מבוא לבינה מלאכותית

דו"ח הפרויקט

אבי קודריאבצב - 311910533
ולנטין וולוביק - 326814449

Preface

We implemented two reinforcement learning solutions:

1) Q-Learning with Q-Table
2) Deep Q-Learning

Both give similar results, in winning over random by small margin

Python requirements

Python version:
3.6.8
Libraries needed :
h5py, numpy, gym, tensorflow, Keras/2.1.6/, PrettyTable
All the requirements are located in setup.py and Pycharm should detect whether certain library is missing

Source code

https://github.com/firluk/Intro2AI

Q-Learning Model

Model description

The model is a classic q-table with state space and action space. Action space is simple fold/play represented as 0/1.

State space:

Every poker state can be broken down to three different categories which produce $\binom{52}{2} \cdot 2 \cdot 4 = 10608$ states:

1. Player hand, which can be any two cards from the deck.
2. Big or Small blind
3. One of four money states (given n is amount of starting chips):
    a. 0<x<½xn
    b. ½xn<x<n
    c. n<x<1.5n
    d. 1.5n<x<2n

Transition from state to state

It was decided that state dependency is too weak - since the state is defined by cards, and cards are random, thus potential reward from the next state will not be used.
The update function is:

```
new_value = (1 - alpha) · old_value + alpha · reward
```

Several attempts were made to implement:

```
new_value = (1 - alpha) · old_value + alpha · (reward + gamma · next_max)
```

But the results were worse.

Implementation

The model was implemented using the most basic modules such as random and numpy. The learning environment was inspired by openAI gym, but was developed from scratch, to suit the nature of the game (for example, step function requires both player inputs simultaneously and returns two rewards). The table itself is a numpy array, for ease of use.

Environment implementation

The environment class held all the reward data. Reward values were chosen on a basis of 0 to 100. Rewards and penalties were chosen and changed empirically. For example loss of some chips is "awarded" by 30, while loss of all chips will return a 0. There are different values to *fold*, *win some*, *win game*, *lose some*, *lose game*. Since all of them been chosen heuristicly and by trial and error there isn't much explanation behind them.

Training implementation

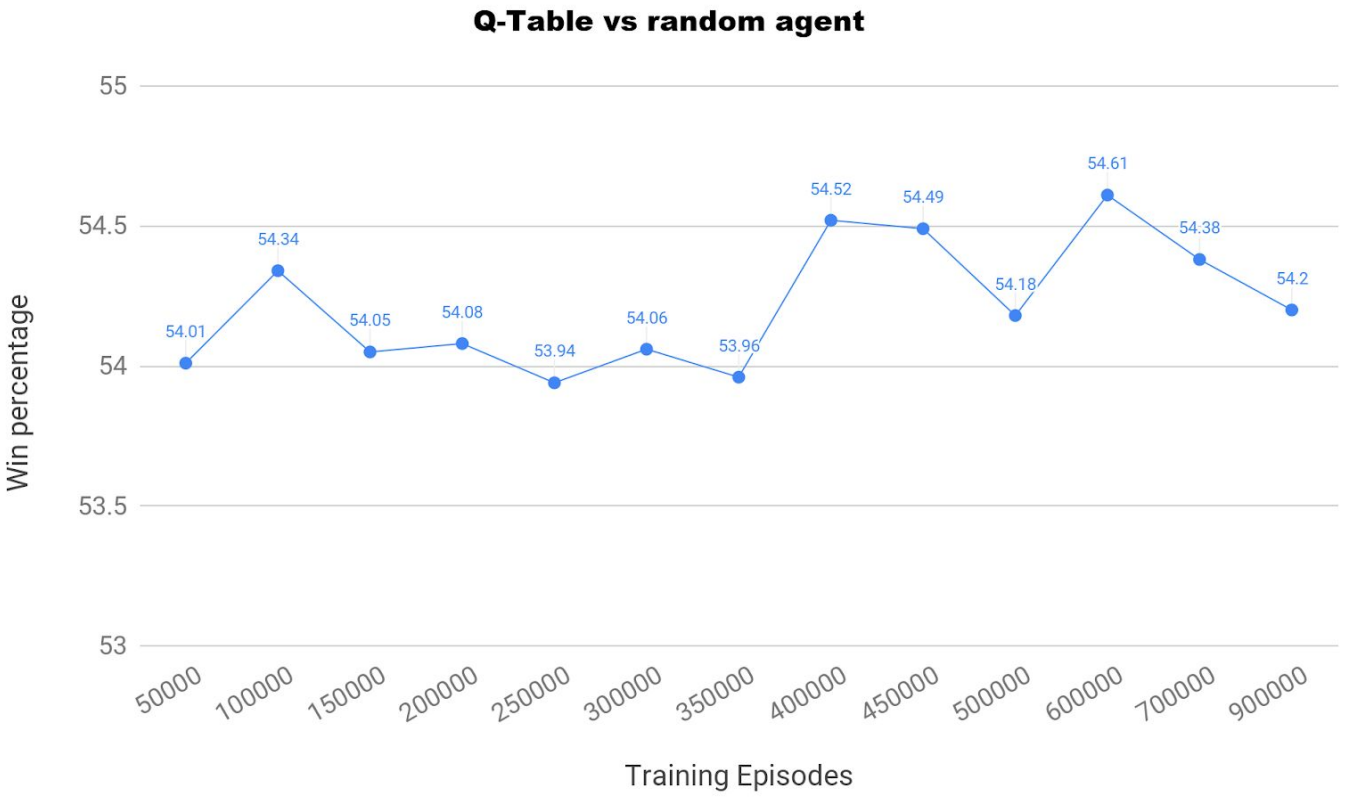
Trainer (gym) had all the data related to learn speed and amount of games.

