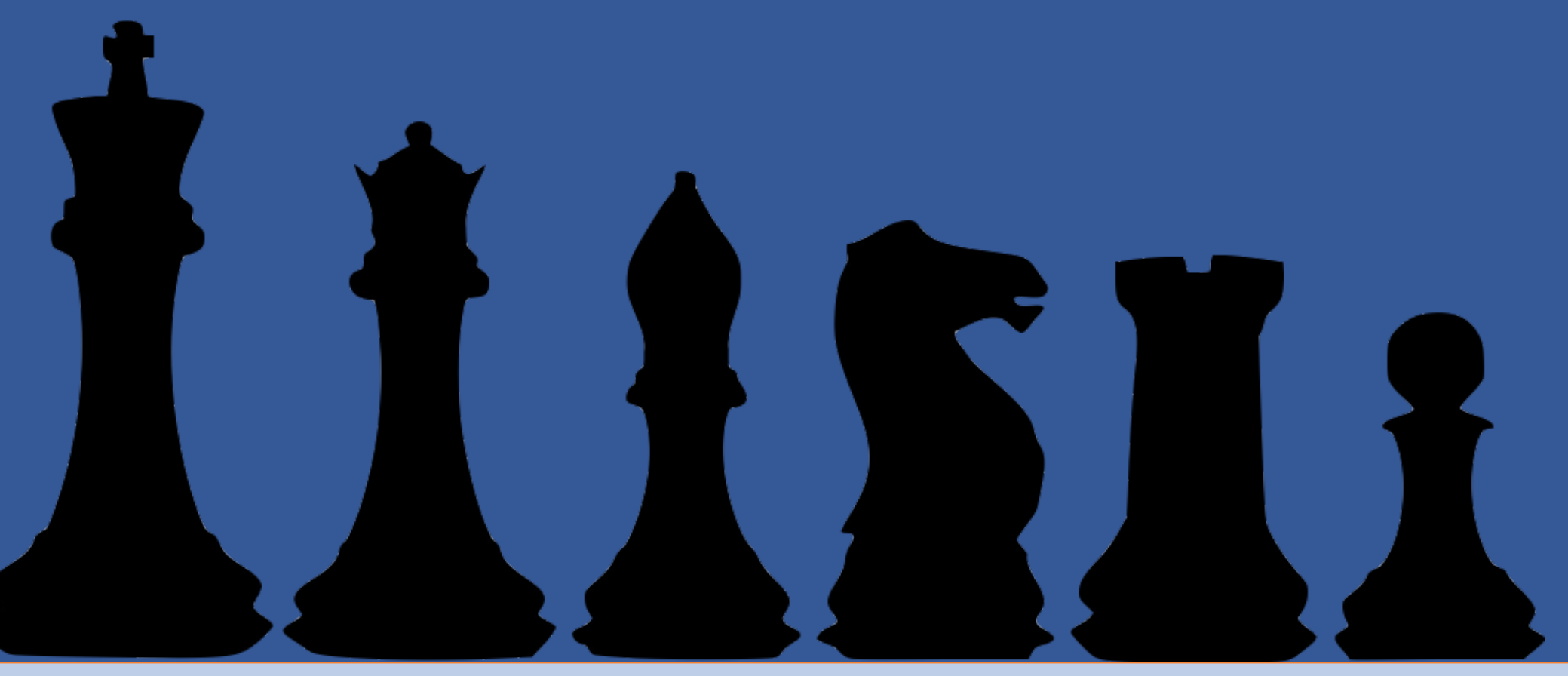
