Convolutional Neural Networks for Pattern Matching in Poker-Type Games

James M. Vrionis

Santa Cruz, California

Abstract

The imperfect information game poker is a stochastic family of card matches defined by a card-dealing and betting. Ranks assigned to singleton cards and combinations of those cards will Outcomes of each game can are dependent on the different combinations of cards relative to rank. The supposition in which the majority of poker games can be solved as pattern matching problems has inspired the creation of a strong user-agent based on unified poker representation. An iterative self-play training method that incorporates prior game actions to continually develop and optimize its proficiency in poker. This user-agent named "Poker-CNN" learns to predict the best strategy directly from the previous cards and bets without being reliant on sophisticated domain knowledge. This approach is applied to three different poker variations: single player video poker, two-player Limit Texas Hold'em and Two-player 2-7 triple draw. Poker-CNN can quickly learn patterns in these three different types of poker games while gaining experience from itself improvements ultimately reaching human expert skill level from heuristic players.

"The contributions of this paper include: (1) a novel representation for poker games, extend-able to different poker variations, (2) a convolutional Neural Network (CNN) based learning model that can effectively learn the patterns in three different games, and (3) a self-trained system that significantly beats the heuristic-based program on which it is trained, and our system is competitive against human expert players."

1. Preliminaries

- Definitions of important concepts
- Notation and Terminology
- Results from other articles that will be used in the sequel.

2. Background

Stochastic games have been a desirable field of research because skill can be determined objectively. This may be true for many games like checkers or chess whereas the many different poker variations can be thought of something way more complex.

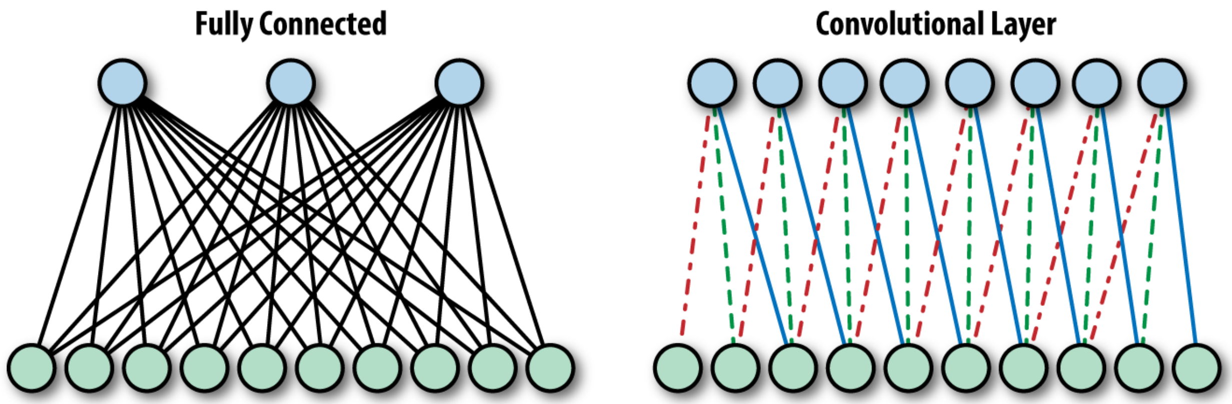

Figure 1: Fully Connected Neural Net vs CNN [1]

The weights of the convolutional neural network have a replicated structure which applies the same weights to every subrectangle of the image producing the total inputs to the next layer. The weights of a convolutional neural network are also called the convolutional kernel, K, and its size, n, is the size of the subrectangles it considers.

