Reinforcement Learning on Shepherding Problem

Katie Li **Yutong Liu** **Peiyu(Gabby) Xiong**

Abstract

Training a dog to herd sheep is time consuming and costly. However, the training process can be viewed as a reinforcement learning (RL) problem where a software agent is trained instead of a real dog. Over the past few years, deep-learning based algorithms such as deep Q-learning (DQN) and its variants have been proposed to overcome the limitations of traditional RL algorithms. In this paper, we discuss the key features of these algorithms and evaluate their performance on the shepherding problem. We find that double deep Q-network (DDQN) achieves the best results in terms of the agent's learning time. Additionally, there is not a sizable difference in learning time between using raw pixels versus handcrafted features as input.

1 Introduction

Herding dogs are widely used for farm assistance. A typical herding task involves circling a stock of animals, moving the stock, and more importantly, driving the stock to the destination. It would be ideal to train a software agent to learn which actions to take to complete the herding task. Once the software agent learns the optimal herding strategy, we can build a robot that acts as a herding dog but is less costly and time consuming to train.

The software agent can be trained using reinforcement learning (RL) algorithms. RL is a subarea of machine learning, concerned with how agents should take actions in an environment with the goal of maximizing some notion of cumulative reward. In the shepherding problem, the agent - dog - seeks the optimal movement strategies to get the group of sheep to the target position.

In this project, we build an environment in openAI to simulate the shepherding process and then apply various popular reinforcement learning algorithms to evaluate their performance. One common issue for most machine learning applications is that it is not clear which algorithm has better performance in certain types of problems. Therefore, we make efforts to compare these algorithms within the scope of the shepherding problem. Once the best reinforcement algorithm is found, a real robot can be made to herd the sheep group in real life.

2 Related Work

Over the past few years, deep learning has been used to overcome inherent limitations in classical RL algorithms. Traditional RL algorithms rely on hand-crafted engineered features, thus lacking scalability. Deep learning has allowed RL to scale to settings with high-dimensional state and action systems.

In 2013, Mnih et al. proposed the original deep Q-network (DQN) method based on Q-learning that uses a neural network to approximate the value function [1]. The DQN model was evaluated on Atari 2600 video games, using raw pixels as input. Since the introduction of the DQN model, many of its components have been fine-tuned to improve performance, leading to the double deep Q-network (Van Hasselt, 2015), prioritized experience replay (Schaul et al., 2015), and dueling network architecture (Wang et al., 2016) [2,3,4].

While Q-learning is a value-function approach for reinforcement learning, there are also policy-based methods that seek to directly optimize the policy that controls the agent. The initial policy gradient method was proposed by Sutton in 1999. In 2015 Lillicrap et al. proposed the deep deterministic policy gradient, a deterministic version of policy gradient that can solve reinforcement learning settings with a continuous action space [5]. The Actor-Critic Method is a system where in which both a policy and value function are learned at the same time.

3 Approach

3.1 Deep Q-Network

In this paper, we focus on applying value-function based algorithms to the shepherding problem. We start by applying the original DQN algorithm, as shown in Algorithm 1 [1]. The success of the algorithm in overcoming the instability caused by highly-correlated game states can be attributed to *experience replay* and a *target network*.

For experience replay, observed experiences $e_t = (s_t, a_t, r_t, s_{t+1})$ are stored in the replay memory $\mathcal{D} = \{e_1, e_2, ..., e_t\}$. Mini-batches from the memory are sampled for training instead of just using the most recent transition (lines 10 and 11). Experience replay reduces the variance of learning updates and uniform sampling reduces the correlation of subsequent training samples.

The target network adds further stability to the algorithm. The target used by DQN (line 12) is

$$y_t^{DQN} = r_t + \gamma \max_a Q(s_{t+1}, a_t; \theta^-).$$

The target is used to compute the loss for each action (line 13). The parameters of the target network, θ^-, are fixed and updated every τ steps from the primary Q-network.

