Applying Machine Learning to a Conventional Data Processing Task—A Quantitative Evaluation

Wolfgang Pree
University of Salzburg
Department of Computer Sciences
Jakob-Haringer-Str. 2, 5020 Salzburg, Austria/Europe
+43 662 8044 – 6488
Wolfgang.Pree@cs.uni-salzburg.at

Felix Hoerbinger
University of Salzburg
Department of Computer Sciences
Jakob-Haringer-Str. 2, 5020 Salzburg, Austria/Europe
+43 662 8044 – 6488

ABSTRACT
Though machine learning (ML) can be applied to a wide spectrum of applications, it has been hardly used and evaluated in the context of conventional data processing tasks. Such conventional data processing tasks are characterized by a set of calculations that follow strict rules, such as in accounting or banking applications. This paper quantitatively evaluates how software which is automatically generated by ML methods and tools compares to software programmed by hand. The assessment of poker hands according to Texas Hold'em rules is a representative example for conventional data processing tasks, because of the various exceptions how to assess and compare hands. For some hand values, the rank (two, three, ... king, ace) of the cards is relevant and the suit (club, diamond, heart, spade) irrelevant, and vice versa. This paper shows how an accuracy of 100% can be achieved for assessing poker hands according to Texas Hold'em rules, with a small set of labeled training data compared to the number of possible hands. We also evaluate quantitatively the effect of the labeling quality on accuracy.

CCS Concepts
• Software and its engineering → Software creation and management → Software development techniques → Automatic programming.

Keywords
Machine learning; supervised learning; neural networks; convolutional neural networks; feed-forward neural networks; data labeling; labeling quality; robustness.

1. INTRODUCTION
The essence of supervised Machine learning (ML) is that *data write programs*. Instead of describing in pain-staking detail how to process input data to obtain correct results (see Figure 1a), examples of correct results (so-called labeled data) are used for neural network (NN) training. The adjusted weights and other parameters of the particular NN represent then the 'program' which delivers hopefully the correct results, not only for the training data but for all possible input data. Figure 1b illustrates this inverted world view where the program results from (labeled) data. As the training of a NN is automated, this schematic representation of *data write programs* illustrates how NNs deliver the vision of automated programming.

One typical application of the supervised ML approach is in image processing, which touches several different domains, such as the medical diagnosis of Magnetic Resonance Imaging (MRI), the natural language description of images, or autonomous driving. It turned out that the classical way of programming image processing (Figure 1a) is significantly inferior to NNs (Figure 1b) so that since a few years image processing relies only on NNs.

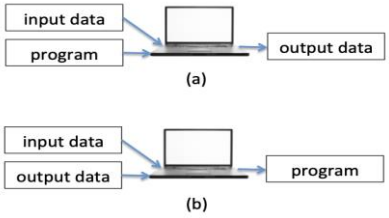

Figure 1. (a) Conventional programming paradigm versus (b) ML—data write programs.

Sometimes it is argued that ML is adequate for tasks such as object recognition, which are straight-forward for biological NNs, in particular, human brains, but not for conventional computing tasks, such as calculations. Though humans usually have no problem recognizing objects in images, most humans would struggle, for example, to compute the exact result of complicated calculations, let's say the multiplication of two 6 digit numbers, without extra help (calculator, or paper and pen). Conventional programming, that is, expressing in a programming language the details how input should be processed to obtain the expected results, is regarded as adequate means for most data processing tasks, if the algorithms, processing rules/constraints and calculations are straight-forward and known. One reason is that data processing is required to be 100.0% correct, not just close to 100 percent.

Both the source code for generating the labeled data as well as the Colaboratory [1] notebooks used in the evaluation are available for download [2].

2. TEXAS HOLD'EM HAND VALUES
Texas Hold'em [3] is a popular variant of the poker card game with exactly defined rules. The card deck consists of 52 cards: 13 cards of each suit (club, diamond, heart, spade). A hand in Texas Hold'em consists of five cards. Thus, if one draws a hand out of

ICMLC 2020, February 15–17, 2020, Shenzhen, China

ACM ISBN 978-1-4503-7642-6/20/02…$15.00
DOI: https://doi.org/10.1145/3383972.3384042

the 52 cards, overall 2,598,960 (almost 2.6 million) possible hands can be drawn. No card is put back once it is drawn, thus we calculate 5-permutations of 52 as 52 x 51 x 50 x 49 x 48 / (5 x 4 x 3 x 2 x 1).

