LEARN2PLAY

CIS 8395 – Big Data Experience – Spring 2020

Abstract

The final paper for a Reinforcement Learning project by group Learn2Play

Group Members:

ANIRUDH CHAUDHARY

HAO NGUYEN

RICHARD MORE

QUAN LE

Table of Contents

Introduction

In this project, we are dealing with a paradigm of machine learning i.e. Reinforcement learning (RL) which holds the implementation potential in multiple disciplines, not limiting to, game theory, control theory, operations research, information theory, simulation-based optimization, multi-agent systems, swarm intelligence, statistics and genetic algorithms. It is useful to consider Reinforcement Learning as a possible solution when the problem statement deals with large state spaces or is looking to augment human behavior by providing decision support. Also, in areas where simulations are used to teach machines through trial and error while dealing with highly complex systems.

Our aim is to develop a system powered by Reinforcement Learning algorithm to achieve superhuman performance in a game and implement the learnings gained in fields like Robotics, Finance, Health & Medicine and Media & Advertising. We have decided to use the Open AI gaming platform and Atari 2600 console as the means to achieve the objective of the project. In the subsequent sections, we will be discussing about Reinforcement learning in detail, along with its types. After which, we will cover the architectural overview of the project form Reinforcement learning perspective as well as implementation perspective using AWS. Finally, we will discuss about the Reinforcement learning elements pertaining our project.

Machine Learning

Brief introduction

Before we jump into the Reinforcement Learning, we would like to give a brief overview of machine learning. As we learned from previous class, machine learning is an application of Artificial Intelligence that has the capability to automate the analysis model through learning and improving.

Have you ever wondered how did your grandmother or mother know when and where to get the best avocadoes for a best price? Back in the days, people would remember the avocado's season and best places from the word of mouth. The chance of getting the best avocadoes at the best price are usually varying. Nowadays, the Millennials tend to be busy with their city lifestyle, so they are less likely to bother remembering the last time they bought good avocadoes. If there is a predictive model to help the Millennials from wasting money, it would be done through Machine Learning.

There are three learning types in Machine Learning: Supervised Learning, Unsupervised Learning, and Reinforcement Learning.

Supervised learning

Supervised Learning is the machine learning task of learning a function that maps an input to an output based on X to y relationship where X is independent, and y is dependent.

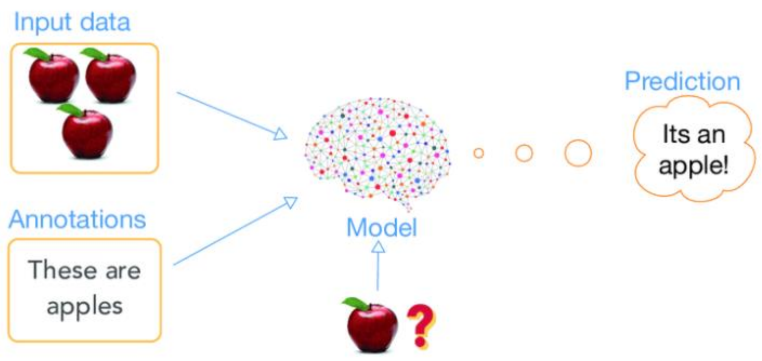


Figure 1: An example of Supervised Learning (Yan 2018)

In Figure 1, we have an input data with features like shapes and colors and the label as apple and other fruits. Model is built by learning from the set of training samples. If we feed in a testing sample, in this case, different features of a fruit. Model will give a prediction of an apple.

Unsupervised learning

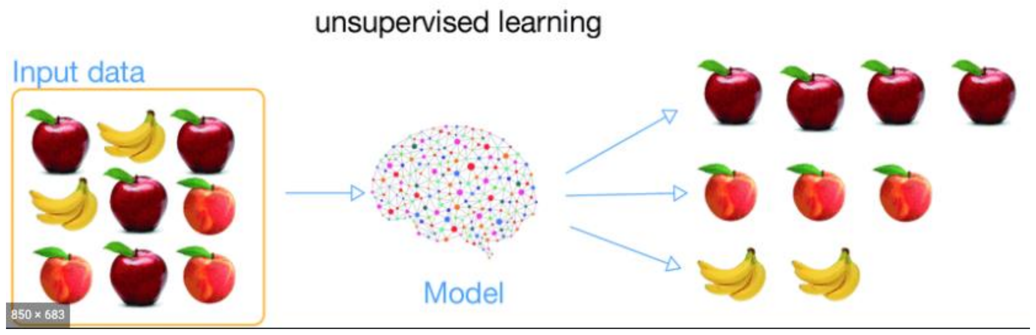


Figure 2: An example of Unsupervised Learning (Yan 2018)

While Supervised learning has X and y relationship, Unsupervised Learning has only X. It, instead, will group and interpret such data based on only the input data. In Figure 2, we have a set of fruits without any specified label. Through techniques like clustering analysis, the algorithm will group or find hidden patterns in the data. Therefore, it will return a group of apples, a group of tomatoes, and a group of bananas.

Reinforcement learning

Reinforcement Learning is an orthogonal approach meaning X and Y are independent. It emphasizes on learning the feedback through trial-and-error and continuously interacting with a dynamic environment without providing standards of correctness. The basic theory of reinforcement learning technology is "enabl[ing] the action of the system choice to obtain the largest cumulative reward value of environment" which is known as positive feedback; otherwise, the system will diminish the actions that gives negative feedback (Qiang 2011). If we relate reinforcement learning to learning how to bike, you will learn how to bike through every time you can balance and every time you fall. Eventually you will understand how to balance yourself, you will be able to ride a bike.

According to Arulkumaran who is one of the authors that conducted and published a brief survey on Deep Reinforcement Learning with IEEE in 2017, he stated that Reinforcement Learning technology contains environment, agent, policy, reward signal, value function, and model. The environment is the observation of a task or simulation. The agent is the algorithms that interact with the environment. Agent can be divided into two categories: Exploration and Exploitation. Exploration agent acts based on trial and error, while the exploitation agent acts based on knowledge gained from the environment. Policy is the Machine's stimulus-response rules or associations. Reward Signal defines whether a policy should be changed or not. Value Function shapes the agent's behavior by specifying an event's long terms goodness. Model mimics the environment and makes inferences about its future behavior.

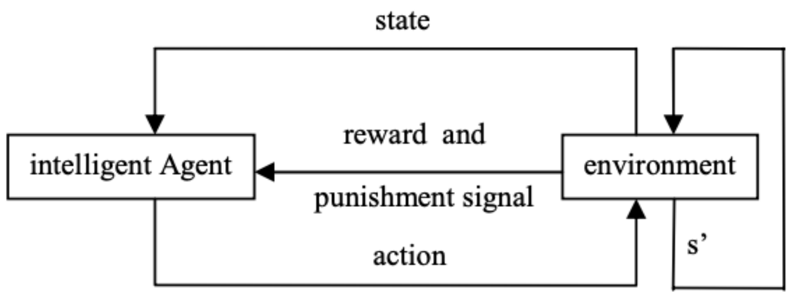


Figure 3: Reinforcement Learning Process Diagram (Qiang 2011)

In Figure 3, the Agent first accepts the input of environment's State. Agent will give an output Action to be performed on the environment. Then, the Environment will change its state. Then, Environment will give the Agent the reward or punishment signal and the new state. Lastly, the reinforcement learning model will learn an action strategy that gives the largest cumulative reward value.

Types of RL

There are two types of Reinforcement Learning: Model-free Learning and Model-based Learning.

Model-free learning

Model-free Learning is also known as direct method where it will sample reward and transition function by interacting with the world. It contains two types of policy: On-policy (e.g. SARSA) and Off-policy (e.g. Q-Learning). They are somewhat similar concepts for algorithm except that in SARSA, we take actual action, and in Q Learning, we take the action with highest reward.

Before we jump into explanation, we would like to throw out some vocabularies. Optimal Policy is a policy where you are always choosing the action that maximizes the "return" of the current state. Update policy is how your agent learns the optimal policy. Lastly, behavior policy is how it behaves.

State–action–reward–state–action (SARSA) is an algorithm for learning a Markov decision process policy. The agent learns optimal policy and behaves using the same policy. Because the update policy is the same as the behavior policy, SARSA is an on-policy. An on-policy learner learns the value of the policy being carried out by the agent including the exploration steps.

A real-life example of SARSA is training the machine to play chess. Chess game already has a set of rules to follow, so it follows one policy. The main policy is making the move on each turn that would cancel out the opponent's piece and the move must follow the rules for the pattern of that piece. The training will reward or punish that actions based on which agent makes a move. Depend on the role of the pieces, it will have a different reward and punishment value. For example, the queen's move might cost 20 points while the knights cost 10 points.

Q-Learning is another model-free reinforcement learning algorithm to learn a policy by telling an agent what action to take under what circumstances. The agent learns the optimal policy by using absolute greedy policy and behaves using other policies. Because the update policy is different from the behavior policy, the Q-Learning is an off-policy. An off-policy learner learns the value of the optimal policy independently of the agent's actions.

