Reinforcement Learning for Turkish Card Game Called “Batak”

Metehan Cekic
Department of Electrical and Computer Engineering
University of California, Santa Barbara
metehancekic@ucsb.edu

Can Bakiskan
Department of Electrical and Computer Engineering
University of California, Santa Barbara
canbakiskan@ucsb.edu

"Batak" is a Turkish card game similar to bridge. We used reinforcement learning methods, specifically Asynchronous Advantage Actor Critic (A3C) algorithm to teach a neural network to play this game. The network learns by playing against agents that play at random. Subsequently, it surpasses the score obtained by pure random play and gets close to human level play scores.

1 Introduction

Recently, advances in reinforcement learning achieved good results on playing various games such as arcade Atari games (4), Chess, Shogi, and famously, Go (2). There have also been works where same techniques have been applied to games where there is more randomness and imperfect information such as Big 2 (1) and Hearts (6). Along the lines of such efforts, in this project we wanted to apply the technique used in (5) called "Asynchronous Advantage Actor Critic (A3C)" to teach a neural network to play the Turkish card game called "Batak". There were no open source implementations of Batak so we wrote the environment that our model interacts with from scratch. Then we trained our network using A3C and evaluated against random agents.

In Section 2, we give a detailed description of Batak. In Section 3 we outline the A2C method which is the underlying method that is extended in A3C. In section 4, A3C method is explained followed by Section 5 which describes the environment we wrote along with the details of the model. Finally in Section 6 we display the results and discuss our work and the future directions in Section 7.

2 Game Description

Batak is a card game widely played in Turkey. It is similar to Bridge, where after the deck is dealt, players bid -based on their judgment of the value of their hand- in an auction to determine the *trump* suit. In Batak, each *play* consists of 13 *tricks*. A trick is where each player plays 1 card. The play goes as follows: in the first trick, the player that won the auction starts. They play a card from a non-trump suit. Each player has to play a card from the same suit and increase the card’s value. If they can’t increase they can play a lower value card of the same suit. If they don’t have any card of that suit, they can play a card from the trump suit. If they don’t have any card from the trump suit, they can play a card from any other suit they have. In each trick afterwards, the winner of the previous trick starts and other players play according to the same rules outlined above. There are two important rules. Number one, if you have a card of the same suit as the initial suit of the trick, you have to play and you have to increase if you can. Number two, players can’t play the trump suit as the first card of a trick unless trump suit has been "unlocked" by a player in an earlier trick by him/her

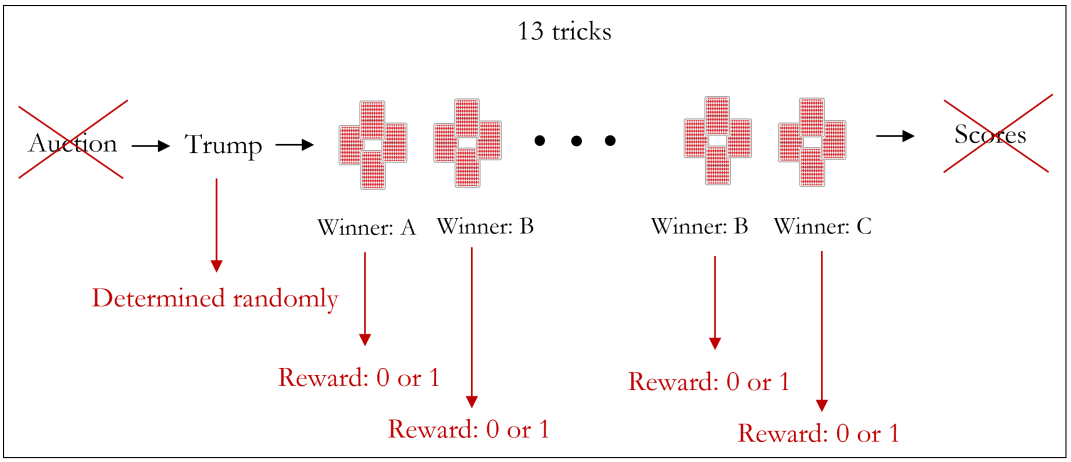


Figure 1: High level simplified game description.

not having the suit of that trick, and playing trump suit. In the end, scores are assigned by the number of tricks each player won. If the winner of the auction scored less than their bid, they are assigned negative of the bid as the score. So the advantage of winning the auction is being able to determine the trump suit, the disadvantage is the high negative score in the case of failure to win tricks as many as the bid. In a game, there are usually 10 plays, the goal of the game is to get the highest score.

