Reinforcement Learning for Video Games

Gitika Meher
Department of Electrical and Computer Engg.
University of California San Diego
9500 Gilman Dr, La Jolla, CA 92093
gkarumur@eng.ucsd.edu

Nasha Meoli
Department of Electrical and Computer Engg.
University of California San Diego
9500 Gilman Dr, La Jolla, CA 92093
nmeoli@eng.ucsd.edu

Ambareesh S J
Department of Electrical and Computer Engg.
University of California San Diego
9500 Gilman Dr, La Jolla, CA 92093
asreekum@ucsd.edu

Farheen Ahluwalia
Computer Science Engg.
University of California San Diego
9500 Gilman Dr, La Jolla, CA 92093
fahluwal@ucsd.edu

Gokce Sarar
Electrical and Computer Engineering
University of California, San Diego
San Diego, CA 92093
gsarar@ucsd.edu

Abstract

We aim to investigate various reinforcement learning techniques as they apply to two games on the OpenAI Gym environment; one from the discrete space and the other from the continuous space. We implement variants of the following algorithms: Policy Gradient, DQN and a 'hybrid method' known as Actor-Critic with the aim of comparing the number of episodes it takes for the model to solve the environment. In addition to this we used a trained model for transfer learning on a new game in the OpenAI Retro environment. We then collate the results, comparing and analyzing the various models.

1 Introduction

In this project, we aim to implement various RL techniques for learning to play Lunar Lander[1] and Car Racing [2] via the OpenAI Gym[3] toolkit. The goal of Lunar Lander is to land a bot safely on the landing pad (marked in between two ags). There are four discrete actions available for the bot: do nothing, fire left orientation engine, fire main (upwards thrust) engine, fire right orientation engine. The goal of the car racing game is to get from the start of the track to the end of the track while keeping the car on the track. The car racing is implemented in continuous space and must be discretized to a suitable action space. Detailed explanations of these techniques is mentioned in sections below.

2 Motivation

With the historic achievement of AlphaGo, developed by Google, winning 4 Go games (in a series of 5) against Lee Sedol in 2016 (the World Champion) reinforcement learning is in an exciting period. The game of Go proved to be significantly more difficult for machines to solve than chess. This is due to the drastically higher number of possible configurations of the game board, it is an impossible task to solve with brute force alone. Combining Deep Learning and Reinforcement Learning has opened up a huge maze of possibilities. We want to experiment on different video games using different algorithms branching from simple Markov Models.

3 Literature Review

Reinforcement learning (RL) dates back to the early days of work in statistics, psychology, neuroscience and computer science. In the last couple of years, it has attracted increased interest in the machine learning and artificial intelligence communities. Recently, RL has been used for robot motion planning, self-driving cars, playing games in simulated environments and for other similar applications. In particular, RL has been applied in Atari games in multiple works in the past [4]. In Atari games the future (image-)frames are dependent on control variables or actions as well as previous frames. Even though frames in Atari in are not composed of natural scenes, frames are high-dimensional in size, can involve tens of objects with one or more object being controlled by the actions directly and many other objects being influenced indirectly, can involve entry and departure of objects, making the scenes fairly complex. Oh, Junhyuk et al proposed and evaluated two deep neural network architectures that consisted of encoding, action-conditional transformation, and decoding layers based on convolutional neural networks and recurrent neural networks, and were the first to make and evaluate long-term predictions on high-dimensional video conditioned by control inputs.

Deep Q-Learning Networks (DQN) are based on Q-Learning which was introduced by Watkins in 1989[5]. Q-learning is used to learn an action-value function Q(s,a) which gives the expected long term reward of the agent if it takes action a in state s. Knowing Q(s,a) for all states and actions simplifies finding an optimal policy in to a simple greedy algorithm. Q-learning is proven to converge for a finite state-space. When states consist of real numbers, this state-space becomes infinite and its state-action table can no longer be stored in memory. In DQN, a neural network is trained to learn an approximation of the action-value function in such scenarios.

The Asynchronous Actor Critic algorithm[6] was released by Google's DeepMind group in 2016 , and it made a splash by essentially obsoleting DQN. It was faster, simpler, more robust and able to achieve much better scores on the standard battery of Deep RL tasks. The algorithm was explored for various application including Box2d's car-racing.

In the paper World Model[7], an approach to train a model quickly in an unsupervised manner to learn a compressed spatial and temporal representation of the environment is described. By using features extracted from the world model as inputs to an agent, they train a very compact and simple policy that can solve the required task. They discuss the use of an autoencoder, RNN and a controller unit to solve reinforcement learning problems with continuous output space like car racing. Minh, Silver et al [8] made the first deep learning model to successfully learn control policies directly from high-dimensional sensory input using reinforcement learning. They used a model with a CNN, trained with a Q-learning, with input raw pixels and output as a value function estimating future rewards. They tested their algorithm on seven Atari 2600 games from the Arcade Learning Environment, with no change in the architecture or learning algorithm, and was then outperforming all previous approaches on six of the games and surpasses a human expert on three of them.

Policy Gradient is an end-to-end Deep Reinforcement Learning technique that is also rooted in Markov Decision. Jaderbag et all[9] introduced, the Spatial Transformer, which explicitly allows the spatial manipulation of data within the network which is a differential module which can be inserted into existing convolutional architectures, giving neural networks the ability to actively spatially transform feature maps, conditional on the feature map itself, without any extra training supervision or modification to the optimisation process. Spatial transformers results in models which learn invariance to translation, scale, rotation and more generic warping, resulting in state-of-the-art performance on several benchmarks, and for a number of classes of transformations.This knowledge inspired an experiment in our model for Lunar Landing Reinforcement Learning.

Actor-critic methods combine the policy and value networks, using the feedback from the critic(value function) to teach the agent (policy). These methods provide a suitable trade-off between variance and bias [10].

