MSE 486 Directed Study
Report #2 - Implementation

# 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

July $3^{rd}$, 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 and PyTorch will be used since it is the most user friendly language, however the core of PyTorch is written in C++, so as to preserve high speeds. 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]. Finally, training of the bot will be done through neural fictitious self-play (NFSP), and Deep Q Learning networks (DQN).

## 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 [4].

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 [5]. 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 [5]. 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.

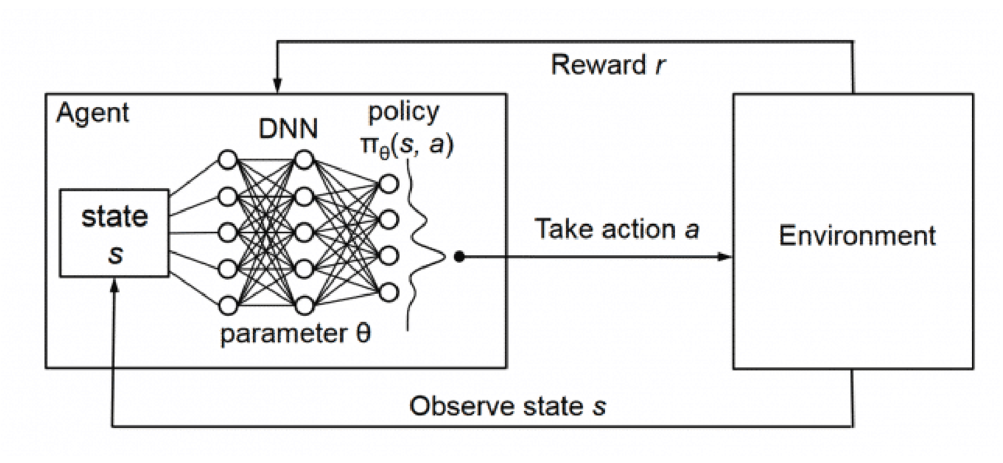


Figure 2: DQN RL Architecture [2]

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 [6] 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 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.

### 2.1 PyPokerEngine

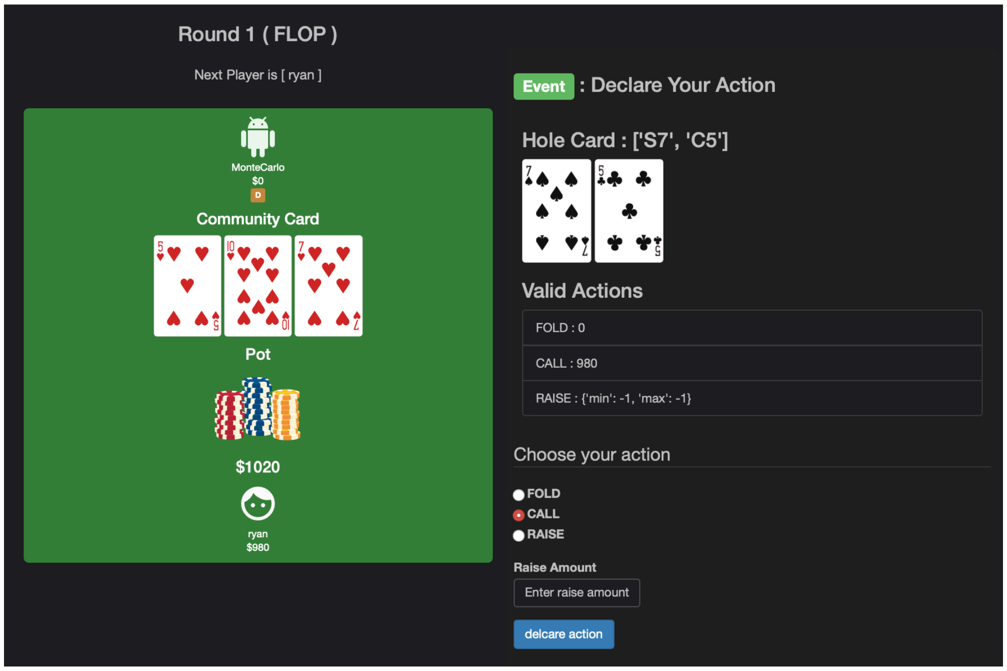
After exploring many options online, a well developed poker game engine written in Python has been chosen to utilize for the environment. The library can be found on GitHub [7] as well as supporting documentation for reinforcement learning users [8], another reason it was chosen for this project. You 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 [7], 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.

### 2.2 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* [9]. 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:


Round 1 ( FLOP )

Next Player is [ ryan ]

Event : Declare Your Action

Hole Card : ['S7', 'C5']

Valid Actions

FOLD : 0

CALL : 980

RAISE : {'min': -1, 'max': -1}

Choose your action

○ FOLD
[✓ CALL]
○ RAISE

Raise Amount

Figure 3: 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.

