Application of self-play deep reinforcement learning to “Big 2”, a four-player game of imperfect information

Henry Charlesworth
Centre for Complexity Science
University of Warwick, Coventry
United Kingdom

Abstract

We introduce a new virtual environment for simulating a card game known as “Big 2”. This is a four-player game of imperfect information where players aim to play all of their cards as quickly as possible. The novelty of the game compared to other card games like Poker is mainly in its complicated action space, where combinations of up to 5 cards a time can be chosen from an initial hand of 13 cards. This makes it challenging for many commonly used reinforcement learning algorithms, and we suggest that it could be a useful testbed for testing new multi-agent learning algorithms. We then use the recently proposed “Proximal Policy Optimization” algorithm(Schulman et al. 2017) to train a neural network to play the game, purely learning via self-play, and find that it is able to reach a level that outperforms a number of amateur human players after only a relatively short amount of training time.

Introduction

Big 2 is a four player card game of Chinese origin, played widely throughout East and South East Asia (it also commonly goes by the names “Big Deuce” and “Deuces”, amongst others). There are many regional variations in the rules, however the basic idea involves a standard deck of 52 playing cards being shuffled and dealt out to four players, such that each player starts with 13 cards. Players then take it in turns to either play a hand or pass, with basic aim of being the first player to be able to discard all of their cards (see section 2 for more details about the specific rules). In this work, we introduce a virtual environment to simulate the game which is ideal for the application of multi-agent reinforcement learning algorithms. We then go on to use this to train a deep neural network to learn how to play, purely using self-play reinforcement learning.

In general, multi-agent environments pose an interesting challenge for reinforcement learning algorithms, and many of the techniques which work well for single-agent environments cannot be readily adapted to the multi-agent domain(Lowe et al. 2017). Approaches such as Deep Q-Networks(Mnih et al. 2015) struggle because multi-agent environments are inherently non-stationary, since the other agents in the environment are themselves improving with time. This prevents the straightforward use of experience replay which is necessary to stabilize the algorithm. Standard policy gradient methods also struggle due to the large variances in gradient estimates that arise in the multi-agent setting, which often increase exponentially with the number of agents. Although other methods are able to partially overcome these issues (for example Proximal Policy Optimization(Schulman et al. 2017), which we use in this work), there is still an active research effort to improve the state of the art multi-agent reinforcement learning algorithms. One necessary component of this is having a good range of different, challenging environments to test new algorithms out on. Whilst there already exist a number of environments that are useful for this purpose, such as the OpenAI competitive environments (Bansal et al. 2018) and the Unity(**?**) platform, we believe that Big 2 could be a useful addition to these since it is relatively accessible whilst still requiring complex strategies to play well. In particular, there are a number of reasons why this is an interesting environment to study. The first of these is that it is a game of imperfect information, since each individual player is unaware of what cards their opponents hold and so do not have access to a full description of the current game state. A second reason is that it is a four-player game. To date, a majority of the most remarkable successes that have arisen from self-play deep reinforcement learning such as AlphaGo(Silver et al. 2016) and AlphaZero(Silver et al. 2017) have been confined to two-player games of perfect information, e.g. Chess, Go and Shogi. Whilst there have also been significant successes in Poker playing programs such as Libratus(Brown and Sandholm 2017) and DeepStack(Moravčík et al. 2017) these have largely been confined to the heads-up (i.e. two-player) versions of the game, and require much more computational power to make a decision compared to the approach we take in this work — for example DeepStack uses a heuristic search method adapted to imperfect information games, whereas we only use the trained neural network. In addition to this, the action space of Big 2 is much more complicated than in Poker, with up to 1695 actions available in a given state. Another approach which warrants a mention and does directly apply self-play deep reinforcement learning is “neural fictitious self-play”(Heinrich and Silver 2016). Here an attempt is made to learn a strategy which approximates a