3.1 State of the Art

Asadi et al[11],used a direct approach to compute multi-step predictions where they proposed a simple multi-step model that learns to predict the outcome of an action sequence with variable length. The result is currently state of the art for our problem. The model can also provide the outcome of running a policy rather than just a fixed sequence of actions. The results on Atari Breakout shows that the multi-step model outperforms the other one-step model in terms of frame prediction after multiple time-steps.

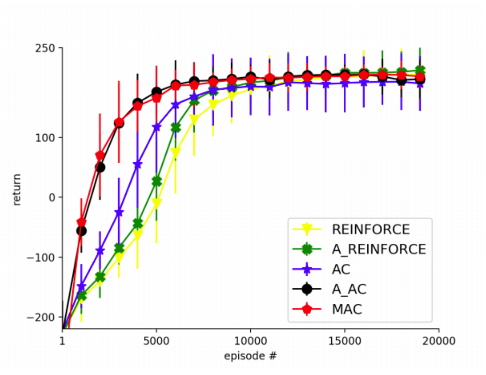

Figure 1: Allen and Asadi et al., 2018

Google DeepMind has a canonical paper on using Asynchronous Methods for Deep Reinforcement Learning [6], where a lightweight framework for deep reinforcement learning using asynchronous gradient descent for optimization of deep neural network controllers. They introduced asynchronous variants of four standard reinforcement learning algorithms and showed that parallel actor-learners have a stabilizing effect on training allowing all four variants(Mean Actor Critic, and variants of DQN) to successfully train neural network controllers. The best performing method, an asynchronous variant of actor-critic, has the best performance compared to available methods, on the Atari domain while training for half the time on a single variantsmulti-core CPU instead of a GPU. Furthermore, asynchronous actor-critic succeeds on a wide variety of continuous motor control problems as well as on a new task of navigating random 3D mazes using a visual input. In terms of general umbrella algorithms used for the purpose, asynchronous actor-critic is the best algorithm currently. Figure 2, shows that these methods were also devised for other games for Atari, like Pong.

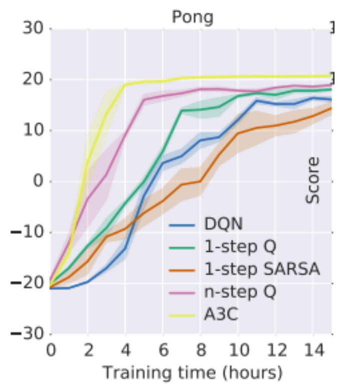

Figure 2: Google DeepMind, 2016

4 Datasets

We are working primarily on OpenAI's LunarLander-v2 environment[1]. Our primary objective in this project was experimenting with different reinforcement techniques. To this end we decided to choose a convenient environment which would enable us to work and improve on the learning algorithms faster with limited available computational resources.

4.0.1 The Environment

The state vector consists of the coordinates,velocities, angle, angular velocity, and ground contact information of the lander. Six of the eight variables are continuous, with only the last two being discrete (0 or 1). There are four discrete actions available for the bot: do nothing, fire left orientation engine, fire main (upwards thrust) engine, fire right orientation engine. Landing pad is always at coordinates (0,0). Coordinates are the first two numbers in state vector.The agent must learn how to control 4 “levers” to safely land the module. It’s episodic; the agent is able to have multiple tries and restarts. While the output (action) is discrete, the input (state space) is infinitely enumerable across 8 continuous dimensions. The high dimensionality and continuous state space pose especially challenging examples of the lemmas of delayed reward, credit assignment, and exploration vs. exploitation. This makes the problem doable but not very straightforward at the same time.

The environment in lunar Lander is continuous both in states and action spaces. The screenshots (from each frame, a 96 x 96 x 3 RGB image) of the Car Race track were captured as the states. The agent is the car. The action space when discreteised (as discussed below) has three distinct actions (steering (left or right), acceleration (gas) and deceleration (brakes)). It is also episodic.

4.0.2 Rewards

In Lunar Lander the agent receives a positive reward for landing closer to the target zone. The agent gets a negative reward for moving further away from the landing zone. Also, the agent receives a small negative reward every time it carries out an action. This is done in an attempt to teach the agent to land the rocket as quickly and efficiently as possible. If we were to simply give it a reward for landing the rocket the agent would be able to do it but it might take much longer than it should and use excess fuel. This will give us the flexibility of exploring and contrasting various approaches with and without negative rewards in later stages.

Reward for moving from the top of the screen to landing pad and zero speed is about 100..140 points. If lander moves away from landing pad it loses reward back. Episode finishes if the lander crashes or comes to rest, receiving additional -100 or +100 points. Each leg ground contact is +10.

Firing main engine is -0.3 points each frame. Solved is 200 points. Landing outside landing pad is possible. Fuel is infinite, so an agent can learn to y and then land on its first attempt.

In Car racing, the reward was was -0.1 every frame and +1000/N for every track tile visited by the agent in an episode, where N is the total number of tiles in the track.

We use Open AI gym along with box2d for the rendering of the simulation and the underlying physics engine respectively. Box2d is a 2D rigid body simulation library for games.

5 Methodology

5.1 Policy Gradient

Policy Gradient tries to optimize the policy space, by directly approximating the action probabilities. So in Policy Gradients, we use a neural network (or other functions) to directly model the action probabilities[12]. Based on the rewards the agent gets, we train the model to learn the best policy given the state. For training of the network, we sample from the softmax of the actions to create the target. Nevertheless, with most of the games, we don’t know if an action is good or bad till we reach the end of the game. What is done is all the gradients are calculated as before and if at the end of the game, the agent won, we multiply all of the gradients with 1, whereas if the agent lost, we multiply all of the gradients with -1, so that in the long run the good actions are going to get promoted more.

With LunarLander we get a reward after each move and at the end we have an accumulated score at the end of each episode. We call the game a success if the accumulated score is above a threshold and as a failure if the accumulated score is under a threshold. The threshold is chosen according to the played episodes in a batch. In our case we used batches of 100 episodes and we set the threshold according the the percentile we set as success or failure. We built upon the implementation of the policy gradient network with a thresholding system for just positive reward as described in [13].

