Project 2:
Time / Sequence Processing

Using an RNN to Predict the Winner of a Chess Game

Andy Varner
12153597
Spring 2023

Introduction / Project Description

The broad focus of this work is the investigation of time series neural networks. Time series neural networks are specialized networks designed to model and make predictions based on time series data, especially for language modeling, pattern recognition and forecasting. Common architectures for these networks include recurrent neural networks (RNNs) and long short term memory networks (LSTMs). In this work, we conduct a study of the RNN using an adapted chess game dataset. Specifically, we pose the question, "Can an RNN predict the winner of a chess game based on the first N moves?"

RNN Background / Theory

In this section we provide an overview of RNN network architecture and learning algorithm used in this project.

RNN Architecture

An RNN takes in input data one at a time and in sequence as shown in Figure 1. At each time step, the RNN produces an output known as the hidden state, which contributes to the output of the next time step. In this way, the network has one-time-step memory. Outputs are available at each step of the network, but for our chess game classification task, we will only consider the Nth, or final, output. In this work, Input 1, Input 2, etc. represent individual moves in a single game of chess. The output is a one-hot vector predicting whether the winner is White or Black.

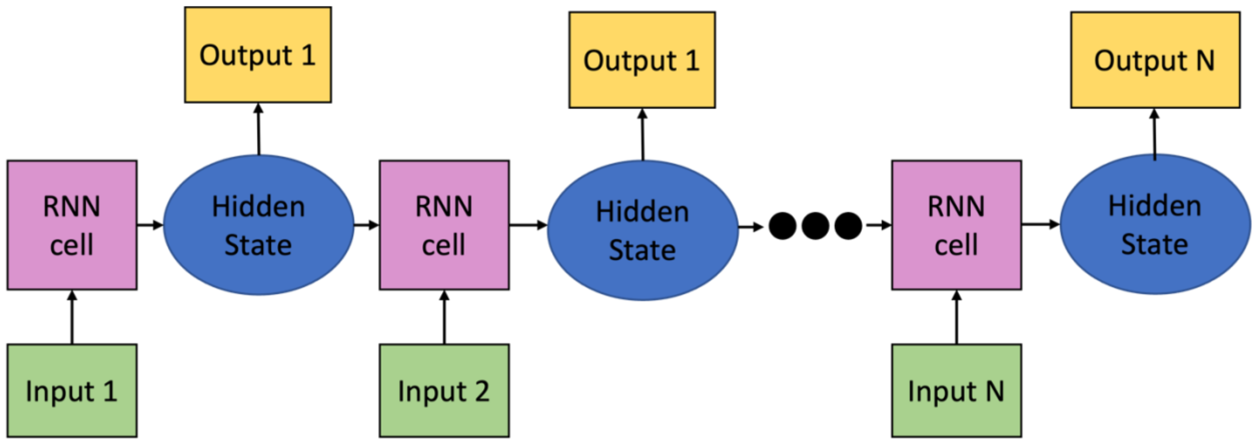


Figure 1. Overview of RNN architecture

The hidden state, h_t, is given by

$$h_t = f_w(h_{t-1}, x_t)$$

where f_w is the RNN function parameter, x_t is the input at time step t, and h_{t-1} is the previous hidden state. Most networks use tanh as the function parameter, so the hidden state aggregates both the previous hidden state and the current time step,

