Defeating the Invaders with Deep Reinforcement Learning

Christian L. Martinez-Nieves
CS229, Stanford University
chris151@stanford.edu

Abstract

Great strides have been made in the field of AI to achieve human-like performance on everyday tasks, as can be seen in reinforcement learning (RL) type problems. The purpose of this project is to demonstrate how reinforcement learning algorithms can be used to play a video game like Space Invader with human like performance, and compare how Deep Q-learning and Deep Deterministic Policy Gradient (DDPQ) algorithms proposed in [1] and [3] perform comparatively on the same games. To experiment with this game, the OpenAI Gym Atari emulator is used to run it, while at the same time giving us access to valuable information about the environment. More specifically, the environment supplies our RL agent with a set of values representing the individual pixels of a given game frame image, which the agent 'analyzes', using the algorithms mentioned, to determine the current state of the environment as well as what action to take next to optimize its reward in the game. In doing so, deeper comprehension of how certain architectures, as well as how certain hyper parameters, influence the agent's performance will become evident. Success of the experiments will be based on how well the agents scored (on average) at the end of the game when compared to a human player's and random agent's average scoring.

1. Introduction

It remains a long-standing ambition of AI related fields to develop algorithms capable of achieving human-like performance in tasks that we tend to perform daily. Reinforcement learning (RL) is one of many such fields attempting to close the gap between human and computer behavior. Although extracting useful information from an environment is still a challenge, advancements in Deep Learning have aided in this process of feature extraction. Although applying deep learning techniques to RL proves to be particularly challenging due to the lack of labeled training data, learning from sparse and noisy reward scalars, and the fact that the underlying data distribution changes as the algorithm identifies new patterns, neural networks (particularly convolutional), have proven very useful to overcoming these hurdles.

Video games can be considered good mediums to test out and improve of RL algorithms, mainly because of the wide variety of complexities one can find using the same interface. Many games are readily available all with a set of different restrictions and vastly different control policies, which proves useful when testing out how well an algorithm adapts to different scenarios.

We thus focus on two particularly popular Deep RL algorithms which prove very useful when it comes to learning control policies from raw video frames. Specifically, each algorithm is used to implement an agent which will use the RL techniques to learn to play a game like Space Invaders. Both algorithms use convolutional neural networks (CNN) in order to learn from high dimensional state spaces, like images. Both algorithms are also *off-policy* (combines action policy estimation and exploration) and *model-free* (learns from samples of the state space without explicitly constructing an estimate for it).

1.1. Deep Q-Learning (DQL)

Deep Q-Learning is a variant of the Q-learning algorithm which attempts to approximate the Q function using a deep neural network (DNN). DQL is usually used in conjunction with the Experience Replay technique to tackle the issue of correlated data and changing data distributions [1] [2]. Experience replay samples from past transitions to try and smooth the training distribution over many past behaviors. The network is also trained with a target Q network to give consistent targets during temporal difference backups [3].

1.2. Deep Deterministic Policy Gradients (DDPG)

DDPG improves on top of many of the advances made by the DQG algorithm. However, where DQL approximates a value function and uses that to compute a deterministic policy, DDPG approximates a deterministic policy directly using an independent function approximator with its own parameters. On the other hand, DDPG tends to be much simpler to implement and scales relatively easy to more difficult problems and larger networks, and can also learn good policies on lower-dimensional observations [3].

2. **Related Work**

Reinforcement learning has made several advancements in the last decades, each being the foundations for the next innovation's success. One of the earliest and most common well known successful RL applications was the TD-Gammon algorithm [4], which used a neural network with a single hidden layer and temporal difference methods to help an agent learn to play Backgammon. However, the concepts developed for it proved to be inadequate for other use cases. A more notorious and still widely popular RL technique is that of Q-learning, described in [5]. Q-Learning also uses temporal difference techniques, like [4], and provides agents with the ability to perform well when dealing with Markov Decision Processes (MDP), by allowing the agent to explore the consequences of actions without having to build maps of the whole domain (*model-free*). Nevertheless, it was later seen that combining the model-free properties of Q-Learning with *off-policy* learning (non-linear function approximators) could make the Q-network diverge. This hindered the use of off-policy learning in subsequent years.

Policy gradients (PG) are currently another significant player in RL, and are a more recent innovation. Unlike Q-learning based methods, which are value based function approximators which try to find deterministic action policies, vanilla policy gradient methods approximate a stochastic policy directly using an independent function approximator with its own parameters [6]. However, as shown by [7], the introduction of Deterministic Policy Gradients (DPG), proved to be instrumental in simplifying the computation of the action policy, as the stochastic(vanilla) PG integrates over both the action and state spaces, while the DPG only integrates over the state space. DPG uses this deterministic policy gradient to derive an off-policy actor-critic algorithm which significantly outperforms the stochastic PG.

