MSE 486 Directed Study
Final Report

2 Player Texas Hold'em Poker Reinforcement Learning Python Bot

Ryan Fielding - 301284210
rafieldi@sfu.ca
778.886.8199

Supervisor: Dr. Ahmad Rad, Ph.D.
Associate Director, School of Mechatronic Systems Engineering,
Simon Fraser University

August 14^{th}, 2020

Abstract

This paper outlines the project progress for a directed study in machine learning, through implementation and development of an AI based bot (*RLPlayer*) to learn and understand different strategies to play poker. Particularly, the game of Texas Hold'em will be the basis for this project, potentially beginning with Leduc Hold'em since it is a much simpler version of the game.

To initiate the project, the first couple weeks will be dedicated to project research, pertaining to coding language, type of machine learning to implement, project goals, requirements and more. Implementation and development will be done in Python, through the AI of reinforcement learning.

To elaborate on the overall development, first, a basic poker environment will be initialized. There is no need to develop this from scratch as there are plenty of sources to implement a gaming environment in many languages. Following, basic game playing bots will need to be setup, as well as strategic bots and honest bots. Each level of trained bot will get more and more advanced.

In parallel, to satisfy the learning outcomes of this course as well as provide a sound understanding of machine learning, research pertaining to neural networks, types and architectures of machine learning, and intensive research involving reinforcement learning will be completed. Primarily during the first few weeks of the semester. Following, an implementation period will ensue to initiate environment setup and initial machine learning implementation. Lastly, the two phases will tie together to merge research and implementation through multiple stages of training, with various parameters and according to different strategies, resulting in a well developed reinforcement learning poker playing bot.

Training these bots will mostly consist of self play and reward based learning, however other strategies will be researched, including that of DeepStack, Libratus, Pluribus and many others that are successfully based on machine learning. The reason for two player poker, instead of 3, 4 or even 6, is because with a lower number of players at play, a rule based gaming strategy is far more effective, and much easier to implement and train. As the number of players increase, the game becomes a multi-player game that requires more sophisticated strategies and potentially even player modelling.

In terms of technical specifics, Python, PyTorch and TensorFlow will be used to implement the environment and AI. In terms of computationally intensive neural network training, rather than purchase of a high quality GPU, use of CUDA to run on a Mac OS GPU, or many other hardware solutions, Google Colab. will be used since it enables the use of free GPU's for training of python neural networks for up to 12 hours [1]. The overall resulting neural network architecture follows that of a Double Dueling Deep Q Network, and will be trained against basic Heuristic based bots on Colab. In terms of results, an exceptional AI was built producing competitive play against statistical model based bots, and substantial performance against an amateur human player.

Contents

List of Figures

1 Background

The complex game of poker requires reasoning with hidden information. When one plays a game, not only do they quantify the value of their own hand, they also aim to predict what cards the opposing players hold. Other players can even bet high when they're hand is not valuable at all, a strong tactic called *bluffing*. Additionally, there are other factors that weigh in, including pot size, call size, remaining stack and more. For background information on Texas Hold'em and how it is played, see the reference [7].

Figure 1: Online Texas Hold'em

Two player Texas Hold'em presents an excellent opportunity to implement machine learning, as rule based strategies work fairly well, according to many experienced sources [8]. Game theory, with higher numbers of players, is far less helpful. Additionally, this becomes incredibly computationally expensive as it drastically increases the number of possibilities, making it much harder to make a sound decision at the time of it's turn. Bots known as Libratus and DeepMind's Go bot used 100 CPU's and 2000 CPU's, respectively [8]. However, a much less expensive AI has been developed, known as *Pluribus*, which only required 2 CPU's to run. It was developed via reinforcement learning and self play, and such a tactic will most definitely be employed for the development of this project.

1.1 RL Overall Architecture - DQN

To provide a better visual understanding of what the integrated system with the deep Q network (DQN) will look like, see the figure below.

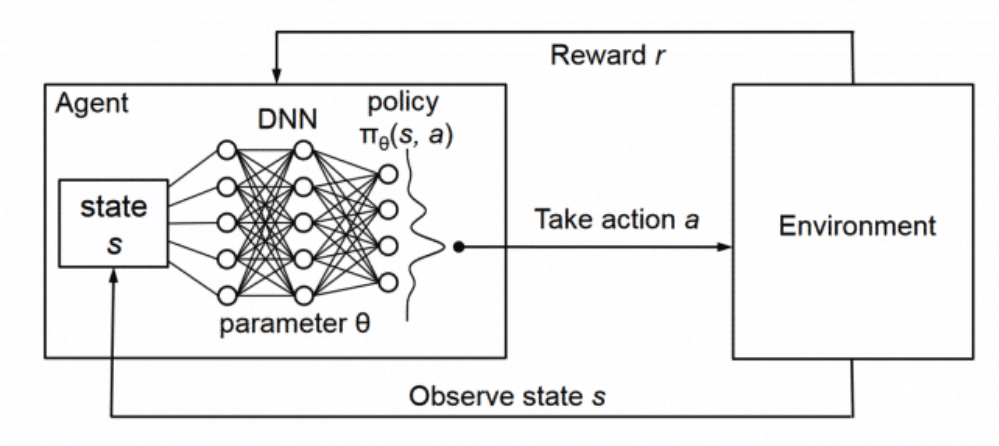


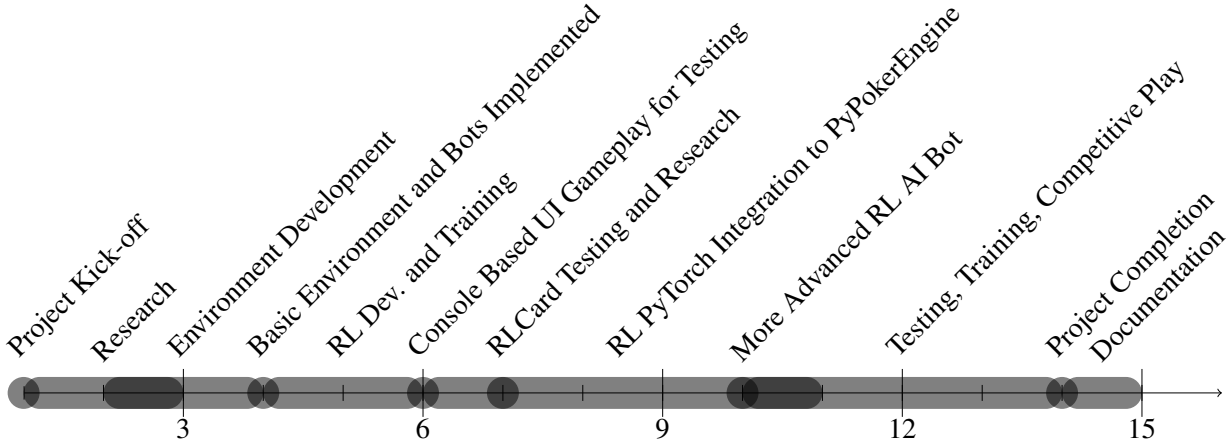
Figure 2: DQN RL Architecture [2]

For the RLPlayer, the agent, to make a decision on what action to take, the game state will be passed through the feedforward trained deep Q network upon every single change of state (after each player's turn and on the flop, turn, river - see [9] for poker terminology). Of course, the DQN will be trained through self play, until an optimal poker bot AI converges. Appendix A displays the few lines of code written in Keras (another python ML library, similar to PyTorch) to build a basic 5 layer neural network.