A hand has exactly one out of ten values: high card, one pair, two pairs, three of a kind, straight, flush, full house, poker, straight flush, and royal flush. For some values, only the card suit is relevant. For example, a hand is a straight, if the ranks (two, three, four, five, six, seven, eight, nine, ten, jack, queen, king, ace) are in ascending order and adjacent. So a hand with a seven, eight, nine, ten and jack card is an example of a straight (= has the value 'straight') independent of the cards's suits. In case of a straight, the ace (which is usually the highest rank) might also count as rank one. Thus, ace, two, three, four, five is also a straight.

A hand is a flush if all five cards are of the same suit. The rank of each of the cards is irrelevant, except for the assessment whether the flush is a straight flush or royal flush. A hand is a straight flush, if it is a flush and a straight. A hand is a royal flush if it is a straight flush with the ace as highest card, that is, the cards in the hand have all the same suit and the ranks ten, jack, queen, king, ace. A hand has the value ‘quads’ if it has four cards of the same rank. A hand has the value 'three of a kind' if the hand contains three cards of the same rank. A hand has the value 'one pair' if the hand contains two cards of the same rank. A full house is a hand with a pair and three of a kind. Two pairs is the value of a hand with two pairs. All hands other than the ones described above have the value high card.

3. HAND ASSESSMENT

Various implementations of Texas Hold'em hands are available on the web. A dataset consisting of approximately one million labeled hands is available in the UCI Machine Learning Repository as Poker Hand Data Set [4]. The application of neural networks on this dataset has been investigated in [5, 6]. For our purpose, as we try to meaningfully investigate generalization properties of different models, we choose a smaller dataset with a total of 4 thousand poker hands. Furthermore, the extremely uneven distribution of hand values was ignored by generating an equal amount of hands for each value but could lead to interesting research in the future. The test set also is evenly distributed among the values, with a total of 240k poker hands.

To give some insight into the task at hand, we briefly introduce our implementation of the hand assessment. We define a two-dimensional array as core data structure for representing the five cards of a hand. This kind of array allows an elegant and lean implementation. Figure 2 exemplifies the hand representation by means of a hand that has two pairs. For the sake of readability, we omit the zeros in the array.

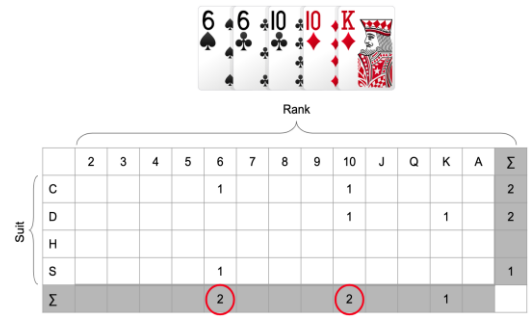

Figure 2. Representation of a sample hand (two pairs).

A one in the suit rows/rank columns indicates a particular card. For example, the diamond king is represented by the 1 in the second row D (for diamond) and column K (for king). The gray cells contain the sums of the rows and columns. The column sum is relevant for assessing hand values for which the rank is the decisive criterion. In the sample hand in Figure 2, the two 2s, marked with a red circle, indicate the hand value of two pairs. A poker, for example, would have a 4 in one of the column sum cells, a three of a kind a 3, and a full house a 3 and a 2. Whereas a flush has a 5 in one of the four row sum cells.

We call the 2-dimensional Java array cardsCalculator. For detecting a Flush or Quads, one simply has to look for a 5 in the rightmost column or a 4 in the lowest row respectively. More challenging is an elegant implementation of method isStraight(). Basically, it has to check whether five consecutive 1s are in the column sum row. Additionally, we have to consider the special case that an ace is used with rank value one. Figure 3 shows the implementation of isStraight().

```
private boolean isStraight() {
   String sumOfColsStr="";
   // put the number of aces as first digit in the string:
   sumOfColsStr+= new Integer(cardsCalculator[4][12]).toString();
   for (int col= 0; col < 13; col++)
      sumOfColsStr+= new Integer(cardsCalculator[4][col]).toString();
   // assertion: 'sumOfColsStr' is the bottom ∑ line of cardsCalculator
   // as string, with the number of aces (sum of column 'A') as
   // first digit and as last digit
   return sumOfColsStr.contains("11111");
}
```

Figure 3. Java method isStraight().

