Chess2vec: Learning Vector Representations for Chess

Berk Kapicioglu
OccamzRazor
berk@occamzrazor.com

Ramiz Iqbal*
MD Anderson Cancer Center
riqbal@mdanderson.org

Tarik Koc
OccamzRazor
tarik@occamzrazor.com

Louis Nicolas Andre
OccamzRazor
louis@occamzrazor.com

Katharina Sophia Volz
OccamzRazor
volz@occamzrazor.com

Abstract

We conduct the first study of its kind to generate and evaluate vector representations for chess pieces. In particular, we uncover the latent structure of chess pieces and moves, as well as predict chess moves from chess positions. We share preliminary results which anticipate our ongoing work on a neural network architecture that learns these embeddings directly from supervised feedback.

1 Introduction

In the last couple of years, advances in machine learning have yielded dramatic improvements in tasks as diverse as visual object classification [9], automatic speech recognition [7], machine translation [18], and natural language processing (NLP) [12]. A common thread among all these tasks is, as diverse as they may seem, they all involve processing input data, such as image, audio, and text, that have not traditionally been amenable to feature engineering. Modern machine learning methods that enabled these breakthroughs did so partly because they shifted the burden away from feature engineering, which is difficult for humans and requires domain expertise, towards designing models that automatically infer feature representations that are relevant for downstream tasks [1].

In this paper, we are interested in learning and studying feature representations of chess positions and pieces. Our work is inspired by how learning vector representation of words [12, 13], also known as word embeddings, yielded improvements in tasks such as syntactic parsing [16] and sentiment analysis [17]. In a similar fashion, we want to study how learning vector representations of chess positions and pieces can deepen our understanding of chess and potentially improve downstream machine learning models designed to play chess.

Research on building computer programs that play chess have a long history which dates back to the 1950s [14, 3]. In recent years, computer chess researchers have focused increasingly on using deep learning to train chess programs [10, 5, 15]. Lai [10] trained a neural network which evaluated positions, trained it via self-play, and showed that it was comparable to an international master. Omid et al. [5] trained a deep neural network called DeepChess that achieved grandmaster-level performance by performing pairwise evaluations of chess positions. Silver et al. [15] designed a deep reinforcement learning algorithm called AlphaZero, which after 24 hours of training via self-play, defeated a world-class chess program.

Unlike previous research on computer chess, our goal is not to design an algorithm which learns how to play chess at a grandmaster level. Instead, we want to undertake a systematic study of constructing

*Research conducted while author was an intern at OccamzRazor.

32nd Conference on Neural Information Processing Systems (NIPS 2018), Montréal, Canada.

vector representations for chess pieces using a variety of methods. We assess the quality of these embeddings both quantitatively, by using them in downstream prediction tasks, as well as qualitatively. To the best of our knowledge, this is the first research paper that studies methods for learning vector representations of chess pieces.

The paper proceeds as follows. In section 2, we introduce preliminaries used in the rest of the paper. In section 3, we conduct a qualitative analysis of the latent structure of chess pieces and moves using principal components analysis (PCA). In section 4, we quantitatively demonstrate that we can improve our vector representations by applying nonnegative matrix factorization (NMF) and hashing techniques.

2 Preliminaries

The fundamental challenge for machine learning based chess programs is to learn the mapping between chess positions and optimal moves [10, 5, 15]. A chess position is a description of where pieces are located on the chessboard. In learning, chess positions are typically represented as bitboard representations [2].

A bitboard is a 8×8 binary matrix, same dimensions as the chessboard, and each bitboard is associated with a particular piece type (e.g. pawn) and player color (e.g. black). On the bitboard, an entry is 1 if a piece associated with the bitboard's type and color exists on the corresponding chessboard location, and 0 otherwise. Each chess position is represented using 12 bitboards, where each bitboard corresponds to a different piece type (i.e. king, queen, rook, bishop, knight, or pawn) and player color (i.e. white or black). More formally, each chess position is represented as a $8 \times 8 \times 12$ binary tensor.