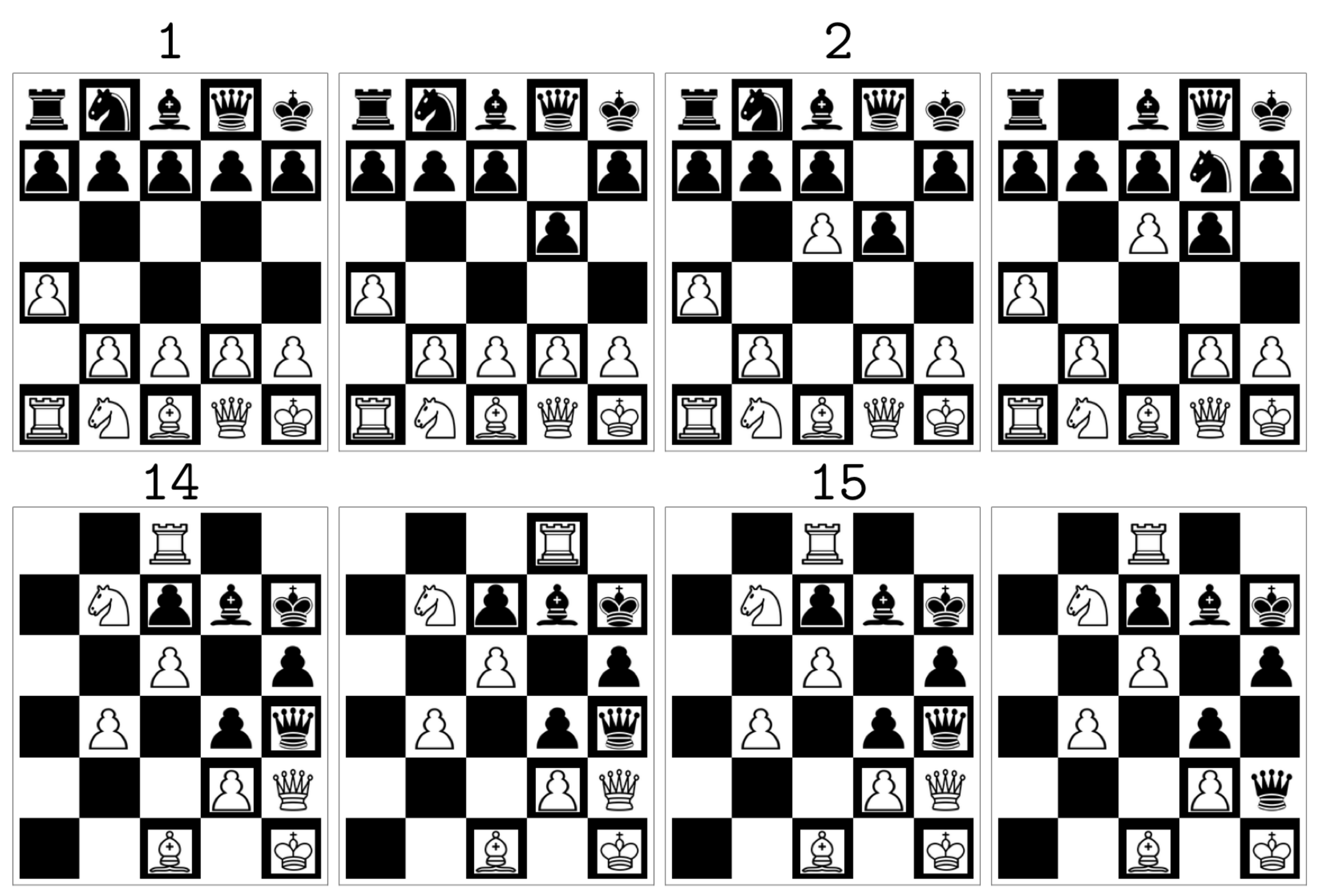


