Comparison of deep learning reinforcement methods when learning rules in a custom environment

Charlie Gaynor

CharlieJackGaynor@gmail.com

(linkedin.com/in/CharlieGaynor, github.com/CharlieGaynor)

November 4, 2021

1 Introduction

Trumps is a popular card game, played with a standard pack of 52 cards. There are some very interesting variants of Trumps, one of which (boerenbridge) was the inspiration for this project. After being defeated by my girlfriend numerous times in a row at boerenbridge, I wanted to create an A.I that could lead me to victory (totally different from cheating). A step back from that is teaching an A.I the rules of the game, which is the focus of this project. More accuartely, the focus is to compare how well different reinforcement learning methods learn the rules in varying, albeit very similar, condiitons.

We will use 4 different methods:

- ♠ Epsilon Greedy
- ♠ Epislon Greedy with replay / offline learning
- ♠ Policy Gradients
- ♠ Actor Critic

Since this project utilises reinforcement learning, our Reinforcement Learning Bot (RL-Bot) will play against a built-in random A.I and will attempt to learn from its mistakes

2 Game description

For this project, we will be focusing on two players only, and the most basic variant of the game (other variants are more fun, I promise).

2.1 Set up

- 2 players (up to 7 in reality)
- Each player is dealt up to 7 cards from the deck
- The next card is turned faced up, which is the ‘trump suit’

2.2 Game Play

The first player to act (the youngest usually, at least that is what my younger-than-me girlfriend tells me…) plays a card from their hand into the ‘pile’. The next player **must** play a card of the same suit if they have one, else they can play any card. Once the cards are all played, the winner of that round is decided in this order:

- If there are any trump cards, the largest trump wins
- Otherwise, the largest card of the original suit (the suit which the 1st player played) wins

The winner of the previous round then plays first in the next round. Continue this until there are no cards left. The person with the most rounds won at the end is the winner!

3 Enviroment

I made a custom environment (essentially a playground for our RLBot) using python (see my GitHub at the top of the page for more info). The environment includes a random AI opponent (RAI), who randomly plays allowed cards at each step (shocker!). At every point where RLBot can play a card the environment returns: state, reward, and a flag for if the game is done

3.1 State

The state comprises of the 'observations' needed for RLBot to assess what is happening in the game. In this project, we are just focused on teaching the rules, so after some experimentation, I settled on returning the following 3 features.
The state is encoded as a simple numpy array of length 105 (one of the biggest rooms for improvement I believe, see end of section 7).

3.2 Action

The possible actions are simply integers in the range [0,51], representing the cards of the deck. The suit of a card would be the integer // 13 (floor division), and its value would be the integer modulo 13. For example, card 20 would be of suit ‘1’ and value 7.
Thus, the challenge our bot has here is picking cards that are actually in their hand, and with the right suit – that is the right value // 13.

Table 1: State features

Feature	**Description**
One-hot encoded list of cards in RLBot's hand	52 numbers, each of which can be 1 or 0, depicting whether not that card is in the agent's hand
One-hot encoded list of cards in the 'pile'	52 numbers, each of which can be 1 or 0, depicting whether not that card is in the pile
Trump	Integer [0,1,2,3] representing the 'trump' suit.

3.3 Reward

If our RLBot makes a mistake, they immediately get a reward of -1 (or -5 for 5 cards), and the game is over. Otherwise, they get +1 for every round they win, and 0 for every round they lose.

4 Training schedule

To train and evaluate the performance of our bots, I will use 3 seperate tests. These are:

- ♠ Trumps with 1 card, our bot always plays first, and only the first feature
- ♠ Trumps with 2 cards, our bot always plays first, and only the first feature
- ♠ Trumps with 5 cards, the first player alternates, and all 3 features

By 'always plays first', I mean our bot will be the first to play a card at the start of each game, i.e. when both players have a full hand.