So we can apply policy gradient to LunarLander in the sense that we multiply the gradients related good episodes with 1 and related to bad episodes with -1. However, as the agent learns and get better, our understanding of good and bad episodes change. That's why, we got an implementation of the policy gradient network with a thresholding system. We have two thresholds for positive and negative rewards. For the actions, whose reward is above the threshold for positive reward, we associate them with a reward of 1 and for the actions which are below the threshold for negative reward (penalty), we associated them with a reward of -1 and for the ones which are in-between we gave a reward of 0. For training a custom cross-entropy loss is implemented, where the reward is incorporated as can be seen below:

$$L = -\sum_{c=1}^{M} r_o y_{o,c} \log(p_{o,c})$$

where y is the targets, p is the softmax outputs, o is the observations, c is the classes and r is the rewards as 1,-1 or 0.

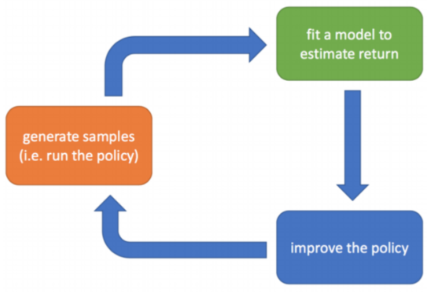
fit a model to estimate return
generate samples (i.e. run the policy)
improve the policy

Figure 3: Reinforcement Learning with Policy Gradient

5.1.1 Implementation Details

We used two types of feed-forward fully connected architecture whose dimensions can be seen in Fig. 4:

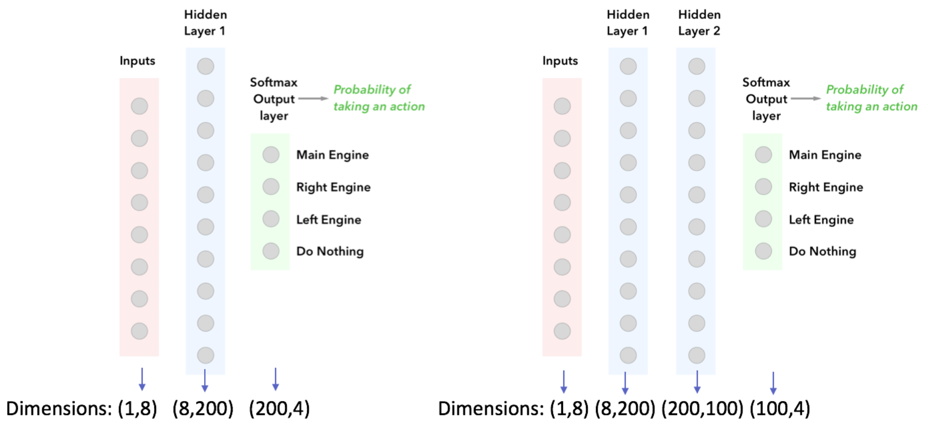

Figure 4: Architectures for Policy Gradient Method: (Left) architecture with one hidden layer, (right) architecture with two hidden layers

We are going to specify which architecture we used in the Results based on their performances. We used Adam optimizer with an initial value of 0.01 or 0.001, which we are going to also specify in the Results section.

We applied 4 sets of reward combination, where we apply the threshold according to the percentile values among the episode. Below we specify the percentile values we used for thresholding:

1)Positive: 80%, Negative: 0% (Only positive rewarding)

2)Positive: 80%, Negative: 20%

3)Positive: 80%, Negative: 50%

4)Positive: 0%, Negative: 20% (Only negative rewarding)

5.2 Q-Learning and DQN

Q-learning learns the activation-value function Q(s,a), which means that for a particular state it learns the values of each action. For the current state, an action is sampled and then its associated reward and new state are found. Based on this new state, the next action is determined which has the maximum Q-value.

So with Policy Gradients we directly try to optimize in the action space, whereas Q learning tries to approximate the Q function. In other words, Q learning tries to infer the optimal policy as $argmax_a Q(s, a)$, whereas Policy Gradients try to optimize the policy space, by directly approximating the action probabilities. By combining Q-learning with Deep Neural Networks, more complicated games could be learned. With a deep neural network architecture, the aim is learning the value of all actions given a state.

5.2.1 DQN with Replay Memory

With neural networks, we aim to give the network an input batch, which is a good approximation of all the input space. But with reinforcement learning, we learn as we play the game in an episode, which means that our input batch consist of very similar samples. That's why it has been shown that using a replay memory from which we can sample for training would help the network to learn much better. That's how we can have iid inputs so that the network won't learn biased. During training, a random batch of tuples $<$ s,a,r,s0 $>$ - where s denotes the next state, a is action, r is reward, s0 is the previous state- are picked from memory and network is trained on them. This helps network learn better using a set of varying experiences.

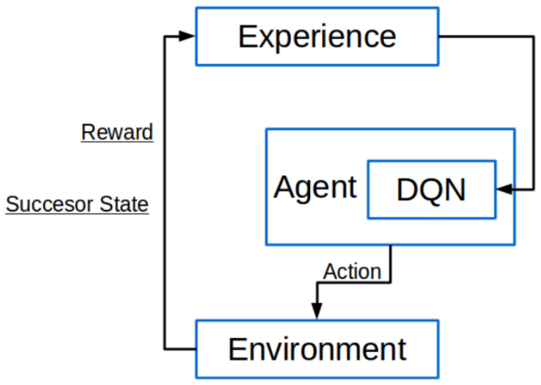

Figure 5: DQN Model with Replay Memory

5.2.2 DQN with Lunar Lander

DQNs are mostly trained on consecutive images as game scenes with a CNN networks so that it can learn the states of the game. By giving consecutive images, the state of the agent is implied. In a simpler game like LunarLander, since we have 8-dimensional states as given, we can also train the network directly with those states. That's why in this project we worked on those states directly to experiment with different learning methods. For DQN models we used the same network architectures for comparison purposes whose architecture match the one we used in Policy Gradient with two hidden layers and can be seen in Fig. 4 on the right with two hidden layers.