$$h_t = tanh(w_{hh} * h_{t-1} + w_{xh} * x_t),$$

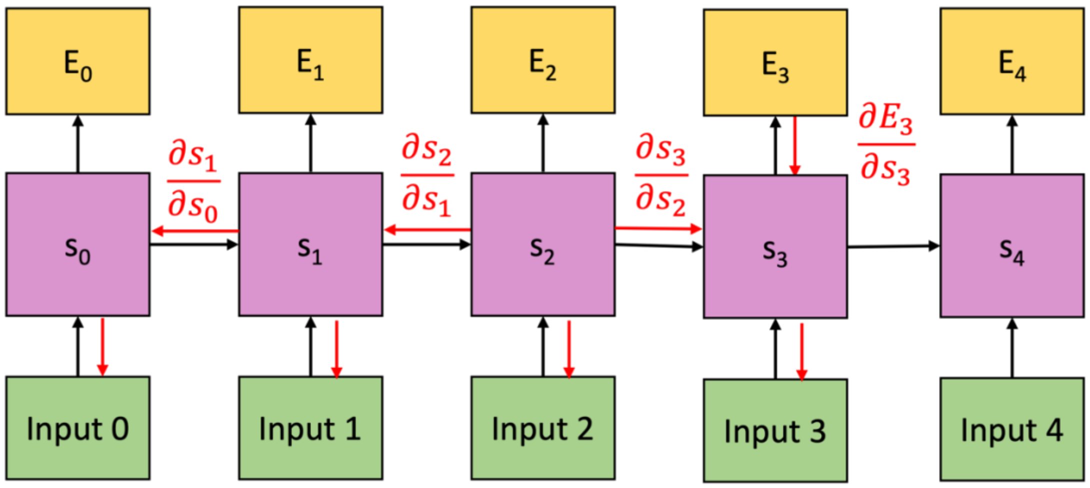
such that the hidden state is on the domain [-1, 1] for stability. The output, y_t is given by

$$y_t = w_{hy} * h_t$$

where h_t is the hidden state just computed and w_{hy} is the new mapped outcome. The hidden state will be fed back into the RNN cell with the next input until the number of epochs is completed.

Training and Backpropagation

Our RNN follows a supervised learning scheme where each sample (game) consists of a set of inputs (the moves of the game), as well as a ground truth label representing the actual winner of the game. Using cross-entropy loss, we calculate the gradient of the loss function for backpropagation, thereby updating the weights in the model. The RNN cells have shared weights throughout the process, so only the input data and hidden state at each time step are unique. The backpropagation process is shown in Figure 2.



E_0
E_1
E_2
E_3
E_4
$\frac{\partial s_1}{\partial s_0}$
$\frac{\partial s_2}{\partial s_1}$
$\frac{\partial s_3}{\partial s_2}$
$\frac{\partial E_3}{\partial s_3}$
s_0
s_1
s_2
s_3
s_4
Input 0
Input 1
Input 2
Input 3
Input 4

Figure 2. Visualization of RNN backpropagation algorithm

Experiment Design and Evaluation of Results

In addition to building the network architecture, we must choose and augment a dataset that will suit the network's needs.

The Chess Dataset

The data used for this project is pulled from Kaggle [1], an aggregation of over 20,000 games played by over 1500 players. This dataset provides a lot of interesting information about the chess games, such as player ratings and opening moves, but we are only interested in `winner` and `moves`. Additionally, the data is stripped down to include only games that do not end in a draw and have more than five moves. Furthermore, three custom datasets are created: Dataset 1 contains only the first ten moves of each game, Dataset 2 contains the first 20 moves, and Dataset 3 contains the first 50 moves. For games that have fewer moves than the maximum value, the list of moves is padded, so that each game contains the same number of values for `Turns`. Each dataset necessarily contains a different number of games (fewer maximum turns, larger dataset), but the train and test datasets are dynamically sorted such that the ratio is 80:20.

Table 1. The three datasets and their corresponding maximum number of turns

Dataset	Maximum Turns
1	10
2	20
3	50

The data must be encoded to be used as inputs for the RNN. The `winner` is encoded as a one hot vector, since this is really just a two-class problem with one winner and one non-winner. The first index contains White, and the second index contains Black, so if Black wins, the one hot vector is [0, 1].

The `moves` require a custom encoding scheme. Each move contains a minimum of four pieces of information which are encoded into a (1,5) vector: player turn [0], the piece that is moved [1], and file/rank [2]/[3], which together refer to the two-dimensional position on the board where the piece lands. Additionally, a move may contain a special modification, like 'capture' or 'check' which is encoded in index [4]. All possible mods, in addition to the encoding scheme, are listed in Table 2. The table lists encoding values for the algebraic-notation values. Where it may not be obvious, algebraic notation translations are given in parentheses. Also, encoding values which are not used are colored green in the table.