The bitboard representation can alternatively be specified by assigning a 12-dimensional binary vector to each piece type and color. Under this interpretation, given a chess position, each of the 64 chessboard locations is assigned a 12-dimensional vector based on the occupying piece. If the location has no piece, it is assigned the zero vector, and if it has a piece, it is assigned the sparse indicator vector corresponding to the piece type and color. Note that, with this representation, all black pawns would be represented using the same vector.

In this paper, our goal is to explore new d-dimensional vector representations for chess pieces to serve as alternatives to the 12-dimensional binary bitboard representation. As part of this effort, we develop novel ways to apply matrix factorization techniques such as PCA and NMF to chess data.

We use various open-source tools for training and evaluating our models. To generate chess moves, we use Stockfish [4], currently the highest rated open-source chess engine in the world . We choose an interactive chess engine over a static chess database because we plan to evaluate our models in a reinforcement learning setting. To interact with Stockfish, we use a Python library called python-chess [6] .

3 Latent structure of pieces and moves

In this section, we analyze the latent structure of chess pieces and moves using principal components analysis. We conduct the analysis on two different datasets: one which contains legal moves and another which contains expert moves. Aside from gaining qualitative insights about chess, we also obtain our first vector representations.

3.1 Principal component analysis (PCA)

PCA is a method for linearly transforming data into a lower-dimensional representation [8]. In particular, let data be represented as the matrix $X \in \mathbb{R}^{n \times p}$ whose columns have zero mean and unit variance. Each row of X can be interpreted as a data point with p features. Then, PCA computes the transformation matrix $C^\star \in \mathbb{R}^{p \times d}$ by solving the optimization problem

$$\begin{array}{ll} \underset{C}{\text{minimize}} & \left\| X - XCC^T \right\|_F^2 \\ \text{subject to} & C^T C = I \end{array}. \tag{1}$$

Intuitively, $C^\star$ linearly transforms the data from a p-dimensional space to a d-dimensional space in such a way that the inverse transformation (i.e. reconstruction) minimizes the Frobenius norm. The

Table 1: Principal component scores of chess piece types (legal moves)

	PC1	PC2	PC3	PC4	PC5
pawn	-11.2100	10.9796	5.8056	28.7984	0.0000
knight	-22.2174	40.2092	-11.5757	-12.5815	0.0000
bishop	-26.1704	-34.2642	-20.2917	-2.0117	0.0000
rook	51.0099	4.1676	-1.4797	-1.9351	0.0000
queen	21.3252	-10.6139	-8.7151	-1.9646	0.0000
king	-12.7373	-10.4783	36.2566	-10.3054	0.0000

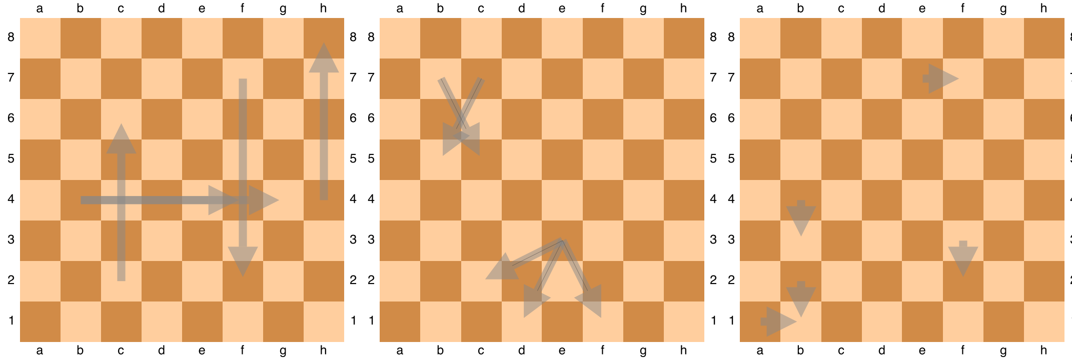

Figure 1: The top five moves associated with the legal move loading vectors. The first loading vector positively weighs horizontal or vertical moves that jump at least two squares and exclude pawn and king moves, and negatively weighs single square diagonal moves. Hence, rook has the highest score and bishop has the lowest score. The second loading vector positively weighs L-shaped knight moves and negatively weighs multiple square diagonal moves. Hence, knight has the highest score and bishop has the lowest score. The third loading vector consists of single square horizontal and vertical moves that exclude pawn moves, and negatively weighs multiple square diagonal moves. Hence king has the highest score and bishop has the lowest score.

