Artificial Neural Networks for playing Spades

Simon van Hus
6147879

April 6, 2020

Thesis for Bachelor Artificial Intelligence at Utrecht University
7.5 ECTS

Supervised by dr. S. Renooij
Second assessment by dr. B. Harvey

1 Introduction

Even since before the dawn of computers people have been trying to have machines play human games. While the most famous example would be chess, computer researchers have also been diligently at work on creating AI for card games in the past thirty years. In my research, I'll be building an AI to play the card game Spades. In this paper, we'll first discuss my main research question, some background information on Spades, and AI for card games. We'll then continue with selecting a fitting approach, its place in the field of AI, followed by the methods used. After that, we'll interpret the results, discuss them, and close off with a general conclusion to the questions discussed next.

My main research question will be whether or not a neural network can be trained to play Spades at a human level. I'll also be looking into the sort of architecture that can be used for such an AI, but that will be discussed in the methods section (2).

1.1 The game of Spades

Spades is a 1930s card game from the Whist family of trick-taking card games, also including games like Bridge and Hearths. It is most often played with four players (for our purposes not in teams), who start a round by getting dealt thirteen cards each, after which they bid on how many tricks they think they'll be able to take. If you make your bid, you get points out of it. Like with other trick-taking games, a trick is taken by playing the highest card of the colour played by the first player in the trick. Unlike its familial games however, spades always trump. Hence the game's name. A more detailed set of rules is provided in Appendix A.

1.2 Card game AIs

Over the last thirty years many card game playing AIs have emerged, using techniques such as planning algorithms, Bayesian learning, Monte Carlo tree search (MCTS), and neural networks.

The most played card game in the world is the trick-taking game of Bridge. It's not surprising then that most card-game AIs have something to do with Bridge. In 1998 AI planning took the Bridge community by storm as it won the world championship in computer Bridge (Smith, Nau, & Throop, 1998), emulating the human way of planning rather than relying on tree search. Since then tree search has made a resurgence, with MCTS now being the most reliable way of getting high level play (Russell & Norvig, 2016; Whitehouse, Powley, & Cowling, 2011). We also find MCTS in Poker (Rubin & Watson, 2011), and here we also find a different technique, using Bayesian networks (Heiberg, 2013), and while this latter AI played a good game, it wasn't quite up to par with human players. Finally the most AI-like way – in my opinion, because this one's really learning – of tackling this problem is with neural networks. However, such a thing hasn't yet been done outside of the double-dummy Bridge problem (Mossakowski & Mańdziuk, 2004; Dharmalingam & Amalraj, 2013), which is a dumbed down version of Bridge with perfect information.

1.3 Selecting an AI for Spades

Of course all methods discussed have merit in creating an AI for Spades. To find a method that can hold it's own against human players however, we'll need to look at AIs that are able to grapple with the nuances of an entire round of Spades from start to finish. Bayesian networks may be able to give us that, but runs into a problem of complexity. In the Poker paper, the game is therefore abstracted to make the Bayesian network more manageable. Poker is the perfect game for this, as any combination of cards already has a decided category to fall into (the different Poker outcomes). This isn't the case for Spades however, and while abstraction isn't strictly necessary, it would be too complex to not use abstraction: thirteen random cards in the starting hand, any combination of thirteen card in the each opponent's hand, and up to four cards on the table. A Bayesian network would consist of a directed graph with nodes for all these cards, and arcs connecting them; most importantly, for every node a set of probability distributions would need to be specified whose size is exponential in the number of parents of the node.

MCTS has proven itself time and time again for a variety of card games, however it falls apart for Spades due to the way the game is set up. The game only evaluates moves at the end of a round, when the tricks are evaluated. This doesn't necessarily have to be a problem, if we just try and maximize the tricks taken overall. However, this would require us assigning some arbitrary value to taking a trick, which would be tricky, and still does nothing to address the size of the search tree, which is massive: $52! \simeq 8.06e67$ nodes.