Algorithm 1 Deep Q-Learning with Experience Replay

```
 1: Initialize Replay Memory $\mathcal{D}$ to capacity N
 2: Initialize action-value function $\mathcal{Q}$ with random weights
 3: for episode = 1, M do
 4:     Initialize sequence $s_1$ = $x_1$ and prepossessed sequenced $\phi_1 = \phi(s_1)$
 5:     for $t = 1, T$ do
 6:         With probability $\epsilon$ select a random action $a_t$
 7:         otherwise select $a_t$ = max$_a Q^*(s_t, a; \theta)$
 8:         Execute action $a_t$ and observe reward $r_t$ and image $x_{t+1}$
 9:         Set $s_{t+1} = s_t, a_t, x_{t+1}$
10:         Store transition $(s_t, a_t, r_t, s_{t+1})$ in $\mathcal{D}$
11:         Sample random minibatch of transitions $(s_j, a_j, r_j, s_{j+1})$ from $\mathcal{D}$
12:         Set $y_j$ = $\begin{cases} \mathrm{r}_j, & \text{for terminal } s_{j+1}, \\ \mathrm{r}_j + \gamma \max_{a'} Q(s_{j+1}, a_j; \theta^-), & \text{for non-terminal } s_{j+1} \end{cases}$
13:         Perform a gradient descent step on $(y_i - Q(s_j, a_j; \theta))^2$ according to equation 3
14:         Every $\tau$ steps, update target parameters $\theta^- = \theta$
15:     end for
16: end for
```

3.2 Double Deep Q-Network

Van Hasselt's double deep Q-network is based on double Q-learning, which aims to reduce over-estimations in the q-function by decomposing the max operation in the target into action selection and action evaluation. The target used by the DDQN is

$$y_t^{DDQN} = r_t + \gamma Q(s_{t+1}, \operatorname*{argmax}_a Q(s_{t+1}, a; \theta_t); \theta_t^-).$$

Like DQN, two neural networks are used to train the Q-values. The current Q-network, with parameters θ_t, is used to select the maximizing action of the state s_{t+1}. The target Q-network, with parameters θ_t^- is then used to evaluate the value of this action.

3.3 Prioritized Experience Replay

Under the assumption that agents may learn more effectively from some transitions than from others, Schaul et al. proposed a prioritized experience replay. Each experience is given a priority p_i that is proportional to the absolute temporal-difference (TD) error. The TD-error,

$$\delta_t = y_t - Q(s_t, a_t),$$

is the distance between the value function and target. The TD-error is stored in replay memory along with every sample and updated at each learning step. When it is time to sample a mini-batch, the probability of choosing an experience i is $P_i = \frac{p_i}{\sum_i p_i}$.

3.4 Dueling Network Architecture

In the DQN algorithm, a single-stream Q-network is used. The dueling network architecture presented by Wang et. al decomposes the single stream into two streams, one that estimates the state value functions $V(s)$ and one that estimates the advantage of each action $A(s, a)$. The two streams share a convolutional feature-learning module and are combined using an aggregating layer that outputs the Q-values for each action $Q(s, a)$. The insight behind the dueling architecture is that the importance of a state can be learned without having to learn the effect of each action for the state [4]. Figure 1 visualizes the differences between the two architectures[1].

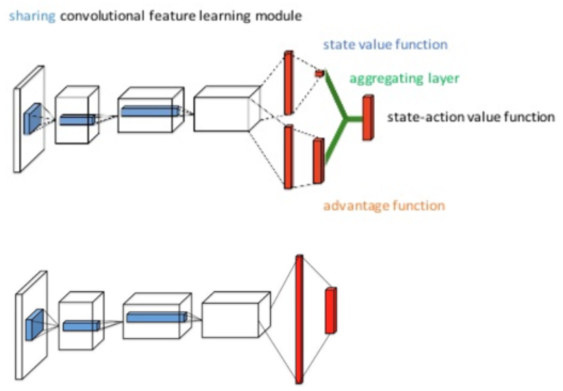
sharing convolutional feature learning module
state value function
aggregating layer
state-action value function
advantage function

Figure 1: Dueling Q-network (above) and single-stream Q-network (below)

3.5 Training DQN with RGB Arrays

An attractive feature of deep Q-learning is that with the aid of deep neural networks, we can use visual descriptions of the learning environment as input to train agents. We experiment with both hand-engineered features and image inputs to investigate whether different input types can affect the performance of the algorithms. To simplify the training process on raw pixel input, we make the learning environment smaller to reduce the state space. Therefore, the results of the experiments are not directly comparable.