1.2 Other Information

1.2.1 Timeline

A revised timeline for the project can be seen below.



Project Timeline for the Summer 2020 Semester, by Week

1.2.2 Meeting Schedule

Meetings will be held at 11am PT, recurring weekly on Tuesdays.

2 Research

2.1 Machine Learning and Neural Networks

In order to achieve an in depth understanding of neural networks and machine learning, the course *Intro to Deep Learning with PyTorch* on Udacity has been selected. Over the course of the first 4 weeks of this semester, the module *Introduction to Neural Networks* has been completed, which outlines the concepts behind neural networks and how they are trained and tuned. This section will not dive into explaining the many concepts of neural networks, but will elaborate on how reinforcement learning applies to the game of poker and presumed topics that will need to be implemented.

2.1.1 Udacity Course Concepts

Luis Serrano, a course developer, guides the user through many of the common concepts of neural networks, including, but not limited to, the following:

- Classification problems - linear boundaries and higher dimensions
- Perceptrons- non linear, perceptron algorithm,
- Error Functions - log-loss error function, discrete vs continuous
- Softmax, One-Hot Encoding
- Probabilities, Cross-Entropy
- Logistic regression, gradient descent
- Non-linear models, neural network architecture
- Feedforward and backpropogation
- Training optimization, testing, over and under fitting
- Early stopping, regularization, dropout
- Local minima, random restart, vanishing gradient, momentum and many other common concepts

The course provides a good understanding of the concepts of neural networks, but does not dive deep into the concepts around reinforcement learning. It then goes to discuss the proper implementation of a neural network in PyTorch, but that will be explored throughout the implementation phase of this course (weeks 4-8).

2.2 Reinforcement Learning

2.2.1 Why RL?

The motivation for reinforcement learning (RL) lies in it's low computational expensiveness. Imagine you are playing chess, before you make a move, you are thinking what all the other player's potential moves are, even 2 or 3 moves ahead of the present. For a computer to do this, in a game with many possibilities and many players, it is incredibly expensive to do in a short period of time. Reinforcement learning is "an area of machine learning concerned with how software agents ought to take actions in an environment in order to maximize the notion of cumulative reward" [**?**].

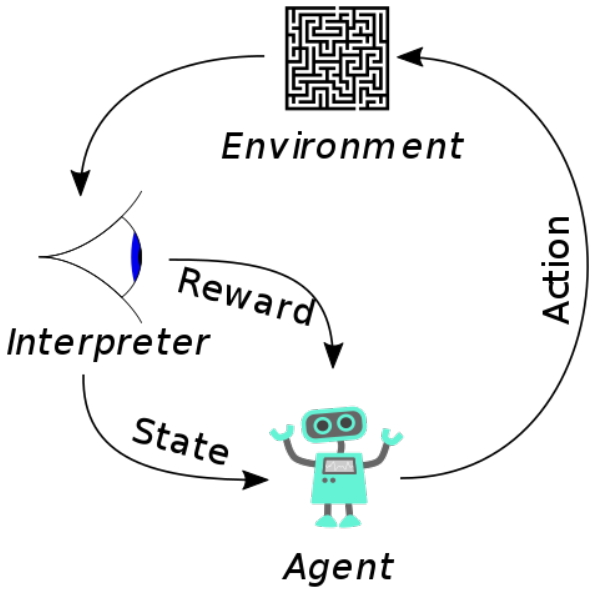


Figure 3: A Typical Reinforcement Learning Approach [**?**]

If you are familiar with the game of poker, the image shown above is nearly directly applicable to the game. Thus, implementing this type of learning, coupled with iterative self play, can lead to a very powerful poker playing bot, especially in two player poker. Potential for implementation in more than two player poker may be explored and tested depending on effectiveness of that of two player.

2.2.2 Elements of RL

The reference listed in [10] provided an incredibly well written introduction to reinforcement learning, hence most of the following information can be referenced there. In addition to an agent, an environment, and the other aspects listed in the previous sections, the following list discusses the other main elements of RL.

- *A policy:* essentially defines the rules of the agent. For example, you cannot bet when it is not your turn, or if you've folded your cards. The agent's possible actions at any given state.
- *A reward:* the goal of the agent. Basically, a reward is received by the agent from the environment upon every single discrete change in state. The agent's sole goal is to maximize

the reward's it receives. Of course in the game of poker, this reward will be a high value if the agent wins a hand, and low if it loses, but not too low if it folds without losing any money, for example.

- *A value function:* instead of an instant reward, the agent must also consider the effect of it's actions in the long run, quantified by the value function. For example, a state may have a low reward but have a high value, since it generally induces other states that have high rewards. In the case of poker, this is analogous to a player betting high when they have a very low stack. They may not have much money to bet, thus each bet has a low reward, but assuming they win the hand, the action of betting will have a high value as they will make money in the end. Value estimation is essentially the most important aspect to RL, as it determines how the agent will choose it's course of action.

- *A model:* this element essentially mimics the environment to predict how it will change in response to an action, before actually taking that action. Since the game of poker is nearly impossible to model because it is a game of imperfect information, and it would be incredibly difficult to model different players during gameplay, this will not be included in the poker RL implementation.

2.3 Deep Q Learning (DQN)

Q learning aims to iteratively populate a 2D matrix of Q values based on current environment states and performing actions. It essentially uses trial and error to map an action to a state and output the reward, thus quantifying the value of certain actions given a state. This type of learning uses the Bellman equation to create a Q learning function [3]. After finding the optimal Q matrix, the agent will pick it's action at any given state by finding the highest Q value for that state. This works well for low dimensional problems, but highly dimensional problems render this method impractical, hence the introduction of Deep Q Learning.

Deep Q Learning, on the other hand, uses a neural network to predict the reward at any given state. In the game of poker, there is no notion of how many states there will be - player A could bet X amount at any given moment, player C could fold, etc. This is another reason this method works effectively for high dimensional games. Lastly, instead of a 2D matrix to store all possible Q values, it uses a Q network that takes the current state as input and outputs Q values for all possible actions (bet, call, fold, bluff, raise). The figure below displays an excellent visual understanding.

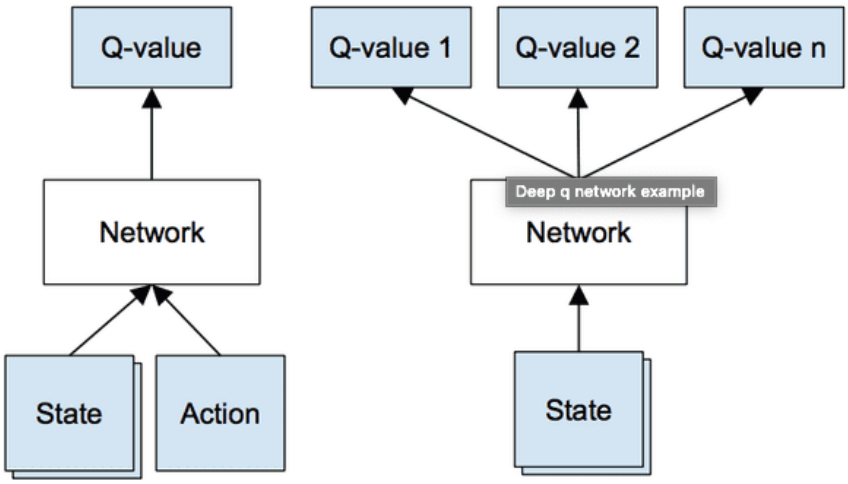


Figure 4: Q Learning vs Deep Q Learning [3]

Applying this to the game of poker will be tedious and trivial, however in [3], the algorithm performed very well at approximating a Nash Equilibrium for two player head's up push-or-fold poker - a very simplified version. It will most definitely be explored

2.3.1 Overall Architecture - DQN

To provide a better visual understanding of what the integrated system with the deep Q network (DQN) will look like, see the figure below.