- Alpha - 0.02 - Learn rate
- Epsilon - 0.5 - exploration vs exploitation
- Gamma - 0.1 - factor of future reward (not implemented in the final model)

The gym simulates full game experience, except that in the beginning of the game, chips are not split evenly, but randomised, to help populate the table faster. The randomization occurs only in the beginning of the game, not between the rounds.
`env = PokerEnv(self.nc, 0, True, 1)` - 1 stands for 100% chips randomization, 0.4 for 40%
Episode is one round of the game, where each players receives their cards and makes a binary decision. Then their hands are resolved, the winner receives an awards and the loser receives a penalty.
The episode ends, but the game continues to run (given both players still have chips) and agents are presented with new cards.
In the training environment one player gets to exploit his knowledge and the other is automatically all in. Both their results are incorporated into the table. Every episode they switch turns.
The average amount of penalty received by agent is: 61.15

Performance

The model was tested against random action opponent.
The model wins around 54.5% of the time.



Training

In order to execute q-table training run `.\train_table.py` in the console. In the end of the run `qtablenpc.npz` will be produced in `Qtable` directory. The readable output file will be generated as well named `Q_table_dump_end.txt`

Running

In order to test the q-table agent against the random opponent run `.\main.py q r 10000 20` in the console.
Where:

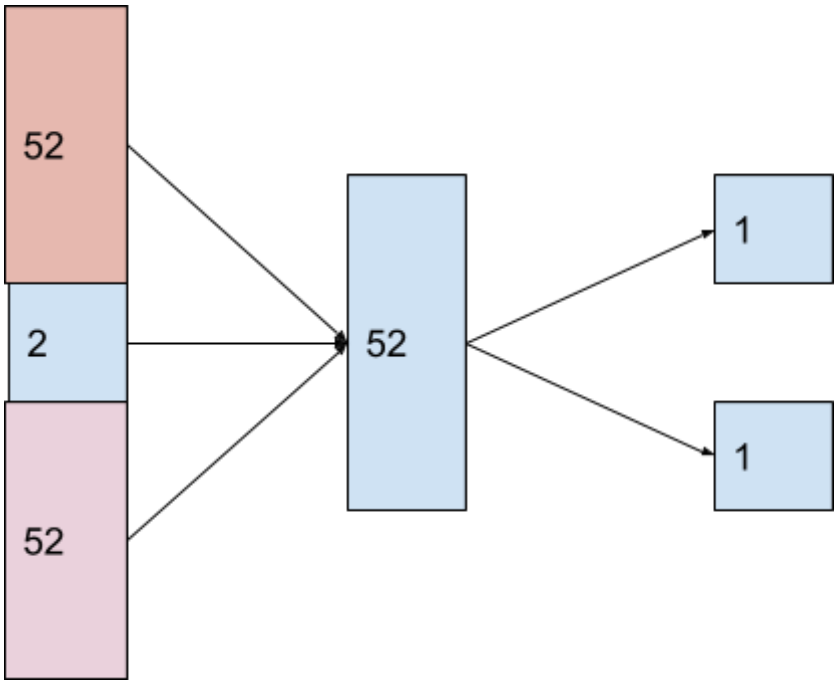
- ➢ q - is indicator for first player as q-table
- ➢ r - for second player as random
- ➢ 10000 - for number of games to be played
- ➢ 20 - number of chips that each player gets