Table 2. Encoding scheme for chess moves

Vector Index and Designation	Map – Encoding Values							
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
[0] Player	White	Black	–	–	–	–	–	–
[1] Piece	– (pawn)	N (knight)	B (bishop)	R (rook)	Q (queen)	K (king)	–	–
[2] File	A	B	C	D	E	F	G	H
[3] Rank	1	2	3	4	5	6	7	8
[4] Mods	0-0-0 (queenside castle)	0-0 (kingside castle)	x (capture)	+ (check)	# (checkmate)	= (promotion)	–	–

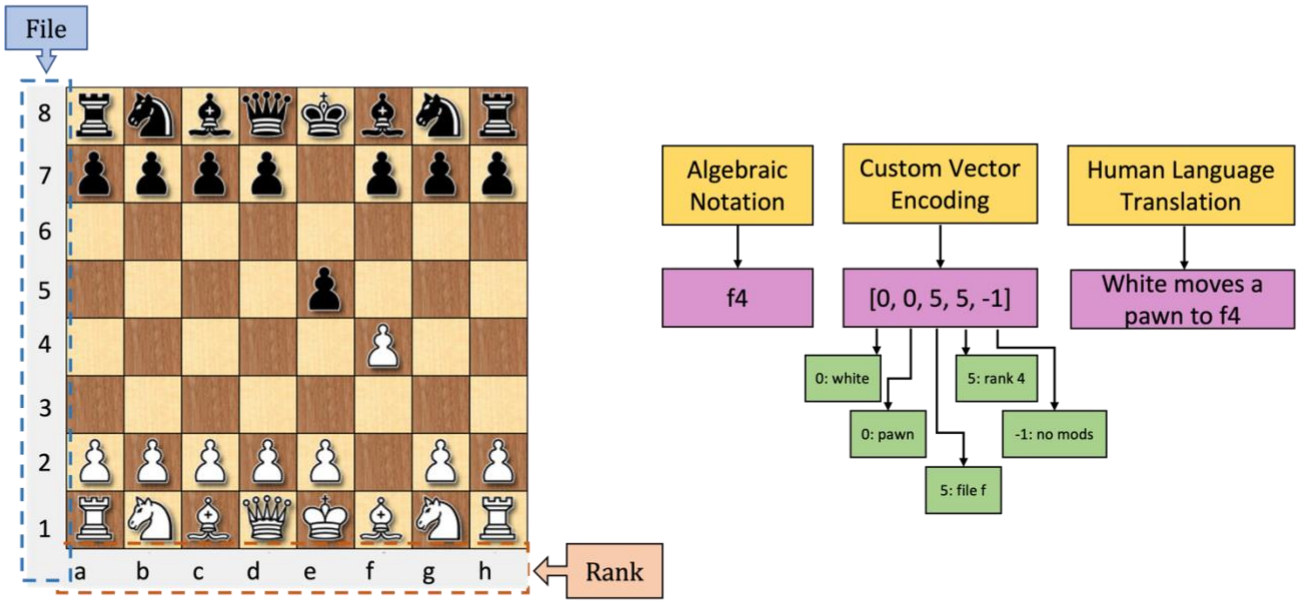


Figure 3. Chessboard visualization, explaining algebraic notation and custom vector encoding. On left: chessboard depicting rank and file. On right: algebraic notation, vector encoding, and human language translation for the white move shown on the board.

Algebraic notation is the standard method of documenting and explaining chess moves. This notation system uses a distinct set of coordinates to identify each square on the chessboard. The vertical columns, ‘a’ through ‘h’ are referred to as *files*, and the horizontal rows, 1 through 8 are referred to as *ranks*. Consequently, every square has a unique identification consisting of a letter indicating the file, followed by a number denoting the rank, as shown in Figure 3.

Evaluation Methods

We keep track of loss and develop confusion matrices to guide evaluation of the networks. From the confusion matrices, we determine accuracy, recall, precision, and F1 score for each experiment.