5 Network

The number of possible states for 5 cards is

$$\sum_{i=1}^{5} \binom{52}{i} \times \left(\binom{52}{1} + \binom{52}{0} \right) \times (4) \approx 10^8$$

This is ignoring symmetry arguments (trump could be fixed as suit '0', then 3 suits are identical etc), but still, the state space is just far too large to learn the value of every state * action, and store it in a table like the traditional tabular approach for Q learning. The tabular approach also fails as the value for an action, given the state, is not static since it depends on what card our opponent has, which is random.

So, we can't just update a table, what next? Our best bet is to use a model-free approach, namely create a neural network (NN). The NN will take in a bunch of integers for the input space, perform some calculations in the hidden layer, and spit out values for all actions. Now defining the best neural network is a very difficult job, and I did not want this to be the focus of this project. Therefore I define a simple network, with just one hidden layer of size 10 * observation space (52 or 105). This will be a limiting factor in the models learning, but one I am happy with for now. I also decided to use the ELU (Exponential Linear Unit) activation function, as this had more success than other popular alternatives such as ReLU.

6 Exploration methods

How should we best select actions when training? If we only ever stick to what we know (exploitation), instead of trying new things (exploration), we may never discover the best actions.

To give a comparison; imagine you move to a new city, and only recognise one restaurant. If you just choose your current favourite when going out to eat, you would only ever eat at the same place, and never try the others, potentially missing a wealth of better experiences. This is relevant in many walks of life (e.g. choosing a career, hairdressers, which way to cook an egg etc), but the importance increases with the timescale of the problem. For example, if you can only ever go to a restaurant once again for the rest of your life, you should probably exploit, but if you have 1000 visits left, get exploring!

To combat this, we must have some way of exploring the unknown, so we can (hopefully) discover all the best moves. Each model below will calculate and output 'Q values' for each action (given the state), using the architecture described in section 5. Then each model will select moves depending on these Q values, albeit in a slightly different way.

The Q values will all be updated by first defining a loss function and then by using Stochastic Gradient Descent (SGD) to take a step in the direction of minimal loss. Each method defines a different loss function, which gives rise to different Q values.

6.1 Epsilon Greedy

Epsilon greedy is perhaps the most popular and simple method of exploration. For each game RLBot plays we have a parameter, epsilon, which represents the probability we will randomly pick an action regardless of the Q values. This ensures we try new things (exploring) instead of just doing what we already know (exploiting). As training goes on we reduce epsilon, to practice more on the better moves.

To be precise:

♠ With probability ϵ pick a random action

♠ Else with probability $(1 - \epsilon)$ pick the action with the highest Q value

The loss function for Epsilon Greedy is like so:

$$L(\theta_i) = \mathbb{E}_{s,a,r,s'}[(y_i - Q(s,a;\theta_i))^2]; \text{where } y_i = r + \gamma \cdot max_{a'} Q(s', a'; \theta_{i-1})$$

Here, θ are the network parameters, s are the states, a actions, r rewards, and γ is known as the discount rate. Intuitively, we are trying to match the Q values with; The reward received + the value of the next state (* discount rate). Applying recursively we can see the value of the next state is intrinsically linked to the expected reward.

6.2 Policy Gradients

We no longer pick a parameter epsilon. Now instead we apply a softmax function to all the Q values to obtain probabilities;

$$\mathbb{P}(a_i) = \frac{exp(Q_{a_i})}{\sum_j exp(Q_{a_j})}$$

We then sample an action using the probabilities and update the network parameters according to the Policy Gradient loss function. For a given game we have:

$$L(\theta_i) = -\sum_k \log \mathbb{P}(a_k|s_k, \theta_i) * \text{Discounted Rewards}_k - \text{entropy}$$

Where Discounted rewards (D.R) is calculated like so:

$$D.R_k = \sum_{t=k}^{len(\text{R})} R^t \gamma^{t-k} \text{ ; and R is an array of rewards recieved at each step}$$

The entropy term favors spread-out, symmetric probabilities, as opposed to dense asymmetric (e.g. all probabilities being the same vs only one action having any probability density). This term encourages more exploration as opposed to exploitation. This is slightly counter-productive for this training, but I wanted to leave it in as it's more representative of how the models would be trained on more complex tasks.
Eventually, the impossible moves should have probability ~ 0, and so are never picked anyway.