Because of time limitations we decided to make both our implementation and the learning easier by getting rid of the auction process altogether. In our implementation, the trump suit is determined at random, and scores are assigned as 0 and 1 to each player according to who won that trick. (Rather than at the end of a play.)

3 Advantage Actor-Critic Algorithm (A2C)

In this section, we give motivation to use A2C algorithm and explain how it works. The main property of the A2C algorithm is that it not only optimizes a policy but also trains a value function for the current state to assess how good is that policy for that specific state. In general, policy-based methods and value-based methods have a number of upsides and downsides. A2C algorithm is designed to combine the benefits of both value based and policy based approaches.

Value-based algorithms require a value function to be calculated for every possible action, although it is more sample efficient. On the other hand, policy-based methods can easily utilized on continuous action spaces due to its direct optimization of policy. However, low sample efficiency causes them to converge slower.

A2C algorithm estimates both policy and value function via deep neural network. Value function corresponds to the worth of that state to be in and the policy gives the probabilities of each available actions for that state.

$$\pi(a|s;\theta) : \text{the probability of action } a \text{ given the state } s$$

$$V_\pi(s) : \text{How good is it to be in state } s$$

The value function is mathematically defined as expected reward of a trajectory after the state that agent is at that time in, which is only dependent on s.

$$V_{\pi_\theta}(s) = \mathop{\mathbb{E}}_{\tau}(R_\tau|s_0 = s, \pi_\theta)$$

Where τ represents the trajectory until game ends. Hereafter, for simplicity we drop θ from π, which means π is parameterized by θ. Critic of the agent evaluates the policies by computing the advantage function of specific action in a given state, which is defined as follow:

$$A_\pi(s,a) = Q_\pi(s,a) - V_\pi(s) : \text{The advantage of making action } a \text{ in state } s$$

Where Q function is another value function which depends on both state and action, particularly it corresponds to the value of an action a at a state s. Observe that with given definitions:

$$V_\pi(s) = \sum_{a \in A} \pi(a|s) Q_\pi(s,a)$$

The algorithm utilizes the fact that there is a trajectory obtained from a game play (An agent interacted with the environment with a given policy function) by computing Q values from empirical discounted reward:

$$Q_\pi(s,a) = R = \sum_{i=0} r_{t_i} \gamma^i$$

where γ corresponds to the how important are the future rewards compared to the current reward (1 means equally important, whereas 0.5 means each time step decreases the value of future reward by half).

As we have 2 different outputs for our deep neural network, namely policy and value functions, we have two different loss functions. The value loss is defined as mean squared error (MSE) between empirical discounted reward and the value output. This loss motivates neural net to learn to assess policy better after each iteration.

$$Loss_{value} = \sum (R - V_\pi(s;\theta_v))^2$$

Similarly, we want to increase the probability of an action which has positive advantage to increase empirical reward, therefore policy loss is defined by:

$$Loss_{policy} = \log \frac{1}{\pi(a|s;\theta_p)} * A_\pi(s,a;\theta_v) - \beta H(\pi)$$

The loss function includes the last term to increase entropy of the action distribution, in other words, we push neural network to explore more. Typical β value is 0.001, it is a good hyperparamemeter to adjust between exploration and exploitation which is a dilemma we always encounter in reinforcement learning algorithms. More inclination towards higher β values causes neural net to explore more, whereas low β values stick with the locally optimum policy.

4 Asynchronous Advantage Actor-Critic Algorithm (A3C)

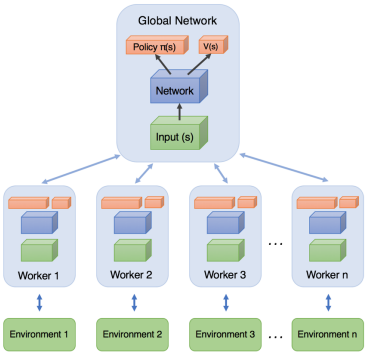
Asynchronous gradient descent is proposed by (5), which boosts the performance of traditional reinforcement algorithms like A2C and Q learning. The main idea of framework is having a master neural network and updating it with the gradients computed from some other worker networks.