Convolutional neural networks form a subclass of feedforward neural networks that have special weight constraints. CNNs facilitate learning at a much higher rate of efficiency given fewer examples when two-dimensional translation invariance is evident. It's important to realize a fully connected feedforward neural network has a very large number of parameters when compared to a CNN of the same size. Small number of weights make parameter estimation a necessity to determine if a good can be found quickly. [2]

$$y_{x,y} = \left(1 + exp\left(-\sum_{u=-(n-1)/2}^{(n-1)/2} \sum_{v=-(n-1)/2}^{(n-1)/2} x_{x+u,y+v} K_{u,v}\right)\right)$$

where y is the activity of a hidden unit at position (x, y), and x are the input units. [2]

The objective of Poker-CNN is to use CNN and a ranking system applied to three tensor-based frameworks. This will develop a general learning model designed to teach itself how to play and in time, dominate professional human players at many different types of poker games.

3. Related Work

Counterfactual Regret Minimization is used as a solution to heads-up No Limit Texas Hold'em. Given some game state, this method will examine every possible outcome to eventually reach an equilibrium solution using different search strategies. Domain knowledge and massive amounts of data are necessary for CFRs to develop an effective user-agent. It's important to mention on of the most successful implementations of a CFR based Neural Network. DeepStack [3], a CFR based solution to Texas Hold'em, does carry the title of the first computer program to defeat professional poker players at heads-up No-limit Texas Hold'em (NLTH), an imperfect information game with over 10^{160} games states. [4]

The process uses recursive reasoning of CFR to handle information asymmetry. Prior to play DeepStack does not compute and store a complete strategy. Instead, situations are considered in real time, examining each particular situation accordingly. This is accomplished using a standard feed-forward network containing seven fully connected hidden layers each containing 500 nodes and parametric rectified linear units (28) for the output. DeepStack takes pot-size and players ranges as a function of the public cards as inputs and will output vectors of counterfactual values for each player and hand as fractions of the pot size. This is the first computer program to beat professional poker players in No-Limit Texas Hold'em, it's approach dramatically reduces worst-case exportability when compared to the abstraction paradigm of Poker-CNN.

An important distinction that must presented is the difference between perfect and imperfect information games. Games like chess and backgammon can be defined as perfect information games because both players know the state of the game at all times. When any type of action is taken or performed the true nature of this action is known to all other participants involved whereas imperfect information can are simultaneous in nature. Each new game state is can be regarded as the starting state of a new game. By comparison, strategies for a perfect information game is much easier to implement, especially when defined relative to each decision point.

Value-based reinforcement learning such as TD- and Q-learning are powerless towards imperfect counterparts. With that in mind a gradient search performed on the parameter spaces simultaneously shift probability distributions in a more successful outcome. This special case of the lagging anchor algorithm promotes much better results. [5]

4. Games of Poker

Video Poker

Video poker (VP) is a single-player game of draw poker, easily found in many casinos. The house randomly deals five cards for a 1$ *dollar fee and shortly after one can either keep their current cards or trade-in some number of cards (1-5) from one to five for a new set of card(s). The objective of VP is simple: match the best possible hand of five cards Given 32 possible choices, based on the players, final 5 cards where payout is calculated by a table of matches..* [6]

Limit Poker

(heads-up limit) Texas Hold'em (LTH): Heads-Up limit Texas Hold'em, is two player card game lasting four rounds with no more than four bets per round. Both players have a fixed amount of money known as stack size at the start of each game. An ante is required to start every game and is considered the first round of betting where one player is responsible for ***small blind*** *whilst the other must pay the* ***big blind*** *(a max of three additional bets during this round). Private and public cards are dealt and if the players make it through 3 more rounds a showdown will take place which is where both players end up showing each other their hands where there is only one winner..* [4].

2-7 Triple Draw

Just like (heads-up limit) Texas Hold'em, 2-7 Triple Draw Poker is a multi-round game capable of many players. This game is like a mixture of the previous two games, which means this variation of poker would be the most difficult to develop a model for that would be competitive against professional human players. Just like LTH this game has multiple rounds of betting but in this game you can also choose to trash some number of your cards or keep them. Another difference this game exhibits is low card is high, in other words a low card straight would be the best hand. [6].

