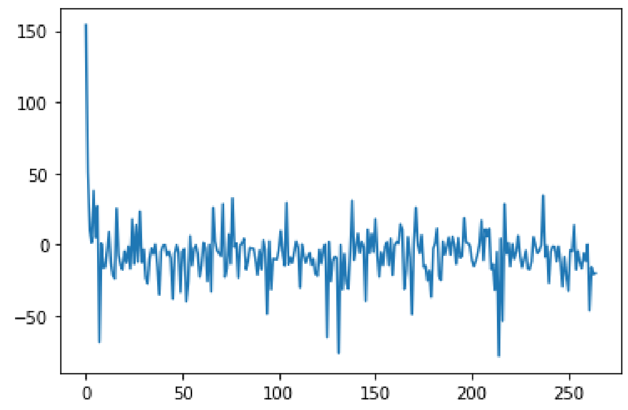

Figure 6. Cumulative reward for every episode during our second training of PPO

Algorithm	Average Score
BC (Ours)	0.79 $\pm$7.6
BC (Baseline)	5.57 $\pm$6.01

Table 3. Table of performance of behavioral clone

3.3. Behavioral Cloning

As previously stated, MineRL provides hours of expert data for each task environment. We trained a two layer convolutional neural network with 64 feature maps per layer, 5x5 kernels with an output dimension equaling the number of possible actions, and a cross entropy loss similar to that which we used in the classification example in homework 2; the action space was subject to change but, empirically, we (and other participants in the competition) have found that a simplified action space as described in 2 worked the best with the RL solutions. As mentioned previously, the behavioral cloning algorithm can be reduced to a classification task; in Figure 7, a sample piece of data and the label is presented. From the MineRL data, we extracted 21,000 image action pairs. For training we used a subset of 20,000 images and validated on the other 1,000 images. As shown in Figure 8, the loss of the classifier drops off significantly in the first several training episodes; we hypothesize that this is the case because the training dataset is severely skewed and not i.i.d. whatsoever. This is a common issue with behavioral cloning because, during expert play, each state in the environment is not visited equally. Still, we were able to achieve around 70% accuracy on the test set.

The results when we tested the behavioral clone in the MineRLNavigateDense-v0 environment were less promising than our performance on the test set. Figure 9 shows the performance of the behavioral clone in the Dense environment over 250 episodes. This method has been proven to underperform both by us in this project and the competition organizers. Table 3 shows compares the performance of our behavioral clone and that of the organizers.

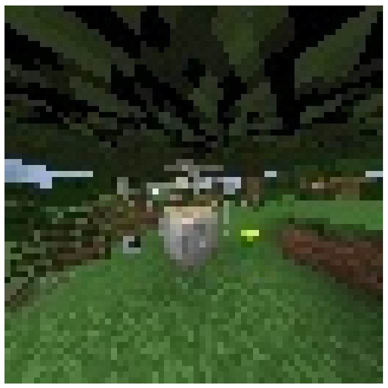

Figure 7. MineRL Data example with action label a_4: Move forward and Attack

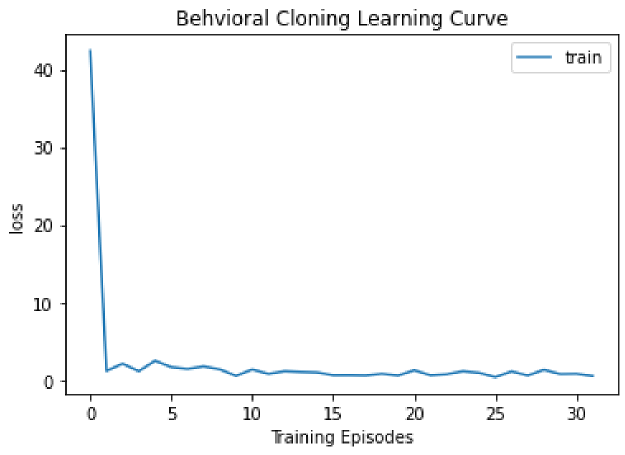

Figure 8. Training Loss for Behavioral Cloning Classifier

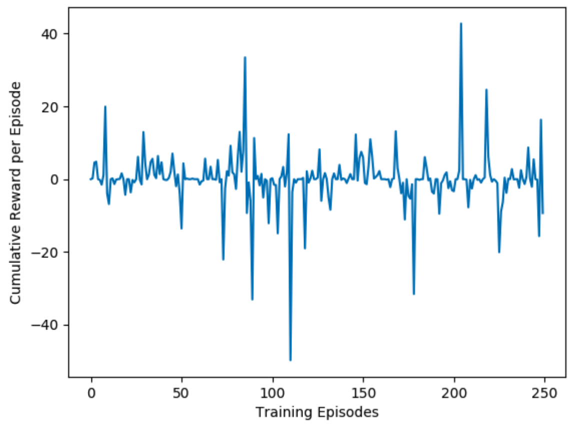

Figure 9. Cumulative Reward over 250 episodes of Behavioral Cloning

In order to garner some understanding as to why this method did not work, we applied network visualization methods such as Saliency Maps [9] to see what the network was looking at when formulating its action decision. As we can see in Figure 10, the network seems to de-emphasize the sky and put more weight on the green/brown land around the agent. This is a good behvior to learn but not nearly enough to replicate the actions of an expert. Ideally, we would see the network focusing on a tree in the Tree Chop environment or the target block in the Navigate environment.