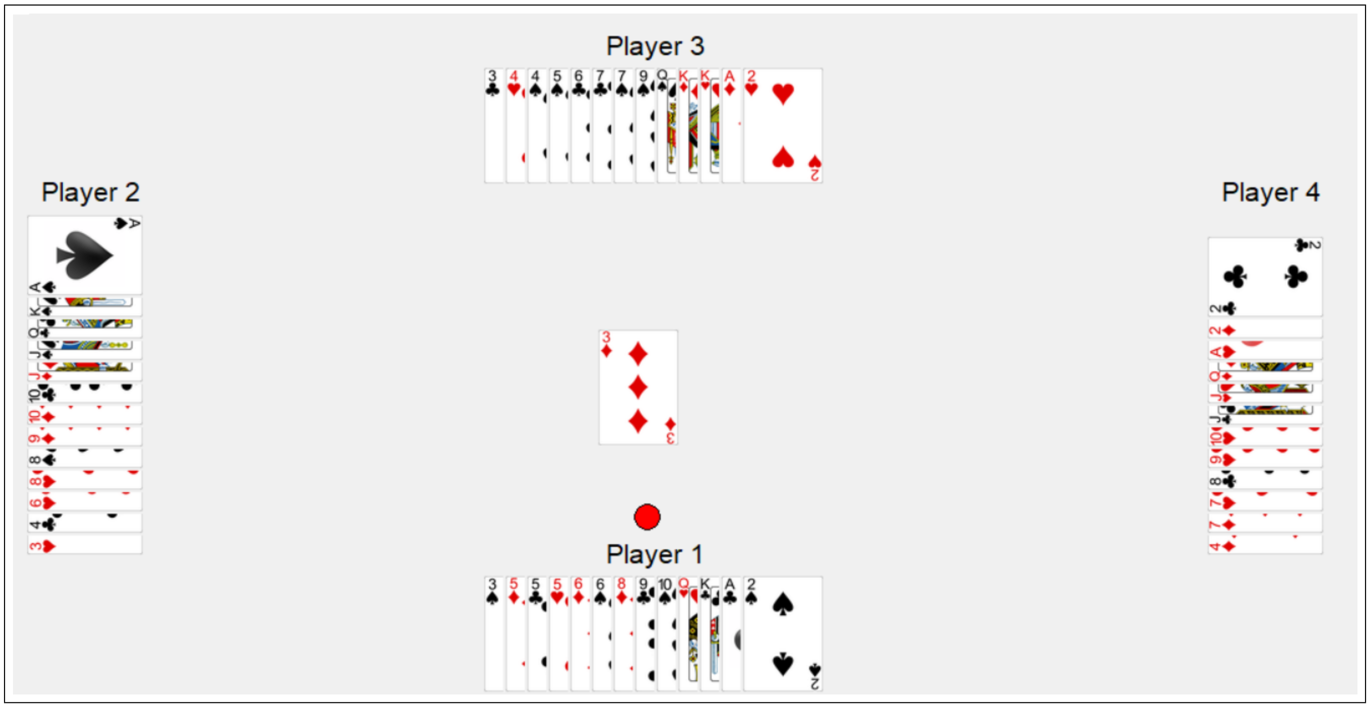

Figure 1: A typical start to a game (although note that players are not aware of the cards held by the other players). All 52 cards are dealt out so that each player begins with 13 cards. The player with the 3 of diamonds (here player 4) must start, and plays this as a single card hand. Subsequent players must play a higher single card or pass (skip their go). This continues until everyone passes, at which point the last player who played a card gains "control". A player with control can choose to play any valid 1,2,3,4 or 5 card hand (see text for details). Subsequent players must then play a better hand of the *same number of cards* or pass, until someone new gains control. This continues until one player has managed to play all of their cards.

Nash equilibrium, however it has only been applied to simple games with no more than two players. As such, it would most probably struggle to learn to play Big 2 well.

Rules and basic strategy

In this section we give a detailed description of the rules of the game (or rather, the rules of the particular variation which we are studying), as well as some brief comments on the basic strategy. At the start of each game a standard deck of playing cards (excluding jokers) is dealt out randomly, such that each of the four players starts with 13 cards. The "value" of each card is ordered primarily by number, with 3 being the lowest and 2 being the highest (hence Big 2), i.e. $3 < 4 < 5 < 6 < 7 < 8 < 9 < 10 < \text{J} < \text{Q} < \text{K} < \text{A} < 2$, and then secondly by suit, with the following order: Diamonds $<$ Clubs $<$ Hearts $<$ Spades. Throughout the rest of the paper we will refer to cards by their number and the first letter of their suit, so for example the four of hearts will be referred to as the 4H. This means that the 3D is the lowest card in the game and the 2S is the highest. In this variant of the rules, the player who is dealt the 3D has to play this card first on its own. The next player (clockwise) then either has to play a higher single card or pass, and this continues until either each player passes or someone plays the 2S. At this point, the last player to have played a card is "in control" and can choose to play any single card or *any valid poker hand*. These include pairs (two cards of the same number), three-of-a-kinds (three cards of the same number), four-of-a-kinds (four cards of the same number), two-pairs, straights (5 cards in numerical order, e.g. $8, 9, 10, \text{J}, \text{Q}$), flushes (5 cards of the same suit), full-houses (3 cards of one number, 2 of another number) and straight-flushes (both a straight and a flush). Subsequent players must then either play a better hand *of the same number of cards*, or pass. This continues until everyone passes, at which point the last player gets control and can again choose to play any valid hand that they wish. The game finishes once one player has gotten rid of all of their cards, at which point they are awarded a positive reward equal to the sum of the number of cards that the three other players have left. Each of the other players is given a negative reward equal to the number of cards they have left — so for example, if player 1 wins and players 2,3 and 4 have 5, 7 and 10 cards left respectively then the rewards assigned will be $\{22, -5, -7, -10\}$. This provides reasonable motivation to play to win in many situations, rather than just trying to get down to having a low number of cards left.