Finally, method assessHand() calls the various methods in the appropriate order to determine a hand's value. Figure 4 shows method assessHand().

```
private HandVal assessHand() {
        if (isStraight() && isFlush() && cardsCalculator[4][12] == 1
        && cardsCalculator[4][0] != 1)
                return HandVal.ROYALFLUSH;
        if (isStraight() && isFlush())
                return HandVal.STRAIGHTFLUSH;
        // ... analogous for the other hand values
        if (isHighCard())
                return HandVal.HIGHCARD;
        return HandVal.NOTVALID;
}
```

Figure 4. Java method assessHand().

The method assessHand() is also a private method, which the Hand constructor calls. The returned hand value is stored in an instance variable that has a getter method.

JUnit tests round out the Java implementation consisting of the classes Hand, Card, and Deck as well as the enumerations Suit, Rank and HandVal. This Java implementation is also used for generating labeled data for an ML-based hand assessment.

3.1 Input Formats

As this problem is a playground for testing different methods rather than an actual application, we try different input formats that can lower or increase its complexity, especially in two ways: using an input format that is invariant to the permutation of the five hand cards, as is the problem itself, will reduce the number of possible inputs by a factor of 5!=120 while significantly reducing the complexity for the neural network to learn. Also, we can

introduce an encoding that explicitly takes into account the suits and ranks of each card, also reducing the complexity of the task.

The single cards are represented in two ways: Once, simply as a one-hot encoded vector with 52 bits, and once with a separate encoding for rank and suit, giving a vector of length 4+13=17. By using all five cards separately as input, giving a total input dimension of 5x52 or 5x17 respectively, we ignore permutation invariance. We can introduce it directly into our encoding by summing up the vectors of the five individual cards in either one of the representations above. In the first case, this leads to 52 bits, of which 5 will be 1 and the rest 0. In the second case, we will basically get the lowest row and rightmost column from Figure 2, which should be particularly easy to learn for the network.

As for the input, the encoding of the target can also be varied. The most straightforward representation is a one-hot encoding of the ten possible values, as described above. This introduces some problems regarding the extreme imbalance in the distribution of hands. Out of the 2,598,600 hands, only 4 are royal flush and 36 are straight flush, not even enough to create a proper training and test set. We can instead evaluate a hand by all values it could possibly be asserted, not just by the highest. A full house for example also classifies as a pair, two pairs and threes. A straight flush is both straight and flush, eliminating the need to explicitly represent it in our target labels. Also, we no longer take into account the royal flush explicitly, as it would be covered as the straight flush with the highest possible tiebreakers. As any hand is at least high card, this value can also be ignored and we are left with 7 bits representing the values from pair to four of a kind, straight flush now identified by straight + flush. Besides alleviating the problem of extreme imbalance, this further introduces additional information for most labels, making training slightly more sample efficient. Unless otherwise specified, the following experiments use a training set of 2400 hands, 300 for each of the eight values that have to be treated explicitly.

3.2 Overparameterization

In all the experiments that will be laid out in the following sections, the benefits of extreme overparameterization to the generalization error were clearly notable. For a training set of 2400 hands, a very small network, for example of two dense layers with 32 units each and ReLU activation followed by the output layer, is sufficient to obtain 100% training accuracy. However, the generalization properties of such a network are relatively poor, giving approximately 65% test accuracy. It has been shown empirically that overparameterized networks, that is networks with many more degrees of freedom than necessary to achieve zero error on the training set, often lead to much better generalization [7, 8]. While not yet completely understood, this phenomenon seems to be particularly strong when combined with the ReLU activation function [9].

For the problem of poker hand assessment, we tried to take advantage of this improvement in generalization in several ways starting from the three-layer network mentioned above. Adding more layers did not improve the generalization error in any of the experiments, while making the hidden layers wider, that is adding more units, did. The most effectively generalizing network we discovered during our experiments had three hidden layers with upwards of 2000 units each, and approximately 9 million parameters in total. Further increasing the number of hidden units did not notably improve generalization and the same results were true for each input format and number of training data.