4 Experiment

4.1 Environment

The reinforcement learning environment, *sheep-v0*, is constructed in the same manner as other environments in OpenAI Gym [6]. The animal behaviors are modelled under Strömbom's Model with a few extensions to facilitate RL research [7]. Similar to classical control problem in OpenAI gym, the environment offers two types of input for training, hand-engineered features and RGB arrays. Reward is defined differently under the two cases. When training with hand-engineered features,

[1]https://www.slideshare.net/carpedm20/dueling-network-architectures-for-deep-reinforcement-learning

Table 1: functions in sheep-v0 environment

Name	Type	Details
step()	Function	Execute action command, return the new observation, reward and whether the sheep is close enough to the target (whether the episode is done)
updateLocation()	Function	Calculate locations of sheep based on Strömbom's model
getReward()	Function	Since the observations are not raw pixel inputs, we engineered the reward to depend on both the location of the herd and the distance between the center of the herd to the agent.
ifDone()	Function	Check if the agent has finished the current episode, the agent successfully finish current episode if the centroid of the herd is within the FINISH Radius from the target position
observation	Space Set	**when training with hand-engineered features:** The observation contains hand-engineered features to train the agent. Features include the locations of individual sheep, the location of the agent, the location of the flock's centroid, and the distance between the flock's centroid and the target etc. *since these features are hand-engineered, they can be changed easily* **when training with RGB arrays:** RGB array of size SCREEN-WIDTH x SCREEN-HEIGHT x 3
action	Space set	Action space includes 4 discrete actions which are UP, DOWN, LEFT, RIGHT

there are more immediate rewards from the environment, whereas when training with RGB arrays, the reward is defined to be more sparse.

At the beginning of each training episode, a new *sheep-v0* environment is initialized. Figure 2 provides a visualization of the environment. The grey dots represent sheep, the green dot represents the target destination, and the red dot represents the agent (dog). Based on Strömbom's model, the five weighted forces that drive the sheep flocking are attraction to the centroid of its n-nearest neighbors, separation from other sheep, inertia force, predator avoidance and random actions [7]. Therefore we have implemented the environment such that every sheep adjusts its speed based on the positions of other sheep and the agent, as well as some randomness. The agent gains information about the sheep and target using the API functions and chooses an action from the action space. Table 1 summarizes the key components of our environment.

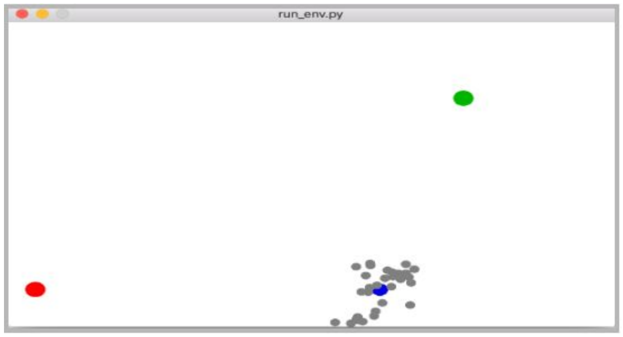

Figure 2: Visualization of sheep-v0 environment

4.2 Network Architecture and Hyperparameters

We use the Tensorflow backend to implement the network architecture. When handcrafted features are used as input, the Q-network and target network consist of two fully-connected layers. When trained with RGB array as input, we first preprocess the original observation from the environment of

Table 2: Hyperparameters from algorithms

Hyperparameter	Description	Value Used
Batch size	Size of sampled batch from experience replay (number of transitions to average over when calculating the gradient for value function)	32
Memory Size	Number of transitions kept in experience replay memory	30000
ϵ_{max}	ϵ is the portion of time that the agent will follow the learned policy while the rest of time will act randomly	0.55
ϵ increment	The epsilon value will start from 0 and increase by value of ϵ increment until it reaches the value of ϵ_{max}	0.001
τ	Number of iterations before updating target parameters	300
γ	Decay rate for discounting future rewards used in target	0.9

size SCREEN-WIDTH $\times$ SCREEN-HEIGHT $\times$ 3 to a grayscale image of size 84×84. Then, a three-layer convolutional neural network is used to extract the features for training the agent followed by a fully-connected layer to output Q-values for each of the four actions.