Chess Attack: Board Evaluation with Minimax vs Classifier

Jordan Hoffmann
Department of Applied Mathematics, Harvard University, Cambridge, MA 02138

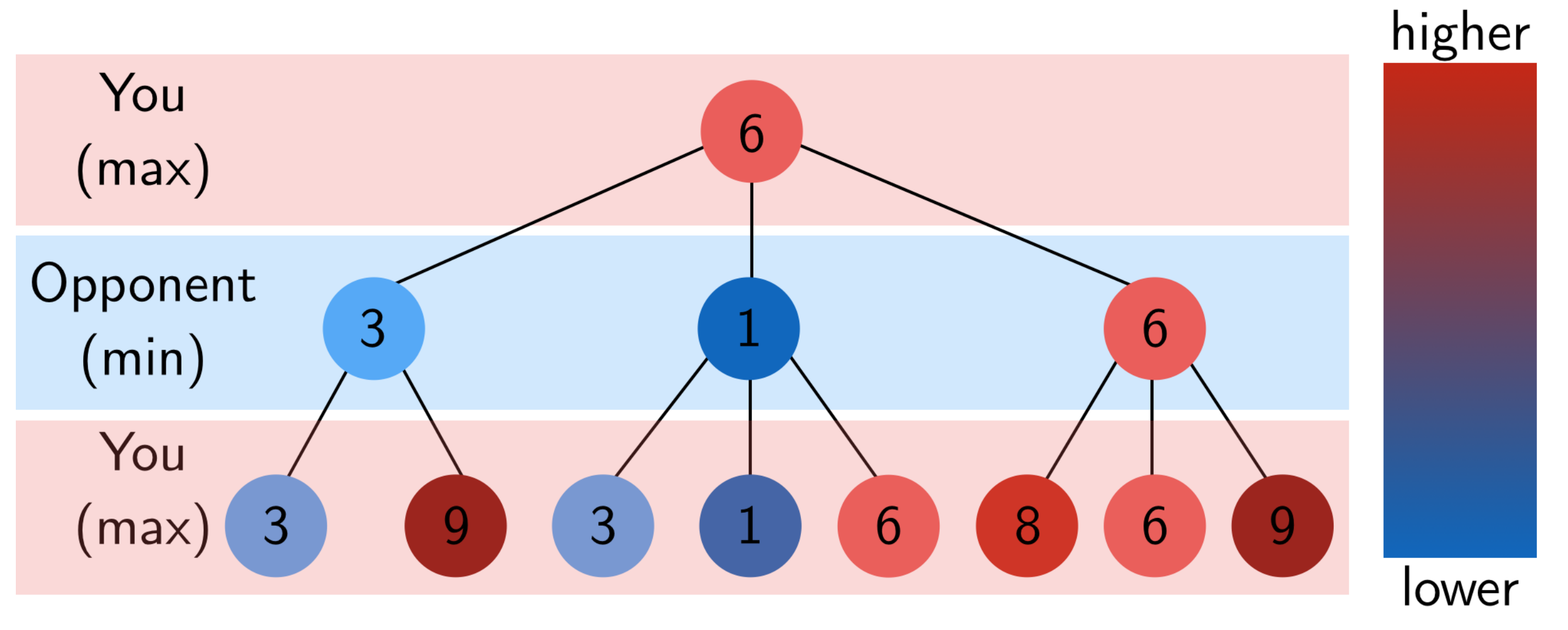
Objective and Motivation

Chess has had a very important role in the development of artificial intelligence and machine learning. Perhaps the first true test of Man versus Machine, the Gary Kasparov – Deep(er) Blue matches of the 1990s marked a pivotal moment in the history of computing. Since then, chess programs have improved markedly, to the point that there is no longer a contest between man and machine. Most state of the art programs use variants of a minimax tree with pruning to search to considerable depths. This approach can be computationally taxing. Here we consider Chess Attack, a variant of chess played on a 6×5 board that is meant to hone endgame skills. We first implement a minimax tree to try to optimize weights of a predetermined parameter vector. We then attempt to replace the 5 ply minimax tree with a single evaluation function.