mapping of X into the d-dimensional space is given by $W = XC^{\star}$. The importance of minimizing reconstruction error will be clear when we quantitatively evaluate the embeddings obtained via PCA.

In order to apply PCA to chess, we need an appropriate way to represent chess data as $X \in \mathbb{R}^{n \times p}$. Ideally, we would like pieces with similar movement patterns to have similar low-dimensional representations, so we represent each piece by its move counts. In its simplest form, a chess move is specified via a source board location that a piece is picked from and a target board location that the piece is moved into. This formulation allows us to avoid specifying the type of the moving piece or whether the target location already contained another piece. There are 64 board locations, thus all move counts can be represented via a 4096-dimensional vector, and since there are 6 piece types, we let $X \in \mathbb{R}^{6 \times 4096}$.

We use two different methods to generate the move counts for each piece. First, we generate counts based only on legal moves. This allows us to qualitatively analyze the latent structure of chess pieces based on their fundamental movement patterns. Second, we generate counts based on expert moves chosen by Stockfish [4], a world-class chess engine.

3.2 Legal moves

In order to generate legal moves, for each piece type and board location, we generate a chess position by placing a piece of that type to the given location on an empty board. We then generate all legal moves for this chess position and increment the corresponding entries in X. This process generates a total of $3{,}764$ legal moves, where pawn generates the lowest at 96 and queen generates the highest at $1{,}456$. We then normalize each row by its total counts, scale the columns to have zero mean and unit variance, and apply PCA.

We list the principal components in Table 1, visualize the loading vectors in Figure 1, and plot the first three components in Figure 2. The loading vectors are directions in the feature space along which

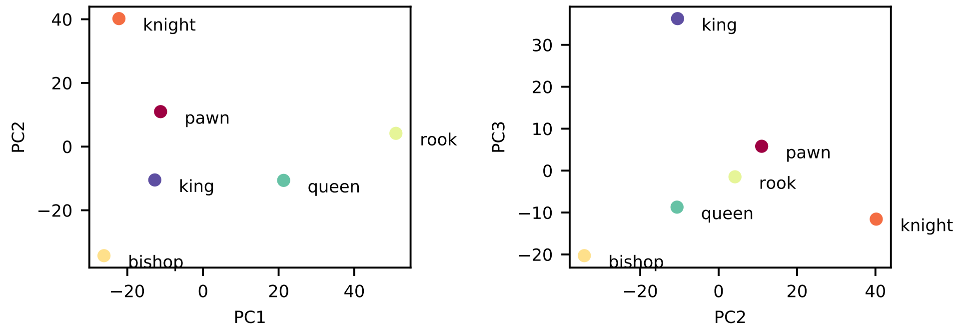

Figure 2: Principal component scores of chess piece types (legal moves). Left: the first principal component versus the second. Right: the second principal component versus the third.

data varies the most. Thus, each loading vector can be interpreted as a fundamental building block of legal chess moves, and the principal component scores depict how each piece type can be described in terms of these building blocks with respect to their movement patterns. Note that, remarkably, the first three principal components explain 89.7% of the variance in the movement data.

As depicted in Figure 1, PCA decomposes legal chess moves into 1) single square diagonal moves and multiple square horizontal and vertical moves that exclude pawn moves, 2) L-shaped moves and multiple square diagonal moves, 3) single square horizontal and vertical moves that exclude pawn moves, and 4) one or two square forward pawn moves. This decomposition not only enhances our intuition about legal chess moves, but it also acts as a baseline to compare future decompositions against.