A confusion matrix is a graphical representation of the overall performance of the network. It shows, for each class, the number of true positive (TP), true negative (TN), false positive (FP), and false negative (FN) classifications made by the network. True positives are the number of samples labeled correctly as belonging to the class. True negatives are the number of samples labeled correctly as not belonging to the class. False positives are the number of samples that are incorrectly labeled as belonging to the class, and false negatives are the number of samples that are incorrectly labeled as not belonging to the class. These results are used to calculate other additional metrics including accuracy. Accuracy is how close a given set of measurements are to their true value.

$$Confusion\,matrix: \begin{bmatrix} TP & FP \\ FN & TN \end{bmatrix}$$

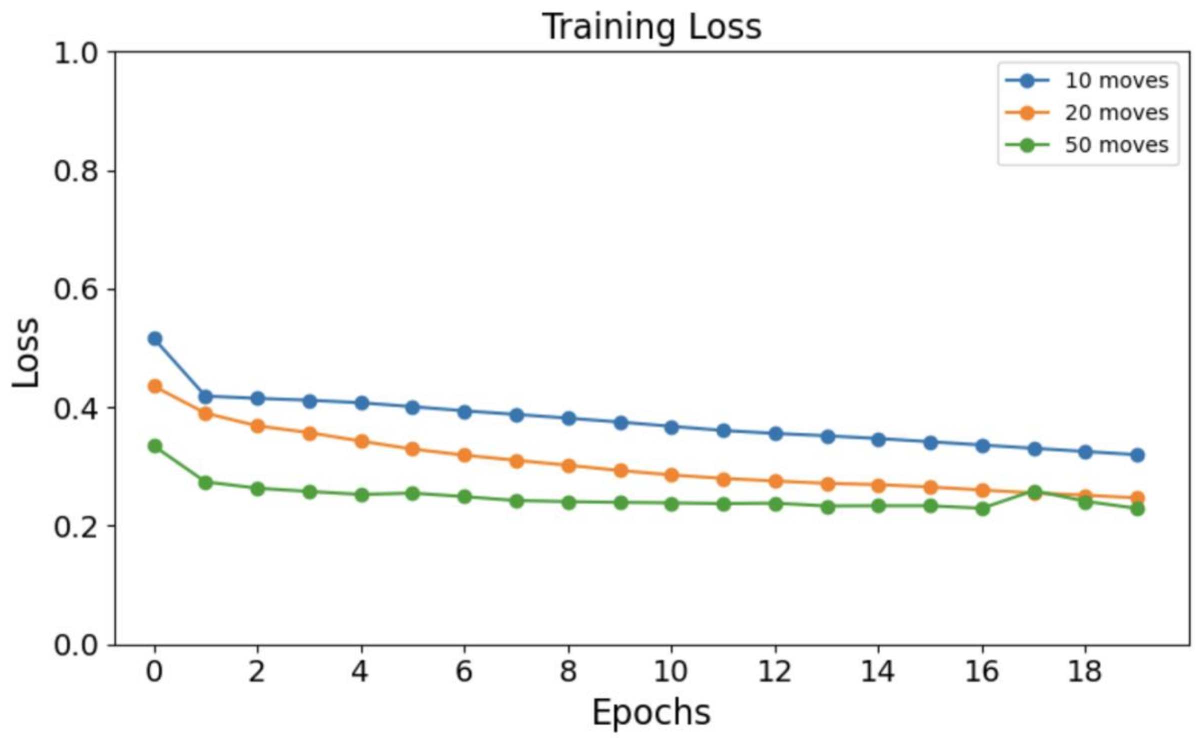
$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

These metrics allow us to assess the performance of the network from different perspectives, showing which classes it performs well on, and which classes it is struggling with. We will periodically use these metrics to assess the impact of hyperparameters on network performance. Next, we discuss the design and results of our experiments.