A real-life example of Q-learning is the automation of traffic network. This research is conducted by IEEE authors Yit-Kwong Chin, Wei-Yeang Kow, Wei-Leong Khong, Min-Keng Tan, Kenneth Teo who presented this article at 2012 Sixth UKSim/AMSS European Symposium on Computer Modeling and Simulation. The researchers try to design a traffic light controller to solve the congestion problem. In this simulated environment, researchers put five agents in the five-intersection traffic network with a reinforcement learning agent at the central intersection to control traffic signaling. There are eight available choices to the agent, and the reward function was defined as reduction in delay compared with previous time step. The researchers use Deep Q-Networks to learn the Q value of the state and action pairs.

Model-based learning

Model-based Learning is also known as indirect method which learns from the existence model instead of interacting with the world. This technique is known for causing exhaustion in the system's memory as it must continuously train based on the given model which can be a large and complex model.

Problem with Reinforcement Learning

In a scenario, where an environment is such that the agent does not receive any feedback based on its actions then it becomes really difficult for the agent to figure out whether the series of actions it has been performing will result in a reward or penalty. Suppose the algorithm has been playing Pong against a human for some time and it's been bouncing the ball back and forth quite skillfully. But then it slips towards the end of the episode and loses a point. The reward for the whole sequence will be negative (-1) so the model will assume that every action taken was wrong, which isn't so. This generic scenario can be inferred for any environment where the feedback received by the agent from the environment is delayed. This situation is termed as the Credit Assignment problem, and it comes into picture because our agent is not receiving immediate feedback after every action. It is understandable that getting instant feedback is not possible in all the situations but the more delayed this feedback is, the more seriousness will be required to address this problem.

In order to overcome this issue, the experts of Reinforcement Learning design manual reward functions by means of which they get to guide the policies of agents towards getting a reward. Getting an award at the right time goes a long way in improving a policy driving the training of an agent. These manual reward functions typically give out a chain of mini awards along the route of getting the big awards which significantly drive the policies, thereby, providing much needed suggestions to the agent. The creation of these manual reward functions takes a lot of understanding of the system and craftmanship keeping the bigger picture in mind. This process is termed as the Reward Shaping.

Summary

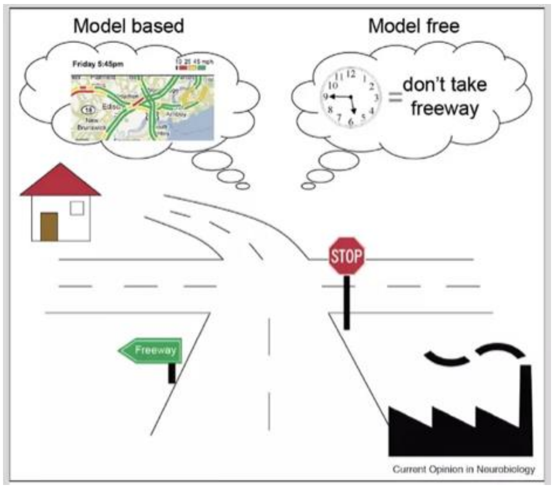


Figure 4: Compare and Contrast between Model-Free Learning and Model-based Learning through autopilot vehicle concept. (Dayan 2008)

Figure 4 is an example of Model-free learning and Model-based learning. This example shows the best route to get home from work. With model-based learning, it will use a model such as Google Maps which has GPS feature, real-time traffic, and more to determine the best route to get home if you need to go home at a certain day of the week and the certain time of the day. In contrast, model-free does not have much information. It will only be giving the time and a condition which is not taking the highway. It will then continuously learn which road will be less busy at that given time and which road will give you the fastest route to home. This implementation is crucial because the world is currently gearing toward automatic vehicles.

Architecture

Reinforcement Learning Side

The figure below (Figure 5) describe the general reinforcement learning model in the real world, and it is also the model that we plan to implement in our project.

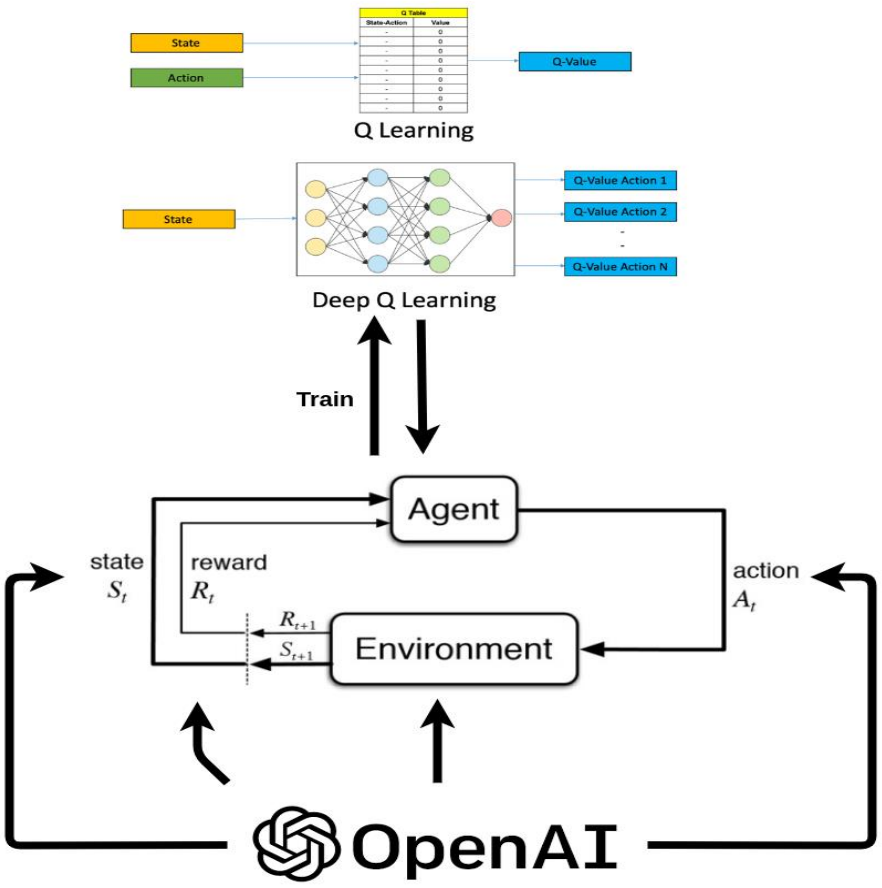


Figure 5: RL architecture (Built by Quan with a mixture of diagrams from Knudggets and Choudhary)

In the architecture, the main operation is the correspondence between Agent and Environment. Agent will first take a random action to see how environment react to the action by providing proper state of agent and its reward. After the first action is the loop of state and reward and action until the loop condition met. In Reinforcement Learning, we won't see any dataflow at all. It is because of the dataset will be generated by continuous interaction between Agent and Environment. Hence, Environment is the most important factor of Reinforcement Learning. Building an environment is not easy, so in real world, we usually use the open source library to help us reduce the environment construction time. In this project, OpenAI is the library that we are choosing for our Reinforcement Learning with Game. OpenAI will not only provide variables related to environment but also preset of state, reward and action to help us save time on set up. On the other hand, with agent operation, we must develop the models to decide the best actions by ourselves. There are two popular models that we may use for our project. One is the Q Learning table which provides the corresponding reward values. In this model, State and Action of Agent will drive the value of reward, so agent will try to maximize the reward (Q value). In contrast to Q Learning table, we can use Deep Q Learning (DQN) model which uses only State as the input and output of reward of next action. In this model, there will be one Q value associated with each individual action, so Agent will choose the action to take rather than find a maximum Q value.

Amazon Web Service Side

In higher level of reinforcement learning side, we need to look at what its infrastructure looks like in the cloud environment. This architecture is very important to provision the security of this project and the future cost of this project in the cloud.