These previous implementations proved, however, insufficient to tackle high dimensional state space problems, like those involving learning from images. However, recent advancements in deep learning, like those of [8], have made learning from high dimensional state spaces possible and have led to the adaption of DPG and Q-learning to high dimensional state spaces, resulting in the DDPG and DQL algorithms.

3. Dataset and Features

The data provided to the models was provided by the OpenAI Gym Atari Emulator [6]. The environment in this case would be the game being emulated (Space Invaders), and can be considered a stochastic environment. The output returned by the environment, which is used to train our agents, consists of the current state of the game, represented as a 210x160x3 RGB image, and the game state is updated every time the agent makes an action. In order to reduce computation complexity, all images returned by the environment are pre-processed by converting them to gray-scale, down-sampling them to have a shape of 110x84, and cropping the images to have a size of 84x84x1 [1].

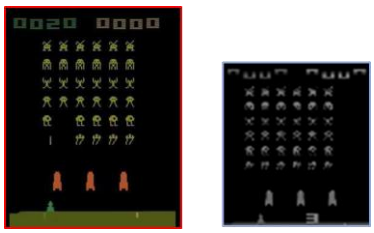


Figure 1. Raw model input (left) and pre-processed model input (right).

Each model input consists a pre-processed image from the game, The input features in this case are all the pixel values in the pre-processed image, which yields a total of 7,056 input features. Since the network models for both DDPG and DQL approaches are convolutional neural networks, features from the inputs are extracted in the convolutional layers of the model. All environment image outputs are stored in the from the agent's Experience Replay buffer, and sampled randomly from it in batches of size 32 in order to train the agent.

4. Method

The two machine learning algorithms of choice for this project are the Deep-Q Learning and Deep Deterministic Policy gradient algorithms, due to their wide range popularity and powerful ability to learn good control policies from high dimensional state spaces. Considering that the goal is to train an agent how to successfully play Space Invaders, the game is discretized into time-steps by the environment, where at each step, the agent must choose an action $A=\{1,\ldots, K=6\}$, and the environment, after updating the internal state and game score, returns an image representing the new game screen (new state), as well as the reward r_t resulting from the action taken. Each RL algorithm uses this information in order to arrive at an optimal action policy which will help them maximize rewards accrued based on game states returned by the environment.

4.1. Deep Q-Learning Agent

In order to handle high-dimensional data, [1] shows how the Q-learning algorithm can be adapted to for these scenarios using CNNs in order to extract features from these high-dimensional state spaces, and using Experience Replay to make Q-Learning converge while using off-policy techniques.

Given a sequence of state, action rewards (s_t, a_t, r_t), $(s_{t+1}, a_{t+1}, r_{t+1})$ …, we want our agent to learn the best possible action policy to play the game and maximize future rewards from this sequence. To do this, we assume that all future rewards are discounted by a factor of γ and define the future rewards discount as

(1) $$R_T = \sum_{t=0}^{T} \gamma^t r_t$$

where T is the time-step at which the game terminates, and define the optimal action value function as $Q^*(s, a)$ in **(2)** to be the expected future reward we get by seeing some state s and following action a,

(2) $$Q^*(s,a) = \max_\pi \mathbb{E}\left[R_t | s_t = s, a_t = a, \pi\right]$$

where π would be the policy mapping sequences to actions. Since $Q^*(s, a)$ obeys the Bellman equation identity, because we can know $Q^*(s', a')$ for the next time-steps, then the optimal strategy would be to maximize the expected value of

$$Q^*(s,a) = \mathbb{E}_{s'\sim\mathcal{E}}\left[r + \gamma \max_{a'} Q^*(s',a') \middle| s, a\right]$$

(3)

Here, s', a' represent the next state and action respectively. Although the idea is to estimate the action-value function Q* using the Bellman equation as an iterative update using **(3)**, in practice we use a non-linear approximator like a neural network with weights w_i to find Q*, because using **(3)** iteratively is impractical because the action value function is estimated separately for each sequence, without generalization. This neural network is what's known as the Q function, and due to the high dimensionality of the state inputs, it's a CNN. Finally, we can use the following loss function and update rule to update the weights w in our Q-function using stochastic gradient descent.