5. Poker Relational Model: URP

Helper function to turn a poker hand (array of cards) into 2D array. if pad_to_fit then pass along to card input creator, to create $14x14$ *array instead of* $4x13$ *NOTE:* $17x17$ *padding! NOTE: Double_row* $= T$ *expands to* $8x13$ *by repeating suit row. Full order: CDHS CHDS. Idea is that any pair can be learned with a single convolution. (Any two suit rows together.).*

```python
HAND_TO_MATRIX_PAD_SIZE = 17
DOUBLE_ROW_HAND_MATRIX = True # False # True # False # Set true for 8x13 matrix, with redundancy.
remap_suit = {CLUB:CLUB, HEART:DIAMOND, DIAMOND:HEART, SPADE:SPADE}
def hand_to_matrix(poker_hand, pad_to_fit=False, pad_size=HAND_TO_MATRIX_PAD_SIZE, double_row=DOUBLE_ROW_HAND_MATRIX):
    # initialize empty 4x13 matrix
    # Unless pad to fit... in which case pad to 17x17
    if pad_to_fit:
        matrix = np.array([[0 for x in range(pad_size)] for x in range(pad_size)], np.int32)
    else:
        matrix = np.array([[0 for x in range(len(ranksArray))] for x in range(len(suitsArray))], np.int32)
    if pad_to_fit:
        if pad_size == 17:
            # add 5 empty rows to start, and 5 empty rows to finish
            suit_offset = 6
            # add empty column to start
            value_offset = 2
        elif pad_size == 15:
            suit_offset = 5
            value_offset = 1
        if double_row:
            suit_offset -= 2
    else:
        suit_offset = 0
        value_offset = 0

    for card in poker_hand:
        #print card
        #print ([suits_to_matrix[card.suit]], [card.value])
        matrix[suits_to_matrix[card.suit] + suit_offset][card.value + value_offset] = 1

    # If double row, now copy rows
    if double_row:
        suit_offset += 4
        for card in poker_hand:
            matrix[suits_to_matrix[remap_suit[card.suit]] + suit_offset][card.value + value_offset] = 1

    return matrix
```

Listing 1: Unified Representation for Poker Games

[7]

6. Poker Games

Big Filters

CNN model Poker-CNN, two convolutional layers, one maxpooling layer, two more convolutional layers, one more maxpooling layer ,one dense layer, dropout layer, filter size is 5×5

Model with 5x5 filter on the bottom for Better visualization while tracking all the layers creating and this will return a full stack.

```
def build_fat_model(input_width, input_height, output_dim,
                    batch_size=BATCH_SIZE, input_var = None):
    print('building fat model, layer by layer...')
    num_input_cards = FULL_INPUT_LENGTH
    layers = []
    l_in = lasagne.layers.InputLayer(
        shape=(batch_size, num_input_cards, input_height, input_width),
        input_var = input_var,
        )
    layers.append(l_in)
    print('input layer shape %d x %d x %d x %d'
              % (batch_size, num_input_cards, input_height, input_width))
    l_conv1 = lasagne.layers.Conv2DLayer(
        l_in,
        num_filters=NUM_FAT_FILTERS,
        filter_size=(5,5),
        nonlinearity=lasagne.nonlinearities.rectify,
        W=lasagne.init.GlorotUniform(),
        )
    layers.append(l_conv1)


    l_conv2_2 = lasagne.layers.Conv2DLayer(
        l_conv2,
        num_filters=NUM_FAT_FILTERS*2,
        filter_size=(3,3),
        nonlinearity=lasagne.nonlinearities.rectify,
```

```
        W=lasagne.init.GlorotUniform(),
        )
    layers.append(l_conv2_2)
    print('convolution layer l_conv2_2. Shape %s' % str(l_conv2_2.output_shape))

    # Skip second max-pool.
    l_hidden1 = lasagne.layers.DenseLayer(
        l_conv2_2, #l_pool2,
        num_units=NUM_HIDDEN_UNITS,
        nonlinearity=lasagne.nonlinearities.rectify,
        W=lasagne.init.GlorotUniform(),
        )
    layers.append(l_hidden1)

    print('hidden layer l_hidden1. Shape %s' % str(l_hidden1.output_shape))

    l_hidden1_dropout = lasagne.layers.DropoutLayer(l_hidden1, p=0.5)
    layers.append(l_hidden1_dropout)

    print('dropout layer l_hidden1_dropout. Shape %s' % str(l_hidden1_dropout.output_shape))

    l_out = lasagne.layers.DenseLayer(
        l_hidden1_dropout,
        num_units=output_dim,
        nonlinearity=lasagne.nonlinearities.rectify, # Don't return softmax! #nonlinearity=lasagne.nonlinearities.softmax,
        W=lasagne.init.GlorotUniform(),
        )
    layers.append(l_out)

    print('final layer l_out, into %d dimension.
                                    Shape %s' % (output_dim, str(l_out.output_shape)))
    print('produced network of %d layers. TODO: name \'em!' % len(layers))
    return (l_out, l_in, layers)
```