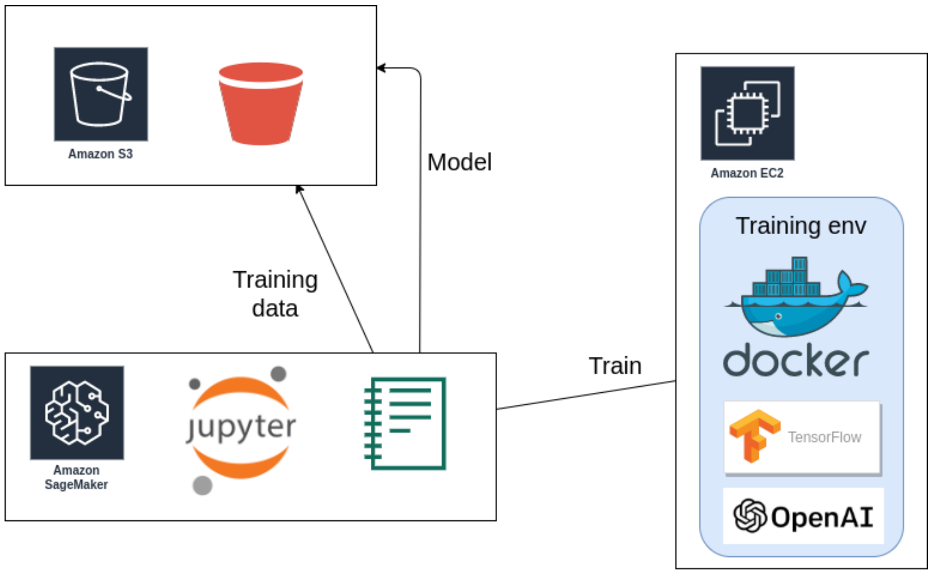


Figure 6: AWS architecture

In Figure 6, we will interact mostly with Amazon SageMaker, a Machine Learning interface on AWS. Amazon SageMaker plays a role of administrator for Jupyter Notebooks that we created to work with cloud infrastructure. In the initiation of SageMaker, we must choose the S3 service to store our training data. Since we need a place to save our model, we will choose the same S3 service to store our models generated by Reinforcement Learning. After finishing set up with Amazon SageMaker on the user's side, an Amazon EC2 instance will be spin up by the SageMaker RL process to run the training process defined in the Amazon SageMaker notebook. The new EC2 will use docker container(s) to tun the actual training scripts and condense the results. The container is defined by the RL estimator that initiates the training.

Data

About the game

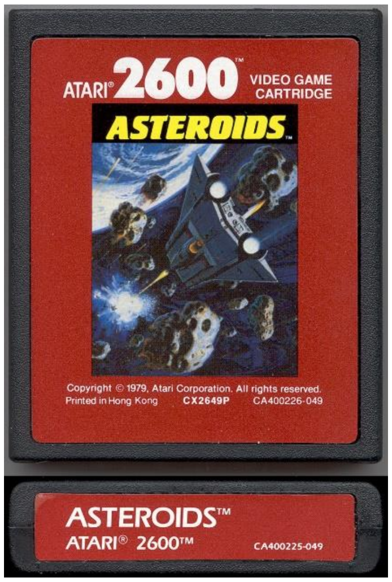

ATARI 2600 VIDEO GAME CARTRIDGE
ASTEROIDS
Copyright © 1979, Atari Corporation. All rights reserved.
Printed in Hong Kong CX2649P CA400226-049
ASTEROIDS™
ATARI® 2600™
CA400225-049

Figure 7: Atari 2600 Asteroids cartridge

The engine we are going to use, OpenAI with retro gym integration is capable of running retro games from consoles like Atari2600, Sega Genesis, and Nintendo Game Boy. We chose to go with the classics, the Atari2600. We have decided to use the famous game called Asteroids from 1979.

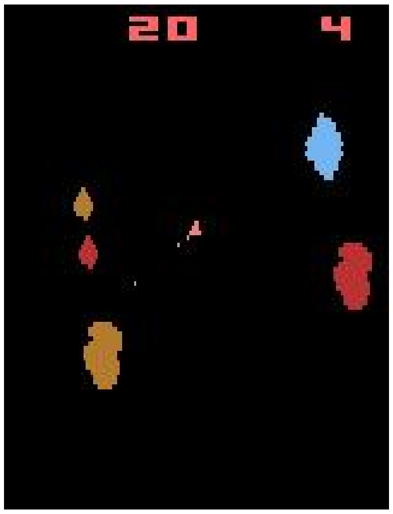


Figure 8: Asteroids gameplay

The game's logic:

- The player controls a single spaceship in an asteroid field which is periodically traversed by flying saucers.
- The object of the game is to shoot and destroy the asteroids and saucers, while not colliding with either, or being hit by the saucers' counter-fire.
- The game becomes harder as the number of asteroids increases. (Wikipedia "Asteroids (video game)"

Reinforcement learning elements

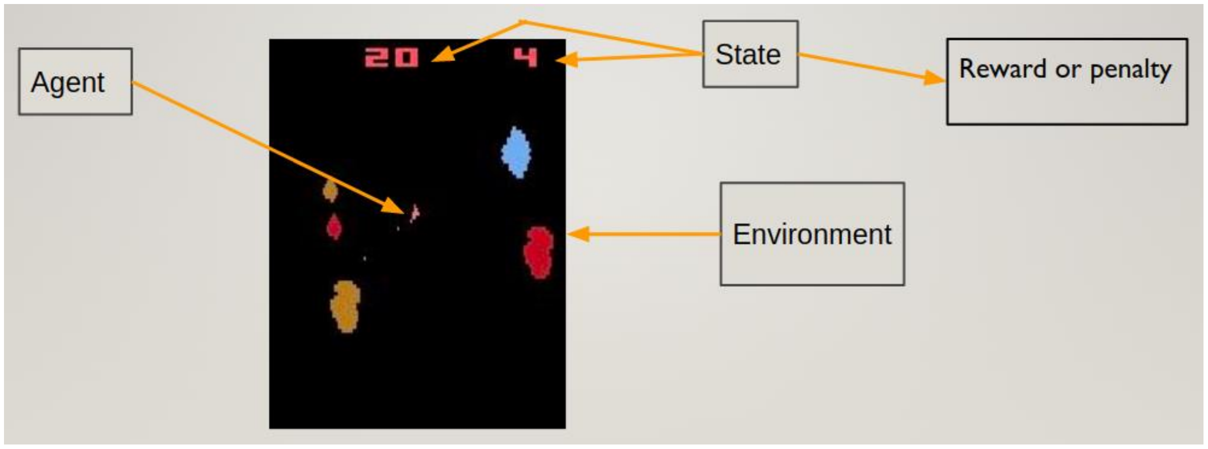


Figure 9: RL elements

Agent

The agent is the object we can control, in the case of the Asteroids game it is the spaceship. The spaceship has two actions:

- Shoot
- Move: accelerate and turn left or right

These actions can be performed at the same time.

Environment – observation

The environment in the whole screen that is show to the user. Each observation in case of the Asteroids game can be either only the agent and the bullets or the surrounding environment with the asteroids.

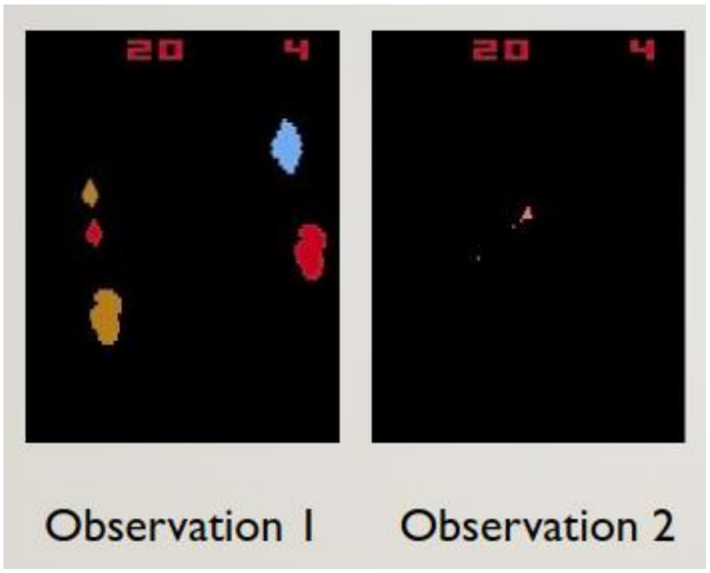


Figure 10: Individual observations

In the case of this separated observation, we must combine two observations to make one useable observation data/image. Since there are only very small difference between observations, we have decided to combine four observations into one.

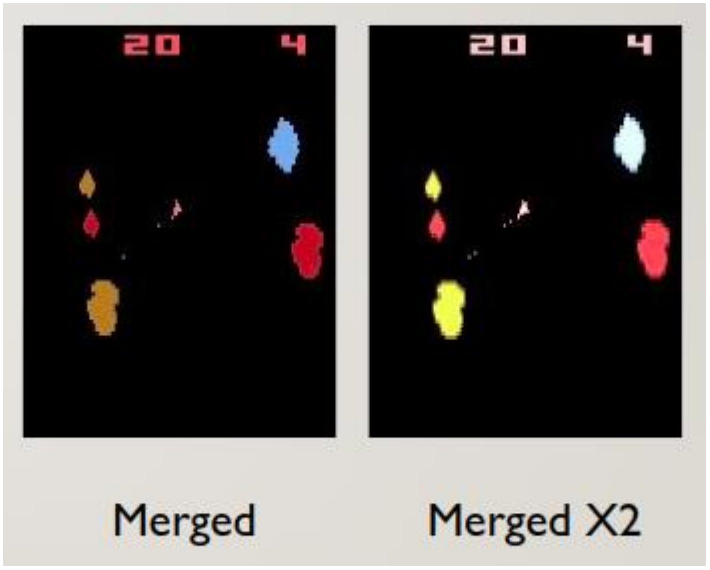


Figure 11: Merged observation pictures

State - done

State is a Boolean object sent by the engine, it is true if the game has ended and false if the game is still going. In case of the Asteroids game, the done is set to false if the agent's life reaches zero.