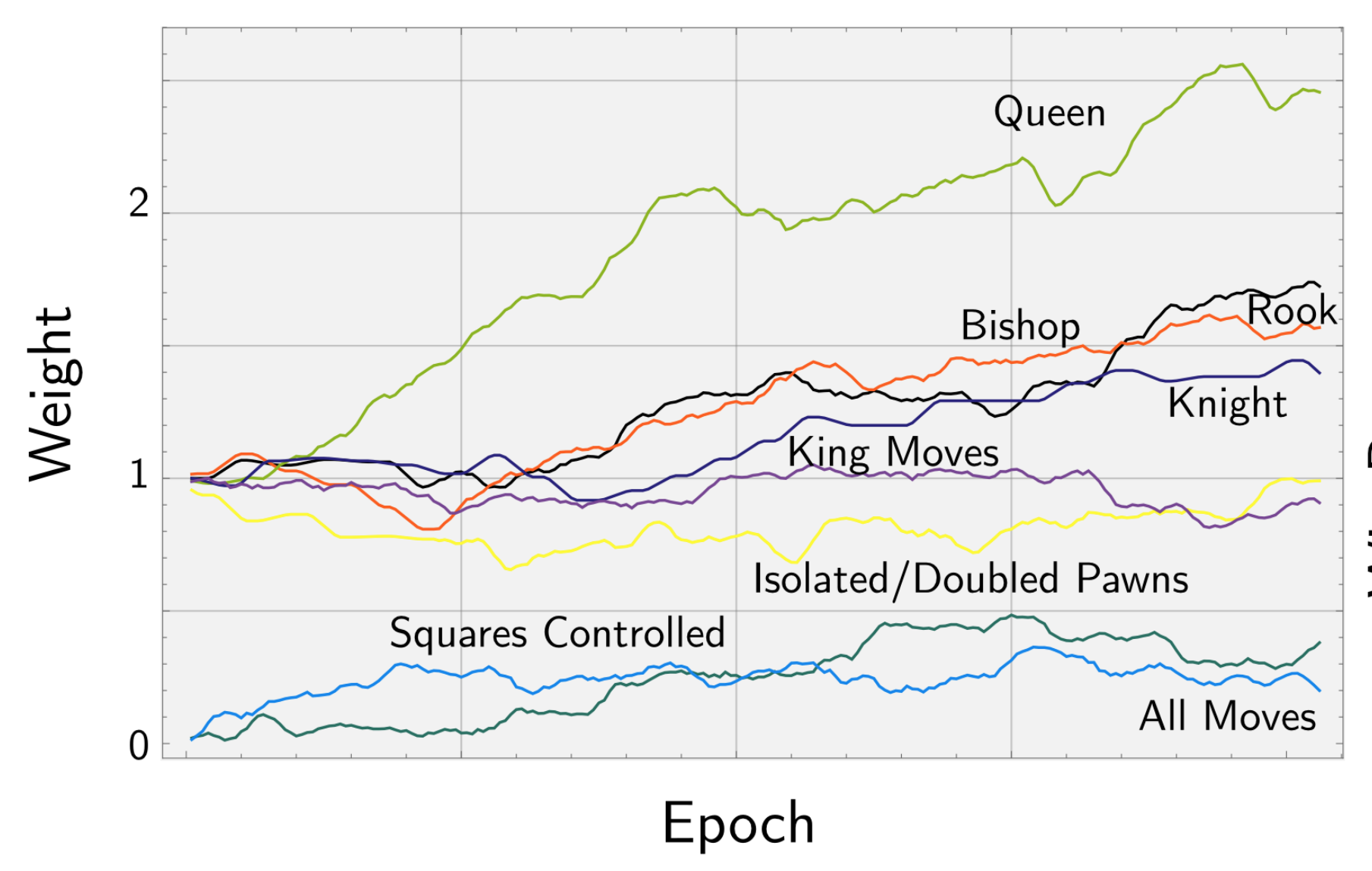
Cost Function Optimization

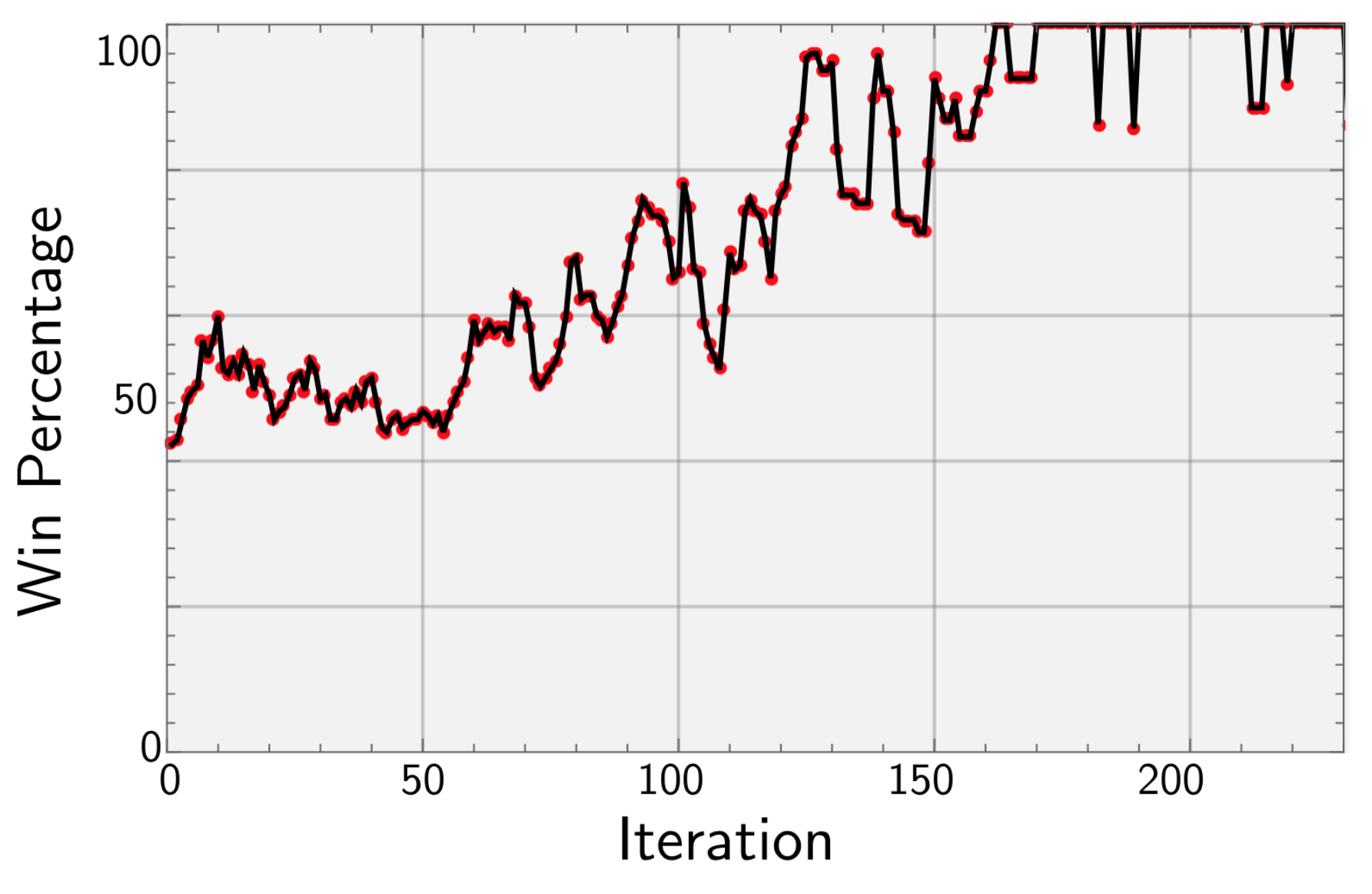
We implement a Chess Attack AI using a minimax tree (truncated version shown **below**), that uses a basic cost function without defined weights. Through self play, we optimize the value of a bishop, rook, knight, queen, no. of available moves, no. of squares controlled, and no. of doubled/isolated pawns. We set a pawn to a fixed value of 1.



Cost Function Optimization Continued

Through various forms of self play, the weights for the cost function were learned. Under the model, many different weighting combinations yield a 100% win rate against random play. Methods of learning included gradient descent as well as Monte Carlo methods.





While the results are bit surprising giving the conventional wisdom, the general trends make sense. It is important to keep in mind that we only search to 5 ply.

Replacing Tree Search with Classification

Searching to a depth of 5 ply without pruning can be very costly. To help, the code is parallelized using MPI and we used Odyssey to help run code. Next, we had the trained program play various