Neural networks luckily don't run into the same problems as Bayesian networks or MCTS. Where they do struggle however, is in the field of training data. Where Bayesian networks don't really train, and MCTS uses simulation, a neural network is often trained using supervised learning. This however is impossible, as that would require us modelling example data for games to test the neural network against (and if we manage that, we wouldn't need an AI to play it for us because we'd already know how to play the game in the best possible way). Instead we'll have to look into unsupervised learning. A great example for a game that manages this with success is checkers (Chellapilla & Fogel, 1999). Here, a neural network is initialized with a set structure and random weight values. A couple of different neural networks then go head to head, and their success in winning the game determines their fitness, which is then used by a evolutionary algorithm, to evolve the weights.
This is also the method I'll be using for the Spades AI.

1.4 The place in the AI field

As discussed before, game-playing AI are nothing new. The goal of these approaches has always either been to beat humans (and other AI) as thoroughly as possible, or to emulate human play. And whilst neither neural networks, evolutionary programming, nor card games, should be seen as the cutting edge of AI, I feel that I've struck a unique balance between the three. Using the combination for a card game hasn't really been

done as far as I'm aware, and even the checkers AI was operating on perfect information. This is an important difference, because this means the AI (or a human player) has access to more information to make choices with. In a card game however, you can't know what cards your opponents have, which requires you to make some informed decisions about the risks you might run into playing certain cards.

2 Methods

2.1 Preliminaries

Before we get into the details of the experiment, we first take a look at the techniques used in finer detail.

2.1.1 Artificial Neural Networks

The main idea of a neural network is to emulate the human brain, by creating a network of (neural) nodes feeding information from an input to an output. Each node may have an activation function, which transforms its input, and a weight associated with each of its outputs. The neural networks are generally divided into layers, each of which take the output of the previous layer as input, and in turn then output to the next layer. A neural network consists of three different kinds of layers, a single input layer, a single output layer, and any number of hidden layers in between them. And whilst the size of the input and output layers is generally set by the problem one's trying to solve, the size and number of hidden layers can vary. However, the number of hidden layers is generally limited to one or two, as any continuous mathematical functions can be mapped by a neural network with one hidden layer, and almost any arbitrary decision can be mapped by network with two hidden layers (Heathon, 2008). The number of nodes in a hidden layer is also a contentious issue, and differs from problem to problem. If you have too many, you'll find many nodes will not be useful and just slow down your network. If you have too few, the neural network might have a hard time finding an effective configuration.

This configuration is found by training a neural network. This is generally done supervised, meaning that the output data of a neural network on some set input data is tested against output data that's known to be right. I'll be using unsupervised learning however, because we have no correct output data to test against playing a game of cards. To do this we'll use the next method: evolutionary programming.

For our experiment we'll be using one of the most simple set-ups for neural networks, a feed-forward neural network. In this set-up, all the information flows in one direction, layer per layer. We'll also be using two hidden layers, because of the reason discussed before. Our set-up will also be fully connected, meaning that each node of some layer l is connected to each node of layer $l + 1$. Finally, we'll also be using a tanh activation function on all our hidden layer and output layer nodes. This way we can ensure our result falls comfortably between -1 and 1.

2.1.2 Evolutionary Programming

Evolutionary programming is one of the four main evolutionary algorithms, who mimic evolutionary processes to iterative select and create better and better systems (like neural networks) for solving a problem. In evolutionary programming, the structure of such a system is already set, and only numeric variables are changed. In our case this means that the structure of the neural network is already determined, but we'll be changing up the weights of the outgoing connections of each node. To be more specific, the evolutionary processes mimicked are reproduction and mutation.

Reproduction is the algorithm creating a new configuration out of two (or more) older configurations. In this case we do this by recombination, where we fill a list of weights by choosing the weight of some index randomly from either 'parent'. This way we create a neural network that has hopefully inherited some of the better features from both parents.

Finally, mutation is changing a random number of weights with random values. This way we ensure we insert fresh 'insight' into the gene-pool, and we won't get stuck with clones of just the best performing neural network from the first generation. Setting a value for mutation high at the beginning ensures high exploration of possibilities. Then slowly decreasing it over time can assure that the best choices stick around for longer.