Information

This return value after an action has completed on the agents contains all our variable known from the game, the number of life remaining, and the score.

Reward

The reward is calculated based on the scenario.json's contents. In case of the Asteroids it is calculated as follows:

- If the number of life remaining is increasing (it happens every 10 000 points) we get 6250 points
- For every score increase we get 10 points. Since the game records point in its memory without the trailing zero the rewards will be the same as displayed in the game.

Actions

In the OpenAI gym environment there can be two different types of action spaces. The first is a discrete action space, which means that we can perform one action at a time; the other one is the multi-binary, which means we can define an action as an array of trues and falses for the buttons. In case of multi-binary model every action is a combination, but in case of discrete we must map action combinations to unique numbers – like 6 to mean UP and SHOOT.

Actions are performed in the environment based on the policy we are currently in.

Data sourcing

Data will be sourced from the experiments, so there will be no external source being used in this project. There are three types of data will be stored: Model, Movie, and Training information. Final model will be stored after the end of training session to S3. Like Model, information in the training period is also stored in S3 to evaluate the learning progress of model. Finally, the movie which stored all moving images of agent and environment are also generate, and this file will be presented in final presentation.

Data cleaning

In case of Reinforcement Learning the data, we collect is clean by nature, but the initial observation can be further processed.

Our observations are arrays of RGB data with a size of 210 pixels long, 160 pixels wide. For the DQN we do not want to overwhelm with redundant data, e.g. black background. For this we are scaling down the observations to half the size, 105 x 80 pixels.

To further reduce the strain of the DQN agent we can convert the picture to grayscale since the colors of the meteorites do not matter for the system, the agent just needs to dodge them or shoot them.

Data storing

Either Q table or Deep Q Learning model will be stored in S3 for purpose of saving the training information for agent. Whenever agent acts or environment react and reward, that information will be store through python and agent will pick them up whenever they receive new state. The circle will be stacked until the end of the iteration in python or specific condition met. To train this complicated RL agent, we will store about one-week data of the training process. If the agent is not smart enough, we will do two weeks of data.

There are a few reasons why we decided to use S3 for this project:

- S3 is cheap with price of $0.023 per GB. We don't have to pay much during this project if we use this solution
- S3 is scalable with a heavy data project like Reinforcement Learning. We can increase our cap for storage as much as we want
- S3 works well with Amazon SageMaker and other services like EC2. Since SageMaker is using S3 anyway to store its variables, we can utilize it also for storing our information
- S3 can handle various types of data and files, so we don't need to worry about Variety of Data

Operations

Checkpoints

Reinforcement Learning takes time to train so it would not be optimal if once a training stops, we must start over again to train the model more. For this purpose, TF-Agents has a checkpoint feature built in (have to be configured separately) that takes the training agent and creates a snapshot of all its components, including the current policy, memory buffer, training step counter, etc. This results in a reasonably large snapshot, in case of the Asteroids training it requires storage of around 3 to 5 gigabytes.

These snapshots can be configured later in any session and used as if the training never stopped. This enables TF-Agents based training to be continued whenever and wherever.

Policy

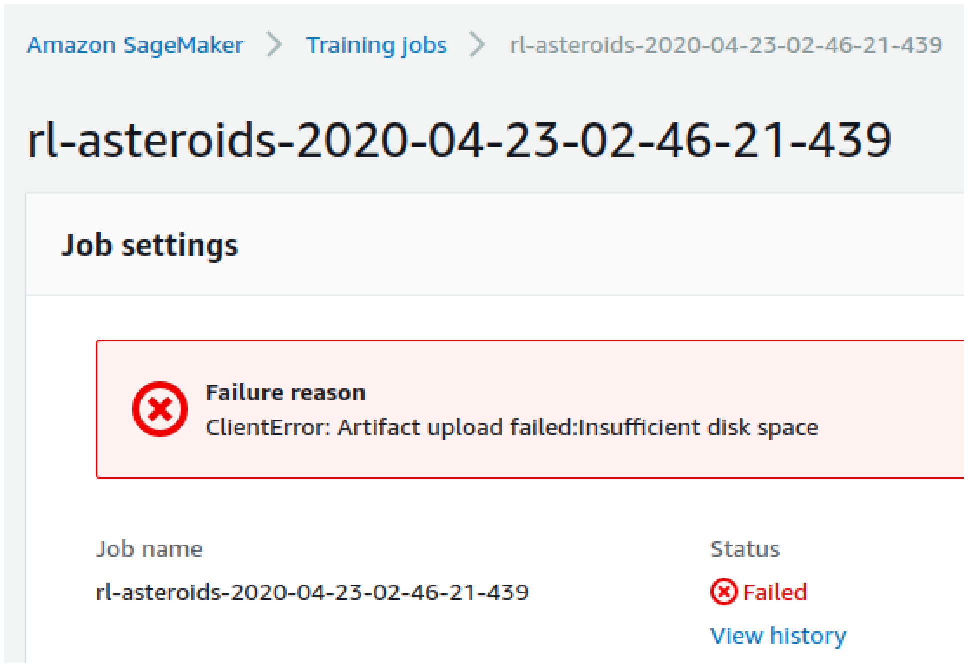
The result of the training process is a policy. The policy is an object that can be used for evaluation, but it is not suitable for further training. In our case there are trained DQN, C51, and REINFORCE policies that can be called by any python application that uses TF-Agents. In contrast to the checkpoints, the policies are lightweight files of around 50 Mb. For a policy to be used we simply need to supply a time step, that is the current environment status, and it will return a suitable action to be performed in that environment state.

Challenges

AWS

As our initial plan was to use the AWS SageMaker provided RL training functions, we encountered the first issues there. It turned out that SageMaker RL is compatible only with Coach and Ray frameworks. This means that in our case we are restricted to run our training using these frameworks.

The Coach framework is used for training basic OpenAI gym environments, like the famous CartPole example where you need to keep a pole up with moving a cart left or right. This gym provides the necessary attributes of the cart and the pole, this means we are dealing with easily interpreted values. Unfortunately, in case of an Atari gym or retro gym the observation is not that easy after a reset or an action has been performed, we are returned with 3 arrays of values representing 3 pictures (RGB), 160 px wide 210 px high. Even after modifying the Coach training script after about one hour, we encounter our issue, namely an out of storage failure.


Amazon SageMaker > Training jobs > rl-asteroids-2020-04-23-02-46-21-439
rl-asteroids-2020-04-23-02-46-21-439

Job settings

Failure reason
ClientError: Artifact upload failed:Insufficient disk space

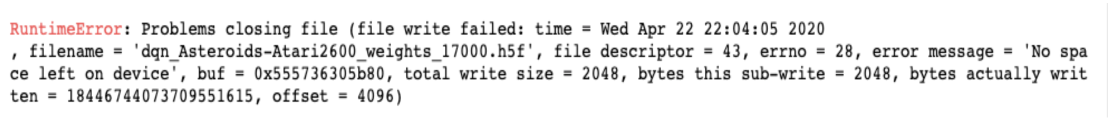
Job name
rl-asteroids-2020-04-23-02-46-21-439

Status
Failed

Figure 12: Resource insufficient error in AWS

As we have noticed, the Couch trainer is configured to make checkpoints frequently, but the frequency cannot be changed or disabled. This approach might be good with a gym like the CartPole, but with a picture-based one we run out of the provided 30 GB soon.

The same issue was encountered when we were training Keras-RL based model under SageMaker, that is trying to store its intermediate and final output onto the SageMaker notebook instance first before we could transfer it to an S3 bucket.



```
RuntimeError: Problems closing file (file write failed: time = Wed Apr 22 22:04:05 2020
, filename = 'dqn_Asteroids-Atari2600_weights_17000.h5f', file descriptor = 43, errno = 28, error message = 'No spa
ce left on device', buf = 0x555736305b80, total write size = 2048, bytes this sub-write = 2048, bytes actually writ
ten = 18446744073709551615, offset = 4096)
```

Figure 13: No Space Error

This above issue could be solved only by limiting the training episodes to about 20, but it resulted in a model that we could not rely on.

Cost of training

The other issue that arises from using cloud infrastructure is the pricing. Although, cloud services are considered cheaper than running your own infrastructure, maintaining it, etc., for testing purposes training RL models is expensive. The SageMaker RL estimators can be configured to train only on more powerful instances like the ml.m4.4xlarge (16 vCPU, 64 GB RAM) which restricts the users to use cheaper instances.

Time to train

Considering the fact that RL models start from a child's level of knowledge where it is just simply exploring the environment, the actions, and the consequences (these are the rewards and penalties in RL) we can see that getting to an "adult" level of control takes time. We have trained one model that was trained with XX million steps, that is YY episodes (game reached game over YY times), which is still not considered enough if we consider that the DeepMind research mentions 50 million training steps for mastering most Atari games.

Hardware utilization

In the case of TF-Agents, not every agent is the same when it comes to the CPU usage. Some agents, like the DQN or C51 agents are optimized to run on multiple CPUs in multiple threads with no issues. On the other hand, the REINFORCE agent can only run in one thread, thus on one CPU. This means that even if TensorFlow would be able to run on CPUs and GPUs as well, the underlying code might not be always optimized for using the available resources.