3.3 Stockfish moves

Instead of analyzing the latent structure of legal moves, we now shift our focus to analyzing the latent structure of expert moves. Part of the challenge continues to be determining how to generate the matrix $X \in \mathbb{R}^{n \times p}$. This time, we generate the count matrix by pitting two Stockfish [4] engines against each other and logging their moves. Unlike before, in order to investigate how pieces of the same type are similar to or different from each other, we let each row of X correspond to a piece rather than a piece type. Hence, we set n to 16, which is the number of white pieces, instead of 6.

We let the engines play against each other for $4{,}000{,}000$ turns, resulting in 23,096 games with an average length of 173 turns. Since we are interested in generating embeddings for settings where the white player wins, we let the white player's think time be sampled from a normal distribution with a mean of 50 milliseconds, whereas we restrict black player's mean think time to 5 milliseconds. In the end, white player wins 45% of the time, black player wins 10% of the time, and a draw occurs 45% of the time. We observe a high percentage of draws because repeating the same move three times causes a draw, and the black player frequently uses that strategy. We only log moves by the white player from the games which the white player won, yielding a total of $881{,}779$ moves, which we use to construct the $X \in \mathbb{R}^{16 \times 4096}$ matrix. We then normalize each row by its total counts, scale the columns to have zero mean and unit variance, and apply PCA.

We list the principal components in Table 2 and visualize the first three components in Figure 4. Each loading vector can be interpreted as a fundamental building block of expert chess moves, and the principal component scores depict how each piece can be described in terms of these building blocks. The first five principal components explain 81.0% of the variance in the movement data, at which point adding more principal components does not increase the explained variance by much. This is because, as seen in Figure 4, pieces of same type have similar principal component scores, and they share the same loading vectors to construct their movement patterns. This is surprising, since PCA has no explicit information that these pieces belong to the same type, and despite their similarities, individual pieces such as pawns do have their own unique movement patterns within the 4096-dimensional movement space.

Table 2: Principal component scores of chess pieces (Stockfish moves)

	PC1	PC2	PC3	PC4	PC5
rook1	62.1762	10.9768	1.9513	1.3136	-2.2036
knight1	-26.5993	44.1739	2.5065	11.0459	0.4398
bishop1	-17.9825	-24.9341	44.8501	17.5337	-6.7910
queen	20.4590	-12.2827	-1.6426	22.9697	1.5908
king	-4.8823	-10.7554	-1.0840	1.0111	38.1625
bishop2	-16.5670	-20.6406	-38.9993	23.5941	-8.2171
knight2	-24.9990	38.6982	2.0482	8.0774	0.2606
rook2	60.8472	10.4907	1.8318	1.3381	-1.3118
pawn1	-4.0662	-4.3803	-2.0495	-9.6416	-4.3501
pawn2	-5.8440	-4.7609	-1.3513	-11.3009	-4.0888
pawn3	-6.6150	-4.7659	-2.0196	-10.4295	-3.0941
pawn4	-8.4415	-4.2057	-1.2503	-10.6654	-1.9597
pawn5	-10.6909	-3.5096	-1.3891	-11.2990	-1.1025
pawn6	-6.4004	-5.1921	-0.6284	-11.2539	-2.0656
pawn7	-6.0409	-4.6343	-1.6150	-11.6597	-2.3890
pawn8	-4.3535	-4.2780	-1.1589	-10.6337	-2.8804