2.2 Simplification

Now, before we start with the neural networks, we first need to make a couple of simplifications to the main rule set of Spades. These simplifications concern the bidding phase, namely that we'll ignore the phase (and bidding) entirely. If we want an AI that can play Spades as good as possible, we don't need to do bidding. To win the game, you need to be able to make the most bids as possible. This means that for each individual round we can just focus on making as many tricks as possible. If desired, one can always make an AI afterwards – that predicts the number of tricks made, using the player's hand as input – to do the bidding. I've also decided against having the neural networks learn the rules, instead we'll just select the highest scoring card that is in compliance with the rules. This is mainly because it'd mean the AIs would first need to learn the rules before they can really start getting better. This is extremely costly, but more importantly might also ingrain certain patterns into a neural network that make it hard for it to change up it's play-style, because it might need to go through a rule-breaking phase before finding a new strategy that's also compliant with the rules. We instead entrust the evolutionary part of the training to pre-select AIs that prefer playing by the rules, as they – intuitively – will be better at playing cards that are able to take a trick.

2.3 Architectures

The neural networks that'll train up for Spades will fall in one of two architectures. Both networks will have two hidden layers, but will have different input and output vectors. The first architecture uses the entire deck as its input vector. This architecture is named 'card-based', as for each of the 52 cards three input nodes exists, and one output node. During the game, each of these 52 cards will be in one of three locations: in the hand of the (AI) player, on the table, in the hand of an opponent, or it's already been played. Each card will get a categorical input representing its position in the game. This is encoded using one-hot encoding, where one of three input nodes is set to one (and the other two to zero) to represent the position being either in the hand of the player, on the table, or played previously. All three of them are set to zero if the card has yet to appear. The output vector is a vector of 52 nodes with a value between -1 and 1 representing each card. The card represented by the node with the highest value will be played, provided that it's in accordance to the rules. If it'd break the rules, we just continue to the next highest valued card.

The second neural network has its input vector divided into three sections, and is named 'hand-based'. The first part will represent the hand of the AI player, which is at most thirteen cards. This part is 65 nodes long, because each card is represented by five nodes. The first node of each card represents the card's rank, and it's value is just a number with one for a card with rank two and thirteen for an ace. The next four nodes are a one-hot encoding of the card's suit. With each node representing one suit (in our case in the order of: clubs, hearths, spades, diamonds). Whenever a card is played from the hand it is replaced by a zero value in this vector. The following fifteen nodes (= three cards) represent the other players' plays. At most three players have gone before you in a single trick, so hence fifteen nodes. Finally we leave a space of 240 nodes (= 48 cards) to be filled up with the cards that have been played in previous tricks. The last trick of a round will never make it here, because after it's played the round ends.
The output vector consists of thirteen nodes, representing the thirteen cards in the hand of the player (in the order they were in at the start of the round). Again, the card represented by the node with the highest value (between -1 and 1) gets played, provided that it's in accordance to the rules.

For both designs we use a rule of thumb for deciding the amount of nodes in the hidden layers, to be precise:

$$\frac{2}{3}(|\text{input}|) + |\text{output}|$$

This means we take two thirds of the of the lengths of the input vector and add the length of the output vector (Heathon, 2008). This results in 156 and 266 nodes for the card-based and hand-based architectures respectively.

Finally, an example of a diagram of both neural networks can be found in figures 1 and 2.