The hyperparameters used in the algorithms are listed in Table 2. The batch and memory size are parameters of the experience replay memory, while γ and τ are related to the target network. A fundamental trade-off in RL is exploration vs. exploitation. DQN and DDQN use an ϵ-greedy exploration policy. The optimal action is taken with probability $\epsilon \in [0, 1]$ and a random action is taken otherwise. Since it is important to explore the environment and accumulate useful transitions before starting to apply the learned policy, we initially set $\epsilon = 0$ and increase ϵ at each iteration so that over time, the agent will progress towards exploitation.

5 Results

We evaluate the effectiveness of each algorithm by comparing the total learning time over 500 episodes for agents under the following models: DQN, DDQN, DQN with prioritized replay, and DQN with dueling architecture. An episode ends when the herd is within a certain distance from the target location. The hyperparameters and testing environment were initialized in the same way for all models. As a baseline, we include the results of a completely random agent.

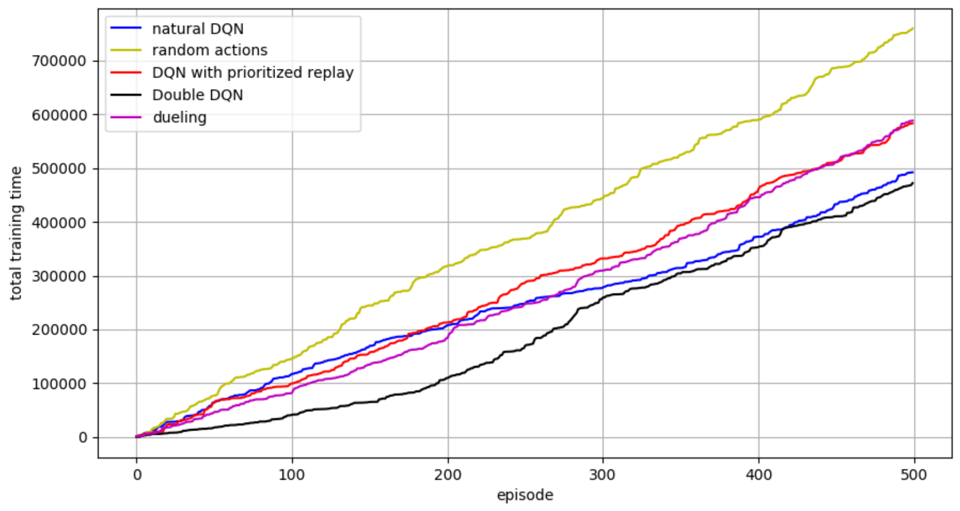

Figure 3: Total training time for four RL models and a random agent

Figure 3 shows the time it takes to finish 500 episodes by the four models as well as the random agent. We can see that the time used without training is always more than that with training, indicating the general effectiveness of the algorithms. Overall, DDQN consistently outperforms the other models. It is also the only model to have a slight upwards curve; the other models are more linear in trend. What is surprising is that in the long run, prioritized replay and dueling network do not improve on the original DQN model. This is contrary to the results achieved by the papers on the Atari testing framework.