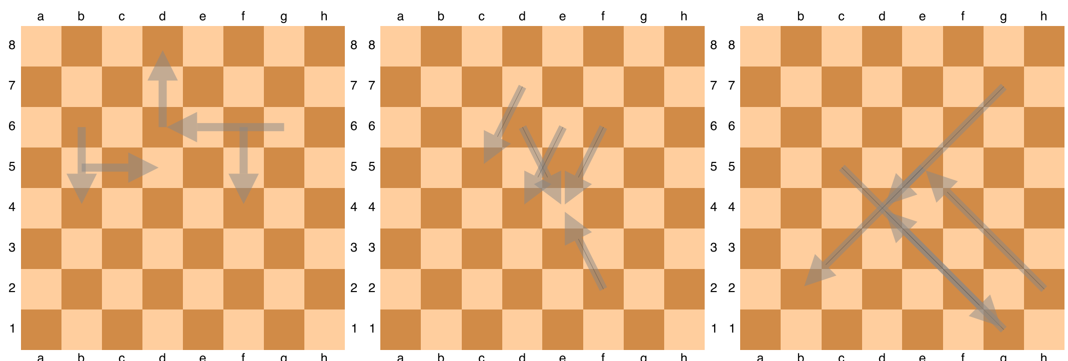

Figure 3: The top five moves associated with the Stockfish loading vectors. The first loading vector positively weighs horizontal or vertical moves that jump at least two squares and exclude pawn and king moves, and negatively weighs L-shaped knight moves. Hence, rook has the highest score and knight has the lowest score. The second loading vector positively weighs L-shaped knight moves and negatively weighs diagonal moves. Hence, knight has the highest score and bishop has the lowest score. The third loading vector specializes in diagonal moves, where moves across black squares are weighed positively and across white squares are weighed negatively.

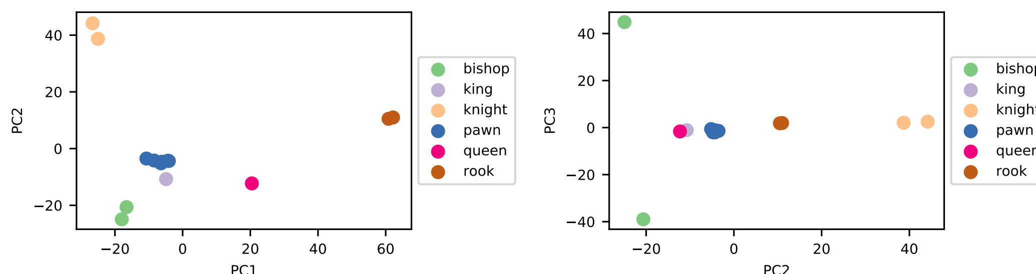

Figure 4: Principal component scores of chess pieces (Stockfish moves). Left: the first principal component versus the second. Right: the second principal component versus the third.

In figure 3, we display the highest scored moves associated with the loading vectors. We note that the first two principal components and loading vectors for Stockfish moves are quite similar to their legal move counterparts. This is also surprising, since the two count matrices are generated quite differently, yet in both cases, multiple square horizontal or vertical moves that exclude pawn and king moves are identified as the most distinguishing movement pattern. The third dimension is the first dimension in which pieces of the same type, namely bishops, are not clustered together with respect to their component scores. In fact, the bishop which moves among black squares has a high component score, and the bishop which moves among white squares has a low component score.

These decompositions are helpful in uncovering fundamental movement patterns and understanding the relationship between pieces with respect to these patterns. To the best of our knowledge, this is the first attempt to use machine learning to conduct such an analysis. In the process, we have also obtained our first d-dimensional vector representations, namely $W = XC^{\star}$.

4 Position-dependent piece vectors

In this section, we attempt to improve upon the vector representations we have obtained so far. In particular, one potential issue is that the piece vectors we constructed are constant with respect to chess positions. For example, the vector representation of a pawn does not vary based on the remaining pieces on the board. We propose a hash-based method to address this issue. We also conduct our first quantitative evaluation.

4.1 Zobrist hashing

Zobrist hashing [19] is a method for constructing a universal hash function that was originally developed for abstract board games. It is typically used to build transposition tables which prevent game agents from analyzing the same position more than once.