For optimizer, we used an Adam optimizer with an initial learning rate of 0.001. We implemented based on [14].

5.2.3 DQN Target Calculation and Loss Function

As explained before the optimal policies can be achieved by estimating the Q(s, a) function, which gives us an estimate of the discounted sum of rewards of taking action a in state s. Playing the action with the maximum Q-value in any given state means that we play optimally.

The input of our neural network will be an initial state s, and its output will be its estimate of $r(s, a) + \gamma max'(Q(s', a'))$, where r is the reward that was obtained when playing action a from state s, γ is our discount rate usually high, and Q(s', a') is the predicted Q value for all actions from the next state using our current model. In Fig. 6, the target formulation can be seen and in Fig. 7 the MSE loss function for DQN can be seen.

$$\underbrace{\hat{Q}(s,a)}_{\text{Q target}} = \underbrace{r(s,a)}_{\text{Reward of taking that action at that state}} + \underbrace{\gamma max_{a'} Q(s',a')}_{\text{Discounted max q value among all possibles actions from next state.}}$$

Figure 6: QLearning Target function

$$loss = \Big(\underbrace{\overset{\text{Reward}}{r} + \overset{\text{Decay Rate}}{\gamma} \max_{a`} Q(s`,a`)}_{\text{Target}} - \underbrace{Q(s,a)}_{\text{Prediction}}\Big)^2$$

Figure 7: QLearning loss function as MSE

Nevertheless, for Q-learning, Huber Loss in Fig. 8 with $\delta = 1$ is shown to be a better loss which can be seen in Fig. 9, due to the fact that it doesn't punish the larger values as harshly, which can hurt the learning process.

$$L_\delta(a) = \begin{cases} \frac{1}{2}a^2 & \text{for } |a| \le \delta, \\ \delta(|a| - \frac{1}{2}\delta), & \text{otherwise.} \end{cases}$$

Figure 8: Huber Loss

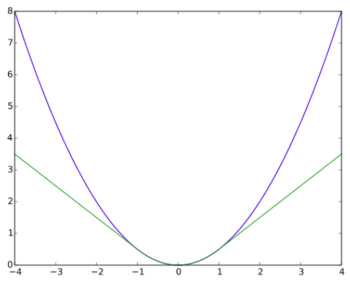
Figure 9: Huber Loss (green) vs MSE (blue)

5.2.4 DQN with Car Racing-V0

The DQN learning algorithm was implemented for the 2-D Car Racing in the Gym environment `(CarRacing-v0)`.The approach was similar to that taken in training the architecture to play Lunar Lander. The objective is to train a the policy network (a DQN in this case) to control the agent to race through a race track from the frames. The episode ends when all the race track tiles are visited or the car drives outside the game boundaries.

The car racing poses a different set of challenges from the lunar lander game as the OpenAI Gym environment is continuous and must first be converted to a discrete space. This required that we choose an action space with countable discrete actions for the purposes of the reinforcement learning problem. The second unique aspect was the longer length of episodes due to the particular definition of failure.

5.2.5 DQN with one network

In this model we only have one network which is updated and also used for the target estimation as we described in the target and loss function section. This type of DQN has the negative side that the model is changing all the time so that the targets are also changing and the output tries to catch a changing output, which makes the convergence harder. That's why the learning is not stable. As solution to it, DQN with fixed Q-Targets are suggested.

5.2.6 DQN with fixed Q-Targets

In this version of DQN, we have two networks, where one of them is kept updated all the time with backpropagation, whereas the second network, the target network, kept fixed for N iterations to produce targets. So that with the first network we can learn the Q-values for fixed target network, where we equate it to the first network in N episodes so that it can get up-to-date.

As a variant of this version we also tested with soft update, in which we update the target network as frequent as we update the first network. However the updating of the target network is not performed by equating directly to the first network, but interpolating the target network's old value with the new value of the first network. For interpolation we use 0.001 multiplied with the new value of the first network.

$$target\_network = (1 - 0.001) * target\_network + 0.001 * first\_network$$

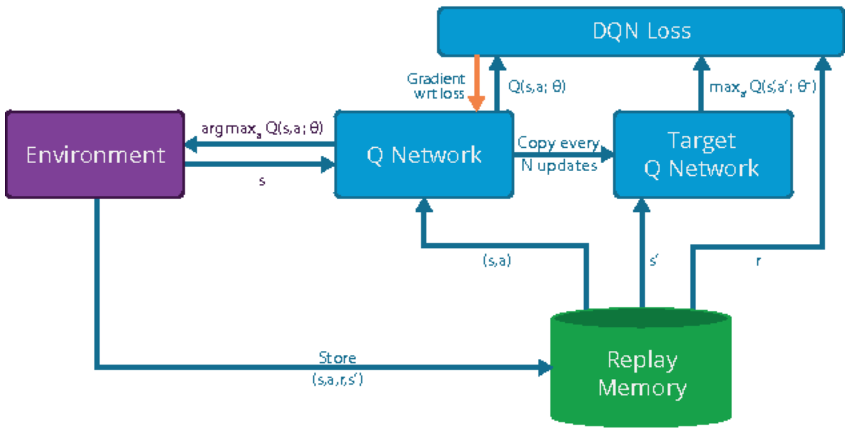

Figure 10: Fixed Target DQN Model

5.2.7 Dueling DQN

Dueling network represents two separate estimators with model-free reinforcement learning. : one for the state value function and one for the state-dependent action advantage function. The main benefit of this factoring is to generalize learning across actions without imposing any change to the underlying reinforcement learning algorithm. We can decompose Q(s,a) as the sum of:
V(s): the value of being at that state
A(s,a): the advantage of taking that action at that state (how much better is to take this action versus all other possible actions at that state).