Listing 2: Big Filter Encoding

[7]

Small Filters

Deep structure with small filters. The size is 3×3 convolutional layers.

```
    print('convolution layer l_conv1. Shape %s' % str(l_conv1.output_shape))
```

```
    l_conv1_1 = lasagne.layers.Conv2DLayer(
        l_conv1,
        num_filters=NUM_FAT_FILTERS,
        filter_size=(3,3),
        nonlinearity=lasagne.nonlinearities.rectify,
        W=lasagne.init.GlorotUniform(),
        )
    layers.append(l_conv1_1)
    print('convolution layer l_conv1_1. Shape %s' % str(l_conv1_1.output_shape))

    l_pool1 = lasagne.layers.MaxPool2DLayer(l_conv1_1, pool_size=(2, 2),
                                            ignore_border=False)
    layers.append(l_pool1)
    print('maxPool layer l_pool1. Shape %s' % str(l_pool1.output_shape))

    l_conv2 = lasagne.layers.Conv2DLayer(
        l_pool1,
        num_filters=NUM_FAT_FILTERS*2,
        filter_size=(3,3),
        nonlinearity=lasagne.nonlinearities.rectify,
        W=lasagne.init.GlorotUniform(),
        )
    layers.append(l_conv2)
    print('convolution layer l_conv2. Shape %s' % str(l_conv2.output_shape))
```

Listing 3: Small Filter Encoding

[7]

7. Inputs

```
class TripleDrawAIPlayer():
    def __init__(self):
        self.draw_hand = None
        self.name = ''
        self.tag = ''
        self.output_layer = None # for draws
        self.input_layer = None
        self.holdem_output_layer = None # For Texas Hold'em
        self.holdem_input_layer = None
        self.bets_output_layer = None
        self.bets_input_layer = None
```

```
        self.use_learning_action_model = False
        self.old_bets_output_model = False
        self.other_old_bets_output_model = False
        self.bets_output_array = []
        self.use_action_percent_model = False
        self.is_dense_model = False
        self.is_human = False

        # Special cases for NLH
        self.imitate_CFR_betting = False # Try to check/bet at ration learned from CFR
    training??
```

Listing 4: Inputs to CNN

[7]

This is the CNN model: but its apparent running multiple models would not be to hard. This is the draw model. Also, outputs the heuristic value of a hand given a number of draws. It is also possible to use multiple models and feed on model into another. This is the main CNN of the whole project. The total code representing the CNN layers is over 2000 lines code. One important take away from the code block is how many layers are needed to represent this model.

Training model. Iterative refining which is defined as playing against itself and previous two iterations of trained the model. Informally this is done to minimize practicing mistakes. Formally, this form of self-play iterative refinement forces the current model to excel against all previous versions of itself independent of its known exploitable weaknesses.