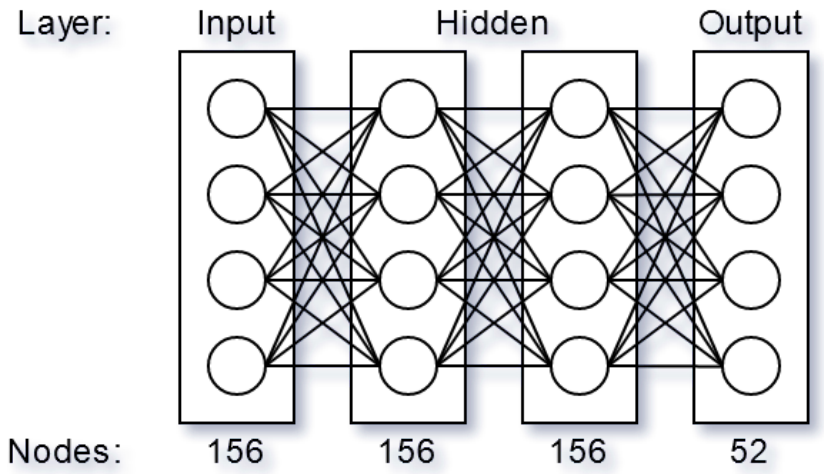

Figure 1: Card-based neural network

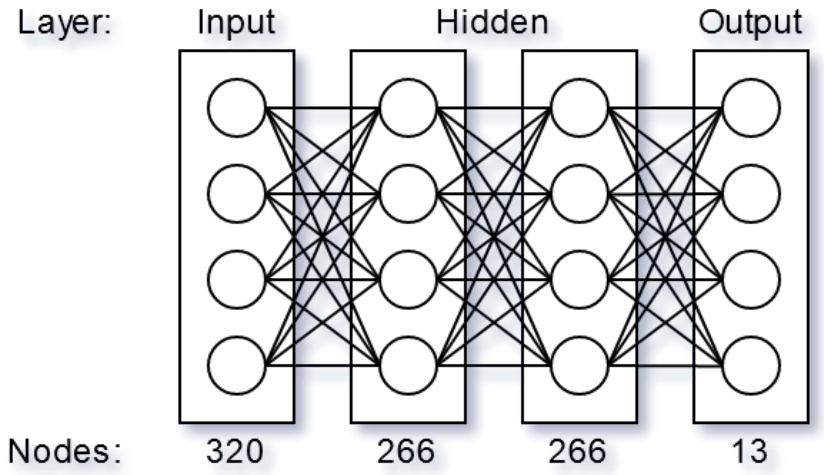

Figure 2: Hand-based neural network

2.4 Evolutionary Programming

To train up our neural networks we use evolutionary programming (Fogel & Fogel, 1995; Eiben, Smith, et al., 2003), where we use evolutionary processes to evolve the weights between nodes in the neural network. To start off we generate a generation of neural networks (in one of the two architectures discussed earlier), initialized with random weights for each connection. We then make the members of this generation go head to head in a couple games of Spades. (We did fifteen games of four rounds each, for randomly generated groups of four). We then look at the performance of each member over all these games, and determine their 'fitness.'

I decided on two ways of determining the fitness of a member. The first 'simple fitness' method, just looks at the amount of tricks made. Each trick made counts as one point to it's total fitness.
The second 'advanced fitness' method, looks at the card the member used for making each trick it made. Cards that are higher ranked have lower scores, because they are beaten by fewer cards, and thus have an easier time making tricks. (The probabilities and a complete score table can be found in Appendix B). Throughout training, only one of the two fitness measures is used, thus separating our results into two categories: using simple fitness and using advanced fitness.

Next, we use this fitness information to create a new generation. This is done by reproduction through recombination. Since all networks of an architecture are the same in shape, we can safely recombine the weights of two well-performing 'parents' into a new 'child' network. The top 50% of a generation are selected for parenthood of the next generation. These parents each recombine with another (randomly chosen) parent of this group, and thus create one child per parent. The next generation is then made up of all these parents, and their children. The generation stays of the same size, only now containing 50% new members, and 50% members from older generations. This also means that well performing members may stick around for multiple following generations.

A second step in the evolution process is mutation. After a new generation is created by parent selection and recombination, there's a chance m for each member of the generation it will mutate. Mutating a member of a generation means that for each weight of that member's network there's a $m/2$ chance to be replaced by a random value between -1 and 1 (at which all weights are clamped). We do this to make sure not too many changes are made, without fixing the mutation parameter in this stage. Since we rely on mutation to get new networks (and strategies), this factor m starts of at 80% and then decreases to 20% (Castillo et al., 2003). This way we trade in exploration for exploitation over time.