Hyperparameter combinations matter

Another challenge in most ML related projects, this way in RL as well, is to find the best hyperparameter combinations. For example, in Keras-RL model, there are three different types of parameters: neural network parameters, data generation parameters, and Agent parameters.

Parameter Type	Parameter Name	Description
Neural Network	Layers	Hidden Layer in Neural Network
	Input	Input of network
	Output	Output of network
	Activation Function	Function to activate certain node
	Node	Number of Node each layer
Data Generation	Record file (.bk2)	Save all operations of an episode and data to reconstruct video
	Memory	Amount of memory allocated to the reinforcement learning job
	Model weight (.h5f)	File with check point to save usage of each variables
	Log file (.json)	Save high level information such as reward, step episode
Agent	Policy	Strict rule of game that agent must follow to achieve our purpose
	gamma	how much importance we give for future rewards?
	nb_steps_warmup	Number of Random steps necessary to build data before applying the policy

	Loss function	Method of evaluating how well specific algorithm models the given data. The smaller number of loss function in each iteration, the better of model.

Table 1: Parameter Lists

The above list is not a comprehensive list of all parameters; however, it is big enough to see how difficult it is for us to make a decision on what we should choose for each parameter (except the output file parameters in the data generation section). Since the agent behaves differently for different combinations, we must test different combinations until we find the most optimal setting. Since training takes time it took us approximately one week to find combinations that work the best in our training environment.

Not to confuse exploration with exploitation

This section explains is one of the most argued challenges we have faced. To be more specific, we had conversations on whether a trained model with 17 episodes (the game trained on reaching 17 game over states) can be considered as a working model or not. On the one hand, we have this model with 17 episodes that is around 40 thousand steps, that performs “well” - just like the random agent. On the other hand, we have the model that was trained on 2 million steps and performs poorly, however, the evaluation every 5 episodes show that it performed “well” when it was around 17 episodes as well, but results decreased by time. The solution for this problem is to frequently evaluate the model and train longer to see the full picture. One other note to keep in mind is that we need to know how the agent and the underlying model (in our case DQN) works, for example, replay buffer – memory – might still contain mostly states from the exploration state, so sampling of the memory takes longer to contain exploitation results.

Knowledge Gap

After being introduced to the reinforcement learning on our first day of class, we were highly interested to challenge ourselves with this project. We took one week to research the topic of reinforcement learning as well as viewing multiple previous projects. We are fascinated at how useful it is for the technologies in this AI era we live in. We attempt to implement reinforcement learning with a retro game because it is easy to generate training data, and it fits with our time constraint. However, we believe that one-week research is not enough to help us with determining the best framework, choice of using clouds, and the methodologies. Hence, we each ran our own models in order to find multiple possible ways to complete our project. We believe there would be much more to explore although reinforcement learning is quite new in this field.

Learnings

Reinforcement Learning is data-hungry

From our research of RL and our own experience while doing this project we concluded that RL's appetite for data is endless. The more data is fed to the system after hours and hours of training, the more it is probable to have some positive outcome while testing the results of that model. Even with a simple game like Asteroids, it took us hundreds of hours of training cumulatively to come up with some acceptable results which can be demonstrated. If we think about the complex scenarios where RL can be implemented like training intelligent MAV (Micro Air Vehicles) or Air combat training simulations where the complexity of the environment and agent interaction exponentially increases, we can only imagine how many hours of training will be required to get considerable results.

Requires hardware with high configuration

While training our RL models on high configuration systems from a commercial perspective like 8 core systems, we were initially under the impression that the current hardware resources will be sufficient to achieve our end goal of the project. As we progressed further, we have started facing the reality that commercially available computer systems are not sufficient to train the models efficiently because of the millions of computations going on while training an RL model and high configurations of hardware is required to withstand such processing elements. We could listen to the buzzing sound of the CPU fan struggling to keep the system cool after only a few hours into model training.

Using cloud for RL is not cheap

The first cloud solution that came to our mind was AWS (Amazon Web Services) which could be integrated with our system to continue working towards our final goal of attaining super-human performance through an RL model in the game of Asteroids. While using the student accounts, we thought that the storage would be sufficient, but we were so wrong. We got charged for $25 while running a training session for over-utilizing the space provided to us on the cloud. Thinking about the industry level RL projects, a company needs to considerable investment in the cloud to build a sustainable model utilizing RL trained models.

Hard to predict model training time

It is almost impossible to come up with a timeline to get a model trained which is producing acceptable results. It totally depends on the interaction between the agent and the environment, the policy being used, the designed model, the reward and penalty functions, and multiple other factors as well. On top of all, as we know that perfection cannot be achieved so one needs to set up its own acceptable standards for a model with respect to its results. The more you train, the better the model gets but a line needs to be drawn where a model starts giving acceptable results to be used in the production environment.

RL is not suitable for mission-critical applications

Reinforcement learning is not suitable for mission-critical applications. When I am referring to mission-critical applications, I am referring to those applications where even the slightest of mistake is not acceptable or has fatal consequences. It is so because RL models during execution choose between exploration and exploitation, the selection choice of which needs to be fine-tuned. Consider an instance where on a busy highway, a polybag came flying in front of a self-driven car which is using Reinforcement Learning. Interpreting that polybag like any other object on the road if the car decides to stop in the middle of the road to avert the collision with the polybag and the adjacent cars, then there is a high possibility that the car behind it will hit it from the back due to the application of sudden brakes. In case of a human driver, such an accident is easily avoidable. Some more research and investment need to be done in RL before it becomes suitable for mission-critical applications.

Results

Quan's Model

Quan's model is using Keras Reinforcement Learning with DQN agent. It includes 420 nodes and one hidden layer with LinearAnnealedPolicy being enforced. Quan's model stops at 50000 steps with episode 20. In this model, the agent performs well at the beginning with high scores (about 1650 game points). However, the number step completed is double comparing to the next episodes. Even though, the next 15 episodes stop at average 760 game points, the number of steps to complete an episode is less than half of the first episode. At episode 17, the agent is can complete episode with 1610 points and half of the total step. In this episode, the agent can chase asteroid and identify the location asteroids. The only problem with the agent now is that it can't dodge asteroids if it flies in discretionary distribution. Overall, my model can perform the best in terms of score and the number of training steps.

Neural Network Models

Elements	**Model**
Number of Layers	2 (2 Dense)
Hidden Layers	2 (420 Nodes)
Number of params	56,342,314
Activation Function	Linear

Table 2: Neural Network Configuration

DQN Models

Element	Model	
Number of steps	50,000	
Sequential Memory:	100,000	
Decay Policy	LinearAnnealedPolicy	
Policy	EpsGreedyQPolicy	
gamma	0.99	
train_interval	100	
Optimizer	Adam optimizer	
Learning Rate	0.00025	
Metric	Mean Absolute Error	
nb_steps_warmup	1,000	
target_model_update	10,000	

Table 3: DQN Configuration

There is one thing that Quan is sharing with us that he was trying to play against with his model to see who is playing the game better. In the first few trials, he was unable to reach 700 Points. The only way he could play better than his model is trying to shoot more randomly. The more he shot, the less likely chance for him to die. For that reason, he concluded that his agent was doing a much better job than him now.

Hao’s Models

Before getting into the model’s summary, we did some research about the different types of reinforcement methods out there from multiple scholarly sources. Figure 20 has shown that DQN has the highest score of around 1458.7 which attempted by Google DeepMind in December 2015. Following our architecture model, we would like to see if there is a way to improve this score, so we decided to use this score as our baseline.

For this model, we attempt to improve Quan’s model by running with a different type of neural network models and increasing the size of training steps as well as expecting more episodes. With the insufficient amount of hardware specs from Hao's MacBook, we attempt to run on the Google Colab which works similarly with Jupyter Notebook. It offers free GPU Tesla K80, 2.3 GHz Intel Xenon CPU, 13 GB of RAM, and 107 GB of storage. It was our first-time using Google Colab, but seeing how fast it runs Quan’s model, we decided to increase the number of steps from 50,000 to 1.75 million. However, it ran for 12 hours and expected to run for 15 hours, but unfortunately, it got terminated because Google Colab is only offering free GPU up to 12 hours continuously. We also learned a lesson that every time the runtime is disconnected, all the files outputting from the code will be recycled and refreshed. Therefore, we decided to change the number of steps back to 50,000 steps due to time constraints for the running.

Comparison Charts

As Quan has mentioned, he was only able to run a basic model for his neural network model due to limited of the CPU usage. Here is the chart that shows how the new neural network would look like and how the score has been dramatically changed after running on Google Colab. Model 1 is the one that mimicking Quan's model and run with different changes in parameters in DQN. Model 2 is adding more layers and tuning with different parameters. The charts below will show the difference between the two models.