## 2.3 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.

```python
from pypokerengine.api.game import setup_config, start_poker
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 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.

### 3.1 RLCard - An RL Toolkit

RLCard: A Toolkit for Reinforcement Learning in Card Games [3]. 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.

### 3.1.1 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 4 below.

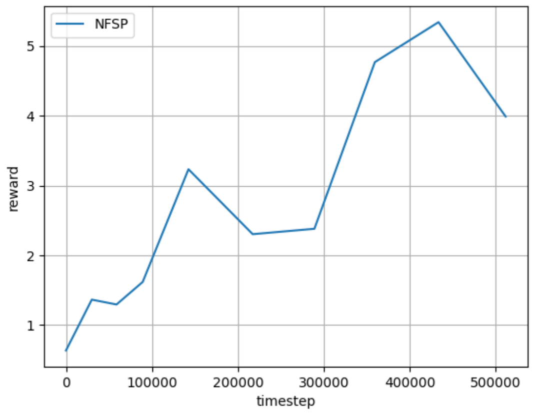


Figure 4: NFSP Agent Training via TensorFlow in No Limit Hold'em

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

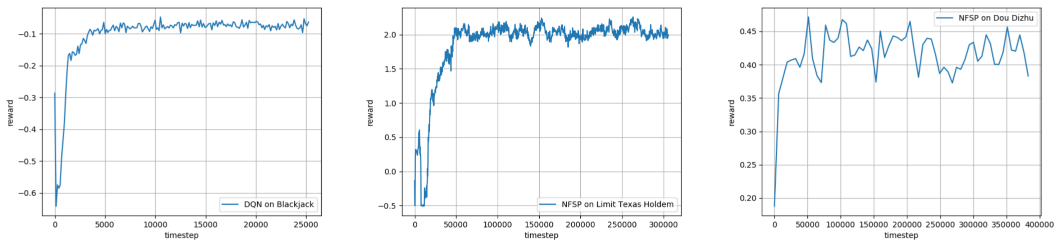


Figure 5: RLCard Typical Training Curves [3]

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 PyTorch

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.

### 3.2.1 DQN and NFSP via PyTorch with PyPokerEngine

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:

```python
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.

### 3.2.2 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.

## 3.3 Agent-Environment Integration

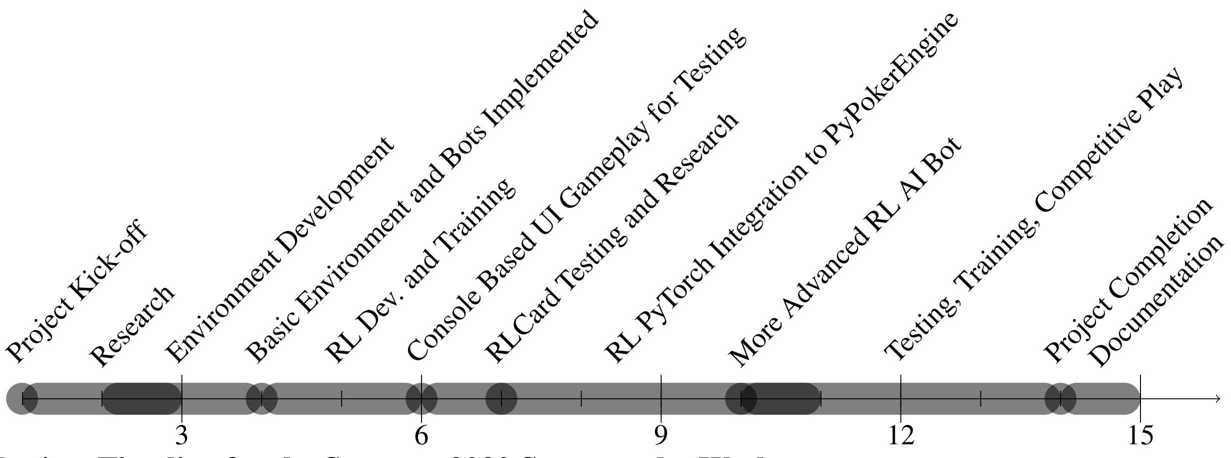
The PyTorch agent (DQN and NFSP) integration and training with the PyPokerEngine environment is the remaining development for the project. To summarize as a list of the goals for the oncoming 4 weeks;

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 Other Information

### 4.1 Timeline

A revised timeline for the project can be seen below.



**Project Timeline for the Summer 2020 Semester, by Week**

### 4.2 Meeting Schedule

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

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

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

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

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

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

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

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

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

## 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
```