2.5 Other details

With two architectures and two fitness functions, we have ourselves four categories to test. To make sure we see enough of each of these categories both were tested for 200 generations. However, we also keep track of the neural network with the highest fitness so far, and if we find a new one, we search for 200 more rounds from it's conception. This way we make sure every strain of neural network – but also it's descendants – has a chance to succeed. Furthermore, to put our networks to the test somewhat decently, each generation is tested on fifteen games of four rounds each. Each game is played in a different group of four randomly selected members of the generation. Also, we generate four starting hands per game, to be used by all groups and then cycle these hands around during the four rounds, so every network gets a fair shot.

3 Results

Before we can discuss the results in Table 1, we must first talk about how to interpret them. We can compare our neural networks to a hypothetical arbitrary player that would play a random card on each trick. Provided it plays only against other arbitrary players, we can expect it'd take 25% of tricks in the long run. This means that as the number of rounds approaches infinity, the average number of tricks taken per round approaches 25% or 3.25. We call this the average tricks per round, or ATpR for short. It should also be noted that true arbitrary play is impossible due to the rules limiting the cards eligible for play, but this doesn't influence the average number of tricks per rounds as the number of rounds approaches infinity.

	Simple Fitness			Advanced Fitness			
	Generation	Tricks	ATpR	Generation	Tricks	ATpR	Fitness
Card-based	18	234	3.9	94	224	3.73	13339
Hand-based	140	231	3.85	62	226	3.77	13487

Table 1: Neural network results with the highest fitness

In Table 1 we see the best performing neural networks (by their fitness). It shows their fitness function, their architecture, the generation they were conceived in, the number of tricks they took, and their ATpR. The advanced fitness group also shows the actual fitness value (which for the simple fitness is equal to the number of tricks taken). It's also important to note that the advanced fitness value of an arbitrary player would be 12090 over the 780 tricks[1].

We see that the best performing network was the card-based network using the simple fitness function, but we can also see that it's performance isn't that spectacular in comparison to the other three networks. In fact, there doesn't seem any pattern as

[1]This is calculated by taking the average fitness of each card assuming the arbitrary player will lead 25% of tricks (= 62), and multiplying this by 25% of the total of 780 tricks: $62 \cdot 0.25 \cdot 780 = 12090$

to a better performing architecture or fitness function, and if there were, the differences would be too small to make any meaningful statement on it.

So let's compare this (card-based simple fitness) network with an arbitrary player then. First we need to make an extra assumption concerning the chance to take a trick. We already made this assumption for the arbitrary player described earlier, but we say it is 25%. This is only true if the four players are of equal skill and as the number of rounds played approaches infinity. We'll go into the caveats of this assumption later on.

We can now create a normal distribution, with $\mu = 3.25$ and $\sigma = 1.5612$, details on which can be found in Appendix C. Now, we'd want to know at what probability our arbitrary player would generate a result of 3.9 (our best ATpR) or better. We can do this with the cumulative distribution function Φ on our generated normal distribution (which would be the same as a Z-test). We find that $\Phi(3.9) \approx 0.661$, so there's a $1 - \Phi(3.9) \approx 0.339$ chance an arbitrary player would play at least this well. This means we can't decisively say using statistics whether or not our neural network is performing better than an arbitrary player, at least not with a confidence interval bigger than 33.9%.

There are a couple caveats to this conclusion however, starting with the one way laid out when describing an arbitrary player. In truth, no arbitrary player exists, because the rules forbid any player from just playing any card at random. There will always be structure to any amount random play. And while it's true that players with equal skill will trend to a trick-taking rate of 25% per round, this will only happen after many games (as the number of rounds approaches infinity).