Deep Learning Model

Model description

The model is a deep neural network build for Deep-Q reinforcement learning.

The architecture is as follows:
106 in input layer densely connected to hidden layer with 53 densely connected to output layer 2:



Input layer:
[0-51] - indicator for player's hole cards (1 - having card with corresponding code, 0 - otherwise)
[52] - indicator for small blind (1 - small blind, 0 - big blind)
[53] - relative money that player has (0.5 - player and opponent same amount, 0 - bankrupt)
[54-105] - probability of card being a community card (1 - card is community card)

Hidden layer:
[0-51] - hidden layer should apply the trained logic

Output layer:
[0] - prognosed reward for fold
[1] - prognosed reward for all-in / call
The same neural network was used for preflop and post-river.

The loss function utilized is MSE and metrics are accuracy of prediction.

The construction of neural network can be summed up by Keras Sequential API definition:

./entities/neuralnetworknpc.py

```
create_empty_model():
    ...
        model = Sequential()
        model.add(InputLayer(input_shape=(106,)))
        model.add(Dense(53))
        model.add(Dense(2))
        model.compile(loss='mse', optimizer='adam', metrics=['accuracy'])
    ...
```

Implementation

Model is implemented by using Keras on top of Tensorflow backend.
We utilized OpenAI's gym as a road-map for building training environment.
./envs/neural_net_poker_env.py/
./entities/neuralnetworktrainer.py/
Deep Q-Learning parameters:
Epsilon = 0.5 - exploration vs exploitation factor
Gamma = 0.5 - discount factor for future reward
Episode is a single round in the game (preflop or post-river),
Reward - money gain

Episode starts with money split between players in randomized fashion, with bias towards equilibrium, and hole cards are drawn from shuffled deck.
States are encoded to represent the drawn cards, small-big blind and money on players.
Both players are neural network based agents being trained simultaneously, for symmetry purposes.
Actions are chosen by both agents based on exploration vs exploitation factor *epsilon* that we defined as 0.5.

If actions include fold - the episode is over with the folding player being punished by negative reward the money lost, whereas calling rewarded with money gained.
Neural network is gonna try to fit to reward, positive or negative

```
self.model.fit(state1, target_vec1, epochs=1, verbose=0)
    if a1: # if action a1 was "call"
    self.model.fit(state2, target_vec2, epochs=1, verbose=0)
```

If both players called - community cards are drawn and episode advances to next step, the players are rewarded with "future reward" by predicting gain.

```
next_reward1[1] = self.model.predict(next_state1)[0, 1]
target1 = reward1 + gamma * next_reward1[a1]
```