Explain the gain/loss using the Evaluating Real-Time Strategy Game States using CNNs. The concept the model will show significant improvement in accuracy over simpler state-of-the-art evaluations resulting learned evaluation function into state-of-the-art RTS search algorithms increases agent playing strength considerably.

```
def encode_limit_bets_string(actions):
    actions_string = ''
    for action in actions:
        if action.type in ALL_BETS_SET:
            actions_string += '1'
        elif action.type == CHECK_HAND or action.type in ALL_CALLS_SET:
            actions_string += '0'
        else:
            continue
    return actions_string
```

Listing 5: Limit Texas Hold'em betting actions

[7]

Check or calling action will be represented by a 0 and a bet or raising action will be represented by a 1. Ignore all other actions and don't deal with non-bets.

```
def pot_to_array(pot_size, pad_to_fit = True):
    pot_to_cards = []
    for rank in ranksArray:
        for suit in suitsArray:
            card = Card(suit=suit, value=rank)
            if pot_size >= 50:
                pot_to_cards.append(card)
                pot_size -= 50
            else:
                break
        if pot_size < 50:
            break
    pot_size_card = hand_to_matrix(pot_to_cards, pad_to_fit=pad_to_fit)
    return pot_size_card
```

Listing 6: Limit Texas Hold'em betting actions

[7]

For limit Texas Hold'em game we assumes 50.0 step and single-precision $4x13$ matrix inputs

Encode pot (0 to 3000 or so) into array by using a faking

Every \$50 of pot is another card so $50 \rightarrow [2c]$, $200 \rightarrow [2c, 2d, 2h, 2s]$

8. Round of Betting

```
def string_ends_big_bets_round(actions_string):
    if not actions_string:
        return False
    if len(actions_string) >= 2 and actions_string[-1] == 'k' and actions_string[-2] == 'k':
        return True
    if len(actions_string) >= 2 and actions_string[-1] == 'c':
        return True
    if len(actions_string) >= 2 and actions_string[-1] == 'k' and actions_string[-2] == 'c':
```

```
        return True
    return False
```

Listing 7: Betting Actions

[7]

Check to see if we are still in a betting round. Then check if opponent has checked, call if our opponent has bet (excluding initial bets i.e., blinds) or check a called bet.

```
def simulate_allin_vs_random(self, num_samples = SIMULATE_ALLINS_COUNT, allin_cache=None):
    if not(self.format == 'holdem' or self.format == 'nlh'):
        return
```

Listing 8: Hand Strength

[7]

Simulate "allin value" vs. "random opponent hand."

Is the hand good or is the hand bad? That is what this code will be testing.

9. Monte Carlo Result Scale

```
MONTE_CARLO_RESULTS_SCALE = 1.0 # 0.1
POT_SIZE = 14
BET_FACED = 15
STACK_SIZE = 16
FOLD_PERCENT = 17 # Same as aggressive%, we need to estimate how often good training data (
    CFR) folds in this similar spot?
```

Listing 9: pokerlib.py: 222:226

[7]

NOTE: If attempting No-limit Hold'em and using AWS for computations, make sure to request and save output bet sizes. compare the results from the bets, inputs and outputs in order to understand if your simulation results are accurate and you user-agent is correctly building off previous less optimal versions of itself.

```
ALLIN_VS_OPPONENT = 18
```

Listing 10: pokerlib.py: 232

[7]

If available for high hand (Limit Texas Hold 'em and No Limit Texas Hold'em) try outputting odds from the Monte Carlo, 0.0-1.0, which is a description of hand strength in theory and in practice.

We predict the expected values of the five output states Match it with one of the five betting actions from To collect average return of every betting choice, chip values are tracked. Folding will assign no value either way (initially). Include future loss(es) or gain(s) directly related to this choice will allow better data.