A natural way to generate position-dependent piece vectors is to modify the construction of the count matrix $X \in \mathbb{R}^{n \times p}$. Previously, each row of X corresponded to a piece or piece type. This time, we expand the rows to correspond to the Cartesian product of pieces and hash buckets. The hash buckets partition the space of all chess positions via Zobrist hashing. Thus, instead of generating a unique vector representation for each piece, we generate it for each piece given a hash bucket, where each bucket represents a random collection of chess positions. In our experiments, we let the number of hash buckets go up to 32,768. Since there are 16 pieces, this yields the sparse count matrix $X \in \mathbb{R}^{524,288 \times 4096}$.

4.2 Non-negative matrix factorization (NMF)

In addition to expanding the count matrix, we also switch our matrix factorization method from PCA to NMF [11]. NMF is specified by the optimization problem

$$\begin{array}{ll} \underset{W,H}{\text{minimize}} & \|X - WH\|_F^2 \\ \text{subject to} & W \geq 0, H \geq 0 \end{array}. \tag{2}$$

In the NMF objective (2), $W \in \mathbb{R}^{n \times d}$ encodes the d-dimensional vector representations and $H \in \mathbb{R}^{d \times p}$ encodes the p-dimensional basis vectors. The corresponding matrices in the PCA objective (1) are $W \simeq XC$ and $H \simeq C^T$, where $\simeq$ simply denotes a semantic (but not a numeric) similarity.

We switch from PCA to NMF for a couple of reasons. First, even though both PCA and NMF attempt to minimize the reconstruction error, the positivity constraints of NMF are more flexible than the orthonormality constraints of PCA. In fact, NMF yields lower reconstruction error than PCA, and as we shall see below, reconstruction error plays an important role in predicting chess moves. Second, NMF yields a different set of vectors than PCA which are also worth investigating.

We demonstrate how matrix reconstruction can be used to predict chess moves from chess positions. As discussed, each row of $W \in \mathbb{R}^{n \times d}$ encodes a d-dimensional vector representation, denoted as $w_{i,j}^T \in \mathbb{R}^{1 \times d}$, that is associated with a white piece $i \in \{1, \dots, 16\}$ and a hash bucket j. Each hash bucket j corresponds to a random collection of chess positions. Let $w_{0,\cdot}$ be the vector representation for the special piece 0 that is always assigned to the zero vector. Let $s \in \mathcal{S}$ represent a chess position,

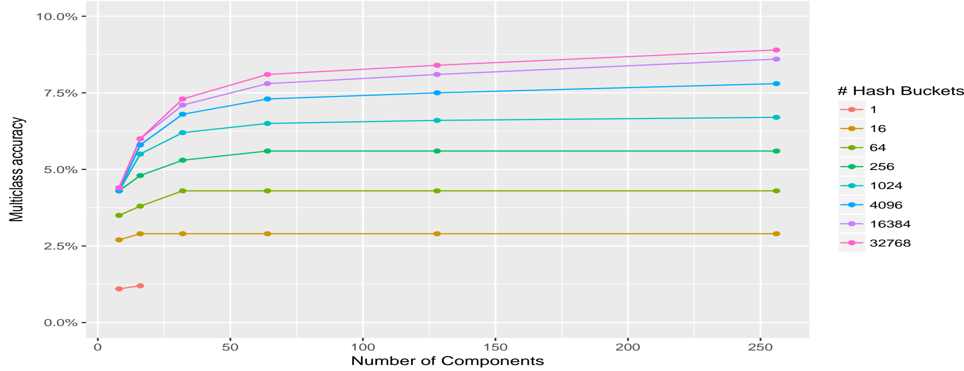

Figure 5: Test multiclass accuracy of matrix reconstruction with position-dependent piece vectors.

which is a description of the locations of all the pieces on a chess board. Let $f_l : \mathcal{S} \to \{0, ..., 16\}$ be a function that maps a chess position to one of the 17 pieces that can be located on board location $l \in \{1, ..., 64\}$, where $f_l(s) = i$ if board location l contains a white piece i, and $f_l(s) = 0$ if l is empty or contains a black piece. Let $h : \mathcal{S} \to \mathcal{J}$ be the Zobrist hash function that maps a chess position to a hash bucket. Then, given a chess position s, $y(s) = \sum_{j=1}^{64} H^T w_{f_j(s),h(s)}$ is the 4096-dimensional prediction vector, and $\arg\max_i y(s)_i$ is the index of the predicted move.