other randomly distributed weights for over 10,000 times, storing every board position and the winner. Every initial position was given a score of 50 which linearly increased to 100 if white won and decreased to 0 if black won. We then trained classifiers on these boards and tried to replace the costly search with single board evaluations, **saving a factor of $>$10,000 evaluations**.

Model Theory and Parameters

We begin by classifying the boards using **multi-class logistic regression**. We begin by constructing a weight matrix **W** and a bias vector b. For each input label Y we can compute

$$P(Y=i|\mathbf{x},\mathbf{W},\mathbf{b}) = \frac{\exp(\mathbf{W}_i\mathbf{x}+\mathbf{b}_i)}{\sum_j \exp(\mathbf{W}_j\mathbf{x}+\mathbf{b}_j)} \text{ for all } i.$$

Where $P(Y=i)$ is the probability of belonging to class i and $\mathbf{W}_i$ is column i of $\mathbf{W}$. Ultimately our classifier will assign each Y to an i that maximizes this value. To train our weight and bias vectors, we minimize the negative log-liklihood

$$\mathrm{NLL}(\theta=\{\mathbf{W},\mathbf{b}\},\mathcal{D}) = -\sum_i \log(P(Y=y_i|\mathbf{x}_i,\mathbf{W},\mathbf{b}))$$