10. Iterative Refining

500,000 generated hands to train this ideology. First,learn"Draws"Network,then use resulting parameter values as initializer for "Bets" Network. Self-play with varying examples. Current version of Poker-CNN will play against as far back as 2 previous versions. Final Poker-CNN for TH will train for 8 self-play epochs. Final Poker-CNN for TD will train for 20 self-play epochs. Minimizes the chance of practicing mistakes.

11. Video Poker Implementation

```
class OneRuleNaivePlayer(CardPlayer):
    def move(self, hand, deck):
        rank = hand_rank_five_card(hand.dealt_cards)
        category = hand_category(rank)
        if category in set([ROYAL_FLUSH, STRAIGHT_FLUSH, FOUR_OF_A_KIND, FULL_HOUSE, FLUSH,
    STRAIGHT]):
            print('--> pat hand. Take our bonus.')
            # Now, complete the [empty] draw!
            discards = hand.draw('')
            deck.take_discards(discards)
            new_cards = deck.deal(len(discards))
            hand.deal(new_cards, final_hand=True)
            return jacks_or_better_table_976_9_6[category]

        values_count = {value: 0 for value in ranksArray}
        suits_count = {suit: 0 for suit in suitsArray}
        for card in hand.dealt_cards:
            values_count[card.value] += 1
            suits_count[card.suit] += 1
        #print(values_count)
        #print(suits_count)

        has_four_flush = False
        has_three_card_royal = False
        for suit in suitsArray:
```

```
            if suits_count[suit] > 3:
                print('--> we has a flush draw')
                has_four_flush = True
            elif suits_count[suit] == 3 and all((card.value in royalRanksSet) or (card.suit != suit) for card in hand.dealt_cards):
                print('--> we has a 3-card royal flush draw')
                has_three_card_royal = True

        draw_string = ''
        if category in set([THREE_OF_A_KIND, TWO_PAIR, JACKS_OR_BETTER]):
            print('good pair+ hand. Keep the pair and freeroll...')
            # Toss any cards not part of a pair/trips
            for i in range(0,5):
                card = hand.dealt_cards[i]
                if values_count[card.value] < 2:
                    #print('Card not part of pair+ %s' % card)
                    draw_string += '%d' % i

        if not draw_string and (has_four_flush or has_three_card_royal):
            print('drawing for a flush')
            for i in range(0,5):
                card = hand.dealt_cards[i]
                if suits_count[card.suit] < 3:
                    #print('Card not part of flush draw %s' % card)
                    draw_string += '%d' % i

        if not draw_string and category in set([ONE_PAIR]):
            print('small pair hand. Keep the pair and freeroll...')
            for i in range(0,5):
                card = hand.dealt_cards[i]
                if values_count[card.value] < 2:
                    draw_string += '%d' % i

        if not draw_string:
            draw_string = '01234'

        # Now, complete the draw!
        discards = hand.draw(draw_string)
        deck.take_discards(discards)
        new_cards = deck.deal(len(discards))
        hand.deal(new_cards, final_hand=True)

        expected_payout = jacks_or_better_table_976_9_6[category]
```

```
        if expected_payout == 0:
            expected_payout = RANDOM_PAYOUT
        return expected_payout
```

Listing 11: VP

[7]

One player with one draw. Keep any pair or better; Otherwise burn all 5 *cards and re-draw. The new cards will be random, not the draw. Check what you got and then complete the process. Next, the frequency for all the ranks and suits is computed.*
Note: *If no big pair or flush draw, keep a small pair (like 33) .*

11.1. Video Poker

Perfect player vs Heuristic player vs. random player All learning models outperform the random action player and the heuristic player.
It was discovered that CNNs make fewer "big" mistakes than Deep Neural Networks (DNN)s. CNNs view the game knowledge as patterns which gives the CNN model a clear advantage of other types of neural networks and makes sense that our CNN is better suited to learn the game via pattern analysis of 2D tensors vs DNNs in 1D space.