6.3 Actor Critic

Actor Critic is essentially another layer on top of the Policy Gradient. For every state we also output a single value (SV), as well as Q values for each action. These SV's essentially describe how 'good' our current position is, and are derived from the same network, but with a single output neuron.
Therefore it is rather intuitive how we train this network. We have:

$$L = \text{Policy Gradient Loss} + \sum_k (D.R_k - SV_k)^2$$

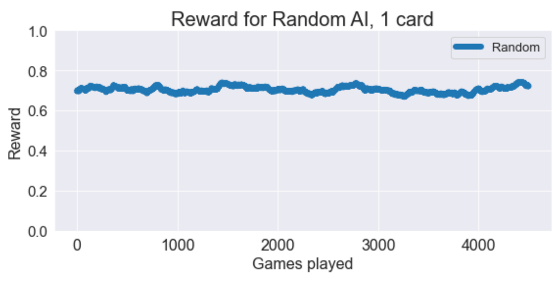
The second term (value term) tells us we want our SV's to match the D.R's - which is great since these give a great representation of the strength of our position (when accurate)

7 Results and Discussion

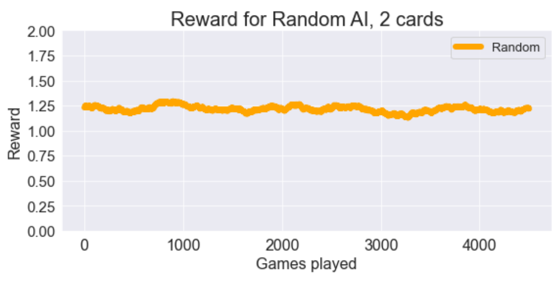
For the epsilon greedy method, I trained two different bots - one with replay / offline learning (i.e plays a bunch of games first and then trains on a random subset) and one without. The differences between them are minimal, but indicate that, with proper vectorisation, training would be much faster on the former. For these methods, I have chosen to plot the 'Evaluation reward', which was calculated by pitching RLBot with epsilon (exploration rate) = 0 vs RAI, at various points in the training.
All Training was done on my local PC (mid-range laptop, nothing impressive). 20,000 games takes approximately 1 minute, which is not terribly long, but could definitely be improved upon.

7.1 Random AI

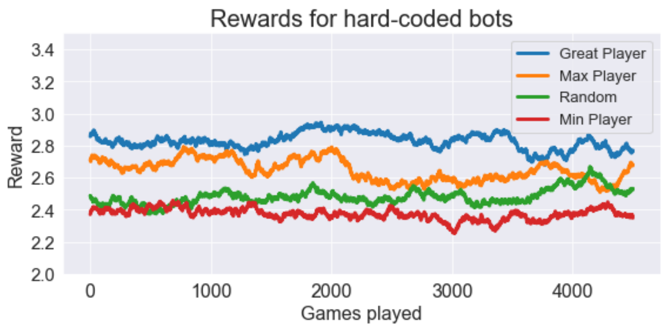
To be able to compare the performance of our trained RLBots, I have gathered data of random bots, who knows the rules, playing in place of the trained bots. The trained bots should aim to be on par with or beat the respective random bots, to confirm they have successfully learnt the rules.