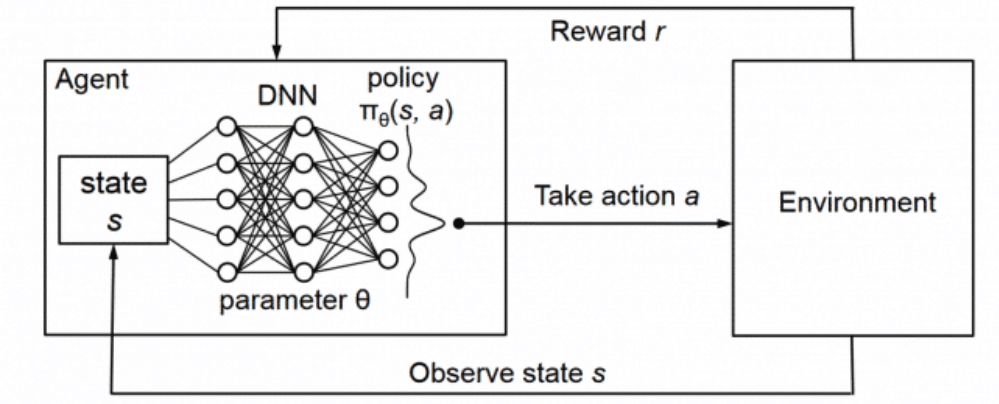


Figure 5: DQN RL Architecture [2]

For the RLPlayer, the agent, to make a decision on what action to take, the game state will be passed through the feedforward trained deep Q network upon every single change of state (after each player's turn and on the flop, turn, river - see [9] for poker terminology). Of course, the DQN will be trained through self play, until an optimal poker bot AI converges. Appendix A displays the few lines of code written in Keras (another python ML library, similar to PyTorch) to build a basic 5 layer neural network.

2.3.2 State Encoding

One cannot simply pass the notions of state to the DQN as how they are represented to humans, all aspects of state must be encoded - or quantified as numbers. Instead of a player's hand of an 8 of Clubs and 10 of Clubs, this must be converted to a vector that can be passed into the DQN. The figure below from [3] displays an example of **one-hot encoding** for this hand.

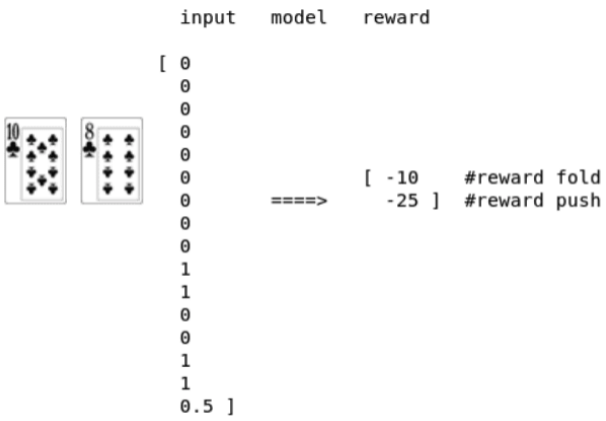


Figure 6: Example of Hand One-Hot Encoding [3]

This example contains rewards for the simplified game of heads up push-or-fold poker. There are many more states that are required to encode for 2+ player texas hold'em, such as:

- RLPlayer hand
- Community cards
- Pot size
- RLPlayer stack size (amount of money)
- Opponents' stack size
- Number of opponents who have not folded
- Minimum bet to play
- Blind sizes

These are the majority of the states that will need to be encoded, however some are already in numerical form, such as the pot size, for example $1,000, or players remaining, 3. Encoding is important as it represents how the DQN interprets and understands the game.

2.3.3 Reward Signal - Bot Strategy

The core **strategy** of the RLPlayer is based around the construction of the reward signal, or function. It determines what actions produce more rewards, affects the value function, Q values, and thus dictates how the player will play. It may lead to an aggressive strategy, a passive one, or others. The following equations represent a general expression of the function for Q.

$$Q(s, a) = r + \gamma max_{a'}(s', a') \tag{1}$$

Which implies that the Q value is equal to the immediate reward plus the future discounted (γ) reward for the most optimal action (a), at a given state (s). For backpropogation of our DQN, the Q learning function below can properly update Q values after performing an action.

$$Q_{t+1}(s_t, a_t) = Q_t(s_t, a_t) + \alpha((r_{t+1}) + \gamma max_a Q_t(s_{t+1}, a) - Q_t(s_t, a_t)) \tag{2}$$

Where α represents the learning rate. These equations, from [3], encompass the general concept of deep Q reinforcement learning through self play - performing actions, observing the changes in state and tuning the decision process based off the reward from these observations. Although they will not be used directly since DQN features a deep neural network to predict the Q value of an action, they are the base concept of Q learning. This is also know as a **Markov Decision Process**. Chapter 3 of [10] greatly discuss this entire subject. Appendix C displays just how simple it is to build a DQN in PyTorch.

3 Implementation

3.1 Python Poker Environment

Along with research, time has been put into the setup and testing of the chosen poker environment, to ensure feasibility of a properly functioning poker playing game. Once this is tested, time can be spent investigating PyTorch and it's integration to the poker bot and it's environment for training through self play.

3.1.1 PyPokerEngine

After exploring many options online, a well developed poker game engine written in Python has been chosen to utilize for the environment. The library can be found on GitHub [11] as well as supporting documentation for reinforcement learning users [12], another reason it was chosen for this project. You can play against your AI via a console, through a GUI, or run a simulation of any number of rounds of AI vs AI, or a combination of bots. The library comes with built in bots, such as a fish player, fold player, random and honest players, which will be used for the training of our reinforcement learning (RL) bot. Appendix D displays just how simple it is to setup an RL bot [11], where the *declare_actions* function is where the bot will make decisions on which actions to take based off the feedforward results of the trained neural network.