11.2. Limit Texas Hold'em

TH: Poker-CNN vs. a Random Player, heuristic player and open source CFR player Former professional poker players to compete 500 hands because of CFR used has known weaknesses Resulting in a competitive model vs human expert.

For example, in limit poker games such as heads-up limit Texas hold, em, the number of information sets can be easily calculated with the single closed-form expression

11.3. 2-7 Triple Draw

CNN player (after 20 iterations of self-play and retraining), and a DNN, trained on the results of the Poker-CNN model. No public 2-7 TD AI currently existed to compete against an expert and champion were asked to play 500 hands against Poker-CNN. In both cases the Poker-CNN model fell short of winning.

12. conclusion

A developed user-agent called Poker-CNN demonstrated a general Deep Convolution Neural Network that worked on three **very** different variations of this imperfect information game. Really studying this material a major red flag arose. The original creator really underrepresented their data. In reference to the Limit Texas Hold'em and 2-7 Triple draw: they have such a more complex game state by comparison to Video Poker that even someone that 2-7 Triple draw: 500 hands to test the Poker-CNN developed for this model. For a heuristic trained agent, this really showed how a generalized self-learning model has room for improvement and could excel in all forms of this immensely large game state.

13. Acknowledgments

my references: [3, 8, 9, 10, 6, 4, 11, 5, 12, 2, 1, 7, 1]

References

1. Lieder I, Resheff YS, Hope T. Learning tensorflow 2018;URL: `https://www.safaribooksonline.com/library/view/learning-tensorflow/9781491978504/assets/letf_0401.png_2018`; [Online; accessed March 17, 2018].

2. Sutskeve I, Nair V. Mimicking go experts with convolutional neural networks 2008;.

3. Moravcík M, Schmid M, Burch N, Lisý V, Morrill D, Bard N, Davis T, Waugh K, Johanson M, Bowling MH. Deepstack: Expert-level artificial intelligence in no-limit poker. *CoRR* 2017;abs/1701.01724. URL: `http://arxiv.org/abs/1701.01724`. `arXiv:1701.01724`.

4. Johanson M. Measuring the size of large no-limit poker games. *CoRR* 2013;abs/1302.7008. URL: `http://arxiv.org/abs/1302.7008`. `arXiv:1302.7008`.

5. Dahl FA. A reinforcement learning algorithm applied to simplified two-player texas hold'em poker. In: De Raedt L, Flach P, eds. *Machine Learning: ECML 2001*. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN 978-3-540-44795-5; 2001:85–96.

6. Yakovenko N, Cao L, Raffel C, Fan J. Poker-cnn: A pattern learning strategy for making draws and bets in poker games. *CoRR* 2015;abs/1509.06731. URL: `http://arxiv.org/abs/1509.06731`. `arXiv:1509.06731`.

7. Yakovenko N. Deep draw 2016;URL: `https://github.com/moscow25/deep_draw`; [Github; poker lib].

8. Nicolai G, J. Hilderman R. Algorithms for evolving no-limit texas hold'em poker playing agents. 2010;:20–32.

9. Simard PY, Steinkraus D, Platt J. Best practices for convolutional neural networks applied to visual document analysis 2003;URL: https://www.microsoft.com/en-us/research/publication/best-practices-for-convolutional-neural-networks-applied-to-visual-document-analysis/.

10. Stanescu M, Barriga NA, Hess A, Buro M. Evaluating real-time strategy game states using convolutional neural networks 2016;:1–7doi:10.1109/CIG.2016.7860439.

11. Heinrich J, Silver D. Deep reinforcement learning from self-play in imperfect-information games. *CoRR* 2016;abs/1603.01121. URL: http://arxiv.org/abs/1603.01121. arXiv:1603.01121.

12. Tesauro G. Td-gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation* 1994;6(2):215–9. doi:10.1162/neco.1994.6.2.215.