In terms of hand comparisons for hands consisting of more than one card, we have the following rules: two-card hands (i.e. pairs) are ranked primarily on number, such that e.g. $[5x, 5y] < [10w, 10z]$ regardless of the suits, and then secondly on suit with the pair containing the highest suit winning, e.g. $[10C, 10H] < [10D, 10S]$. For three card hands only the number is important since we never have to compare three card hands of the same number. For four card hands, when we compare two-pairs only the highest pair is important (so e.g. $[QD, QS, JH, JS] < [KC, KH, 4C, 4H]$), and a four-of-a-kind beats any two-pair. For five card hands we have that: Straight $<$ Flush $<$ Full House $<$ Straight Flush. If we are comparing straights, then whichever one contains the largest individual single card will win, and similarly for comparing flushes. Full houses are compared based on the number which appears three times, so for example: $[2S, 2H, 5C, 5H, 5S] < [3S, 3H, 10H, 10S, 10C]$.

The skill of the game is in coming up with a plausible strategy for being able to play all of one's cards. This often needs to be adapted as a result of the strategies which one's opponents play, and includes identifying situations when the chances of winning are so low that it is best to try and aim for ending with a low number of cards, rather than actually playing to win. This involves knowing when to save hands for later that one could play immediately, but which might turn out to be a lot more useful at a later stage of the game. Whilst there is certainly a significant amount of luck involved in terms of the initial hand that one is dealt, such that the result of any individual game shouldn't be taken to be too meaningful, if one plays against more experienced opponents it will quickly become apparent that there is also a large skill component involved. As such, over a large number of games a good player will have a significant edge over a less experienced player in the long run.

Virtual Big 2 environment

A virtual environment written in Python which simulates the game is available alongside the source code used for training the neural network to play here: `https://github.com/henrycharlesworth/big2_PPOalgorithm`. The environment operates in a way which is fairly similar to those which are included in OpenAI Gym(Brockman et al. 2016) but with a few differences. The primary functions used are:

```
#set up and reset environment:
env = big2Game(); env.reset()
players_go, current_state,
currently_available_actions =
env.getCurrentState()
#play chosen action and update game:
reward, done, info = env.step(action)
```

There is also a parallelized implementation of the environment included. This uses Python's multiprocessing module to run multiple different games at the same time on different cores, which is particularly useful for the method we used to train a neural network to play, which we describe in section 4.

Describing the state of the game

One of the most important steps for being able to train a neural network to play is to determine a sensible way of encoding the current state of the game into a vector of input features. Technically a full description of the current game state would involve information about the actual hand the player has as well as every other hand that each player had played before them, as well as any potentially relevant information about what you believe the other players' styles of play to be. Given that it is possible for some games to last over 100 turns, storing complete information like this would lead to potentially huge input states, containing a large amount of information which is not particularly important when making most decisions. As such, we design an input state by hand which contains a small amount of "human knowledge" about the things that we consider to be important when making a decision in the game. Note that this is the only stage at which any outside knowledge about the game is built into our method for training a neural network, and we have tried to keep this fairly minimal. Full details can be found in Appendix A.

Representing the possible actions

Modelling the available actions takes a bit more thought, since generally there are many ways you can make poker hands from a random set of 13 cards. What we need is a sensible and systematic way of indexing these. The approach we take is to ensure that player's hands are always sorted in order, and then define actions in terms of their indices within the sorted hand. So for example, if we are considering actions involving playing five cards, and a player has the hand $[3C, 3S, 4H, 6D, 7H, 8C, 9D, 10C, KS, AC, AS, 2C, 2S]$, then we could define the action of playing the straight $[6D, 7H, 8C, 9D, 10C]$ in terms of the ordered card indices within the hand (using 0 as the starting index): $[3, 4, 5, 6, 7]$. If instead we were thinking of the flush: $[3C, 8C, 10C, AC, 2C]$, this would correspond to indices $[0, 5, 7, 9, 11]$. This works fine, because the input state to the neural network tells us about which card value actually occupies each of the card indices in the current hand. We can then construct look up tables that convert between card indices and a unique action index (see Appendix B for details and pseudocode). Doing this we find that there are a total of 1695 different actions that could potentially be available in any given state, although a majority of time the actual number allowed will be significantly lower than this.

Training a network with self-play reinforcement learning

To train a neural network to play the game we make use of the "Proximal Policy Optimization" (PPO) algorithm proposed recently by Schulman et al(Schulman et al. 2017). This has been shown to inherit the impressive robustness and sample efficiency properties previously found with "Trust Region Policy Optimization" methods(Schulman et al. 2015a) whilst being significantly easier to implement. It has also been shown to be successful in a variety of reasonably complicated competitive two-player environments, such as "Sumo" and "Kick and Defend" (Bansal et al. 2018). In these examples, huge batches are generated by running many of the environments in parallel which allows the algorithm to overcome the problem of dealing with large variances.

The algorithm is a policy-gradient based actor-critic method in which we use a neural network to output both a policy $\pi(a|s)$ over the available actions a in any given state s, alongside an estimate of a state value function which is used to estimate the advantage $\hat{A}(a|s)$ of taking each action. We make use of the "generalized advantage estimation" (Schulman et al. 2015b) algorithm to do this. Further details of the PPO algorithm (including the hyperparameters used) and the neural network architecture can be found in Appendix C.