Which brings us straight onto our second caveat: we didn't play an infinite amount of rounds. In fact, our neural networks only got sixty rounds to prove themselves, which means their performance might either be higher or lower than their real average would be. Which finally, is also the cause of our third caveat: even an arbitrary player won't make 25% of tricks in a round, because their chances of taking a trick are influenced by the cards they got handed. In fact, looking at Table 2 and 3 in Appendix B, we see that there exists a large variance of trick-taking chances between cards. This also skews the results on scale of only sixty rounds (which – again – would average out to 25% as the number of rounds approach infinity).

These three caveats taken together mean that we have to take the results with a grain of salt. While doing a finer calculation would get to complicated for the scope of this thesis, it's good to be aware of this.

4 General Discussion

In our results, we discuss how our best neural network only plays better than a random player about two thirds of the time, under a couple caveats. There's however one more caveat worth discussing here, which is an inherit caveat due to the game's rules. Namely that every trick needs to be taken. This means that if you have some bad cards, but you're opponents cards are worse, you might still take a trick with those bad cards. In the land of the blind, the one-eyed man is king. In a real game (with bidding) this can lead to a lot of frustration and some interesting strategies, like letting a trick you bid on go to have someone else – who didn't bid on it – pick it up instead (and end up with more tricks than they bid on). However, for the purposes of our neural networks this becomes a limiting factor in how well we can assess the skill of our networks. Because the networks also play with a randomly generated hand, they'll be forced to give some tricks away just because they don't have the cards to take them. This means that neural networks have a harder time distinguishing themselves, and might cause difficulty for the evolutionary algorithm to separate the good from the lucky neural networks.

What is much harder expressed in numbers, is how the AI stands up to human play. It does reasonably well at it's goal, maximizing the number of tricks, but does so in an a seemingly inhuman way. However, there does seem to be some nuance to the play of the neural networks, mainly in what tricks they take. The neural networks seem to prefer taking unlikely tricks, or save their high cards for later, something you wouldn't expect a human to do, because it can put tricks – that otherwise have a high chance of being taken – at risk[2]. This might be an effect of training, or this might indicate incapability, as again it's hard to distinguishing the networks' trained behaviour from that of pure luck.

In a real game much more of the final score hinges on whether a player can make their bid or not, which of course is completely absent from this simulation. This is important to note, because this also rewards players for bidding on and making zero tricks, and creates variety in more aggressive and more conservative strategies. Because we train our neural networks on winning as many tricks as possible, we're actually only training it on the most aggressive strategy possible. If this weren't the case, an AI that can take 3.9 tricks against three other players of equal skill, is actually a very strong player. In the long run this means the AI will outrun the other players in score. However if it only takes two tricks for every three tricks of each other player, it will only score 20% or 2.6 tricks per round, which means it'll lose in the long run. This is unfortunately what seems to be going on, which also shows that the networks are perhaps only on a beginner level of Spades in their performance.

Perhaps changing the architectures around more, or finding a way to include the bidding phase will lead to more interesting results. A possible change to the architecture could be including the likelihoods of winning with each card in the input vector, or creating a direct connection between the cards in the input vector to the last layer,

[2]This is because cards played earlier have a lower chance of being trumped, because all players generally have three or four cards of each colour.

in the hopes of creating a more solid link between the cards and the output. Whilst better ways of creating an AI for card games are known, we shouldn't count out the neural network just yet, nor should we discount evolutionary systems. There's still lots to experiment with.

5 Conclusion

To our main question – whether or not an evolved neural network is suitable for creating a Spades AI – we might just simply answer no. The results certainly point in that direction, however there's a caveat to these findings. After all we were working with a slight simplification, and there are many more ways to apply neural networks and evolutionary programming. We've tried two architectures that seemed obvious, but did not produce spectacular results. This however does not mean that no neural network might ever be able to play Spades at a high level.

Between the two architectures we didn't find much difference in performance. Which is probably because we provided both with the same information, just in different ways. A real difference might appear when providing a neural network with different information. There wasn't a noticeable difference between fitness functions either, other than the slight difference in play style (where those with advanced fitness would prefer making more difficult tricks, in line with the fitness function).