Figure 5 illustrates the improvement through overparameterization for a three-layer network with 2.4k labeled training hands. Each data point represents the average of several runs to somewhat account for random initialization and, importantly, each of those networks gave zero training error. The input format in this case were 52 bits specifying the presence of a given card, thus including permutation invariance but not the explicit split into suit and rank. For all the widths tested in Figure 5, both increasing or decreasing the number of layers deteriorated the test results.

We use Tensorflow [10] for all experiments throughout this work and optimize the models with the Adam Optimizer [11]. We did not notice any improvement in generalization by switching to stochastic gradient descent, as is sometimes found in different problems [12].

While adding more dense layers to the model deteriorated the problem, we further found that we could stack many more convolutional layers [13]. To this end, we first tried to represent the input data as an image with 5 rows and 17 columns and apply the convolutions, which works surprisingly well despite the neither translational invariant or covariant input. A dense layer followed by reshaping the features into a two-dimensional map worked even better and close to the overparameterized dense network. This could even be improved slightly by instead reshaping into a three-dimensional feature map and consequently using 3D-convolutions. While the very wide dense layers did end up with the best generalization, this could be an interesting approach for more complicated problems that require deeper networks.

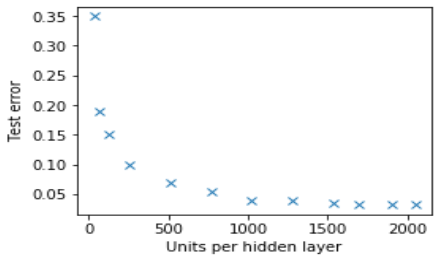

Figure 5. Test error depending on the degree of overparameterization.

3.3 Rank and Suit Encoding

The explicit encoding of rank and suit allows for a more easily processable representation of the cards, significantly decreasing the test error when trained with an equal amount of training data. The assumption that this representation would be easier to learn was confirmed throughout all of our experiments, with the 17 bit representation consistently outperforming the 52 bit representation.

Instead of using an integer between 0 and 51 for representing a card, bits that one-hot encode both suit and rank are introduced: four bits for a card's suit, and 13 bits for a card's rank, thus, overall 17 bits. For a specific card, all bits are initialized with 0. For the suit we define the order CDHS. Depending on the card's suit, the corresponding bit is set to 1. For example, a spade would be represented by 0001, a diamond by 0100. In an analogous way the rank of a card is represented in the 13 bits for the rank. If the card has rank 2, the first bit of the 13 bits is set to one, for rank 3 the second bit, and so on. For example, a card with suit diamond and rank ace would be represented by the following 17 bits: 0100 0000000000001 (for better readability we intentionally added a

space after the four bits representing the suit). This representation has also been used in [6].

3.4 Permutation Invariance

Introducing an input representation that explicitly takes into account invariance to permutations of the five hand cards makes the problem significantly easier to learn. The most straightforward way to introduce invariance with respect to permutations of the five cards is to sum up their encoding vectors. This works for single card representations introduced, resulting in one vector of length 52 or 17 respectively that represents the complete hand. Note that for the case where suit and rank are encoded separately, this sum directly gives the rightmost column and lowest row of the table in Figure 2, making this representation presumably particularly easy to learn.

As expected, the performance of a neural network increases drastically with such an invariant representation. However, we can also introduce permutation invariance in another way, namely by randomly permuting the cards in the input to an otherwise not permutation invariant network. This is in some ways similar to data augmentation, for example in image recognition, where random rotations of the input are supposed to train networks somewhat invariant to rotations. One might expect this to be harder to learn than having the permutation invariance explicitly encoded by a simple sum. In all of our experiments, except where the training set was sufficiently large to achieve almost zero test error, this second method of training outperformed the one with the permutation invariant hand representation. Table 1 summarizes these results.

Table 1. Test accuracy for both single card representations (52 and 17 bits), with or without permutation invariant input and with number of training data n.

	n=800	n=1600	n=2400
52 bits, summed	63.7 ±0.5%	86.9 ±1.5%	96.8 ±0.2%
52 bits, not summed	70.7 ±1.4%	95.5 ±0.2%	98.0 ±0.05%
17 bits, summed	92.1 ±0.4%	99.0 ±0.2%	99.9 ±0.02%
17 bits, not summed	97.4 ±0.4%	99.6 ±0.1%	99.7 ±0.02%

We also note that the networks that were trained on permutations of the non-summed hand representations do not arrive at an actually permutation invariant solution. When presented with all 120 permutations of a single hand, the logits in the output layer vary significantly, their relative standard deviation over these 120 permutations usually being in the range from 10-20%.