$$Q(s,a;\underset{\text{Common network parameters}}{\boxed{\theta}},\underset{\text{Advantage stream parameters}}{\boxed{\alpha}},\underset{\text{Value stream parameters}}{\boxed{\beta}}) = V(s;\theta,\beta) + \left(A(s,a;\theta,\alpha) - \underbrace{\frac{1}{\mathcal{A}}\sum_{a'} A(s,a';\theta,\alpha)}_{\text{Average advantage}}\right)$$

Figure 11: Target for DDQN

With DDQN, we want to separate the estimator of these two elements, using two new streams as shown in the image and then combine them through a special aggregation layer to get an estimate of Q(s,a).As a consequence, by decoupling we're able to calculate V(s). This is particularly useful for states where their actions do not affect the environment in a relevant way. In this case, it's unnecessary to calculate the value of each action.

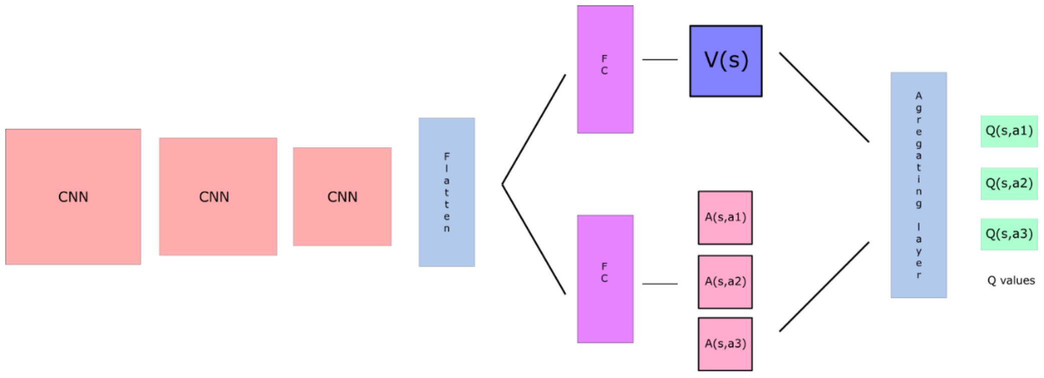

Figure 12: Dueling DQN Model

5.3 Actor Critic

Actor critic is a “hybrid method” which uses elements from both the techniques.DQN and policy value based methods, both perform adequately on this task and learned to complete the simulation. But they take more time to learn because of the randomness and state space size of varying degrees that is built in the training process of each of those methods.. DQN first learns the mapping of all the state action pairs before it is able to learn to simulate the landing. This requires the DQN network to learn the approximate mapping of a very large space of state, action pairs. The policy gradient method updates the weights after every episode using the total reward accumulated. This leads to weighting all the correct state action pairs of the episode to be encouraged the policy gradient method weights the actions after the episode has ended using the total reward collected, thus leading to weighting all the state action pairs of the episode by the same amount. This implies all the some state action pairs are encouraged in some cases and some good actions are penalized in others. On average however this technique is able performs well but takes a longer period of time to converge. Actor critic method combines both the DQN and Policy gradient methods to solve the credit assignment problem of Policy Gradient method. The model has 2 components a. a Critic that measures how good the action taken is (value-based) b. an Actor that controls how our agent behaves (policy-based).

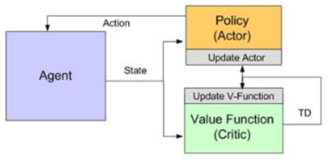

Figure 13: Actor Critic Model

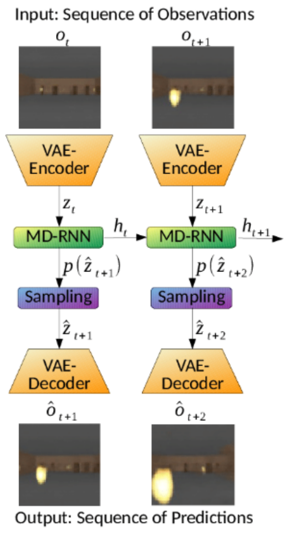

Figure 14: VAE based World Model

6 Experiments

6.1 VAE based World model

This approach uses the the OpenAI Gym environment to wrap around the game to make useful parameters such as pixel data observable. Vision model (Convolutional Variational Autoencoder)accepts pixel data (224x320x3) from the observation as input and starts encoding it. The encoded pixel data is exported as z (a latent vector) and passed to the Memory model and Controller (single layer linear model). The Memory model (Recurrent Neural Network with Mixture Density Network as output) accepts z, hidden state, h (from itself), and actions as input. The Controller takes in z and h and outputs actions for the agent to take based on these parameters. Actions decided by the Controller are fed back into the Memory model and to the environment. The environment wrapper translates actions from the Controller into the game where further observations take place. Most of the information regarding the environment and previous steps taken is contained within the Vision and Memory models. Because of this, a World Model can be constructed from these two.