(4) $$L = \sum_{s,a,r,s'} (Q(s, a; w_i) - (r + max_{a'} Q(s', a'; w_{i-1})))^2$$

(5) $$w = w - \alpha\left[\sum_{s,a,r,s'} \left(Q(s, a; w_i) - \left(r + max_{a'} Q(s', a'; w_{i-1})\right)\right)^2\right] \nabla_{w_i} Q(s, a; w_i)$$

where w_i is current Q-function's weights, while w_{i-1} are the target Q-function's weights. The target Q-function is a separate CNN with the same structure as the normal Q network, and its weights are updated periodically by gradually setting them to be like those of w_i ($w_{i+1} \leftarrow \tau * w_i + (1-\tau) w_{i+1}$ and $\tau << 1$). On the other hand, α represents our learning rate.

Figure 2 shows the pseudocode used to train the DQL agent. It starts by initializing our Experience Replay buffer, which stores a sequence of (s_t, a_t, r_t) values representing, each of the agent's interactions with the environment at time-step t. The off-policy nature of the agent comes from the use of ϵ-greedy action policy, where we select the best action with probability (1- ϵ) and a random action with probability ϵ. During training, this ϵ is annealed linearly between 1 and 0.1, decreasing with each time-step We then train the agent for a given number of episodes, which typically end when the agent losses, updating the parameters periodically using random batches of experiences sampled from the Experience Replay buffer.

4.2. Deep Deterministic Policy Gradient Agent

While DQL has been tested rigorously on Space Invaders, DDPG has not, which allows us to test how well DDPG adapts environments with high dimensional

state spaces, but low (discrete) action spaces. Since DDPG improves greatly upon DQL's techniques, we see many common characteristics, like the use of the Experience Replay buffer and of CNNs to extract features. However, they differ greatly on their approach to estimate the optimal action policy.

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$
  end for
end for

Figure 2. DQL Pseudocode [1]

To do this, DDPG implements an actor-critic architecture, where the parameterized actor function μ(s|θμ) specifies the current action policy by deterministi-cally mapping states to specific actions. The critic Q(s,a), on the other hand, is learned using the Bellman equation, as is done in DQL. DDPG the applies the chain rule with respect to actor parameters, thus getting that the update rule for the actor function by,

(6) $$\theta_\mu = \theta_\mu - \alpha \left[\nabla_a Q(s,a|\theta_Q)|_{s=s_t,\, a=\mu(s_t)} * \nabla_{\theta_\mu} \mu(s|\theta_\mu)|_{s=s_t}\right]$$

which was proved by [7] to be the gradient of the policy's performance.

Since the critic network being updated is also used to update the target network **(4)** and **(5)**, this can easily make the critic update prone to diverge. This is solved by creating a copy of Q and μ, called Q' and μ' to handle target value calculations. Just like in DQL, these target functions are updated by gradually having them track the weights of the learned networks. Finally, the off-policy nature of the agent comes from the use of an exploration policy which adds noise *N* sampled from an Ornstein-Uhlenbeck process [7] to the selected action. E.g. *action* = μ(s|θμ) + *N*.

As seen in figure 3, and like the DQL algorithm, the DDPG agent initializes the Experience Replay buffer, as well as the actor-critic functions. The replay buffer works just like the one in DQL The agent is trained for a set number of game episodes, taking actions based on the mentioned off-policy, and updating the weights of the critic and actor networks based on **(4), (5),** and **(6)**.

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t = 1, T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q)^2)$
    Update the actor policy using the sampled gradient:

$$\nabla_{\theta^\mu}\mu|_{s_i} \approx \frac{1}{N}\sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu}\mu(s|\theta^\mu)|_{s_i}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end for
end for

Figure 3. DDPg Pseudocode [7]

5. Experiments and Results

To ease the performance comparison between DQL and DDPG, a random agent was created and is used as a baseline for DDPG and DQL algorithms. Using random agent as a baseline helps detect bugs when implementing DQL and DDPG because if the algorithms implemented aren't training correctly, their performance would be similar or worse than that of the agent's. On both cases, 3 frames are always skipped between every action that the agent makes, repeating the last action for 3 frames before making the agent select another action based on the environment. All models were trained using Pytorch and an NVIDIA 970GTXm GPU.

The DQL algorithm was then implemented following the experiment outlined in [1]. The created DQL agent uses Adam optimizer with 0.001 learning rate, and betas=(0.9, 0.999). It also uses mean squared error as a loss function and has an experience buffer size of 100000 elements. It was trained for 60 epochs, each performing 45,000 CNN parameter updates with batches of size 32 sampled from the Experience Replay buffer. Rewards were also clipped between -1 and 1 during training to help bound the magnitude of the gradients during backpropagation. Finally, the CNN implemented for the DQL agent consisted of :

- First hidden layer: 16 8x8 convolutional filters with stride=4 and ReLU non-linearity

- Second hidden layer: 32 4x4 convolutional filters with stride=2 and ReLU non-linearity
- Third hidden layer: 256 fully-connected rectifying units.
- Output layer: fully connected linear layer with 6 outputs (number of permitted actions in Space invaders).