All in all, it seems like the lack of neural network AI in card games isn't wholly unfounded, but there's also room for more research, mainly on other configurations of neural networks. I personally have learned a lot on implementing a neural network, and evolutionary programming, which I did without the help of any libraries. Furthermore, this project really made me think about how to represent a domain – in this case the game of Spades – for a neural network. Not to mention it was great fun to (albeit in text-form) see the neural networks go head to head. It has been a worthwhile endeavour.

References

Castillo, P., Arenas, M., Castillo-Valdivieso, J., Merelo, J., Prieto, A., & Romero, G. (2003). Artificial neural networks design using evolutionary algorithms. In *Advances in soft computing* (pp. 43–52). Springer.

Chellapilla, K., & Fogel, D. B. (1999). Evolving neural networks to play checkers without relying on expert knowledge. *IEEE transactions on neural networks*, *10*(6), 1382–1391.

Dharmalingam, M., & Amalraj, R. (2013). Artificial neural network architecture for solving the double dummy bridge problem in contract bridge. *International Journal of Advanced Research in Computer and Communication Engineering*, *2*(12), 4683–4691.

Eiben, A. E., Smith, J. E., et al. (2003). *Introduction to evolutionary computing* (Vol. 53). Springer.

Fogel, D. B., & Fogel, L. J. (1995). An introduction to evolutionary programming. In *European conference on artificial evolution* (pp. 21–33).

Heathon, J. (2008). *Introduction to neural networks with java*. Heathon Research, Inc.

Heiberg, A. (2013). Using bayesian networks to model a poker player. In *Workshops at the twenty-seventh aaai conference on artificial intelligence.*

Mossakowski, K., & Mańdziuk, J. (2004). Artificial neural networks for solving double dummy bridge problems. In *International conference on artificial intelligence and soft computing* (pp. 915–921).

Rigal, B. (2011). *Card games for dummies*. John Wiley & Sons.

Rubin, J., & Watson, I. (2011). Computer poker: A review. *Artificial intelligence*, *175*(5-6), 958–987.

Russell, S. J., & Norvig, P. (2016). *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited,.

Smith, S. J., Nau, D., & Throop, T. (1998). Computer bridge: A big win for ai planning. *AI magazine*, *19*(2), 93–93.

Whitehouse, D., Powley, E. J., & Cowling, P. I. (2011). Determinization and information set monte carlo tree search for the card game dou di zhu. In *2011 ieee conference on computational intelligence and games (cig'11)* (pp. 87–94).

A Spades Rules

Spades is played typically with four players (Rigal, 2011). While it's sometimes played in two teams of two players each, this rule set will assume no such teams. Furthermore, this rule guide will also assume no designated dealer is chosen, and all players deal in turn. Its played with a typical 52 card deck. A game is played in a multiple of four rounds, each existing of thirteen tricks.

Round Preparation and Bidding

To get started, the dealer shuffles the deck, and deals each player thirteen cards. Looking at their own cards, each player (in order, starting with the player left of the dealer) makes a bid. A bid is the amount of tricks a players is aiming to win.

Tricks

The player to start the first trick of a round, is again the player left to the dealer. The player is allowed to play any card of their hand. Then the other players, in clockwise fashion, all play a card of their hands. If available, all following players need to play a card of the same suit. If they have no card of the same suit, they may play any card of another suit. The trick is won by the player who played the highest card in the suit of the first card. However – as the game's namesake eludes to, spades are used as trump cards. Therefore, if a player runs out of the suit that's being called for, they can play a spade to take the trick. If spades have been played by multiple players, the player of the highest spade wins the trick. Of course, if the original suit played is spades, everyone with spades in their hand still needs to follow, and no trumping is possible.

Points and Winning

At the end of a round (thirteen tricks), every player counts their taken tricks. Spades doesn't really have a set scoring system, but players only earn points if they have taken the exact amount of tricks they bid on.

- If the bid is made exactly:

  10 points for making the bid

  5 points for every trick taken

- If the bid is not made exactly:

  -5 points for every trick taken over or under the bidding amount

The winner of Spades is the player with the most points at the end.