3.1.2 PyPokerGUI

Playing poker against a tuned AI bot through a console is a tedious and unrealistic task. After spending hours training an AI to play poker, one should be able to test it in an environment that resembles that of a real casino, hence the implementation of GUI (guided user interface) support for the *PyPokerEngine*, called the *PyPokerGUI* [13]. All settings exist in a **.yaml** file to configure and setup a localhost server to run the python module in any browser. Invoking the following command through a terminal can start the server and initiate the game engine GUI:

python3 -m pypokergui serve poker_conf.yaml –port 8000 –speed moderate

The following figure displays a basic two player game of poker in a browser window:

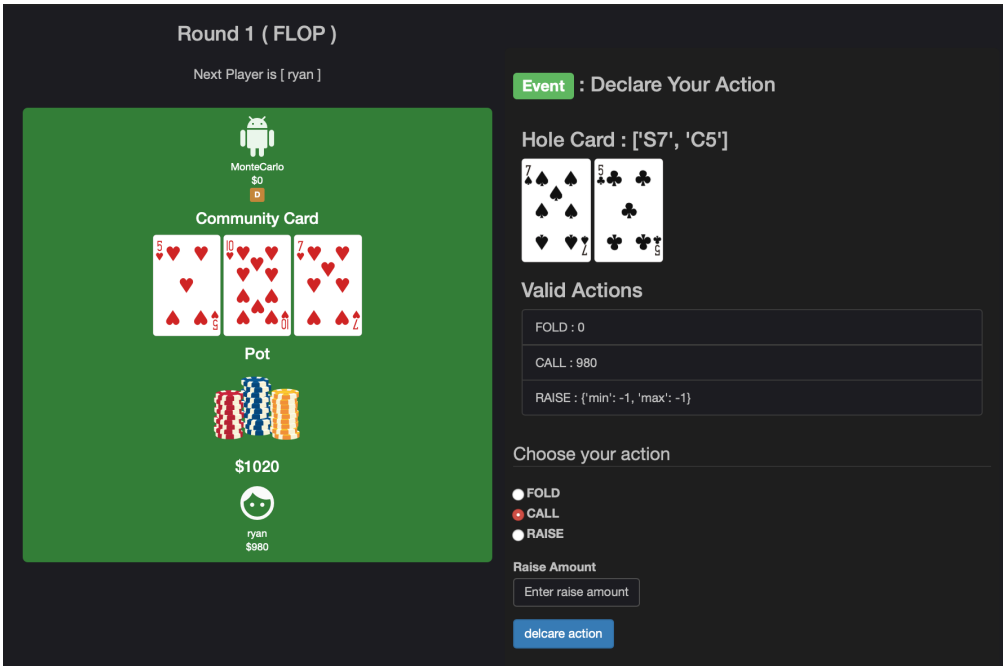


Figure 7: PyPokerGUI Example

Of course, the game settings can be modified in the **.yaml** file to play multiple players of different types, different blind sizes, and more. This ultimately provides an excellent testing method for our trained neural network.

3.1.3 Training and Simulation

In order to train the neural network, another method of testing or simulation must be readily available to run at high speeds that does not require user input. Hence, just a few lines can be written in a Python script to begin a simulation of any number of games of poker, while consistently displaying the results.

```
1 from pypokerengine.api.game import setup_config, start_poker
2 config = setup_config(max_round=10, initial_stack=100, small_blind_amount=5)
```

```
config.register_player(name="p1", algorithm=FishPlayer())
config.register_player(name="p2", algorithm=HonestPlayer())
config.register_player(name="p3", algorithm=RLPlayer())
game_result = start_poker(config, verbose=1)
```

Some example results can be seen below:

```
Started the round 10
Street "preflop" started. (community card = [])
"p1" declared "call:10"
"p2" declared "call:10"
"p3" declared "call:10"
Street "flop" started. (community card = ['H7', 'C7', 'C9'])
"p2" declared "call:0"
"p3" declared "call:0"
"p1" declared "call:0"
Street "turn" started. (community card = ['H7', 'C7', 'C9', 'H6'])
"p2" declared "call:0"
"p3" declared "call:0"
"p1" declared "call:0"
Street "river" started. (community card = ['H7', 'C7', 'C9', 'H6', 'HK'])
"p2" declared "call:0"
"p3" declared "call:0"
"p1" declared "call:0"
"['p2']" won the round 10 (stack = {'p1': 90, 'p2': 150, 'p3': 60})
```

This simulation will be the core of training the RLPlayer - the neural network. Upon every single action (represented by each new line in the code above) , the player will backpropogate through it's network to tune the gains and minimize the error function, until a best fit is reached.

3.2 PyTorch Reinforcement Learning

There are many libraries that implement the many various concepts and architectures of machine learning in Python, such as TensorFlow, Theano, PyTorch, and more. PyTorch has been chosen as it is the most user friendly, while it still preserves high speeds as it's core is written in C++. It will be integrated with the PyPokerEngine environment to train, test and play the RL bot. Techniques to develop numerous types of RL agents will be employed, including Deep Q Neural Networks and Neural Fictitious Self-Play.

RLCard - An RL Toolkit RLCard: A Toolkit for Reinforcement Learning in Card Games [4]. This library includes many pre-built agents using TensorFlow instead of PyTorch, and thus provides an excellent, in depth understanding of reinforcement learning and card games. Additionally, there are numerous functioning environments for different types of card games, such as Blackjack, UNO, Rummy, Texas Hold'em and more. Since RLCard possesses the required development for this directed study, it will merely be used as a reference, and benchmark. Further, it's environment will not be utilized for training, as they are less developed and don't contain a GUI. Hence, the required learning to build a successful RL bot will most definitely be gained.

TensorFlow Training - NFSP Agent To assess the computational expense of training a basic NFSP agent in No Limit Hold'em via TensorFlow on a 2015 Mac OS X, a training session was run. Over the span of approximately 45 minutes, the following results were obtained, shown in figure 8 below.

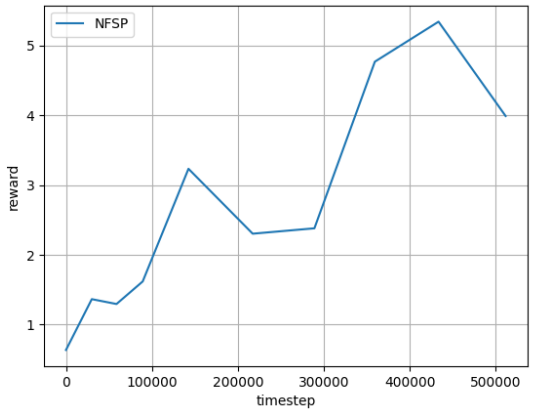


Figure 8: NFSP Agent Training via TensorFlow in No Limit Hold'em

Although clearly there is learning taking place, generally, a training agent undergoes plenty more iterations, such as those seen in the following figure.

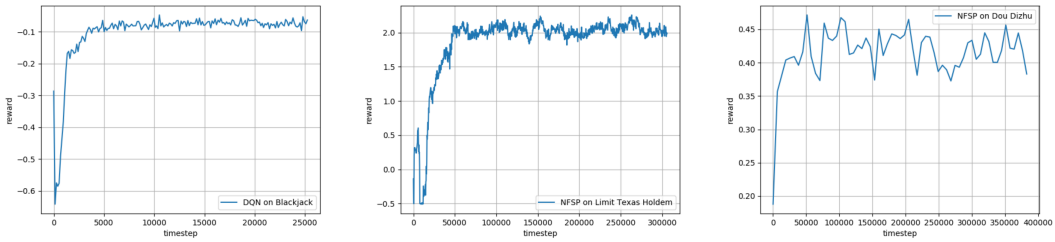


Figure 9: RLCard Typical Training Curves [4]

Thus, it makes much more sense to train the agents on a more expensive piece of hardware, a graphics processing unit - GPU. Since these are built to handle array mathematics in an incredibly short amount of time with respect to a CPU, they can easily shorten the time required to train a neural network. Additionally, Google Colab will be utilized for cloud computing and thus faster training.

3.2.1 Poker Player Class

Ultimately, the goal is to develop an RL agent through PyTorch, in the PyPokerEngine environment. RLCard serves as a great reference and benchmark to do so.