In standard reinforcement algorithms we usually use memory mechanism to feed earlier experience of the agent when updating the neural network. We call this concept as "experience replay", which is helpful to stabilize learning. Running parallel agents independently stabilizes learning without "experience replay" because of the stochasticity in the game each agent experiences totally different action-reward trajectory. Moreover, as all agents play the game in parallel, training time is reduced roughly linear in the number of parallel actors.

4.1 Algorithm

The main algorithm is given in Figure 2. The following seven steps explain the algorithm verbally in simple fashion.

1) Play the game in each worker environment with current policies



Algorithm S3 Asynchronous advantage actor-critic - pseudocode for each actor-learner thread.

// Assume global shared parameter vectors θ and θ_v and global shared counter $T = 0$
// Assume thread-specific parameter vectors θ' and θ'_v
Initialize thread step counter $t \leftarrow 1$
repeat
  Reset gradients: $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$.
  Synchronize thread-specific parameters $\theta' = \theta$ and $\theta'_v = \theta_v$
  $t_{start} = t$
  Get state s_t
  repeat
    Perform a_t according to policy $\pi(a_t|s_t;\theta')$
    Receive reward r_t and new state s_{t+1}
    $t \leftarrow t + 1$
    $T \leftarrow T + 1$
  until terminal s_t **or** $t - t_{start} == t_{max}$
  $R = \begin{cases} 0 & \text{for terminal } s_t \\ V(s_t, \theta'_v) & \text{for non-terminal } s_t \text{// Bootstrap from last state} \end{cases}$
  for $i \in \{t-1, \ldots, t_{start}\}$ **do**
    $R \leftarrow r_i + \gamma R$
    Accumulate gradients wrt θ': $d\theta \leftarrow d\theta + \nabla_{\theta'} \log \pi(a_i|s_i;\theta')(R - V(s_i;\theta'_v))$
    Accumulate gradients wrt θ'_v: $d\theta_v \leftarrow d\theta_v + \partial (R - V(s_i;\theta'_v))^2 / \partial \theta'_v$
  end for
  Perform asynchronous update of θ using $d\theta$ and of θ_v using $d\theta_v$.
until $T > T_{max}$

Figure 2: Left: Overview of master-worker architecture, Right: Detailed description of A3C algorithm (5)

2) Compute gradient descent of losses for each worker neural net at the end of game.

3) Accumulate gradient descent from each worker neural network.

4) Update Master neural network according to accumulated updates

5) Update all worker networks with master network

6) Calculate expected reward.

7) Go back to step 1 until convergence

5 Model and Environment

We decided that in order to capture the sequential information within the game we should either keep a list of all played cards explicitly in some form of memory, or we should use an RNN that would learn to extract the underlying temporal information implicitly. We decided to go with the second option as the first option requires much bigger input sizes which in turn requires bigger number of neurons in the model. With this decision, we determined the essential states of the game that a human player uses when playing and made these the inputs to our model. They are: trump suit (One-hot vector of length 4), hand (Boolean vector of length 52) and current and past trick's cards up to our model's turn, along with who played them.(Boolean matrix of size 7 by 56) Cards are represented by one-hot vector of length 52, players are represented by one-hot vector of length 4, then these vectors are concatenated to form vector of size 56.

Since the inputs have fairly simple structures we chose to use dense layers to process them. The detailed structure of the network is as follows: current and past trick's cards input go into an LSTM (3) with size ...

In Figure 3, overall architecture of our model is displayed, with the LSTM unrolled in time to give a better sense of time dependency.

6 Results

We trained 3 different architectures with Nvidia GeForce GTX 1080 Ti. To be able to evaluate trained agent, we get a couple of people (Who knows and plays the game very well) play the game against random-playing bots similar to trained agent.

Results are given in Figure 4. As can be seen from the figure, all architectures we experimented performs similarly and much better than random playing games. However, all architectures perform slightly worse than humans.