(a) Reward for Random 1st player with 1 card



(b) Reward for Random 1st player with 2 cards



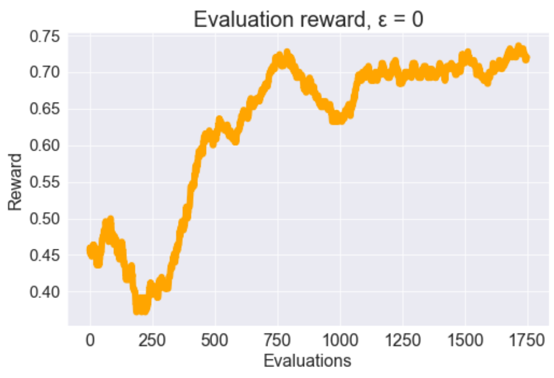
(c) Reward for different hard-coded bots with 5 cards

For the 1 card game, we see that the first player wins much more often than the second, with a win rate of approximately 69.5%. With 2 cards, the first player has an average reward of 1.21, compared to the maximum reward of 2 - this is approximately 60% of rounds won.

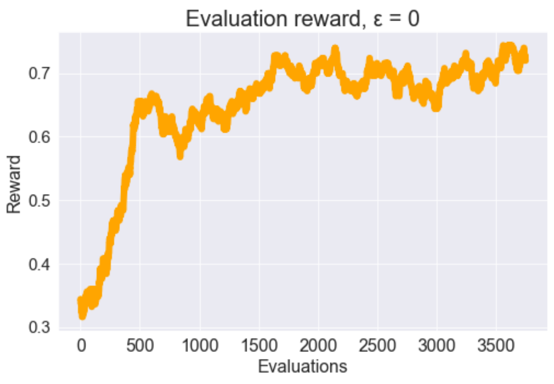
For 5 cards, the game is a little more complicated as now our bots play both first and second at the start of the game, alternating each time. To prove our models can learn to beat a random A.I, I have created and plotted the success of hard-coded A.I's with some basic logic. Our best bot obtains roughly 60% of the score, our worst 48%, and the random A.I 50% as expected.

7.2 One card

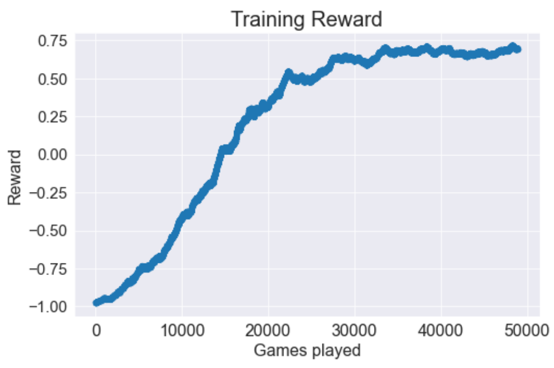
First, we train all methods on the extremely simple game of just playing the card in their hand. All bots handle this well, and quickly. From the previous section, we can see we are aiming for a reward of just under 0.75.



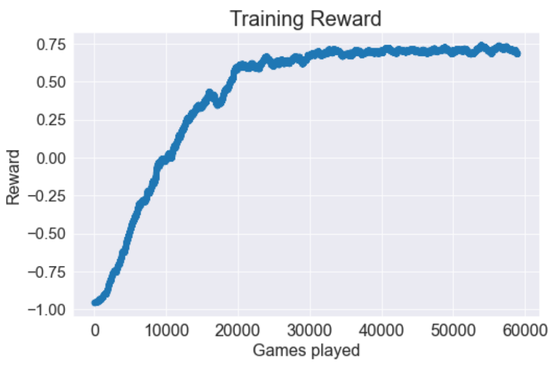
(a) Epsilon Greedy 1, not making mistakes