B Advanced fitness card scoring table

Below we find Table 2 and Table 3 with for each card their chance of winning a trick and their associated fitness score. Fitness scores are divided into whether the player in question played the first card of the trick (was 'leading') or not (was 'following'). You'll find that, because it's easier to make a trick when leading, scores for leading cards are lower. The first table, Table 2, gives us the values for each clubs, hearths, or diamonds card (all three of which are equal), whereas the second table, Table 3, gives us the values for each spades card.

Card	Win chance			Fitness score	
Rank	Leading	Following	Average	Leading	Following
A	74.51%	23.53%	49.02%	25	76
K	70.59%	21.57%	46.08%	29	78
Q	66.67%	19.61%	43.14%	33	80
J	62.75%	17.65%	40.20%	37	82
10	58.82%	15.69%	37.25%	41	84
9	54.90%	13.73%	34.31%	45	86
8	50.98%	11.76%	31.37%	49	88
7	47.06%	9.80%	28.43%	53	90
6	43.14%	7.84%	25.49%	57	92
5	39.22%	5.88%	22.55%	61	94
4	35.29%	3.92%	19.61%	65	96
3	31.37%	1.96%	16.67%	69	98
2	27.45%	0.00%	13.73%	73	100

Table 2: Chance of winning a trick and associated fitness for each club, hearth, or diamond

Card	Win chance			Fitness score	
Rank	Leading	Following	Average	Leading	Following
A	100.00%	100.00%	100.00%	0	0
K	98.04%	98.04%	98.04%	2	2
Q	96.08%	96.08%	96.08%	4	4
J	94.12%	94.12%	94.12%	6	6
10	92.16%	92.16%	92.16%	8	8
9	90.20%	90.20%	90.20%	10	10
8	88.24%	88.24%	88.24%	12	12
7	86.27%	86.27%	86.27%	14	14
6	84.31%	84.31%	84.31%	16	16
5	82.35%	82.35%	82.35%	18	18
4	80.39%	80.39%	80.39%	20	20
3	78.43%	78.43%	78.43%	22	22
2	76.47%	76.47%	76.47%	24	24

Table 3: Chance of winning a trick and associated fitness for each spade

C Normal Distribution

To get the normal distribution associated with trick taking we need to find the mean μ and the standard deviation σ. From our assumption that each player (of equal skill) takes 25% of tricks per round as the number of rounds approaches infinity, we may conclude that the mean tricks per round is $\mu = 0.25 \cdot 13 = 3.25$.

To calculate the standard deviation we can use an expected values table, which we find in Table 4. Note that the chance to take any x tricks in a round is calculated as:

$$P(x) = \binom{13}{x} \cdot 0.25^x \cdot 0.75^{(13-x)}$$

Looking in Table 4, we find our mean μ from summing up the expected values ($P(x) \cdot x$). This helps as a sanity check, because this returns the $\mu = 3.25$ that we already expected. We find our standard deviation σ by taking the square root of the variance σ^2, which is calculated by summing up the square of the difference between the value x and the mean and multiplying it with $P(x)$. We get $\sigma = \sqrt{2.4375} \simeq 1.5612$.

x	$P(x)$	$P(x) \cdot x$	$(x-\mu)^2 \cdot P(x)$
0	0.0234	0	0.2509
1	0.1029	0.1029	0.5212
2	0.2059	0.4118	0.3217
3	0.2517	0.7550	0.0157
4	0.2097	0.8388	0.1180
5	0.1258	0.6291	0.3853
6	0.0559	0.3355	0.4229
7	0.0186	0.1305	0.2621
8	4.6602e−3	0.0373	0.1051
9	8.6300e−4	7.7670e−3	0.0285
10	1.1507e−4	1.1507e−3	5.2427e−3
11	1.0461e−5	1.1507e−4	6.2829e−4
12	5.8115e−7	6.9737e−6	4.4494e−5
13	1.4901e−8	1.9372e−7	1.4165e−6
Totals (μ and σ^2)		3.25	2.4375

Table 4: Expected values table