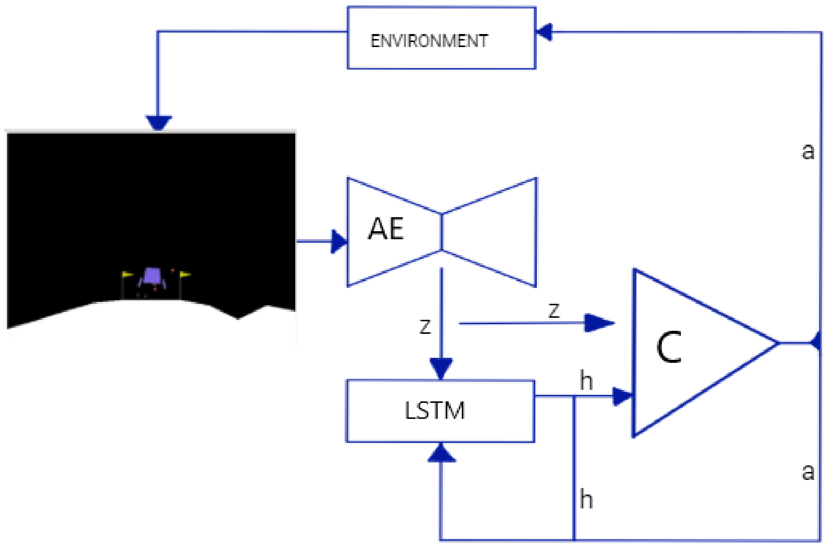

Figure 15: World Model

6.2 Transformer Model Adaptation for Reinforcement Learning

In 2016 Jaderberg et al [9] introduce a Spatial Transformer module, which can be included into a standard neural network architecture to provide spatial transformation capabilities. The action of the spatial transformer is conditioned on individual data samples, with the appropriate behaviour learnt during training for the task in question. Unlike pooling layers, where the receptive fields are fixed and local, the spatial transformer module is a dynamic mechanism that can actively spatially transform an image (or a feature map) by producing an appropriate transformation for each input sample. Notably, spatial transformers can be trained with standard back-propagation, allowing for end-to-end training of the models they are injected in [9].

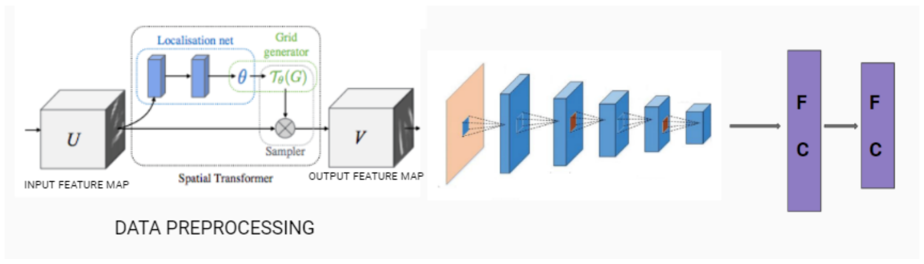

Figure 16: Transformer Model

We use the localization net to extract the best features from the image, after applying the transformations. The resultant output feature map is fed into the input of the DQN network. The transformation is performed on the entire feature map . This allows networks which include spatial transformers to not only select regions of an image that are most relevant (attention), but also to transform those regions to a canonical, expected pose to simplify recognition in the following layers.

6.3 Transfer Learning

With the best trained model, We will implement transfer learning on a game similar to Lunar Lander. For this purpose, SpaceX-like Falcon Rocket Lander environment present as a part of the gym-extensions is chosen. Rocket Lander

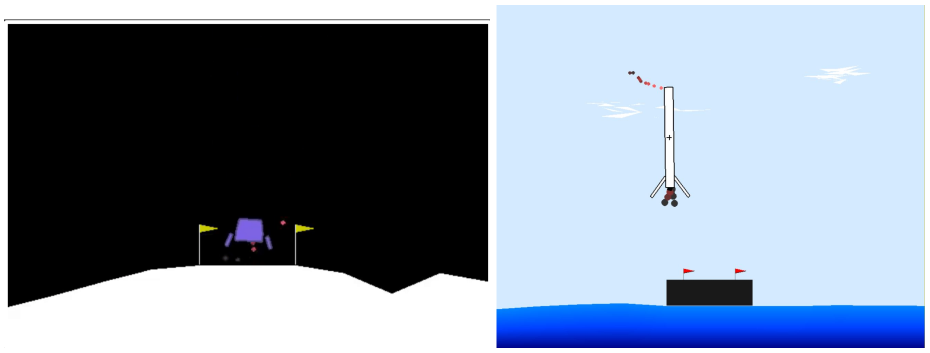

Figure 17: Lunar Lander and Rocket Lander

7 Results and Analysis

7.1 Selection of Reward Threshold for Policy Gradient Learning

In this part we used the two layer neural network, which is described in the Methods part. We used a learning rate of 0.01 with Adam Optimizer. As mentioned in Methods we used the following thresholding percentiles in the episode for training:

1)Positive: 80%, Negative: 0% (Only positive rewarding)

2)Positive: 80%, Negative: 20%

3)Positive: 80%, Negative: 50%

4)Positive: 0%, Negative: 20% (Only negative rewarding)

We trained for 500 iterations with batches of 100 episodes in each iteration and when the agent gets a score more than 200, it is accepted as a successful play. The results can be seen in Fig. 18. We aimed to see how taking different thresholds affect the learning of agent, like if we only train based on the positive rewards or negative reward, does the agent learn at the same level. If we look at the figure, we can see that with only negative rewarding so eventually only penalizing the agent, we get lower results at the beginning, but eventually it reaches to the same level as only positive rewarding. If we combine positive and negative rewarding at the percentiles 50% and 80%, we can't see a significant increase, but if we combine at a level of 20% and 80%, there is a big jump in the score as can be seen in the figure. The reason is setting the negative threshold as 50 among the episodes, leads some not so bad episodes to be evaluated as bad, so that the agent isn't trained optimally. Nevertheless, when we decrease that threshold to 20, we penalize the agent for really bad episodes and give positive reward to really good episodes and treat the actions in-between as neutral. That's how it can differentiate between good and bad episodes really good and learn at a very good level.