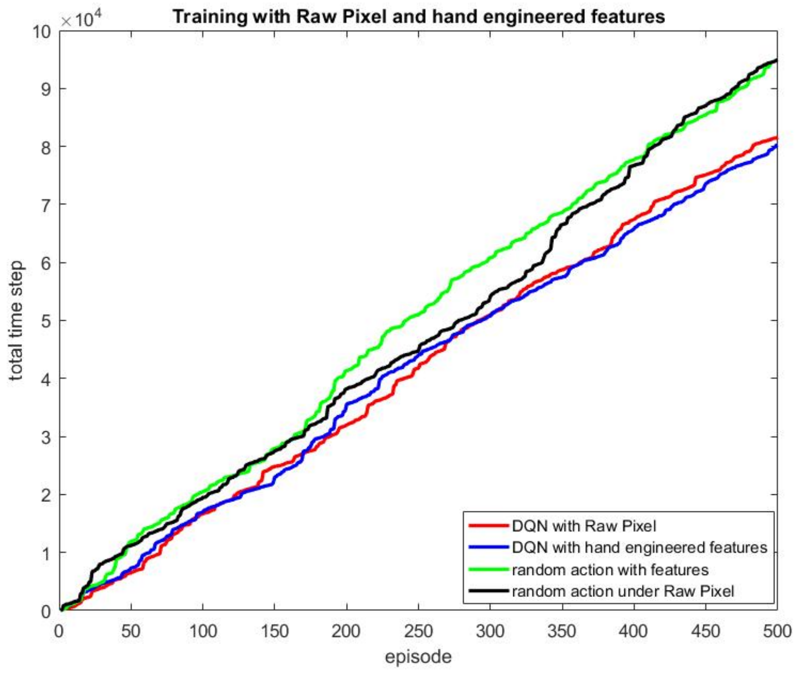

Figure 4: Comparison of training with Raw Pixel and Hand-Engineered Feature

We also compare the training time for different input types. Figure 4 again shows the cumulative time spent for the agents to complete 500 episodes. The baseline for comparison in this case is chosen to be the agent with no training. In this case, there is not a large difference between training with raw pixels and with hand-engineered features. Again, training the agent is generally effective, as the agent with random actions consistently requires more time to finish the same number of episodes. Due to time limit, we could not verify if these trend varies when we increase the number of episodes.

6 Discussion and Future Work

Within the scope of the shepherding problem, we show that DQN and its variants are effective in training the agent. Among all attempted algorithms, DDQN outperforms other deep-learning based models in terms of the total time used to finish 500 episodes. Although one major advantage of DQN is that it incorporates raw pixels as input, we did not see much difference between using hand-crafted features and image input in our experiment.

Ultimately, we were successful in implementing our own OpenAI training environment and applying various deep reinforcement learning models to the shepherding problem. We also show that while some algorithms may be state-of-the-art in some settings, those results may not hold in other settings. Unlike results achieved on the Atari framework by the published papers, prioritized replay and dueling network did not improve on the original DQN model. We also expected the total training time curve to converge, which indicates that it takes less time to finish an episode the more the agent learns. However, all algorithms we tried had a linear trend.

One possible explanation for the discrepancies in our results is that we are not running the learning process long enough. In the published papers, the algorithms usually run for a few days. This was not feasible in our case due to time constraints. Another limitation of our contribution is that we are not able to optimize the hyperparameters in each algorithm. In our experiments, we choose to fix all parameters, with the only variation being algorithmic differences. If we had chosen other values for hyperparameters, the results could have been more close to the referenced papers. We could have also independently tried to find the best hyperparameter values for each model and compare the training results across the models. Given more time, we would definitely seek to experiment with different hyperparameter values.

References

[1] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D. & Riedmiller, M. (2013) Playing Atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.

[2] van Hasselt, H., Guez, A. Silver, D. (2015) Deep reinforcement learning with double Q-learning. *arXiv preprint arXiv:1509.06461*.

[3] Schaul, T., Quan, J., Antonoglou, I. & Silver, D. (2015) Prioritized experience replay. *arXiv preprint arXiv:1511.05952*.

[4] Wang, Z., de Freitas, N. Lanctot, M. (2016) Dueling network architectures for deep reinforcement learning. *arXiv preprint arXiv:1511.06581*.

[5] Lillicrap, T.P., Hunt, J.J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., , Silver, D., Wierstra, D., (2015) Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*.

[6] OpenAI gym: https://gym.openai.com/

[7] Strömbom, D. & Antia, A. (2018. Anticipation induces polarized collective motion in attraction based models. *arXiv preprint arXiv:1710.05692*.