DQN and NFSP via PyTorch with PyPokerEngine BasePokerPlayer Class Generally, the RLPlayer should look something like this, depending on the type of agent being implemented. The following Python-pseudo code displays the approximate setup of RLPlayer:

```
from pypokerengine.api.game import setup_config as env
class RLPlayer(BasePokerPlayer):
    def __init__(self):
        # Define init functions for player
        super().__init__()
        self.wins = 0
        self.losses = 0
        # Setup agent
        self.agent = NFSPAgent(scope='nfsp',
                        action_num=env.action_num,
                        state_shape=env.state_shape,
                        hidden_layers_sizes=[128,128],
                        q_mlp_layers=[128,128])
                # Load pre-trained model
                checkpoint_path = 'models/nfsp.pth'
                checkpoint = torch.load(checkpoint_path)
                self.agent.load(checkpoint)

    def declare_action(self, valid_actions, hole_card, round_state):
        # Define action and amount
        # as the output of a feedforward pass through trained NN
        return action, amount

    def receive_game_start_message(self, game_info):
            # Define state update functions. For example this function:
        self.n_players = env.game_info['player_num']
```

The section *declare_ action* is where the neural net (NN) will operate. It's actions will be determined based off a feedforward pass through it's NN. The sections below will update the states of the RLPlayer based off the game environment. Finally, a *train* function will be setup to train the RLPlayer, through self-play.

Google Colab Google Colaboratory (Colab) allows anyone to deploy Python projects on remote servers with powerful GPU's [1]. By merely inputting a link to a Github repository, one can execute computationally expensive programs even though they do not have direct access to hardware, through Colab. It begins with a free trial and then has a fee that is pay per use. This will be explored after a few tests of training the RLPlayer on a basic Mac OS CPU to better understand just how expensive training really is.

3.2.2 Agent-Environment Integration

The PyTorch agent (DQN and NFSP) integration and training with the PyPokerEngine environment is the remaining development for the project. To summarize as a list of the goals for the oncoming 4 weeks;

1. State encoding of available cards, stack size, pot size, call cost, etc.

2. Reward signal setup, testing and variation (DQN)
3. Build RLPlayer bot, load trained model for feedforward agent decision making (during play)
4. Effective, observable learning while training
5. Testing via Human vs. Bot GUI play

4 Results

Since the completion of Report #2, there have been substantial developments in the completion of a DQN RL bot, trained to play and win at 1 on 1 poker. The following section discusses said developments, including a switch from PyTorch to TensorFlow, the overall double-dueling-DQN architecture, training the model, and testing the trained model.

PyTorch to TensorFlow After spending hours and hours reading through documentation on PyTorch while attempting to properly setup and initialize a bot, I decided to change directions and continue development in TensorFlow (TF). Most RL resources I found online seem to use TF, including the RLCard library, which serves as a very useful benchmark. Further, other resources were found integrating with PyPokerEngine that also use TF. Rather than wasting time trying to learn PyTorch, which isn't the goal of this study, it was a wise decision in switching to TF.

4.1 RLPlayer Class

As mentioned in section 3.2.1, the creation of the RLPlayer will inherit it's attributes from BasePokerPlayer, a generic class for PyPokerEngine poker bots. After extensive developments and testing to write a basic DQN, mediocre results were obtained that did not suffice.

4.1.1 DQN

A basic deep Q network was not merely enough for a complex scenario. The following image below displays the basic architecture for a DQN.

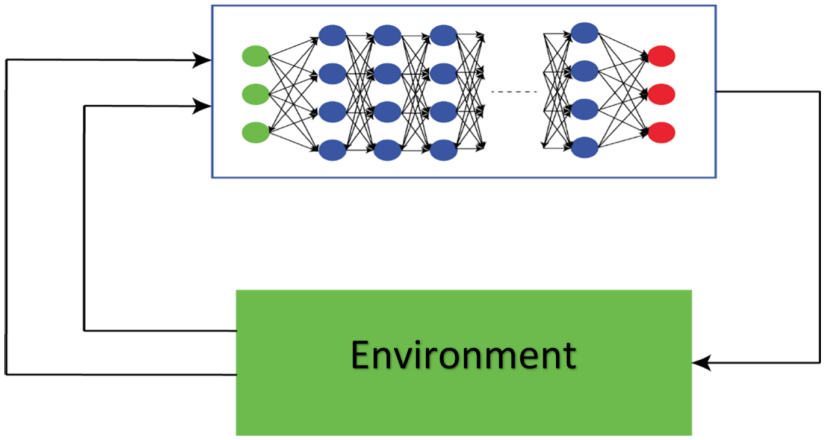


Figure 10: RL via DQN - Basic Architecture [5]

As the bot trains, actions that result in high rewards will essentially *compound* the resulting Q-value for that action. Thus, if the poker bot learns that the action of *bluffing* results in a win - or a high reward - it will continuously take the action of bluffing, even if it's dealt terrible hole cards, such as an unsuited 2 and 8. Thus, we need some way of evaluating an actions potential reward using a memory buffer, or *memory replay* or *experience*.

4.1.2 Double DQN

In order to effectively bring down these high rewarded action values, we can use a secondary DQN that is a carbon copy of the main DQN, however only from the last episode, or round of poker. Thus, this model can be used to obtain a better Q-value. The following figure displays a helpful but quantitative understanding of a double DQN (DDQN).

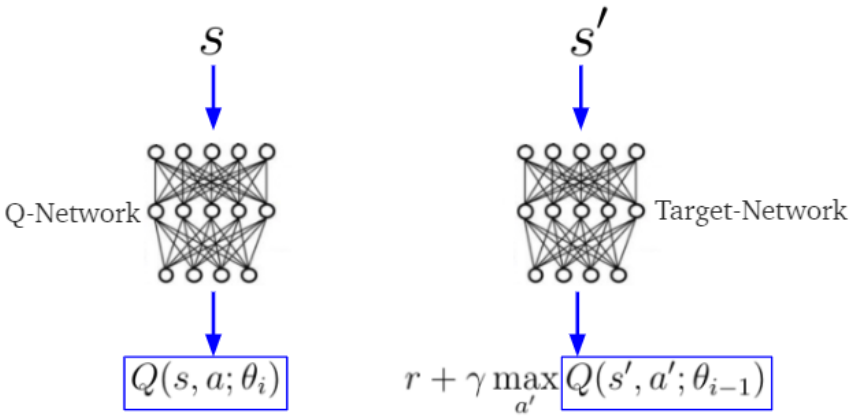


Figure 11: DDQN - Basic Architecture [6]

4.1.3 Double Dueling DQN

Finally, a dueling DDQN (DDDQN) aims to slightly alter the structure of the model:

$$Q(s,a) = V(s) + A(s,a) - \frac{1}{n}\sum_{a'} A(s,a')$$

Where V is the value of state s, and A is the advantage of an action a while in state s. Thus, we can see that the value of a state is completely independent from an action, which definitely solves our initial issue. For example, if the bot is dealt poor cards, such as the unsuited 2 and 8, even though it may have had good experience with bluffing during training, it's current state is poor, and this will now decrease the Q-value of taking that same route.

As for A, advantage, assuming the bot is in a state where all actions have the same value, there would be no optimal action. If we find the mean of all the advantages for actions taken from the secondary DQN and subtract this from our main advantage A, this will ensure that the Q value does not overshoot or explode. The following figure displays a resulting archtecutre of the overall DDDQN that will be used for the RLPlayer.