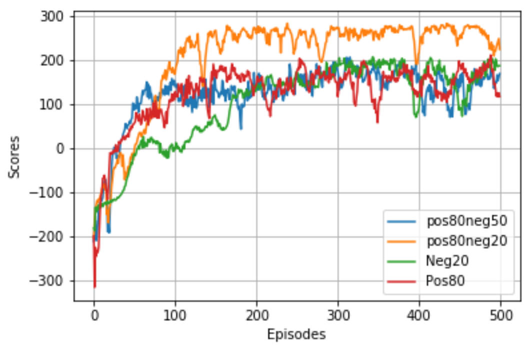

Figure 18: Effects of different thresholding methods on Policy Gradient

7.2 Depth and the Learning Rate effect on Policy Gradient

After observing different thresholding effects, we experimented on the depth with the best thresholding method we found in the previous part: Positive: 80%, Negative: 20%. The results can be seen in Fig. 19. As can be seen when we trained with 3 layers and an initial learning rate of 0.01, the agent learns much faster and gets really good results quicly but we also saw severe oscillations in the scores at the later stages of the learning. This could indicate that the learning rate might be too high. That's why we trained with a lower learning rate, which is 0.001 and as can be seen it learns at a very slow rate, which is not desired in reinforcement learning. We can conclude that the deeper network did better, although it had some oscillations at later stages.

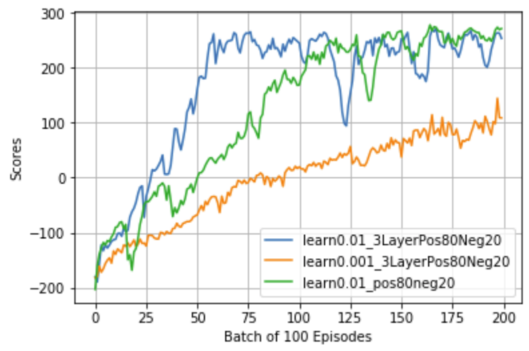

Figure 19: Depth and the Learning Rate effect on Policy Gradient

7.3 Target Update Rules

As mentioned in Methods, we implemented three versions with target network. In the first one, we basically have only one network where we learn and create the targets with the same network. In the second one we do an update of the target network in every 100 step and in the third one, we softly update the target network with an interpolation factor of 0.001. As can be seen in Fig. 20, using the same network gave the worst results as expected. In Fig. 20, on the left we give the results per episode, where on the right we give the mean score for the last 100 episodes, since it is accepted by the community that if the average score of 200 is achieved in the last 100 episodes, the environment is accepted as learned. With the soft and hard update, there isn't a severe difference at the beginning, but the results with hard update oscillate much more severely, that's why soft update is more robust. As can be seen with DQN we reach the score of 200 much fater than Polciy gradient, considering that there the plot shows the result per batch of 100 episodes.

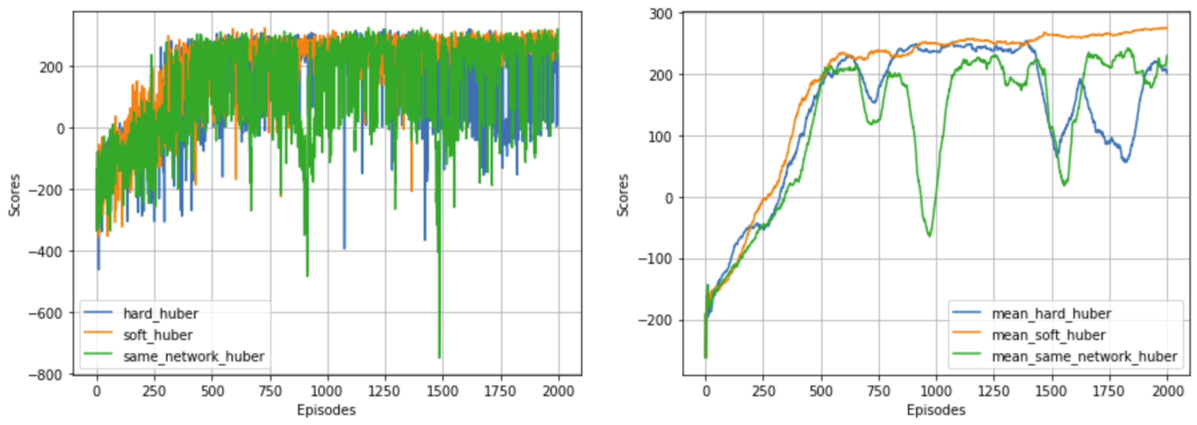

Figure 20: Comparison of Target Update Rule: (left) scores per episode, (right) mean scores of the last 100 episodes

7.4 Loss Function Comparison for Q-learning

As can be seen in Fig. 21, MSE Loss is lower than Huber loss for DQN at the beginning. Huber loss punishes the model less on the larger values as compared to the smaller values when it produces scores similar to the scores of MSE, this leads to less oscillations in learning, so that Huber loss performs better. In MSE if the error is large, the punishment is very high so that the model can change severely. Nevertheless, since our predictions and targets are closely related, a severe change in the target models can end up having harm on the learning procedure. That's why punishing the big errors less as in Huber Loss is beneficial in reinforcement learning.

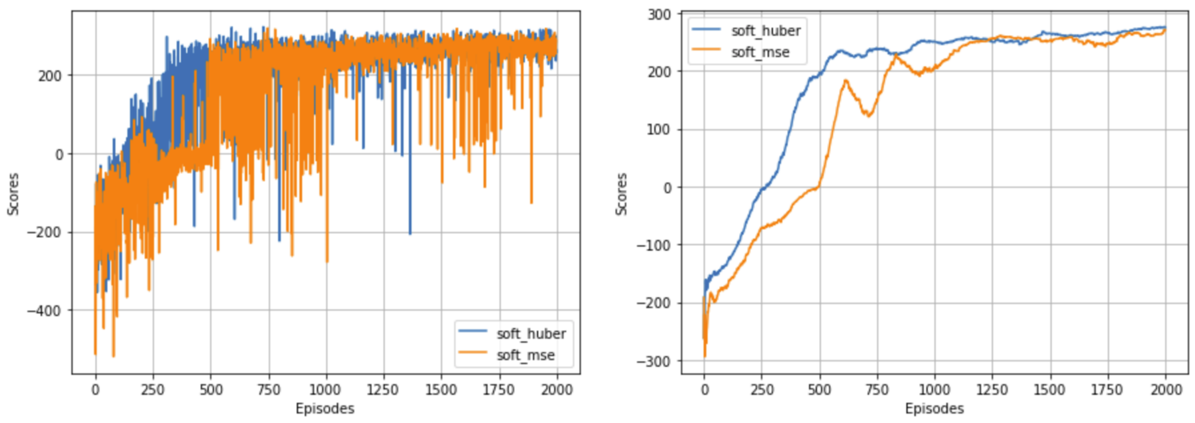

Figure 21: Loss Function Comparison for Q-learning: (left) scores per episode, (right) mean scores of the last 100 episodes

7.4.1 Comparative Analysis across all models

The following results were observed after training for Lunar Lander across different models.