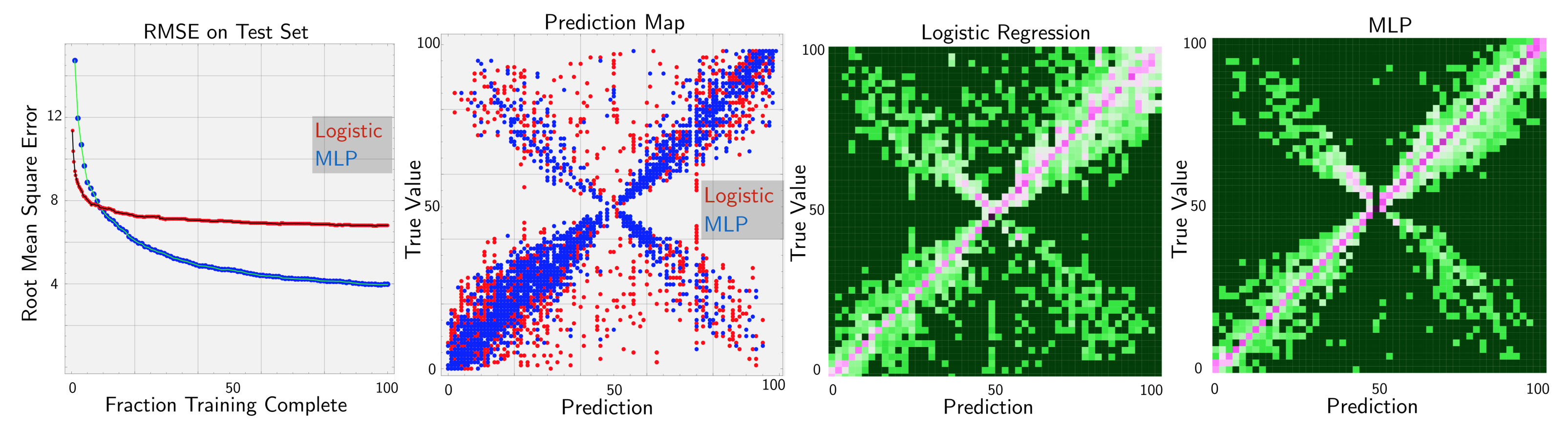
using stochastic gradient descent with minibatches. To better model nonlinear relationships, we add a hidden layer and use a **multi-layer perceptron** (**MLP**). We consider two bias vectors, $\mathbf{b}_1$ and $\mathbf{b}_2$ along with two weight matrices, $\mathbf{W}$ and $\mathbf{V}$. We consider activation functions G and J. We have a model of the form

$$f(x) = G(\mathbf{b}_2 + \mathbf{V}(J(\mathbf{b}_1+\mathbf{W}\mathbf{x}))).$$

Here, we take $J=\tanh(\cdot)$. To help learn the nonlinear relations expected in chess, we chose to include a regularization term that we add to the NLL given by $\lambda||\theta||_2^2$.