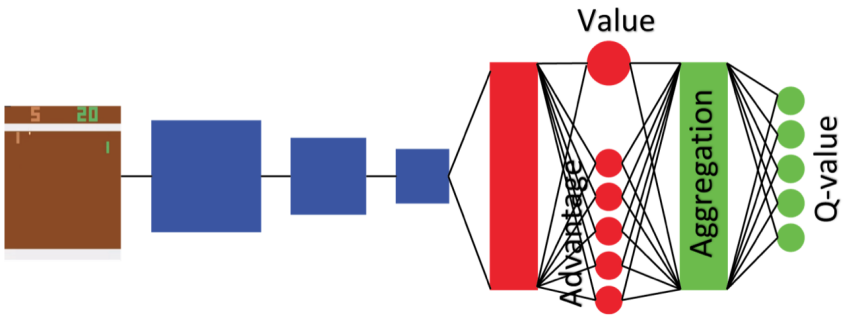


Figure 12: DDDQN - Basic Architecture [5]

4.1.4 Overall NN Setup

Many thanks to *Evgeny Kashin* [14] for his implementation of a DDDQN network. This allowed me to advance my basic PyTorch DQN to an advanced TF DDDQN. In general, the network is setup as shown in appendix E.

4.1.5 Encoding Inputs

The input data to the neural network is encoded as follows:

- **Round state: street.** The street refers to the community cards shown face up on the table, revealing 3 on the *flop*, then 1 more each on the *turn* and finally, *river*.
- **Pot size.** The amount of cash in the main pot (0 at start of each round, plus blinds).
- **Stack size.** The amount of cash the bot has remaining.

- **Dealer position.** Who is currently the dealer.
- **Small-blind position.** Who is currently the small blind.
- **Big-blind position.** Who is currently the big blind.
- **Next player.** Who's turn it is to take action.
- **Round number.** How many rounds of poker have been played since buy in.
- **Other stack sizes.** How much cash other players have.
- **Hole-cards estimation.** Pre-trained model that estimates the power of dealt hole cards courtesy of Evgeny [14].

Overall, this is pretty much all the available information in the game of poker - hence being a game of *imperfect information*. The output of the network returns the action number of the available actions:

- Fold.
- Call. Pay the minimum amount of cash required to stay in the game.
- Raise. Increase the required call to a set amount that others must pay to stay in the game.
  - Amount. If Raise chosen, how much cash to raise by.

4.2 Training

The process of training the network was done by playing against a Heuristic based bot, that uses a MonteCarlo simulation to compute the probability it's cards have of winning. This is what most average amateur poker players do - they're careful with their money and only bet when they have a good hand. Hence, it should perform well against honest players, but may not perform well against a *bluff*, for example.

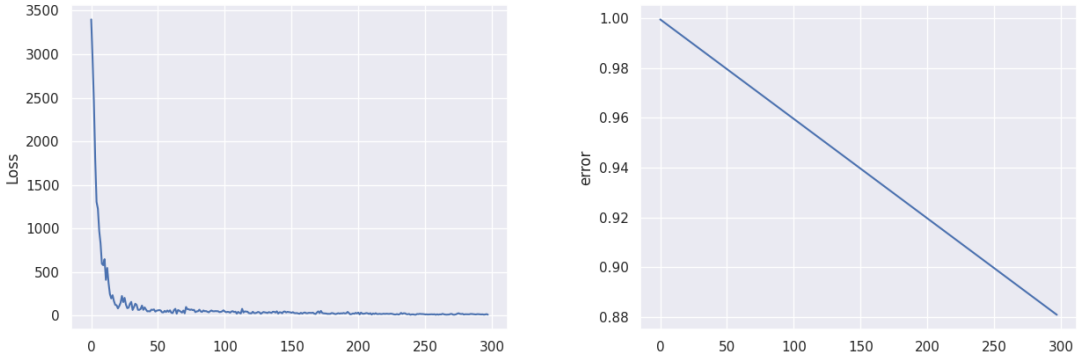
4.2.1 Parameters

The following parameters were used to implement the training script.

```
# Training params.
batch_size = 128
update_freq = 50 # how often to perform regression (train model)
y = 0.99 # discount factor
start_E = 1 # starting chance of random action
end_E = 0.2 # final chance of random action
num_episodes = 500000 # total games of poker to play (unrealistic)
annealings_steps = num_episodes/5 # how many steps to reduce start_E to end_E
pre_train_steps = 1000 # how many steps of random action before training begins
load_model = False # load pre-trained model
path = '/content/drive/My Drive/PokerRLModels/poker-RL/src/cache/models/DQN2Plots/DQN' # where to save the model
h_size = 128 # the size of final conv layer before spliting it into advantage and value streams
tau = 0.01 # rate to update target network toward primary network
is_dueling = True # whether or not to use the dueling architecture
```

4.2.2 Learning Plots

To effectively understand if the agent did in fact learn over thousands of games of poker, numerous metrics were taken. While running the training script over only 1000 episodes (games of poker), and thousands of epochs (actions taken), clearly, there is learning as the overall loss is minimized. Note that values were logged every 50 actions taken, resulting in 300 data points.

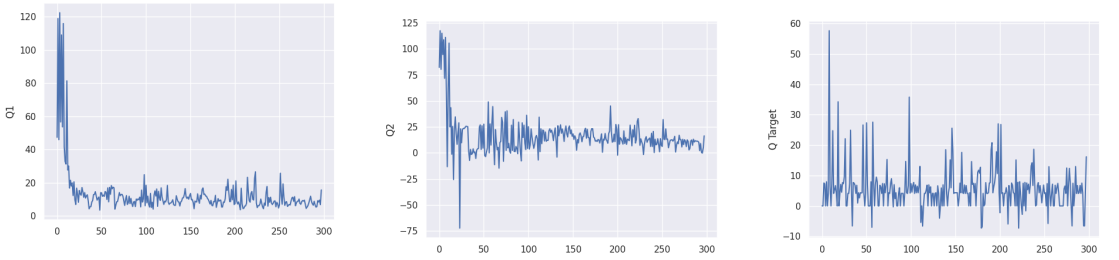


(a) Loss

(b) Error

Figure 13: Training Performance Metrics

Clearly, the AI learns how to perform well against the Heuristic bot, after just 150 games or so. Further, we can see the resulting Q-values for each of the double DQN networks, and the target network.



(a) Primary DQN Predicted Q-Value

(b) Primary DQN Q-Value

(c) Secondary DQN Q-Value

Figure 14: Double Dueling DQN Q-Values - Training

As shown above in figure 14, the networks immediately begin to tune their networks after 1000 actions have been taken, and we see the values converge quickly. Although these indicators look promising, just how well will it perform?

4.3 Testing

4.3.1 Simulation Testing

Firstly, we will test the trained RLPlayer against other various poker bots over 50 games of poker and plot the resulting bar graphs. The following parameters were used for testing:

- Max rounds = 100. This will ensure a single game will finish after 100 rounds, the winner will be whoever has more chips.
- Initial stack = 1000. Each player starts with $1000 worth in chips.
- Small blind amount = 15.
- Number of games = 50. RLPlayer will play 50 games against each bot.