The DDPG algorithm was then implemented following the experiment outlined in [7]. The created DDPG agent uses Adam optimizer with 0.001 learning rate for the actor, and 0.0001 for the critic and betas=(0.9, 0.999). It also uses mean squared error as a loss function for the critic and has an experience buffer size of 100000 elements. It was trained for 20 epochs, each performing 15,000 CNN parameter updates with batches of size 32 sampled from the Experience Replay buffer. Rewards were also clipped between -1 and 1 during training to help bound the magnitude of the gradients during backpropagation. Finally, the CNN implemented for the DDPG agent consisted of the same structure for both the actor and critic with a small difference in the 4th hidden and output layers:

- First hidden layer: 16 8x8 convolutional filters with stride=4 and ReLU non-linearity
- Second hidden layer: 32 4x4 convolutional filters with stride=2 and ReLU non-linearity
- Third hidden layer: 400 fully-connected rectifying units.
- Fourth hidden layer: 300 fully-connected rectifying units. Here the critic injected 6 inputs to the layer representing the actions, thus meaning that it had 406 inputs and while the actor had 400 inputs here.
- The critic outputted a single value while the actor outputted 6 (one for each action).

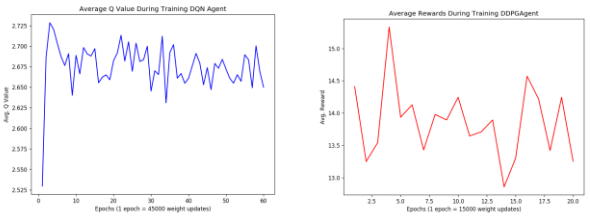
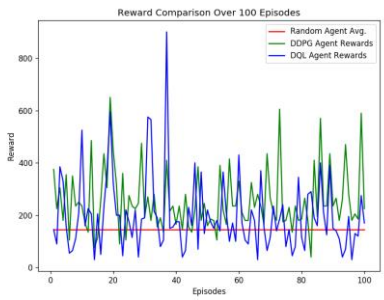


Figure 4. Average Q value (DQL Agent, Left) and average reward per epoch during training (DDPG agent, Right). Bellow is a performance comparison between Random, DQL, and DDPG agents



Rewards Achieved By Each RL Algorithm			
RL Algorithm	Avg. Reward Training	Avg. Reward Test	Top 5 Rewards Test time
DDPG	13.7	255.05	650, 605, 590, 570, 485
DQL	11.9	196.85	900, 595, 575, 565, 525
Random		144.5	555, 385, 305, 275, 245

Table 1. Results from testing the agents

6. Conclusion

Although the average Q value returned per epoch converged nicely during training, the average reward accumulated per epoch didn't seem to improve significantly throughout training. Nevertheless, as seen in table 1, the DQL agent tended to achieve relatively high scores while testing. The same was present during the DDPG agent's training, where DDPG did not show significant improvement during training, but still performed well during testing. This tendency to show little improvement during training was likely due to the decision to use very simple CNNs on both DDPG and DQL, as well as the fact that each input example state passed to the CNNs consisted of a single pre-processed image without stacking. Nevertheless, the simple CNN structures and small input examples passed to the CNNs did reduce significantly the computational complexity of the models during training, speeding up the process while still achieving good testing performance. For the future, more GPUs would be used to train the models and more complex CNNs would also be designed.

References

[1] Mnih, Volodymyr, et al. "Playing atari with deep reinforcement learning." arXiv preprint arXiv:1312.5602 (2013).

[2] Van Hasselt, Hado, Arthur Guez, and David Silver. "Deep Reinforcement Learning with Double QLearning." AAAI. 2016.

[3] Lillicrap, Timothy P., et al. "Continuous control with deep reinforcement learning." *arXiv preprint arXiv:1509.02971*(2015).

[4] Tesauro, Gerald. "Temporal difference learning and TD-Gammon." Communications of the ACM 38.3 (1995): 58-68.

[5] Dayan, Peter, and C. J. C. H. Watkins. "Q-learning." Machine learning 8.3 (1992): 279-292.

[6] Sutton, Richard S., et al. "Policy gradient methods for reinforcement learning with function approximation." Advances in neural information processing systems. 2000.

[7] Silver, David, et al. "Deterministic policy gradient algorithms." Proceedings of the 31st International Conference on Machine Learning (ICML-14). 2014.
[8] Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks." Advances in neural information processing systems. 2012.
[9] Brockman, Greg, et al. "OpenAI gym." arXiv preprint arXiv:1606.01540 (2016).