3.5 Achieving 100% Test Accuracy

Since we have the luxury of an arbitrarily large test set at our disposal for this problem, we can furthermore test for the minimal training set to achieve practically 100% test accuracy. As expected, this depends on the input representation and the kind of neural network used. Overparameterization is crucial, as we could not achieve it with small networks at all. When training data is abundant, we notice that the summed-up permutation invariant representation begins to outperform the other one that trains on permuted data. We therefore compare the summed-up versions of the 52 bit and 17 bit representations introduced above. The difference in the effectiveness of both representations becomes clear as the 17 bit only needs approximately 500 training samples per class, a total 4000, while the 52 bit representation needs approximately triple that number on our very large (240k hands) test set.

3.6 Resilience Regarding Labeling Errors

The Texas Hold'em case study provides a test bed for evaluating the impact of the labeling quality on the results based on the results from the previous section. Our Java program for assessing hands is used for generating the labeled data. So, assuming that the tested Java program is correct, all generated labeled data are correct. Another Java program processes a labeled data text file and changes a specified percentage of the labels so that they become wrong labels. To constrain the incorrectness, a label can only become better or worse by one degree compared to the correct label. For example, a hand value of 'one pair' can either become 'two pairs' or 'high card'. Table 2 shows the results if 1% and if 3% of the labels are not correct by deviating by one degree from the correct label.

Table 2. Impact of labeling quality on classification results

Correctness of labels	Accuracy of classification
100%	100%
99%	99.53%
97%	99.25%

4. HAND COMPARISON

The comparison of two Texas Hold'em hands is straight-forward if the hands are different: high card < one pair < two pairs < three of a kind < straight < flush < full house < poker < straight flush < royal flush. If the hand values are equal, the comparison becomes more complex, but is clearly defined by rules. For example, Peter Norvig, vividly explains these rules in a video [14]. We do not go into all the details but exemplify the rules for the hand value 'full house': a full house is a three-of-a-kind and a pair. If two hands have the hand value 'full house', the ranks decide which hand has a higher value. First, the ranks of the three-of-a-kind cards are compared. The one hand with the three-of-a-kind of a higher rank wins. Note the special case in the following example: a hand might have a pair of aces and three queens. But the other hand with a pair of 10s and three queens wins. So it is not sufficient to compare the hands by comparing their card ranks sorted in descending order. Instead, the sorting requires the additional context of 'three-of-a-kind' and 'pair' in case of the 'full-house'. The provided Java code represents all the implementation details of the comparison rules. It uses array lists that store the ranks of tie-breakers (in the example above, a king rank would be in the tie-breaker list of the one hand and a queen rank would be in the tie-breaker list of the other hand) to decide based according to Texas Hold'em rules whether two hands are indeed equal (that is not possible for 'full-house' hands as only four cards exist for each rank) or which of the two hands has a higher value than the other one.

To evaluate the suitability of ML for this more complex data processing task, we harness the one of the training sets used above, specifically the one with 8k hands, in the following way: one random line represents the first hand, another one the second hand.

Then we use the Java program to get the comparison result: –1 (value of first hand < value of second hand), 0 (value of first hand = value of second hand), 1 (value of first hand > value of second hand). As we might expect the network to first do some operation on the single hands before comparing them, we use a siamese network similar to the ones used before and fuse the two encodings again by two certainly overparameterized Dense layers. The error of our model will obviously depend on the number of pairs of hands that we generate from our original one-hand dataset. Figure 6 illustrates the test error over number of hand pairs generated for our model.

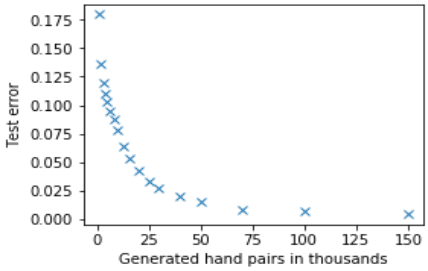

Figure 6. Test error depending on the number of generated hand pairs.

5. BEST HAND OUT OF 7 CARDS

At the end of a Texas Hold'em poker round, each player in the so called showdown has to determine the best hand out of 7 cards, the 2 the player has in the pocket (not known to the others) and the 5 cards on the table. Thus, a player has to find the best hand out of overall 21 hands.