(b) Epsilon Greedy 2, not making mistakes



(c) Policy Gradient, not making mistakes

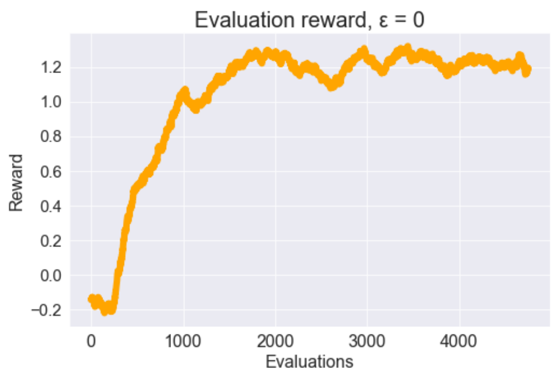


(d) Actor Critic, Still making mistakes

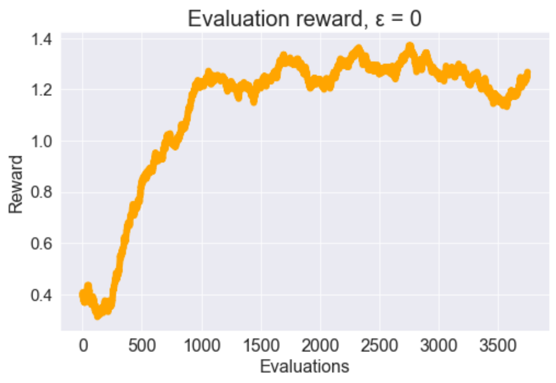
Interestingly, the Actor Critic bot is still making mistakes, even after training for 50,000 games - whereas the epsilon greedy bot makes no mistakes after just 1750 evaluations, which is $\sim$ 20,000 training games (see notebook for more info).
To try and combat this we could either train for a longer time or exclude actions with probability $< \delta$, where delta is in [0.1,0.01,0.001] etc.

7.3 Two cards

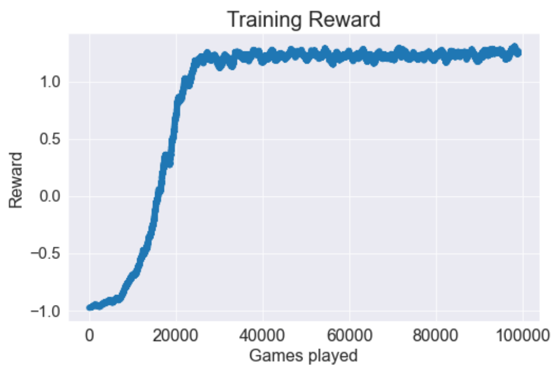
For two cards, the 'pile' is still irrelevant, and so observation space is still just the 52 cards. This is a tiny bit more of a challenge than just 1 card, but not really - as evidenced by the performance of the bots.



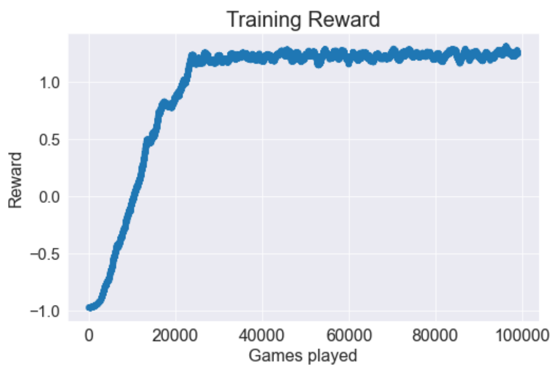
(a) Epsilon Greedy 1, not making mistakes



(b) Epsilon Greedy 2, not making mistakes



(c) Policy Gradient, still making mistakes



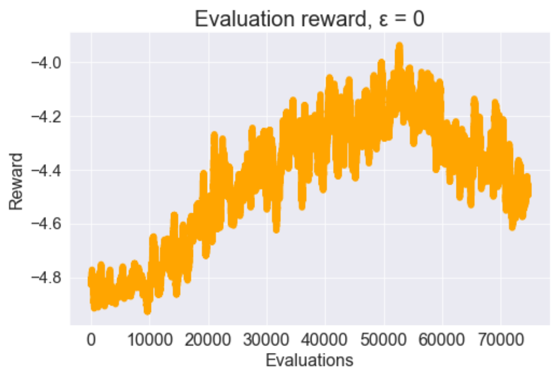
(d) Actor Critic, Still making mistakes

Both the Policy Gradient and Actor Critic models still are making mistakes, but with a nearly perfect average score. All models are learning very quickly what to do, though still epsilon greedy is outperforming.