Board Classification

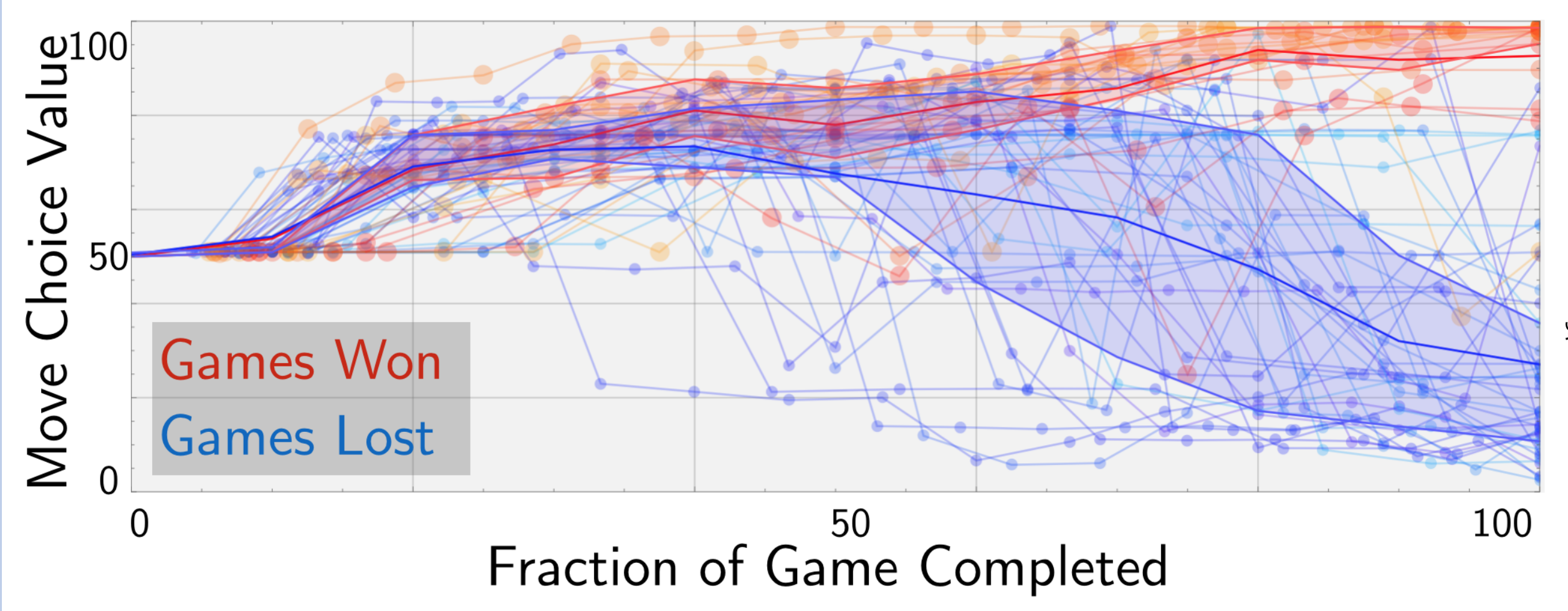
After training our two classifiers, we test our classifier on the test set. We find that the prediction error converges (**far left**) for both models.

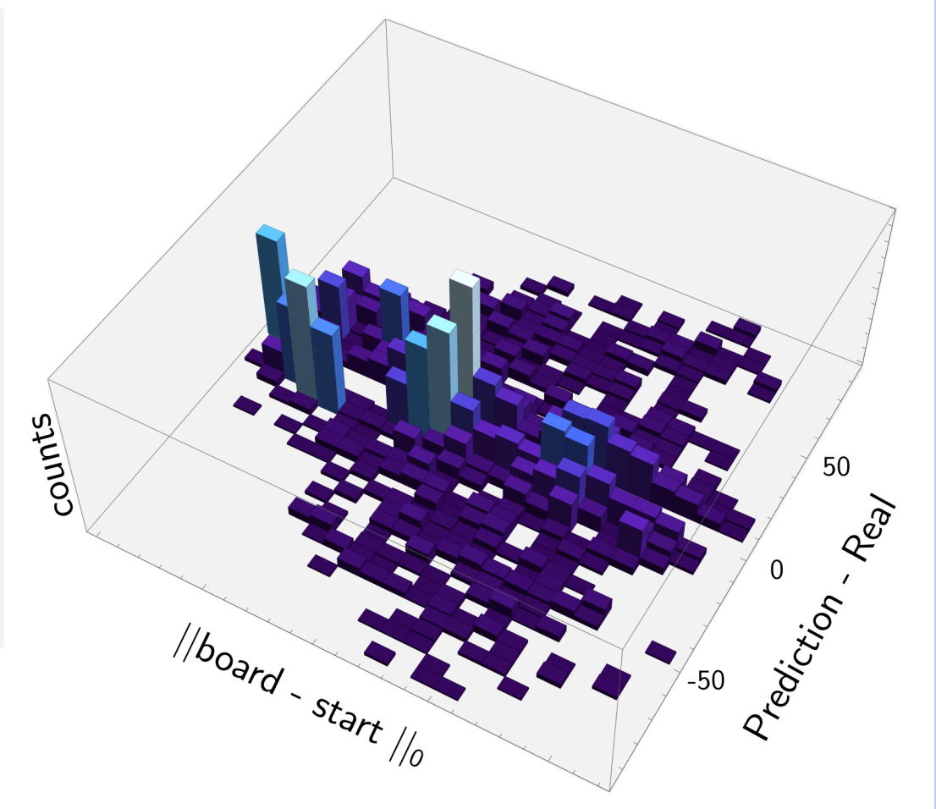


The panels in the **center** and on the **right** show the true and predicted values for the two different classifiers. The incorrect diagonal is an interesting result that significantly impairs the performance.