We initialise a neural network with random weights, make four copies of this, and then get them to play against each other. This means at first they are making moves completely at random. We generate mini-batches of size 960 by running 48 separate games in parallel, each for 20 steps at a time. We then train for 5 epochs on each batch using the ADAM optimizer(**?**). Note that these are significantly smaller than those used in (Bansal et al. 2018), where batches of hundreds-of-thousands were used. We then run this for $150,000,000$ total steps, corresponding to $156,250$ training updates or approximately 3 million games. This was carried out on a single PC with four CPU cores and a single GPU, taking about 2 days to complete. We did not find that it was necessary to use any kind of opponent sampling, hence the neural networks were always playing the most recent copies of themselves throughout the entire duration of training. However, it would be interesting to see if opponent sampling could lead to any further improvements. The hyperparameters we used were chosen to be similar to those which had worked previously for other tasks, but interestingly we did not have to play around with any of these at all to get the algorithm to work well. It is possible we just got lucky, and we have not conducted a rigorous study of different parameter variations, however this seems to back up the claim that PPO is remarkably robust, unlike many other deep reinforcement learning algorithms.

It is also worth noting that unlike AlphaGo and AlphaZero which use "Monte Carlo Tree Search"(Silver et al. 2016) (MCTS), we *do not* supplement the neural network with any kind of search of future game states at all. This means that we only provide the current game state to the neural network, and the decision is made solely based on this. Given that Big 2 is a game which requires significant planning, it is interesting that we are able to achieve such good results in this way. Although we have not tested this, it seems likely that some kind of generalization of MCTS to imperfect information games, of which there are a few, could provide a significant improvement on our results. Having said that, it is a very nice property that the trained agents are able to make their decisions with very little delay.

Results

Fully evaluating how good the network is able to play is not as easy as it may seem. Since this is a game of imperfect information, an ideal thing to be able to do would be to calculate the exploitability of the network's strategy, however this is difficult to do for a complex game like Big 2. As an extremely first metric of the network's performance we simply evaluate its average score when it plays against three random opponents, and track how this improves over time. This is shown in figure 2(a), where we see that initially there is a rapid improvement in performance (at update 0, the average reward is 0, but the graph is cut off). For the rest of the training period we see the increase is more gradual, which is perhaps not surprising as (a) the networks are not trying to get better against random opponents, but against the most recent copy of themselves and (b) there's only so good the performance can get against someone playing randomly since luck plays a significant role in the game. Figure 2(b) shows perhaps a more interesting measure of how the network performs playing against three earlier copies of itself. That is, we take the final trained network and make it play against earlier versions which we saved, evaluating how well it does against all of these. We see that there appears to be continual improvement throughout the training, with the final network beating all of the previous ones.

As a more interesting test we designed a front-end, making it easy for humans to play against the trained network. This is available to try for yourself, linked to from the Github repository (`https://github.com/henrycharlesworth/big2_PPOalgorithm`). We then gathered some data of a different human players against three of the trained neural networks over a number of games. Whilst none of these players could be considered professional, all had a decent amount of experience playing the game and so were not coming into it unprepared. There is a definite issue here in that Big 2 is a game where there is a large variance in the scores achieved, and games involving humans take quite a long time. As such, we were not able to gather as much data as we would have liked. Nevertheless, the data we have gathered is included in Appendix D, and we see that in all but one case the human players ended with a negative score (and this exception was over a relatively small number of games). Since Big 2 is a zero-sum game, a negative score can effectively be counted as a loss, and so we see that the trained network significantly outperformed the humans overall.

Conclusion

In this paper we have introduced a novel environment to simulate the game of "Big 2", specifically designing it as an environment to test reinforcement learning algorithms on. We have also been able to successfully train a neural network to play the game to a standard which exceeds amateur human players, purely using self-play deep reinforcement learning, and without the needing to supplement this with any kind of tree search over possible future states in order to make a decision. The game of "Big 2" is a challenging game for a number of reasons we have discussed, and our trained neural network certainly does not play optimally. As such we would encourage anyone working on multi-agent learning techniques to consider this environment as a test for their algorithms.