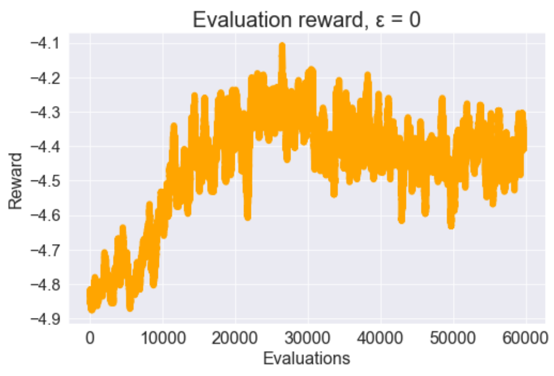
7.4 Five cards

Here is where we see the vast majority of the difference. Now our RLBot has 5 cards in their hand and will need to intelligently factor in the 'pile' to be able to make correct moves. The trump is also fed to the bots in the observations now, which gives the size of the observation space to be 52+52+1 = 105 integers. I now trained for 750,000 games, which took about 1 hr for Epsilon Greedy + Policy Gradient methods. Actor Critic took about 3.5 hours (!), mainly because this method was actually learning to play the game and so took much actions in each game, hence more rounds, hence longer game time.

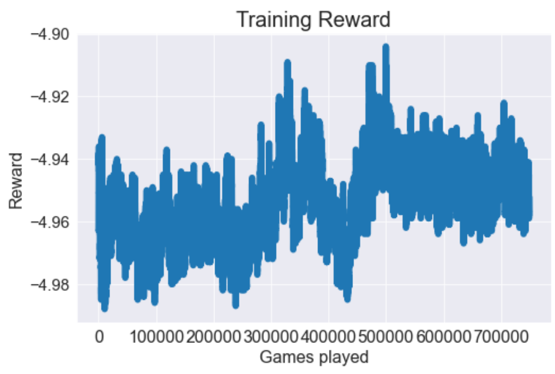
The rewards for making a mistake are now -5, to try and help the bots learn more quickly what to do with all the cards.



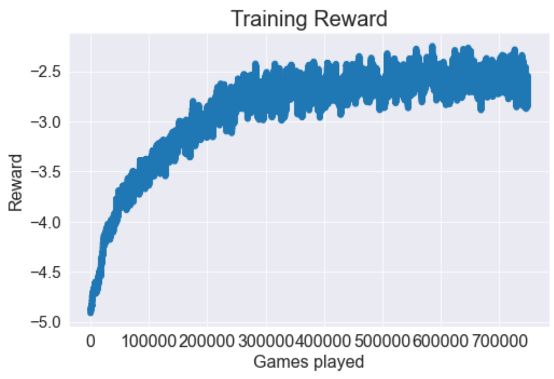
(a) Epsilon Greedy 1



(b) Epsilon Greedy 2



(c) Policy Gradients



(d) Actor Critic, makes real progress!

Epsilon Greedy makes a valiant effort, but no real progress. Policy Gradient is completely lost and doesn't know where to start. Actor Critic makes great progress, yet still makes mistakes constantly.

8 Conclusion

While Epsilon Greedy functions well for relatively easy tasks, it is vastly outperformed by Actor Critic for more complicated tasks. Using offline learning slightly enhances training performance, and also has the potential to allow vectorisation of games, which would massively speed up training (factor of about 10x I'd estimate).

Perhaps with more advanced architecture the Actor Critic model would be able to learn the rules of the game, and even outperform the random AI. I think the biggest improvements to be made probably lie within how we communicate the observations. In a similar fashion to RGB for images, passing observations in stacks and then using Convolutional Neural Networks would be my next line of investigation. We would, however, also want our RLBot to be able to recall what happened during the game (e.g. was our opponent forced to play spades because they had no hearts?). This screams out that LTSM/RNN models should be trialed.

Thanks for reading! If you have any suggestions for improvemens / thoughts / questions / anything else, please contact me on linkedin or email.