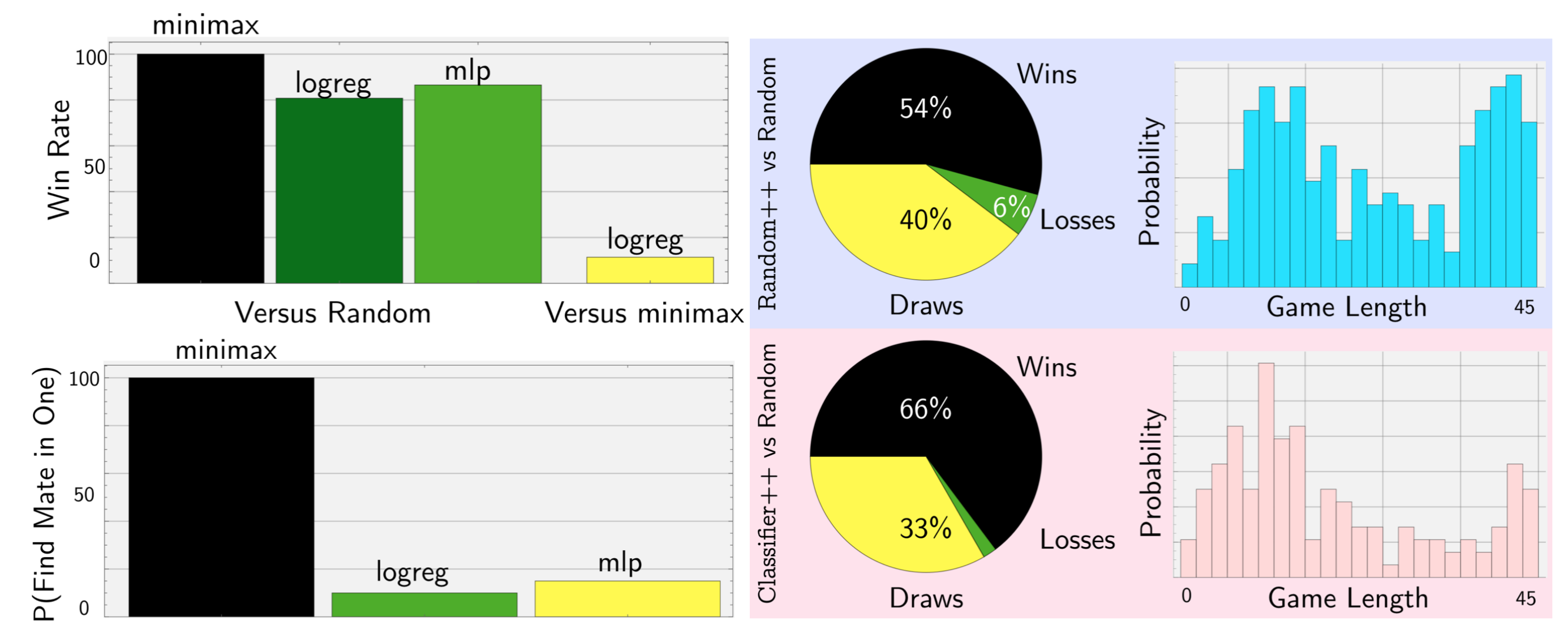
Results and Discussion

We attempted to replace the costly minimax based approach with two different types of classifiers. **Below** we plot the evaluation of the board as a function of fraction of game completed for games that were won (red) and lost (blue). The results are quite interesting: the classifier does seem to know when it winning or losing however when it is losing, it is often not able to classify the board correctly in the moves leading up to its loss.





Above, on the **right** we plot the difference between the real and actual evaluation as a function of difference from starting position. Notice that as the game goes on, the variance increases, but decreases at the end of the game.



Above on the **left, upper** we plot results of various methods to make a move playing against a random player. Below that, we plot the probability of finding a mate in one move, if there is one present. Though the probability is very low, it is slightly better than random. This is perhaps the crucial performance issue. Our first pass addressing this was including a mate-in-one finder (**right**). The performance improved considerably and the game length decreased.