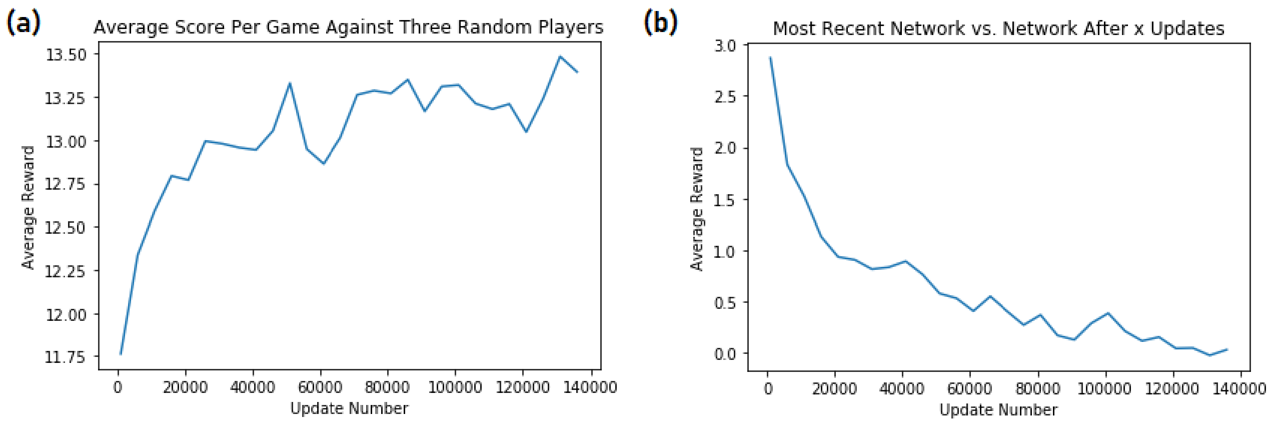

Figure 2: (a) Average score per game of the trained network against three random opponents as the training progresses. (b) The final network against three copies of the network at earlier times in the training. All plotted points are averaged over 10,000 games. Note that the first point plotted is after 1000 updates rather than 0.

Acknowledgements

Thanks to Liam Hawes, Katherine Broadfoot, Terri Tse, Kieran Griffiths, Shaun Fortes and James Frooms for agreeing to play competitive games against the trained network, and to Professor Matthew Turner for reading this manuscript and providing valuable feedback. This work was supported by the UK Engineering and Physical Sciences Research Council (EPSRC) grant No. EP/L015374/1, CDT in Mathematics for Real-World Systems.

Appendix A: Encoding the current game state

Figure 3 shows the input that is provided to the network. Firstly the player's cards are sorted into order of their value (from 3D to 2S) and labelled from 1 up to a maximum of 13. For each card in the player's current hand there are then 13 inputs that are zero or one to encode the card's value, and then four more to encode the suit. As well as this we provide information about whether the card can be included in any combination of cards (i.e. is it apart of a pair, a straight etc). For each of the three opponents we keep track of the number of cards they have left as well as well as certain information about what they've played so far. In particular, we keep track of whether *at any point* during the game so far they've played any of the highest 8 cards (AD - 2S), as well as if they've played a pair, a two pair, a three of a kind, a straight, a flush or a full house. The network is also provided information about the previous hand which has been played (both its type and its value), as well as the number of consecutive passes made prior to the current go, or if it currently has control. Finally, we provide it with information about whether *anyone* has played any of the top 16 cards. This is potentially important for keeping track of which single is the highest left in play, and hence would be guaranteed to take control if played. We cut this off at 16 to reduce the size of the input, and because it is rare for a high-level game to still be going when the highest cards left are lower than a queen.

This is the way we choose to represent the current game state when training our network, and is also the state which is returned by the `env.step()` function in the game environment. However, the `big2Game` class also records all hands which are played in a game, and so it should be relatively simple to write a new function which includes more or less information if this is desired.

Appendix B: Indexing the action space

Here we give the pseudocode for generating "look-up tables" which can be used to systematically index the possible actions that are available in any given state. We consider separate look up tables for actions containing different numbers of cards. In the case of five-card hands it is possible, because of flushes, for any combination of card indices to be a valid hand. This means that under this representation there are $\binom{13}{5} = 1287$ possible five-card actions. The idea is then to construct a mapping between each allowable set of indices $\{c_1, c_2, c_3, c_4, c_5\}$ and a unique action index i. Algorithm 1 does this by creating a matrix "actionIndices5" which can be indexed with the card indices to return i, and then including a reverse-look up table which maps i back to the card indices. In the case of four-card actions there are constraints on the indices that can actually be used to make a valid hand, since the only valid four-card hands are two pairs and four of a kinds. This means that, for example, the combination of indices $[2, 8, 9, 10]$ could never be a valid hand as the cards (which are sorted in order) in positions 2 and 8 could never correspond to the same number, and hence cannot be a pair. Consequently rather than there being $\binom{13}{4} = 715$ possible four-card actions, we find that are there are actually only 330 under this representation. Similar constraints apply to two and three card actions where we find that there are 33 and 31 possible actions respectively, and then trivially there are 13 possible one-card actions. In total this gives us