For this task, the input for the network consists of 7 cards with a bitwise representation as described in section 5. The label for each set consisting of 7 cards is the hand value of the best hand. First we generate a dataset of 8,000 7-card hands, equally distributed among the possible values. As the problem is close to the 5-card hands, this method also performs surprisingly well. Depending on the input representation, we can even use the same network that classified 5 card hands to 7 card hands. To this end, we have to use a summed up representation in order to maintain the same input dimensions. Furthermore, there would be some ambiguity in the 17 bit summed up representation: a 7 card hand could allow to form both a flush and a straight, while not allowing to form a straight flush. This would not be detectable in the summed up 17 bit representations. Therefore, we use the 52 bit representation in this experiment and find that a model with practically 0% test error on the five card test set produces only 0.2% error on the 7 card set without requiring further training.

6. CONCLUSIONS

To sum up, this paper presents a quantitative evaluation of the application of machine learning to what might be called conventional data processing tasks. For us it is surprising how important overparameterization is in the solution of this task. Also, the underperformance of a specifically permutation invariant neural network, one that summed up the individual card representations, when compared to introducing permutations in the training set was surprising. While only true in our case for smaller datasets, it could also have implications for ongoing efforts to encapsulate different types of invariances into the network structure itself in order to reduce the need for more conventional data augmentation. Several aspects would be interesting for further investigations, for example: How can labeled data sets be further minimized? Which other neural network types are suitable for the Texas Hold'em tasks? Can the results presented in this paper be generalized and applied to other conventional data processing tasks? Answers to the above research questions might push the start-of-the-art in automatic programming significantly forward. An economic perspective poses the question whether data labeling, or conventional programming is more effective and efficient. As no real-world program is 100% correct, a quantitative and qualitative comparison of the quality, robustness and maintainability of human-written programs and those generated by ML means would be a research direction that could be further pursued.

7. REFERENCES

[1] Google Colaboratory (Cloud-based Jupyter notebooks). DOI= https://colab.research.google.com/

[2] Colaboratory notebooks used in this project: DOI= https://drive.google.com/drive/folders/1UkubXx2QBua7rxiVRJLgRnRSCYj5l0Q6?usp=sharing

[3] Texas Hold'em. DOI= https://en.wikipedia.org/wiki/Texas_hold_%27em

[4] Dua, D. and Graff, C. (2019). UCI Machine Learning Repository [http://archive.ics.uci.edu/ml]. Irvine, CA: University of California, School of Information and Computer Science.

[5] Gautam, B., Kaviarasan, S., Veena, D. 2016. *NN-based Poker Hand Classification and Game Playing*. 29th Conference on Neural Information Processing Systems (NIPS 2016), Barcelona, Spain.

[6] Jabin, S. 2016. Poker hand classification. In *Proceedings of the International Conference on Computing, Communication and Automation* ICCCA 2016.

[7] Neyshabur, B., Li, Z., Bhojanapalli, S., LeCun, Y., & Srebro, N. (2018). Towards understanding the role of over-parametrization in generalization of neural networks. arXiv preprint arXiv:1805.12076.

[8] Greenwood, M., & Oxspring, R. (2001). The Applicability Of 'Occam's Razor' to Neural Network Architecture. Undergraduate Coursework, Department of Computer Science, The University of Sheffield, UK.

[9] Brutzkus, A., & Globerson, A. (2018). Why do Larger Models Generalize Better? A Theoretical Perspective via the XOR Problem. arXiv preprint arXiv:1810.03037.

[10] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., ... & Kudlur, M. (2016). Tensorflow: A system for large-scale machine learning. In 12th Symposium on Operating Systems Design and Implementation (pp. 265-283).

[11] Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980.

[12] Keskar, N. S., & Socher, R. (2017). Improving generalization performance by switching from adam to sgd. arXiv preprint arXiv:1712.07628.

[13] LeCun, Y., & Bengio, Y. (1995). Convolutional networks for images, speech, and time series. The handbook of brain theory and neural networks, 3361(10), 1995.

[14] Norvig, P. Wild West Poker—Texas Hold'em hand comparison (part of a Udacity course on the Design of computer programs). DOI= https://www.youtube.com/watch?v=4bpc2A3gIuc.