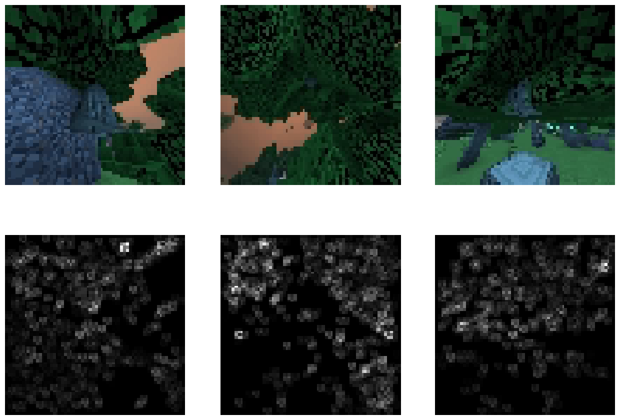

Figure 10. Saliency Maps for the Behavioral Cloning Classifier

4. Conclusion and Future Works

None of our methods performed well. In particular, we were not able to reproduce the results of [5]. The reasons might be the large size of the action space and/or our lack of experience in reinforcement learning algorithm tuning. We also lacked the time to properly try several combinations of hyper-parameters as MineRL environment did not allow us to use any GPU (the environment requires a monitor when using a GPU).

Moreover, a future work might be to try advanced algorithms combining expert data from the MineRL dataset and reinforcement algorithms. We tried to implement DQfD [2] but the supervised training part of our implementation underperformed, similar to our implementation of behavioral cloning. Besides [5], only one other person tried (and failed) to get good results with DQfD with MineRL. There is an open issue regarding the reproducability of certain MineRL baselines.

Additionally, as mentioned earlier, all of the official baselines have become deprecated and are not runnable; this raises a serious issue questioning the reproducibility of the MineRL baselines. Furthermore, some of the baselines are extremely unstable: for instance their PPO implementation displayed a result of 87.83 average cumulative reward over 100 episode for a standard deviation of 59.46. This is emblematic of the performances of the other baselines (many baselines have standard deviations as big as their means). In conclusion, we faced several problems when implementing solutions to the MineRL competition. We were not able to successfully reproduce the baselines from last year's competition.

References

[1] Nikhil Barhate. Ppo-pytorch repository, 2018. Available at https://github.com/nikhilbarhate99/PPO-PyTorch.

[2] Todd Hester, Matej Vecerík, Olivier Pietquin, Marc Lanctot, Tom Schaul, Bilal Piot, Andrew Sendonaris, Gabriel Dulac-Arnold, Ian Osband, John P. Agapiou, Joel Z. Leibo, and Audrunas Gruslys. Learning from demonstrations for real world reinforcement learning. *CoRR*, abs/1704.03732, 2017.

[3] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing Atari with Deep Reinforcement Learning. *arXiv e-prints*, page arXiv:1312.5602, Dec. 2013.

[4] Mojang. Minecraft. [CD-ROM], 2011.

[5] Preferred Networks. Minerl baselines, 2019. Available at https://github.com/minerllabs/baselines.

[6] Adam Paszke. Reinforcement learning (dqn) tutorial, 2017. Available at https://github.com/pytorch/tutorials/blob/master/intermediate_source/reinforcement_q_learning.py.

[7] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust region policy optimization. *CoRR*, abs/1502.05477, 2015.

[8] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017.

[9] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps. *ICLR*, 2013.

[10] Umar Syed and Robert E Schapire. A reduction from apprenticeship learning to classification. In J. Lafferty, C. Williams, J. Shawe-Taylor, R. Zemel, and A. Culotta, editors, *Advances in Neural Information Processing Systems*, volume 23, pages 2253–2261. Curran Associates, Inc., 2010.

Appendix: Reproducibility

Reproducibility checklist can be found appended to this paper. Code for this project exists in a private repository but is uploaded to the gradescope link.

Models

All models are explained in detail in the paper and included in the code. Though, the algorithms are at the center of this paper, not the specific deep models we used.

Datasets

All data can be found in the MineRL package documentation here. The datasets are discussed in the Behvioral Cloning section along with all test splits, etc...

Experimental Results

Because most of results were underwhelming, we did not apply significant statistical methods in our analysis. We discussed the issues and possible fixes to the problems we encountered. All hyperparameters are discussed in the paper and those that are not discussed are the same as those used in the deprecated MineRL baselines found here.