The following results were obtained:

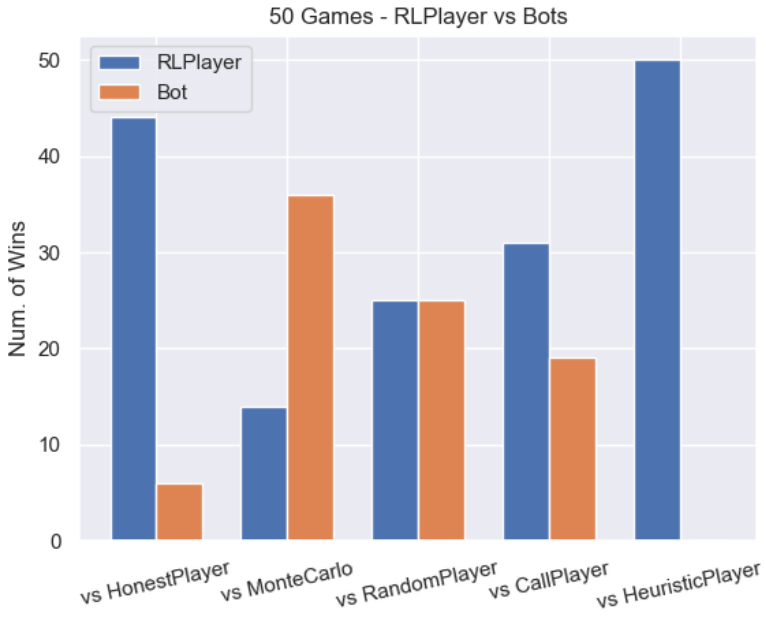


Figure 15: 50 Game Simulation Against Other Bots

By analysing figure 15 shown above, we can see that the bot performs quite well against heuristic based bots; Honest, MonteCarlo and Heuristic. This makes sense as the DDDQN was trained against the HeuristicPlayer bot. However, when playing other bots that appear to play less deterministically, such as CallPlayer and RandomPlayer, the bot appears to perform sub-optimally. Finally, we can see that RLPlayer won every single game against HeuristicPlayer, as it should since they trained over 11,600 episodes. Overall, these are solid results.

4.3.2 Me vs AI

Since I have some experience with poker myself, I figured it would be a good test to play the trained model myself. Using PyPokerGui, it was quite simple to configure and run the poker environment:

```
ante: 0
blind_structure: null
initial_stack: 1500
max_round: 50
small_blind: 15
ai_players:
- name: DQN
  path: ./src/bots/DQNPlayer1v1.py
```

And by invoking:

python3 -m pypokergui serve poker_conf.yaml –port 8000 –speed moderate

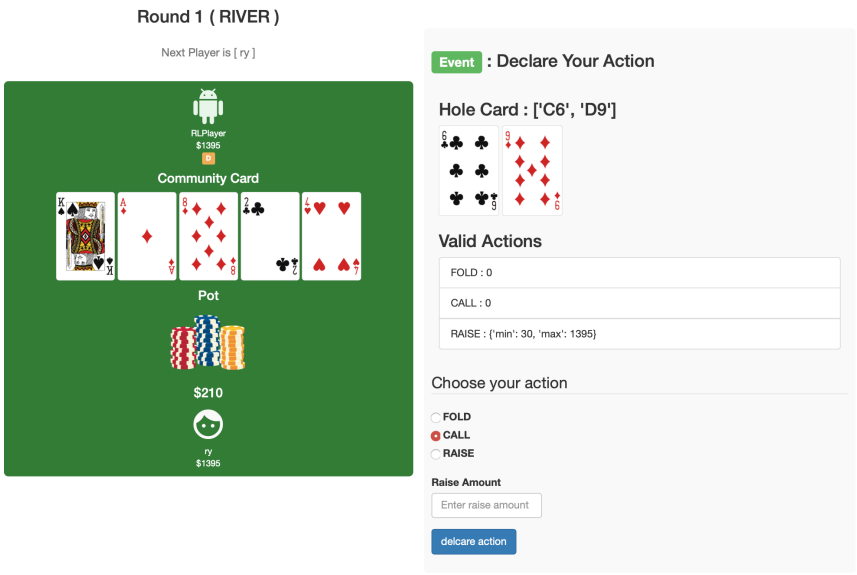


Figure 16: GUI Against RLPlayer Trained Model

Figure 16 above shows the interface used to play against the RLPlayer trained model.

Observations. In summary, RLPlayer is quite aggressive. It consistently bets between $30 and $45, and generally calls. Occasionally bluffing (going all in) would cause it to fold, but not very often. I found that if it had high hole cards it would generally call my bluff. Furthermore, it would also go all in when it was dealt good hole cards, sometimes waiting until the turn or river to do so. My strategy was to play similarly to HeuristicPlayer, as it invoked honest, amateur gameplay that the majority of poker players follow, and also because this is what the AI was trained to beat.

Overall, it was a fairly competitive opponent. I was able to begin winning a few games after an hour or so and a few games of play, after figuring out it's weaknesses and exploiting them. However, the first few games it had me beat, and I'd definitely say it's an above average opponent for anyone looking to play an AI in poker.

5 Conclusions

To conclude, this was a highly productive course. Having no experience with machine learning, it was a great way to foray into the complex realm of machine learning and artificial intelligence. Having dove into concepts of reinforcement learning, deep Q learning, double and dueling DQN, as well as touched on several others, this was a very fulfilling experience.

I would also like to thank Dr. Rad for his supervision and advisement throughout the course of this study.

5.1 Recommendations

Moving forward, I would like to:

- Train the agent against multiple players.
- Train the agent, 1v1, against other types of players.
- Adjust numerous parameters while training.
- Integrate with an online environment with fake money that will allow the agent to be tested against or trained against online poker players.

References

[1] *Welcome to Colaboratory*. [Online]. Available: https://colab.research.google.com/notebooks/intro.ipynb

[2] *Reinforcement learning Part 2: Getting started with Deep Q-Networks*. [Online]. Available: https://www.novatec-gmbh.de/en/blog/deep-q-networks/

[3] *Applying Deep Reinforcement Learning to Poker*. [Online]. Available: https://www.adaltas.com/en/2019/01/09/applying-deep-reinforcement-learning-poker/

[4] *RLCard: A Toolkit for Reinforcement Learning in Card Games*. [Online]. Available: http://rlcard.org

[5] *Deep Reinforcement learning: DQN, Double DQN, Dueling DQN, Noisy DQN and DQN with Prioritized Experience Replay*. [Online]. Available: https://medium.com/@parsa_h_m/deep-reinforcement-learning-dqn-double-dqn-dueling-dqn-noisy-dqn-and-dqn-with-prioritized-551f621a9

[6] *Self Learning AI-Agents III:Deep (Double) Q-Learning*. [Online]. Available: https://towardsdatascience.com/deep-double-q-learning-7fca410b193a

[7] *Texas Hold'em*. [Online]. Available: https://en.wikipedia.org/wiki/Texas_hold_%27em

[8] *No limit: AI poker bot is first to beat professionals at multiplayer game*. [Online]. Available: https://www.nature.com/articles/d41586-019-02156-9

[9] *Glossary of Poker Terms*. [Online]. Available: https://en.wikipedia.org/wiki/Glossary_of_poker_terms

[10] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. The MIT Press Cambridge, Massachusetts London, England: A Bradford Book, 2017.