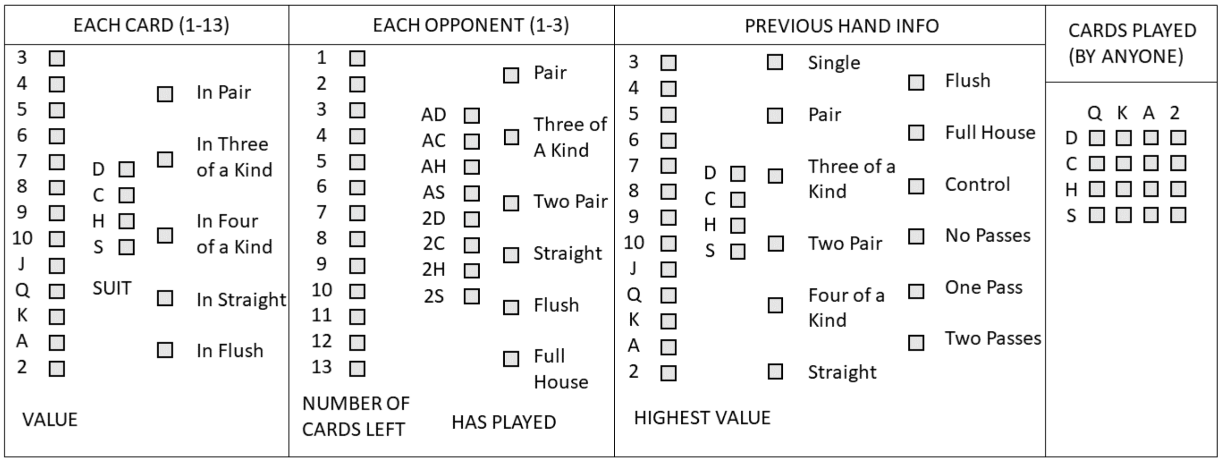

Figure 3: Input state provided to the neural network which encodes the current state of the game. This includes information about the player's own hand as well as some limited information about what each of the opponents has played so far and other things which have occurred during the game up until the present point. This leads to an input of size 412 made up of zeros and ones.

$1287 + 330 + 31 + 33 + 13 + 1 = 1695$ potential moves that could be allowable in any given state (the extra 1 is accounting for being allowed to pass). In the python implementation the big2Game class has a function `availAcs = big2Game.returnAvailableActions()` which returns an array of size 1695 of 0s and 1s, depending on whether each potential action is actually available for the current player in the current game state. This vector is ordered with one-card actions in indices $0 - 12$, two-card actions from $13 - 45$, three-card actions from $46 - 76$, four-card actions from $77 - 406$, five-card actions from $407 - 1693$ and then finally 1694 corresponding to the pass action. The `big2Game.step(...)` function takes an action index (from $0 - 1694$) as its argument and `big2Game.getCurrentState()` returns as its third value a vector of 0s (corresponding to actions allowed in current state) and $-\infty$ (not allowed). This was just because it was convenient to use these values instead of 0s and 1s when using a softmax over the neural network output to represent the probability distribution over allowed actions, but is straightforward to change.

Algorithm 1 Look up tables for five-card actions

[1] **Initialize:** *actionIndices5* as a $13 \times 13 \times 13 \times 13 \times 13$ array of zeros **Initialize:** *inverseIndices5* as an 1287×5 array of zeros **Initialize:** $i = 0$ $c_1 = 0$ to 8 $c_2 = c_1 + 1$ to 9 $c_3 = c_2 + 1$ to 10 $c_4 = c_3 + 1$ to 11 $c_5 = c_4 + 1$ to 12 *actionIndices5*$[c_1, c_2, c_3, c_4, c_5] = i$ *inverseIndices5*$[i, :] = [c_1, c_2, c_3, c_4, c_5]$ $i \mathrel{+}= 1$

Algorithm 2 Look up tables for four-card actions

[1] **Initialize:** *actionIndices*4 as a $13 \times 13 \times 13 \times 13$ array of zeros **Initialize:** *inverseIndices*4 as an 330×4 array of zeros **Initialize:** $i = 0$ $c_1 = 0$ to 9 $n_1 = \min(c_1 + 3, 10)$ $c_2 = c_1 + 1$ to n_1 $c_3 = c_2 + 1$ to 11 $n_2 = \min(c_3 + 3, 12)$ $c_4 = c_3 + 1$ to n_2 *actionIndices*$4[c_1, c_2, c_3, c_4] = i$ *inverseIndices*$4[i, :] = [c_1, c_2, c_3, c_4]$ $i \mathrel{+}= 1$

Algorithm 3 Look up tables for three-card actions

[1] **Initialize:** *actionIndices*3 as a $13 \times 13 \times 13$ array of zeros **Initialize:** *inverseIndices*3 as an 31×3 array of zeros **Initialize:** $i = 0$ $c_1 = 0$ to 10 $n_1 = \min(c_1 + 2, 11)$ $c_2 = c_1 + 1$ to n_1 $n_2 = \min(c_1 + 3, 12)$ $c_3 = c_2 + 1$ to n_2 *actionIndices*$3[c_1, c_2, c_3] = i$ *inverseIndices*$3[i, :] = [c_1, c_2, c_3]$ $i \mathrel{+}= 1$

Algorithm 4 Look up tables for two-card actions

[1] **Initialize:** *actionIndices*2 as a 13×13 array of zeros **Initialize:** *inverseIndices*2 as an 33×3 array of zeros **Initialize:** $i = 0$ $c_1 = 0$ to 11 $n_1 = \min(c_1 + 3, 12)$ $c_2 = c_1 + 1$ to n_1 *actionIndices*$2[c_1, c_2] = i$ *inverseIndices*$2[i, :] = [c_1, c_2]$ $i \mathrel{+}= 1$