Neural Network Models

	Model 1	Model 2
Number of layers	2	5
Number of hidden layer(s)	1	2
Number of params	42,342,314	1,691,310
Activation function(s)	Linear	Linear and Softmax

Table 4: Neural Network Configuration

DQN Models

	Model 1	Model 2
Number of steps	50,000	
Sequential Memory:	100,000	
Decay Policy	Linear Annealed Policy	
Policy	Eps Greedy Q Policy	
gamma	0.99	
train_interval	4	
Optimizer	Adam optimizer	
Learning Rate	0.00025	
Metric	Mean Absolute Error	
nb_steps_warmup	1,000	5000
target_model_update	1,000	10,000

Table 5: DQN Configuration

Result Chart

	Model 1's Linear Activation Function	Model 2's Linear Activation Function	Model 2's SoftMax Activation Function
Highest Rewards	190	1460	34
# of Episodes	53	37	43
MAE	0.401	0.288	0.067
Run Time	9184 seconds (~2.5 hours)	1285 seconds (~21 minutes)	1508 seconds (~25 minutes)

Table 6: DQN Result Chart

Summary

Overall, Model 2 with linear activation function gives the best rewards, and we are satisfied with this number because it is somewhat a number that we are expected to get with DQN. As we mentioned above, from the research section, this is around the best score that scholarly researchers get from using DQN.

Richard's Models

Richard's models were built using the TF-Agents framework, because it is using the new version of TensorFlow, while KerasRL is using the quite old 1.14 version of TensorFlow. This means that this TF-Agents framework is easier to run and debug and it uses the Python control flow instead of a graph control flow, this simplifies the specification of dynamic models since "TensorFlow's eager execution is an imperative programming environment that evaluates operations immediately, without building graphs." (TensorFlow, Eager execution)

In this section we are not using the default action space (except with the last experiment with the OpenAI gym), we only use 9 buttons which is the result of a custom discretized action space with the following allowed actions and combinations:

- Up
- Left
- Right
- Shoot
- Left + Up
- Right + Up
- Shoot + Up
- Left + Shoot
- Right + Shoot

The 5 buttons variation is used to further reduce the complexity for the agent:

- Left
- Right
- Shoot
- Left + Shoot
- Right + Shoot

REINFORCE models

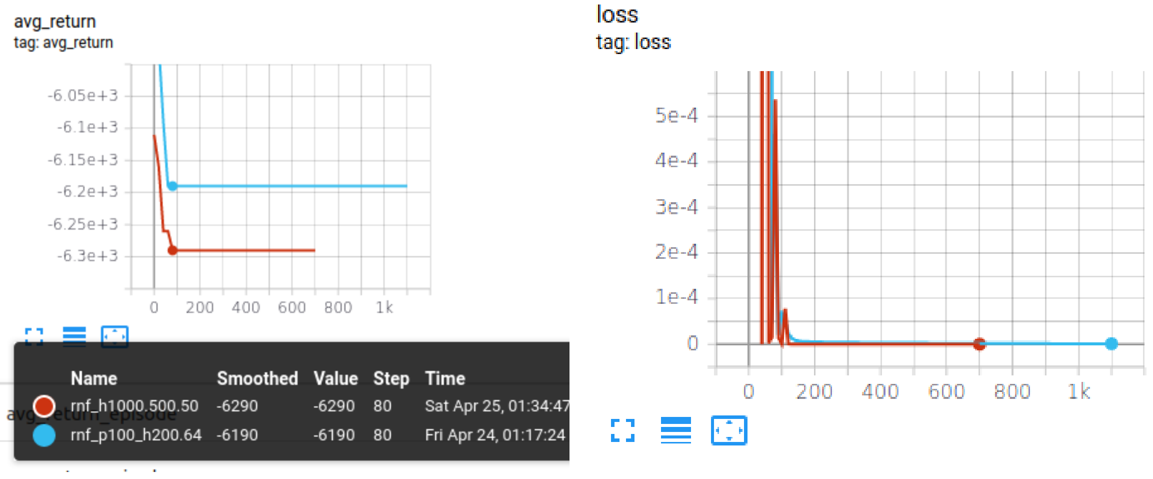
REINFORCE (Monte-Carlo policy gradient) relies on an estimated return by Monte-Carlo methods using episode samples to update the policy. It is the first step-up from the Q learning.

We had two experiments with the REINFORCE agent with the following configurations:

Agent	REINFORCE	
Game	Retro emulation	
Action space	9 buttons	
Hidden layers	2	3
Nodes	200, 64	1000, 500, 50
Penalty on life loss	100 (resulting in –1600 per life)	
Log color	Blue	Red

Table 7: REINFORCE configuration

After these two training sessions we got the following graphs:



	Name	Smoothed	Value	Step	Time
🔴	rnf_h1000.500.50	-6290	-6290	80	Sat Apr 25, 01:34:47
🔵	rnf_p100_h200.64	-6190	-6190	80	Fri Apr 24, 01:17:24

Figure 14: REINFORCE Result

In case of the REINFORCE agent, it had to be run with 2 episodes per training iteration due to its different approach to replay buffer usage (training from the buffer after 2 episodes then empty buffer), so when interpreting we have to keep in mind that every step is 2 episodes. With this in mind, we can see that after 160 episodes both configurations have reached the lowest state and stayed stable there. The same can be seen in the loss graph, that the model stabilized after the previously mentioned 160 episodes.

From these we can clearly see that the REINFORCE agent is not a good solution for our main project and not to mention that is slower than other agents due to its one CPU implementation.

C51 models

Categorical “C51” is a feasible algorithm proposed in a paper to perform iterative approximation of the value distribution Z using Distributional Bellman equation. The number 51 represents the use of 51 discrete values to parameterize the value distribution Z. (Felix Yu)

We had two experiments with the C51 agent with the following configurations:

Agent	**C51**	
Game	Retro emulation	
Action space	9 buttons	5 buttons
Hidden layers	2	
Nodes	200, 64	
Penalty on life loss	None	100 (resulting in –1600 per life)
Log color	Blue	Grey

Table 8: C51 Configuration

After these two training sessions we got the following graphs:

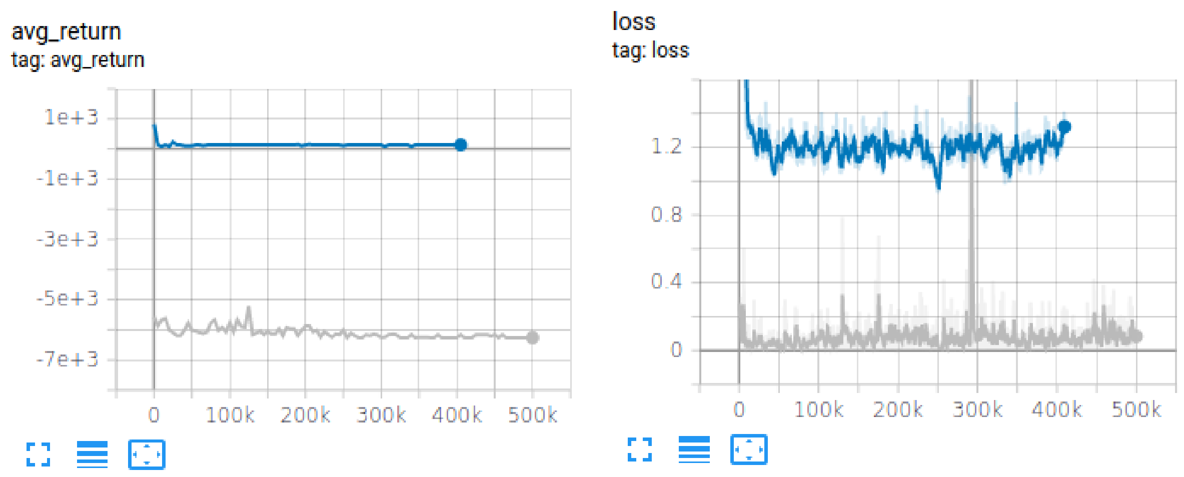


Figure 15: C51 Result

The results from both configurations using C51 agents resulted in similar behavior as the REINFORCE models, after a certain point the agents just try to be stable while not maximizing the score but minimizing it. They take only the least amount of actions, and not really shooting. Even with a random agent, we got higher scores from random shooing than with these agents. In this case, the loss values are low and fairly stable.

DQN models

In the DQN section, we have conducted several different experiments. Some with the OpenAI Retro environment and one with the OpenAI Atari gym called Asteroids-v0.