[11] *PyPokerEngine Github*. [Online]. Available: https://github.com/ishikota/PyPokerEngine

[12] *PyPokerEngine Documentation*. [Online]. Available: https://ishikota.github.io/PyPokerEngine/

[13] *PyPokerGUI Github*. [Online]. Available: https://github.com/ishikota/PyPokerGUI

[14] *EvgenyKashin/TensorPoker*. [Online]. Available: https://github.com/EvgenyKashin/TensorPoker/blob/master/scripts/DQNPlayer.py

Appendices

A Heads Up Push-or-Fold DQN Model

A basic 5 layer neural network written using Keras in Python.

```
def preflop_model():

    input_n = Input(shape=(16,), name="input")

    x = Dense(32, activation='relu')(input_n)
    x = Dense(64, activation='relu')(x)
    x = Dense(16, activation='relu')(x)
    out = Dense(2)(x)
    model = Model(inputs=[input_n], outputs=out)
    model.compile(optimizer='adam', loss='mse')

    return model
```

B Basic PyTorch NN

```
from torch import nn
class Network(nn.Module):
    def __init__(self):
        super().__init__()

        # Inputs to hidden layer linear transformation
        self.hidden = nn.Linear(784, 256)
        # Output layer, 10 units - one for each digit
        self.output = nn.Linear(256, 10)

        # Define sigmoid activation and softmax output
        self.sigmoid = nn.Sigmoid()
        self.softmax = nn.Softmax(dim=1)

    def forward(self, x):
        # Pass the input tensor through each of our operations
        x = self.hidden(x)
        x = self.sigmoid(x)
        x = self.output(x)
        x = self.softmax(x)

        return x
```

C Basic PyTorch DQN

```
class DQN(nn.Module):

    def __init__(self, h, w, outputs):
        super(DQN, self).__init__()
        self.conv1 = nn.Conv2d(3, 16, kernel_size=5, stride=2)
        self.bn1 = nn.BatchNorm2d(16)
        self.conv2 = nn.Conv2d(16, 32, kernel_size=5, stride=2)
        self.bn2 = nn.BatchNorm2d(32)
        self.conv3 = nn.Conv2d(32, 32, kernel_size=5, stride=2)
        self.bn3 = nn.BatchNorm2d(32)

        # Number of Linear input connections depends on output of conv2d layers
        # and therefore the input image size, so compute it.
        def conv2d_size_out(size, kernel_size = 5, stride = 2):
            return (size - (kernel_size - 1) - 1) // stride  + 1
        convw = conv2d_size_out(conv2d_size_out(conv2d_size_out(w)))
        convh = conv2d_size_out(conv2d_size_out(conv2d_size_out(h)))
        linear_input_size = convw * convh * 32
        self.head = nn.Linear(linear_input_size, outputs)

    # Called with either one element to determine next action, or a batch
    # during optimization. Returns tensor([[left0exp,right0exp]...]).
    def forward(self, x):
        x = F.relu(self.bn1(self.conv1(x)))
        x = F.relu(self.bn2(self.conv2(x)))
        x = F.relu(self.bn3(self.conv3(x)))
        return self.head(x.view(x.size(0), -1))
```

D PyPokerEngine Player Setup

```
from pypokerengine.players import BasePokerPlayer
from pypokerengine.api.emulator import Emulator
from pypokerengine.utils.game_state_utils import restore_game_state

from mymodule.poker_ai.player_model import SomePlayerModel

class RLPLayer(BasePokerPlayer):

    # Setup Emulator object by registering game information
    def receive_game_start_message(self, game_info):
        player_num = game_info["player_num"]
        max_round = game_info["rule"]["max_round"]
        small_blind_amount = game_info["rule"]["small_blind_amount"]
        ante_amount = game_info["rule"]["ante"]
        blind_structure = game_info["rule"]["blind_structure"]

        self.emulator = Emulator()
        self.emulator.set_game_rule(player_num, max_round, small_blind_amount, ante_amount)
        self.emulator.set_blind_structure(blind_structure)

```

```
        # Register algorithm of each player which used in the simulation.
        for player_info in game_info["seats"]["players"]:
            self.emulator.register_player(player_info["uuid"], SomePlayerModel())

    def declare_action(self, valid_actions, hole_card, round_state):
        game_state = restore_game_state(round_state)
        # decide action by using some simulation result
        # updated_state, events = self.emulator.apply_action(game_state, "fold")
        # updated_state, events = self.emulator.run_until_round_finish(game_state)
        # updated_state, events = self.emulator.run_until_game_finish(game_state)
        if self.is_good_simulation_result(updated_state):
            return # you would declare CALL or RAISE action
        else:
            return "fold", 0
```

E DQNPlayer TF Setup

```
self.scalar_input = tf.compat.v1.placeholder(tf.float32, [None, 17 * 17 * 1])
self.features_input = tf.compat.v1.placeholder(tf.float32, [None, 13])

xavier_init = tf.compat.v1.keras.initializers.VarianceScaling(scale=1.0, mode="fan_avg", distribution="uniform")

self.img_in = tf.reshape(self.scalar_input, [-1, 17, 17, 1])
self.conv1 = tf.compat.v1.layers.conv2d(self.img_in, 32, 5, 2, activation=tf.nn.elu, kernel_initializer=xavier_in
self.conv2 = tf.compat.v1.layers.conv2d(self.conv1, 64, 3, activation=tf.nn.elu, kernel_initializer=xavier_init)
self.conv3 = tf.compat.v1.layers.conv2d(self.conv2, self.h_size, 5, activation=tf.nn.elu,
                                kernel_initializer=xavier_init)

#self.conv3_flat = tf.contrib.layers.flatten(self.conv3)
self.conv3_flat = tf.compat.v1.layers.flatten(self.conv3)
self.conv3_flat = tf.compat.v1.layers.dropout(self.conv3_flat)

self.d1 = tf.compat.v1.layers.dense(self.features_input, 64, activation=tf.nn.elu, kernel_initializer=xavier_init
self.d1 = tf.compat.v1.layers.dropout(self.d1)
self.d2 = tf.compat.v1.layers.dense(self.d1, 128, activation=tf.nn.elu, kernel_initializer=xavier_init)
self.d2 = tf.compat.v1.layers.dropout(self.d2)

self.merge = tf.concat([self.conv3_flat, self.d2], axis=1)
self.d3 = tf.compat.v1.layers.dense(self.merge, 256, activation=tf.nn.elu, kernel_initializer=xavier_init)
self.d3 = tf.compat.v1.layers.dropout(self.d3)
self.d4 = tf.compat.v1.layers.dense(self.d3, self.h_size, activation=tf.nn.elu, kernel_initializer=xavier_init)

if is_double:
    self.stream_A, self.stream_V = tf.split(self.d4, 2, 1)
    self.AW = tf.Variable(xavier_init([self.h_size // 2, total_num_actions]))
    self.VW = tf.Variable(xavier_init([self.h_size // 2, 1]))

    self.advantage = tf.matmul(self.stream_A, self.AW)
    self.value = tf.matmul(self.stream_V, self.VW)

    self.Q_out = self.value + tf.subtract(self.advantage, tf.reduce_mean(input_tensor=self.advantage, axis=1, kee
else:
    self.Q_out = tf.compat.v1.layers.dense(self.d4, 5, kernel_initializer=xavier_init)
```

```
37
38  self.predict = tf.argmax(input=self.Q_out, axis=1)
39
40  self.target_Q = tf.compat.v1.placeholder(tf.float32, [None])
41  self.actions = tf.compat.v1.placeholder(tf.int32, [None])
42  self.actions_onehot = tf.one_hot(self.actions, total_num_actions, dtype=tf.float32)
43
44  self.Q = tf.reduce_sum(input_tensor=tf.multiply(self.Q_out, self.actions_onehot), axis=1)
45
46  self.td_error = tf.square(self.Q - self.target_Q)
47  self.loss = tf.reduce_mean(input_tensor=self.td_error)
```