RNN Design and Results

For the first experiment, we build a vanilla RNN with four recurrent layers and a hidden size of 12. The model is trained with cross entropy loss and optimized using the Adam optimizer. The nonlinearity used is tanh, which is the default in Pytorch's RNN. We want to know if the model performs better with more data, so we use datasets with 10, 20, and 50 maximum moves as described above. The results are shown in Figure 3 and Figure 4. The confusion matrices of Figure 4 are normalized due to each dataset having a different number of total games.

Table 3. Training loss for datasets with 10, 20, and 50 moves across 20 epochs



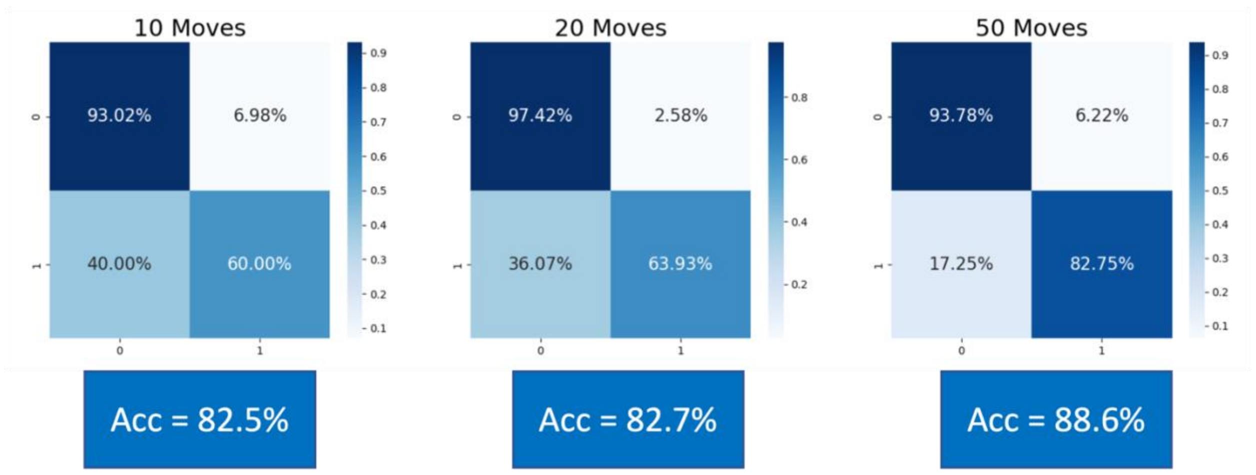


Figure 4. Confusion matrices and accuracy for each dataset

Although loss drops in the first epoch, and continues to decrease, the rate at which it decreases is extremely slow and never drops below 0.2 within 20 epochs. This stagnation in learning is reflected in mediocre accuracy results, which are not improved by much as the number of moves given to the model increases. Still, the model is able to perform significantly better than 50%, so we are confident that the model has potential to perform this task. Note that the model is much better at predicting a white win than a black win. White has the advantage, so there is a bias in the dataset (For example, in dataset 3, White makes up 54% of the wins).

To try to improve performance, we adjusted some network parameters such as learning rate, number of features in the hidden state, and number of recurrent layers. Initial results are not promising! Figure 5 shows worse performance for both increasing and decreasing the number of layers. Learning stagnates at a higher loss value than when the number of layers was set to four, and in both cases we see worse accuracy. Similar stagnation and roughly 80% accuracy was showng for increasing the size of the hidden dimension to 20 and then 32, so we can conclude that, for this particular problem, number of hidden units in the RNN and number of recurrent layers are irrelevant to accuracy and ability to learn.