The intuition behind this prediction is as follows. Each row of X, denoted as $x_{i,j}^T \in \mathbb{R}^{1\times 4096}$, represents the number of times a white piece i has been associated with a particular move for chess positions in $\mathcal{S}_j = \{s \in \mathcal{S} \mid h(s) = j\}$. Since equation 2 minimizes reconstruction error, given a chess position s, $H^T w_{f_j(s),h(s)} \approx x_{f_j(s),h(s)}$. Thus, $y(s) \approx \sum_{j=1}^{64} x_{f_j(s),h(s)}$ approximates the number of times a move has been selected for all the chess positions that are hashed into bucket j.

4.3 Quantitative evaluation

In this subsection, we evaluate the accuracy of making move predictions using matrix reconstruction and position-dependent piece vectors. We split the Stockfish data, which consists of 881,779 moves, into 80% train and 20% test. We use the train data to generate the count matrix $X \in \mathbb{R}^{524,288\times 4096}$, apply NMF, and obtain W_{train} and H_{train}. We then iterate over each chess position s in test data, replace their bitboard vectors with the relevant rows from W_{train}, and compute $y(s)$.

In figure 5, we display the multiclass accuracy on held-out test data. As we increase the number of hash buckets, the held-out accuracy improves. With 32768 hash buckets, this method predicts 8.8% of Stockfish moves correctly, on a task with 4096 classes.

References

[1] Yoshua Bengio, Aaron Courville, and Pascal Vincent. Representation learning: A review and new perspectives. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2013. 1

[2] Wikipedia contributors. Bitboard — Wikipedia, The Free Encyclopedia. `https://en.wikipedia.org/w/index.php?title=Bitboard&oldid=841136708`, 2018. 2

[3] Wikipedia contributors. Computer chess — Wikipedia, The Free Encyclopedia, 2018. 1

[4] Wikipedia contributors. Stockfish (chess) — Wikipedia, The Free Encyclopedia. `https://en.wikipedia.org/w/index.php?title=Stockfish_(chess)&oldid=855275247`, 2018. 2, 3.1, 3.3

[5] Omid E. David, Nathan S. Netanyahu, and Lior Wolf. DeepChess: End-to-end deep neural network for automatic learning in chess. In *ICANN*, 2016. 1, 2

[6] Niklas Fiekas. python-chess: a pure Python chess library. `https://python-chess.readthedocs.io/`, 2018. 2

[7] Geoffrey Hinton, Li Deng, Dong Yu, George Dahl, Abdel Rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara Sainath, and Brian Kingsbury. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, 2012. 1

[8] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An Introduction to Statistical Learning*. 2013. 3.1

[9] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *NIPS*, 2012. 1

[10] Matthew Lai. Giraffe: Using deep reinforcement learning to learn to play chess. *arXiv*, 2015. 1, 2

[11] Daniel D. Lee and H. Sebastian Seung. Learning the parts of objects by non-negative matrix factorization. *Nature*, 1999. 4.2

[12] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. In *NIPS*, 2013. 1

[13] Jeffrey Pennington, Richard Socher, and Christopher Manning. Glove: Global Vectors for Word Representation. In *EMNLP*, 2014. 1

[14] Claude E. Shannon. XXII. Programming a computer for playing chess. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 1950. 1

[15] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. *arXiv*, 2017. 1, 2

[16] Richard Socher, John Bauer, Christopher D. Manning, and Andrew Y. Ng. Parsing with compositional vector grammars. In *ACL*, 2013. 1

[17] Richard Socher, Alex Perelygin, and Jy Wu. Recursive deep models for semantic compositionality over a sentiment treebank. In *EMNLP*, 2013. 1

[18] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *NIPS*, 2014. 1

[19] Albert L Zobrist. A new hashing method with application for game playing. *ICCA Journal*, 13(2):69–73, 1970. 4.1