The first experiment had the following configuration:

Agent	**DQN**
Game	Retro emulation
Action space	9 buttons
Hidden layers	1
Nodes	100
Replay buffer	20 000
Train length	500 000 steps

Table 9: DQN 1 configuration

This configuration resulted in the following graphs:

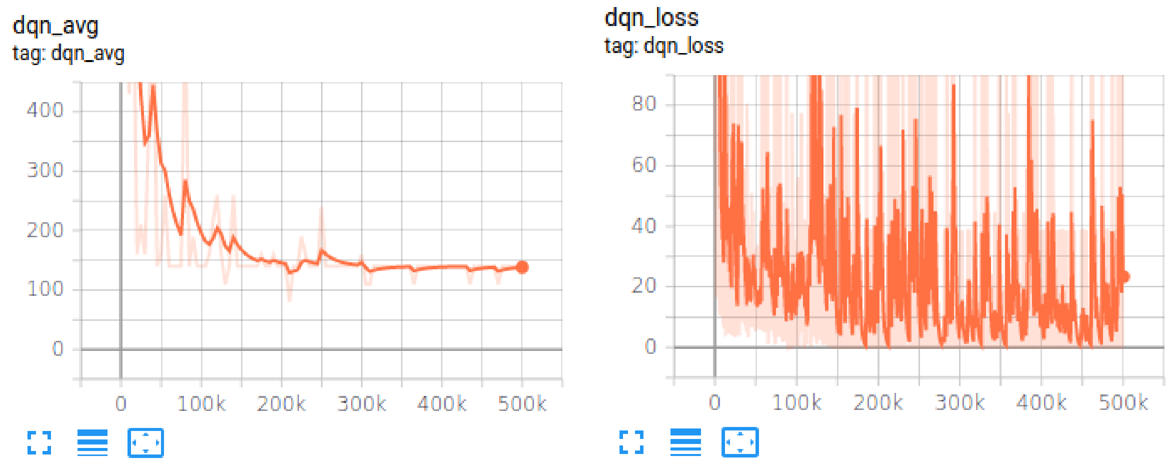


Figure 16: DQN 1 Result

From the average evaluation results, we can see that at around 100 thousand steps in the training session the agent has reached the point we have seen with the other agents, the game minimized the score to stay stable with the scores. This stabilization should have resulted in close to zero loss values, however, we can see that the loss is still highly variable but getting closer to zero.

The next trainings we have started used the following configurations:

Agent	**DQN**	
Game	Retro emulation	
Action space	9 buttons	10 buttons (9 plus “stay” action)
Hidden layers	2	1
Nodes	200, 64	100
Penalty on life loss	100 (resulting in –1600 per life)	
Replay buffer	10 000	20 000
Log color	Blue	Purple

Table 10: DQN 2 Agent configuration

With the experiment in purple the specialty of it is that we have included an action called “stay” which will result in the agent doing nothing in that step, this will make the time pass and If the agent is in motion then the agent will continue in the same direction. Whit this inclusion we wanted to test our hypothesis that the agent after some training will decide to do exactly nothing. Here are the findings:

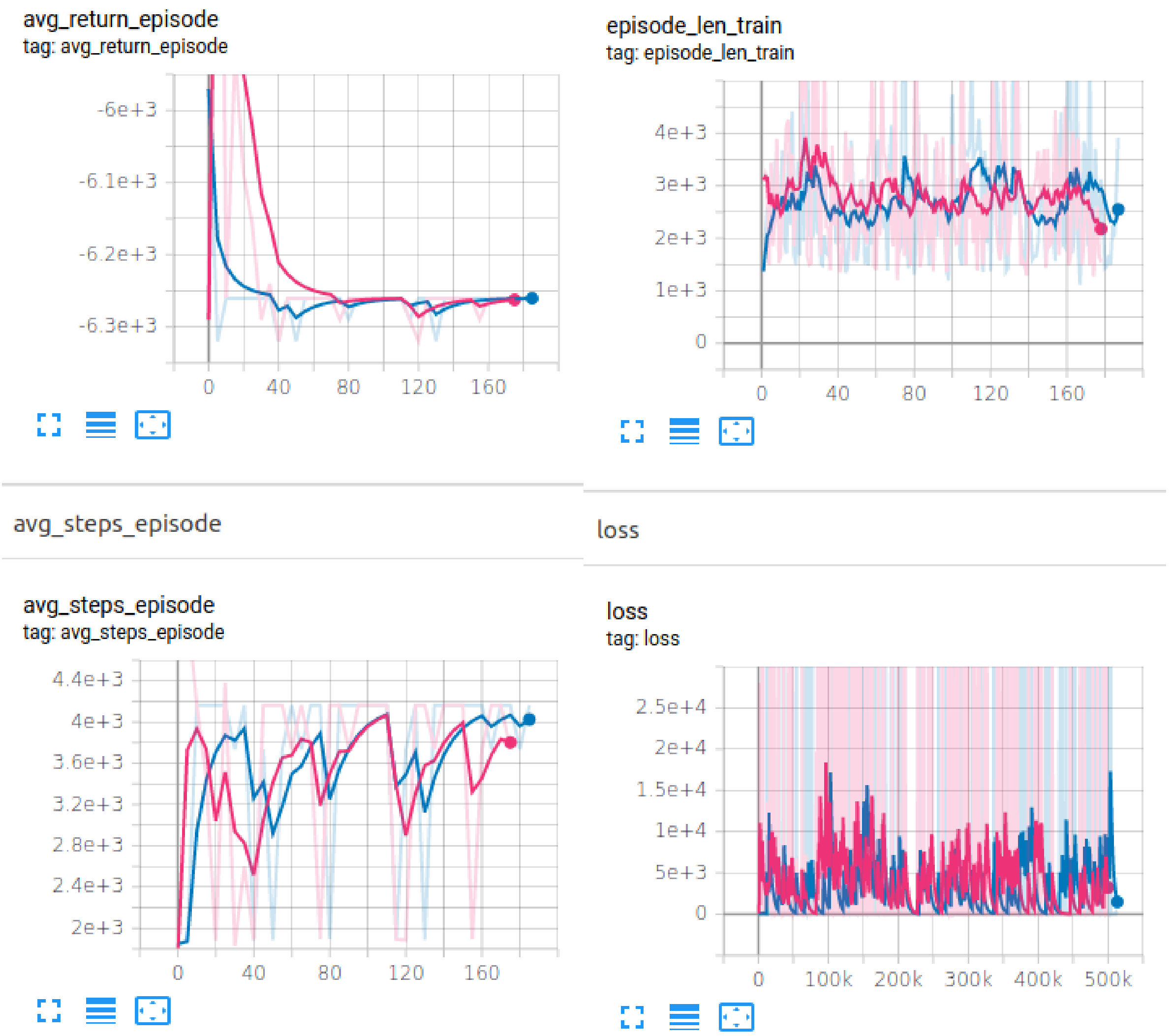


Figure 17: DQN 2 Result

For this experiment we have generated more logs, on the left we can see the logs from the evaluations, like before, and on the right, we can see the whole training processes episode length report and the loss. Since this section was run using the increment by episode method instead of the increment by step method. This meant that in the graphs, except for the last one, one step is one episode. From this we can see that we had around 170 episodes of training in both cases, both running for 500 thousand steps, so one episode lasted on average around 3000 steps (graph with tag episode_len_train).

After all the experiments we were not surprised to see that these agents have stabilized themselves too with the lowest scores they could reach. After generating the footage using the policies, we could verify the hypothesis that including a "stay" action resulted in less useful actions to be performed, and most of the time the agent chooses to do nothing.

The model that was trained the longest is a DQN model with the following configuration:

Agent	DQN
Game	OpenAI version Asteroids
Action space	Default 14 buttons
Hidden layers	1
Nodes	100
Replay buffer	20 000
Train length	2000 episodes

Table: 11: DQN 3 Configuration

This configuration resulted in a model that was trained for 2.5 days, with a final step count of around 2 million.

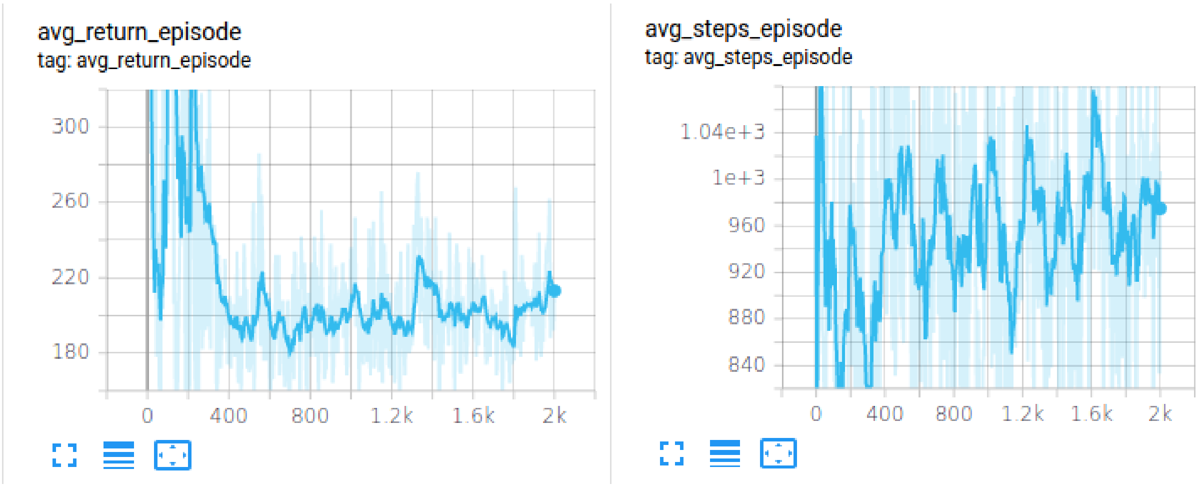


Figure 18: DQN 3 Result

As it can be seen from figure X, after around 400 episodes the agent reaches the state where it will not score as much, and after that point, the model stabilizes to score around 190 points with a small amount of variations. Overall, between the 400th and 2000th episode the model does not seem to improve on its internal algorithm to maximizes its score, more like it just wants to reach a stable state. However, the average steps for the 5 evaluation tests show that there is a huge variation in the number of steps it takes to reach a game over state.