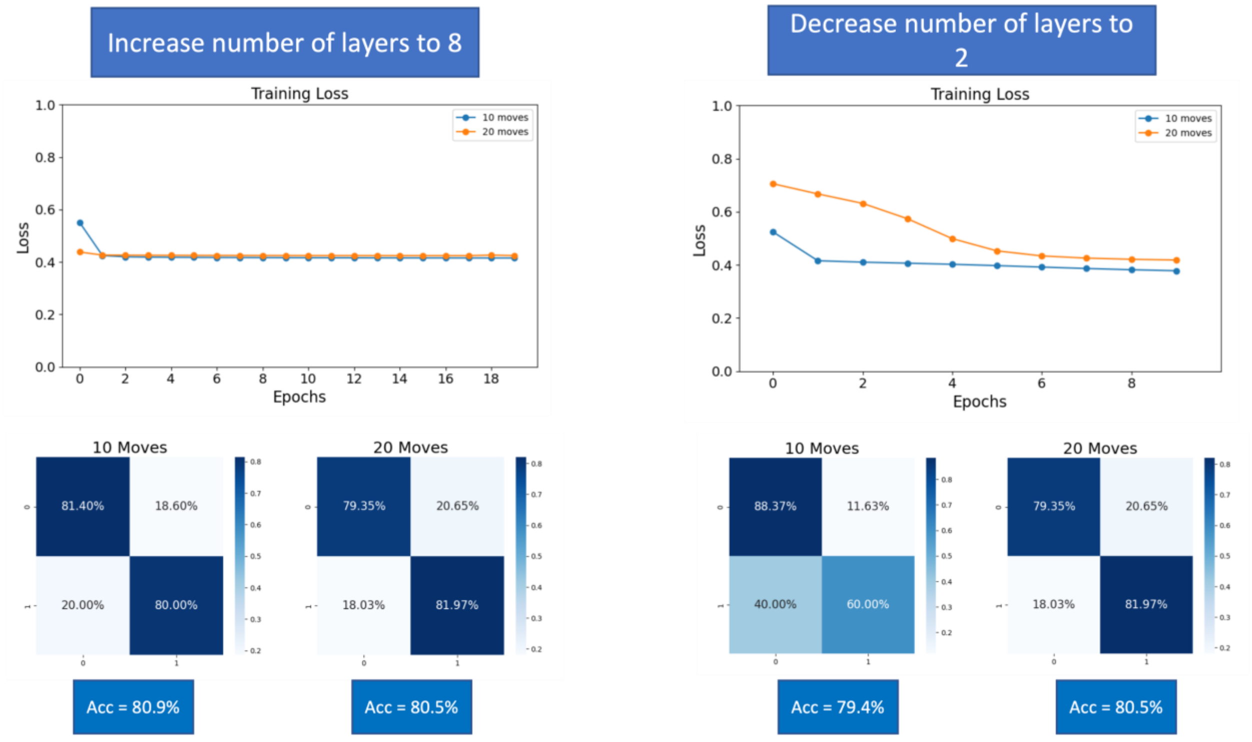


Figure 5. Results of changing the number of recurrent layers

In addition to changing these hyperparameters, we attempted to tune learning rate and changed the activation function to ReLu. In each case, we saw either degraded or similar performance to what is shown in Figure 5. Presently, the best we can do with this dataset and the RNN architecture is 80% accuracy. Note that we did not include games with 50 moves for these experiments because they were very time-consuming. Based on Figure 4, we might expect that this dataset would continue to result in better performance than the other two.

Conclusions and Future Work

The RNN's ability to predict the winner of a chess game based on the first 10, 20, or 50 moves is disappointing, but it seems apparent that giving the model more moves helps it predict with better accuracy. It is possible that making some changes in the data preprocessing stage could show some improvements. We could switch to a one-hot encoding scheme for the moves rather than the (1,5) we are currently using. Even more compelling, however, is the idea of switching to an LSTM. The RNN is known to have a vanishing/exploding gradient problem, and this could account for the stagnation seen in our loss function across all configurations of this model. Also, a chess game is complicated, and its patterns do not become apparent until more than two or three moves are established. The RNN only has short term memory, but the LSTM contains both the RNN's hidden state for short term memory, with an additional cell state which contains long term memory, whose context is propagated through the layer. Not only would an LSTM likely help with learning, but this additionally context would be extremely valuable for this type of problem, requiring "long" memory.

References

[1] "Chess Game Dataset (Lichess)," *www.kaggle.com*. https://www.kaggle.com/datasets/datasnaek/chess