Appendix C: Details about the training algorithm/ neural network architecture

If the weights and biases of the neural network are contained in a vector θ then to implement the PPO algorithm we start by defining the "conservative policy iteration" loss estimator (Kakade and Langford 2002)

$$L^{CPI}(\theta) = \hat{E}_t\left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}\hat{A}_t\right] \quad (1)$$

where here the expectation is taken with respect to a finite batch of samples generated using the current policy parameters θ_{old}. Trust region policy optimization methods maximize this loss subject to a constraint on the KL divergence between π_θ and $\pi_{\theta_{old}}$ to prevent policy updates occurring which are too large. PPO is able to achieve essentially the same thing by introducing a new hyperparameter $\epsilon \ll 1$ and instead using a clipped loss function that removes the incentive to make large policy updates. If we define $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$, then PPO considers instead maximizing the following "surrogate loss function":

$$L^{CLIP}(\theta) = \hat{E}_t\left[\min\left(r_t(\theta)\hat{A}_t, \text{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right)\right)\right] \quad (2)$$

We then also introduce a value function error term, as well an entropy bonus to encourage exploration, such that the final loss function to be optimized is

$$L(\theta) = \hat{E}_t\left[L^{CLIP}(\theta) - a_1 L^{VF}(\theta) + a_2 S[\pi_\theta](s_t)\right] \quad (3)$$

where a_1 and a_2 are hyperparameters, S is the entropy and $L^{VF} = (V_\theta(s_t) - V_t^{target})^2$ is the squared-error value loss. We estimate the returns and the advantages using "generalized advantage estimation", using the following estimate:

$$\hat{A}_t = \delta_t + (\gamma\lambda)\delta_{t+1} + \cdots + (\gamma\lambda)^{T-t+1}\delta_{T-1} \quad (4)$$

where T is the number of time steps we are simulating to generate each batch of training data, γ is the discount factor, λ is another hyperparameter and $\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$ (with r_t being the actual reward received at time step t).

When a batch is generated by running N separate games each for T time steps and the advantage estimates are made training then occurs for K epochs using a minibatch size of M. The hyperparameters we used for our training were the following: $N = 48, T = 20, \gamma = 0.995, \lambda = 0.95, M = 240, K = 4, a_1 = 0.5, a_2 = 0.02$ with a learning rate $\alpha = 0.00025$ and $\epsilon = 0.2$ which were both linearly annealed to zero throughout the training.

In terms of the neural network architecture we used this is shown in figure 4. We have an initial shared hidden layer of 512 RelU activated units which is connected to two separate second hidden layers each of 256 RelU activated units. One of these produces an output corresponding to the estimated value of the input state whilst the other is connected to a linear output layer of 1695 units which represents a probability weighting of each potentially allowable move. This is then combined with the actually allowable moves to produce a probability distribution. The rationale for having a shared hidden layer is that there are likely to be features of the input state that are relevant for both evaluating the state's value as well as the move probabilities, although we did not run any tests to quantify whether this is really significant. All layers in the network are fully connected.

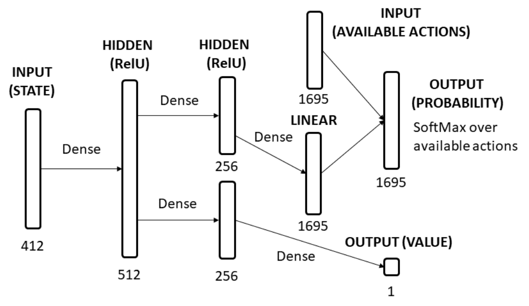

Figure 4: Architecture of the neural network used.

Appendix D: Results against human players

Results against seven different human players are shown in table 1.

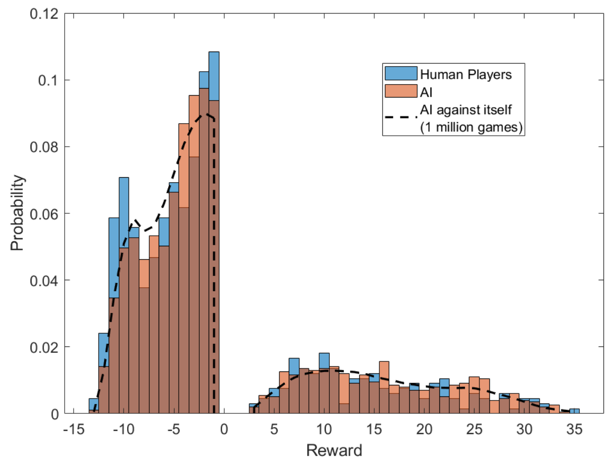

Figure 5: Probability distribution of the rewards received from the games between the AI and various human players (see table 1 for a summary of results). For comparison the black line is the probability distribution from four of the fully-trained neural networks playing against themselves over 1 million games.

Although we only have a relatively small data set and Big 2 is a game of large variance in the scores, it is clear that on the whole the neural network quite significantly outperforms the human players. Of the seven players who played only one of them finished with a positive score, and this was from a relatively small number of games. If we look at the total scores of all of the human players combined we find an average score of -0.96 ± 0.38 per game, which shows that on the whole the trained neural network seems to have a significant advantage.