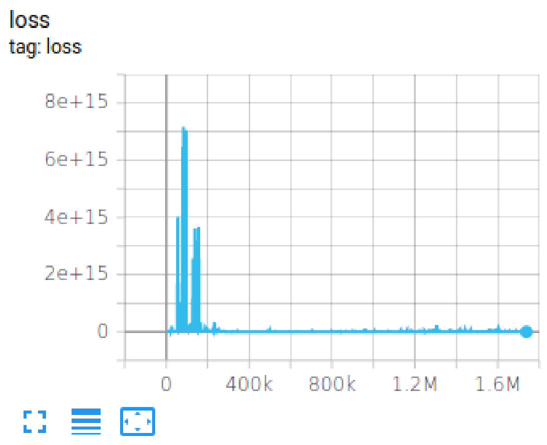


Figure 19: DQN 3 Loss Result

The graph about the loss we can see that after initial randomness the actions tend to result in a low loss value and the model tries to stabilize itself. It could be seen from the previous models that on average an episode takes somewhere between 3000 and 5000 steps to complete with the OpenAI Retro environment. From this, we can assume that the Atari gym environment provided by OpenAI has been optimized in some ways for their use case.

Future improvements

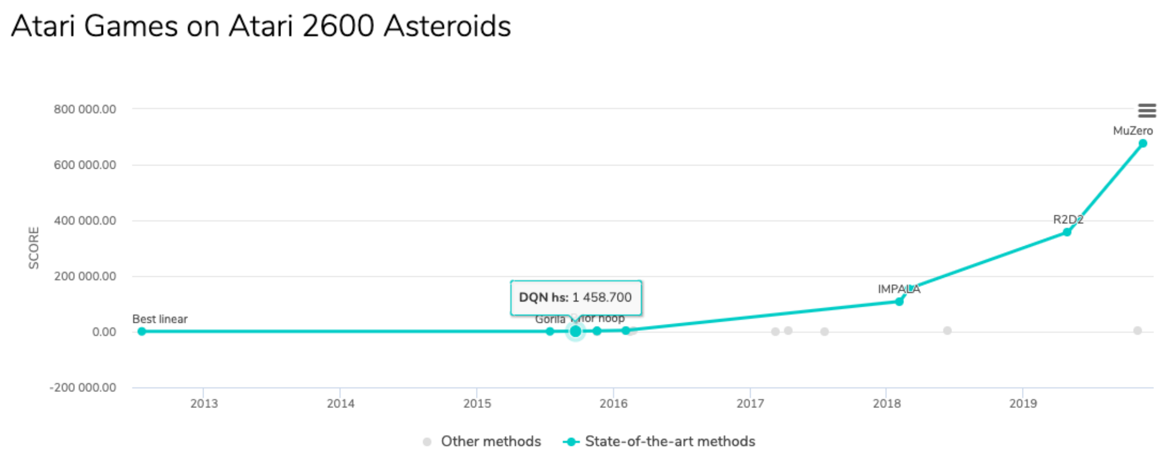


Figure 20: methods and scores for Asteroid (For more information, please visit https://paperswithcode.com/sota/atari-games-on-atari-2600-asteroids)

This research conducts by Google DeepMind and multiple scholarly paper sources. According to Figure 20, DQN is no longer the best reinforcement learning method for the Atari Asteroid game. In fact, the new method MuZero which is a model-based reinforcement learning algorithm and is known as Google's next generation of AlphaZero because it achieves the same strength as AlphaZero without being told the rules has become the number one method for Atari games. In November 2019, Google DeepMind and the University of London collaborated on training Atari games, Chess, Go, and Shogi with MuZero. The result for Asteroids reward is 678,558.64 which is 600 times better than DQN's score. According to the Guinness World Records, the highest score of Asteroids played by a human is "41,838,740, achieved by John McAllister (USA), who played for 58 hours in an attempt verified by Twin Galaxies on 5 April 2010". Even though there has not been any model that beats this score for Asteroids, Muzero has been one of the few methods out there that has beat human scores in other Atari games. For future implementation, we would like to use different methods out there to train the model and improve our reward scores.

References

34133-asteroids-atari-2600-media [Photograph found in Mobygames.com]. (n.d.). Retrieved April 16, 2020, from https://www.mobygames.com/images/covers/l/34133-asteroids-atari-2600-media.jpg

Amazon Web Service Logo [Photograph found in Wikimedia.org]. (n.d.). Retrieved April 16, 2020, from https://upload.wikimedia.org/wikipedia/commons/thumb/9/93/Amazon_Web_Services_Logo.svg/1200px-Amazon_Web_Services_Logo.svg.png

Arulkumaran, K., Deisenroth, M. P., Brundage, M., & Bharath, A. A. (2017). Deep Reinforcement Learning: A Brief Survey. *IEEE Signal Processing Magazine, 34*(6), 26-38. doi:10.1109/msp.2017.2743240

Asteroids (video game) - Wikipedia. (n.d.). Retrieved April 18, 2020, from https://en.wikipedia.org/wiki/Asteroids_%28video_game%29

Choudhary, Ankit. “Introduction to Deep Q-Learning for Reinforcement Learning (in Python).” *Analytics Vidhya*, 6 May 2019, www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/.

D802aff83bfcd5a62645c30aba35388a-atari-2600-alt [Photograph found in Hyperspin-fe.com]. (n.d.). Retrieved April 16, 2020, from https://hyperspin-fe.com/siteuploads/downloads/screenshots/monthly_10_2016/d802aff83bfcd5a62645c30aba35388a-atari-2600-alt.png

Dayan, Peter & Niv, Yael. (2008). Reinforcement learning: The Good, The Bad and The Ugly. Current opinion in neurobiology. 18. 185-96. 10.1016/j.conb.2008.08.003. https://www.princeton.edu/~yael/Publications/DayanNiv2008.pdf

Deep Q-Networks [Photograph found in Analyticsvidhya.com]. (2019, April 18). Retrieved April 14, 2020, from https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/

Figure 1 [Photograph found in Kdnuggets.com]. (2018). Retrieved April 13, 2020, from https://www.kdnuggets.com/2018/03/5-things-reinforcement-learning.html

“Highest Score on Asteroids (Atari, 1979).” *Guinness World Records*, www.guinnessworldrecords.com/world-records/highest-score-on-asteroids-(atari-1979)?fb_comment_id=617186045070955_940109289445294.

MacGlashan, J. (2018, April 29). What is the difference between model-based and model-free reinforcement learning? Retrieved April 14, 2020, from https://www.quora.com/What-is-the-difference-between-model-based-and-model-free-reinforcement-learning

Mao, L. (2019, March 14). On-Policy VS Off-Policy in Reinforcement Learning. Retrieved April 18, 2020, from https://leimao.github.io/blog/RL-On-Policy-VS-Off-Policy/

Openai-cover [Photograph found in Openai.com]. (n.d.). Retrieved April 15, 2020, from https://openai.com/content/images/2019/05/openai-cover.png

Qiang, W., & Zhongli, Z. (2011). Reinforcement learning model, algorithms and its application. *2011 International Conference on Mechatronic Science, Electric Engineering and Computer (MEC),* 1143-1146. doi:10.1109/mec.2011.6025669

Sagemaker (2020) "Using Reinforcement Learning with the SageMaker Python SDK". 1.56.1.post1 Documentation, sagemaker.readthedocs.io/en/stable/using_rl.html.

"Eager Execution." *TensorFlow Core*, TensorFlow, 2 Apr. 2020, www.tensorflow.org/guide/eager.

Yan, Ma & Liu, Kang & Guan, Zhibin & Xu, Xinkai & Qian, Xu & Bao, Hong. (2018). Background Augmentation Generative Adversarial Networks (BAGANs): Effective Data Generation Based on GAN-Augmented 3D Synthesizing. Symmetry. 10. 734. 10.3390/sym10120734.

Y. K. Chin, W. Y. Kow, W. L. Khong, M. K. Tan and K. T. K. Teo, "Q-Learning Traffic Signal Optimization within Multiple Intersections Traffic Network," 2012 Sixth UKSim/AMSS European Symposium on Computer Modeling and Simulation, Valetta, 2012, pp. 343-348.

Yu, Felix. "Distributional Bellman and the C51 Algorithm.", Github, 24 Oct. 2017, flyyufelix.github.io/2017/10/24/distributional-bellman.html.