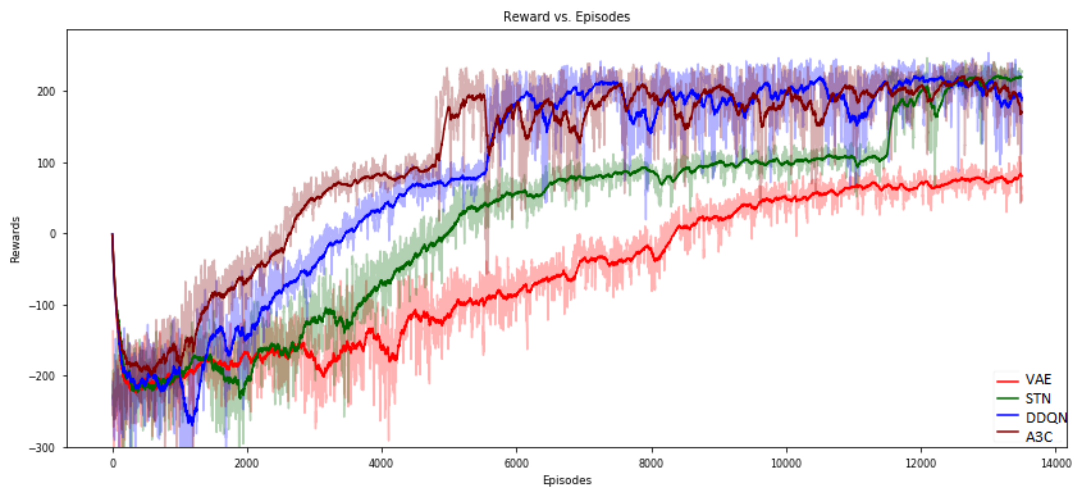

Figure 22: Comparison Across different Models

As expected Actor Critic is the best performing model of all. Our results weren't as good as the state of the art models (in terms of final scores achieved), but probably could have done better with longer training procedures.

DDQN works reasonably well and has comparable results. The convergence happens after 6000 epochs too. The model works well possibly because of incorporating the advantage model, which proportionally incorporates rewards rather than doing it directly, making it better than normal DQN.

STM, performed not as well as the other models because Spatial transformation probably doesn't work as well for the Lunar Lander because a different spatial orientation is actually a different state in the game. The spatial orientation change is definitely a plus for other tasks like image recognition, but has to handled differently for this particular task.

We are hypothesizing that VAE will eventually converge over 15000 episodes. The model, we believe, will work much better for far more complex environments, and probably was overkill for a simple game for Lunar Lander. It had too many learnable parameters and probably thus fared worse off than the simpler models which was adequate for the given environment.

8 Individual Contributions

All author's implemented their own versions of their baseline, which was later combined into a single source code to set up baselines for all other experiments. The work on different experiments was similarly divided equally. Ambareesh worked on Actor Critic models. Various experiments with DQN and explorations on Transfer Learning were carried out by Nasha. Gitika was responsible for implementing the Duelling DQN. Farheen was responsible for the VAE architecture. Gokce was responsible for Policy Gradient and baselines for DQN.

9 Particulars

Github link

Presentation

References

[1] Lunarlander-v2. `https://gym.openai.com/envs/LunarLander-v2/`, 2016. Accessed: 2019-03-09.

[2] Car racing-v0. `https://gym.openai.com/envs/CarRacing-v0/`, 2016. Accessed: 2019-03-09.

[3] Gym. `https://gym.openai.com/`, 2016. Accessed: 2019-03-09.

[4] Junhyuk Oh, Xiaoxiao Guo, Honglak Lee, Richard L Lewis, and Satinder Singh. Action-conditional video prediction using deep networks in atari games. In *Advances in neural information processing systems*, pages 2863–2871, 2015.

[5] Christopher John Cornish Hellaby Watkins. *Learning from delayed rewards*. PhD thesis, King's College, Cambridge, 1989.

[6] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937, 2016.

[7] David Ha and Jürgen Schmidhuber. World models. *arXiv preprint arXiv:1803.10122*, 2018.

[8] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.

[9] Max Jaderberg, Karen Simonyan, Andrew Zisserman, et al. Spatial transformer networks. In *Advances in neural information processing systems*, pages 2017–2025, 2015.

[10] Vijay Konda and John Tsitsiklis. Actor-critic algorithms. In *SIAM Journal on Control and Optimization*, pages 1008–1014. MIT Press, 2000.

[11] Kavosh Asadi, Evan Cater, Dipendra Misra, and Michael L Littman. Towards a simple approach to multi-step model-based reinforcement learning. *arXiv preprint arXiv:1811.00128*, 2018.

[12] Andrej Karpathy blog. Deep Reinforcement Learning: Pong from Pixels. `http://karpathy.github.io/2016/05/31/rl/`, 2016. Accessed: 2019-03-09.

[13] Donal Byrne. Landing a rocket with simple reinforcement learning. `https://medium.com/coinmonks/landing-a-rocket-with-simple-reinforcement-learning-3a0265f8b58c`, 2018. Accessed: 2019-03-09.

[14] `https://github.com/udacity/deep-reinforcement-learning/tree/master/dqn`.