	Player 1	Player 2	Player 3	Player 4	Player 5	Player 6	Player 7	Total
Games Played	250	127	100	55	50	50	31	664
Games Won	68 (27.2%)	25 (19.7%)	19 (19.0%)	21 (38.2%)	5 (10.0%)	4 (8.0%)	7 (22.5%)	149 (22.5%)
Final Score	-128	-118	-154	104	-100	-231	-10	-637
Average Score	-0.51	-0.93	-1.54	1.89	-2.00	-4.62	-0.32	-0.96
Standard Error	0.68	0.86	0.88	1.54	1.09	0.93	1.52	0.38
AI Scores	51, 58, 19	15, -78, 181	73, -143, 224	-71, -15, -18	86, 8, 6	137, 116, -22	124, -77, -37	415, -131, 353
AI (1) Average	0.20 ± 0.67	0.12 ± 0.95	0.73 ± 1.04	-1.29 ± 1.23	1.72 ± 1.48	2.74 ± 1.70	4.00 ± 2.07	0.63 ± 0.41
AI (2) Average	0.23 ± 0.67	-0.61 ± 0.89	-1.43 ± 0.84	-0.27 ± 1.35	0.16 ± 1.39	2.32 ± 1.75	-2.48 ± 1.31	-0.20 ± 0.39
AI (3) Average	0.08 ± 0.65	1.43 ± 1.09	2.24 ± 1.12	-0.33 ± 1.20	0.12 ± 1.29	-0.44 ± 1.48	-1.19 ± 1.37	0.53 ± 0.41

Table 1: Data from games of seven different human players vs. 3 of the trained neural networks. Standard errors on the average scores are calculated as $\sigma_m = \sigma/\sqrt{N}$ where σ is the standard deviation of the game scores and N is the number of games played.

We can also look at the probability distribution of the rewards (figure 5) to potentially get more insight into how the neural network plays compared with the human players, although really we do not have enough data to say anything conclusive. One of the main differences we see is that the human players seem to find themselves left with a large number of cards more frequently than the AI does, perhaps as the AI is better able to identify situations where the chances of winning is very low and so knows just to get rid of as many cards as possible. It also seems like the AI is slightly better at ending the game early, and therefore achieving the higher scores. This could also be the reason why human players tend to have more cards left more often, although it's difficult to say anything concrete here.

References

[Bansal et al. 2018] Bansal, T.; Pachoki, J.; Sidor, S.; Sutskever, I.; et al. 2018. Emergent complexity via multi-agent competition. In *ICLR*.

[Brockman et al. 2016] Brockman, G.; Cheung, V.; Petterson, L.; Schneider, J.; et al. 2016. Openai gym. *arXiv preprint arXiv:1606.01540*.

[Brown and Sandholm 2017] Brown, N., and Sandholm, T. 2017. Superhuman ai for heads-up no-limit poker: Libratus beats top professionals. *Science*.

[Heinrich and Silver 2016] Heinrich, J., and Silver, D. 2016. Deep reinforcement learning from self-play in imperfect-information games. In *NIPS Deep Reinforcement Learning Workshop*.

[Kakade and Langford 2002] Kakade, S., and Langford, J. 2002. Approximately optimal approximate reinforcement learn- ing. In *ICML*, volume 2, 267–274.

[Lowe et al. 2017] Lowe, R.; Wu, Y.; Tamar, A.; Harb, J.; et al. 2017. Multi-agent actor-critic for mixed cooperative-competitive environments. *arXiv preprint arXiv:1706.02275*.

[Mnih et al. 2015] Mnih, V.; Kavukcuoglu, J.; Silver, D.; Rusu, A.; et al. 2015. Human-level control through deep reinforcement learning. *Nature* 518:529–533.

[Moravčíc et al. 2017] Moravčíc, M.; Schmid, M.; Burch, N.; Lisy, V.; et al. 2017. Deepstack: Expert-level artificial intelligence in heads-up no-limit poker. *Science* 356:508–513.

[Schulman et al. 2015a] Schulman, J.; Levine, S.; Moritz, P.; Jordan, M.; et al. 2015a. Trust region policy optimization. *arXiv preprint arXiv:1502.05477*.

[Schulman et al. 2015b] Schulman, J.; Moritz, P.; Levine, S.; Jordan, M.; et al. 2015b. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*.

[Schulman et al. 2017] Schulman, J.; Wolski, F.; Dhariwal, P.; Radford, A.; et al. 2017. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.0634*.

[Silver et al. 2016] Silver, D.; Huang, A.; Maddison, C.; Guez, A.; et al. 2016. Mastering the game of go with deep neural networks and tree search. *Nature* 529:484–489.

[Silver et al. 2017] Silver, D.; Schrittwieser, J.; Simonyan, K.; Antonoglou, I.; et al. 2017. Mastering the game of go without human knowledge. *Nature* 550:354–359.