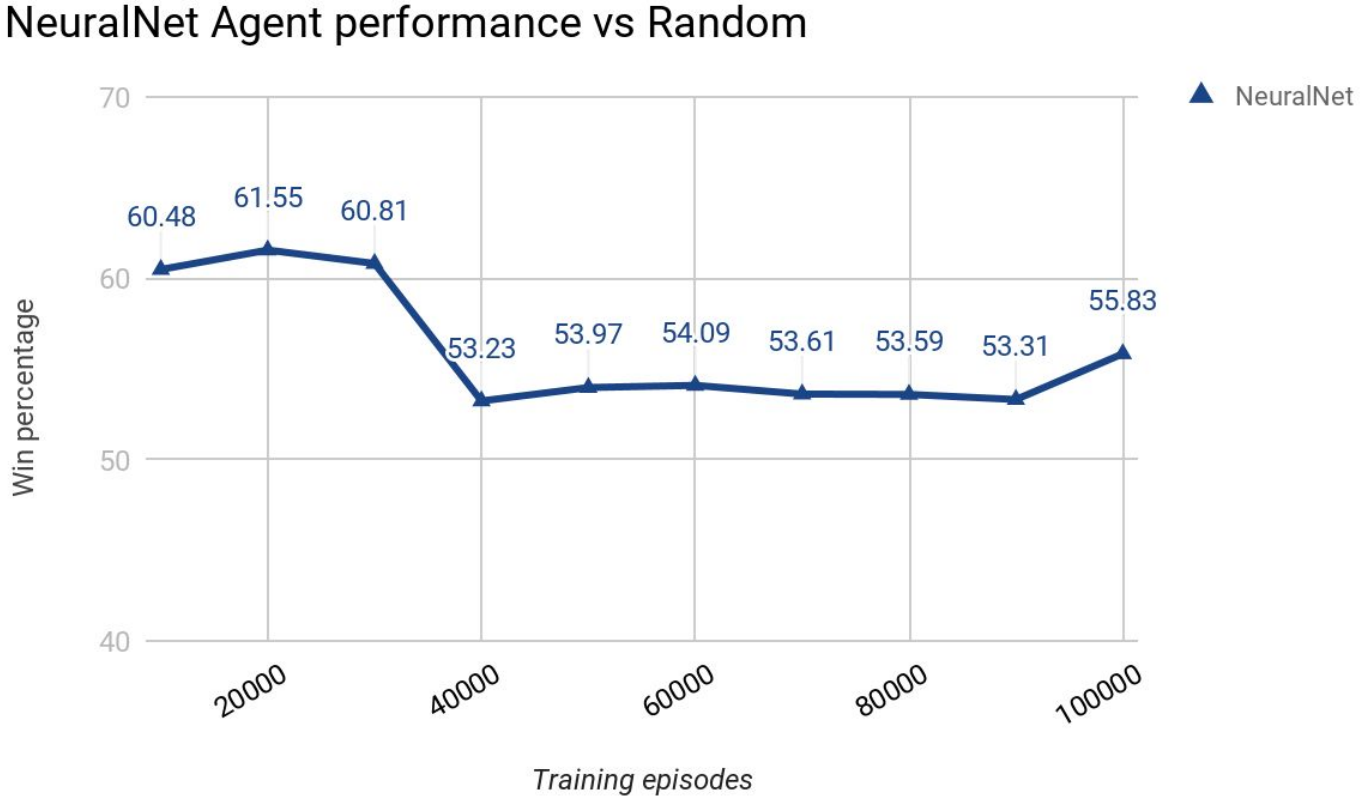
Network tries to fit to new value and proceeds to next step.
During next step the action is predefined due to simplicity of the toy-poker.
Winner is chosen by comparing both hands and rewards are as in zero-sum game either 0-0 if the neither player had better hand or sum to zero, by awarding one with money gain, and punishing another with money lost.

Both agents are trained to fit to their new value.

Episode ends, even if players have money left, and environment is reset.

Performance

When run opposite random we get the following results
[5565, 4435]
In total: Player1[NeuralNet] has won 5565
In total: Player2[Random] has won 4435
Average game length when Player1[NeuralNet] has won 4.03
Average game length when Player2[Random] has won 5.81
Successful bluffs by Player1[NeuralNet] 4341 / 7786
Successful bluffs by Player2[Random] 2210 / 4234
time elapsed: 60.56s
Average reward/punishment during training : -0.1 ~ -0.2



Training

Training the model is done by running

```
.\train_neural.py 100000 1000000 20 q_table.txt
```

[pulse] [episodes] [number of chips] [dump file for readable q_table]
This will train a neural for 1,000,000 episodes and save model every 100,000 episodes

Running

In order to test the q-table agent against the random opponent run `.\main.py n r 10000 20` in the console.
Where:

- n - is indicator for first player as neural network built from **.\NeuralNet\my_model.h5**
- r - for second player as random
- 10000 - for number of games to be played
- 20 - number of chips that each player gets