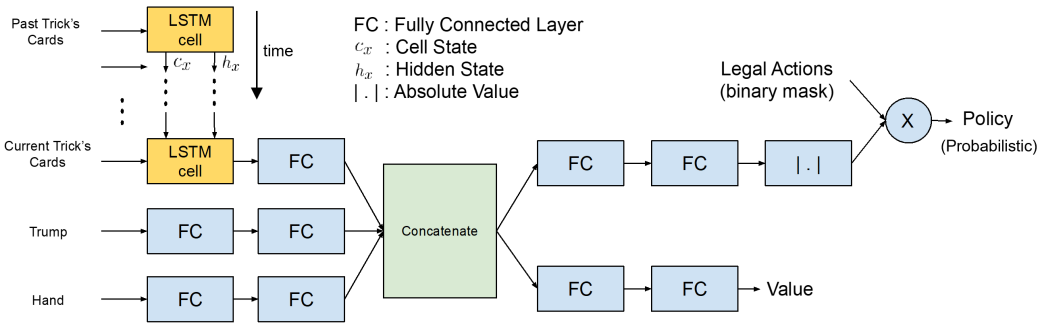


Figure 3: Model used for calculating value and policy.

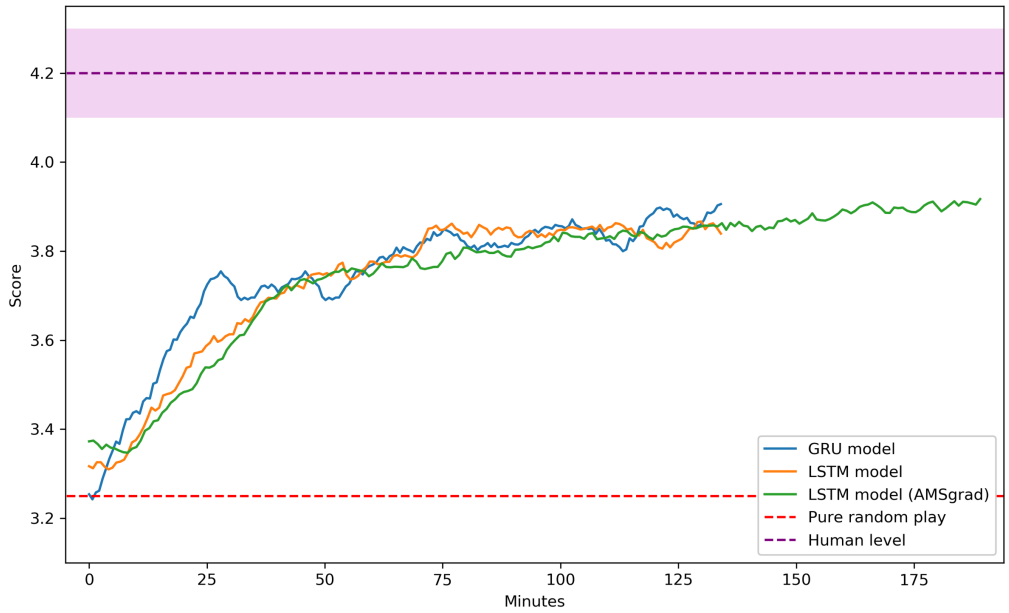


Figure 4: Results of training. Score is the average value of tricks won by the master model against other random agents.

7 Conclusion

In this project, we wanted to implement and evaluate the performance of A3C algorithm on the Turkish game called "Batak". We also wanted to observe the effect of architecture and optimization method for convergence. As can be seen from the results section, the architectures we tried so far perform more or less same.

We trained agent in the game without auction, next step for this project would be to improve bots to be able play with auction (learn and bid accordingly). Moreover, we plan to develop agent by making it play against itself (Instead of random bots).

References

[1] Charlesworth, H. Application of self-play reinforcement learning to a four-player game of imperfect information. 2018.

[2] David Silver*, Julian Schrittwieser*, K. S. I. A. A. H. A. G. T. H. L. B. M. L. A. B. Y. C. T. L. F. H. L. S. G. v. d. D. T. G. D. H. Mastering the game of go without human knowledge. *Nature*, 2017.

[3] Sepp Hochreiter, J. S. Long short-term memory. *Neural Computation*, 1997.

[4] Volodymyr Mnih, Koray Kavukcuoglu, D. S. A. G. I. A. D. W. M. R. Playing atari with deep reinforcement learning. 2015.

[5] Volodymyr Mnih, Adrià Puigdomènech Badia, M. M. A. G. T. P. L. T. H. D. S. K. K. Asynchronous methods for deep reinforcement learning. *ICML 2016*, 2016.

[6] Wagenaar, M. Learning to play the game of hearts using reinforcement learning and a multi-layer perceptron. 2017.