Chess Opening Classification

Solving a nuanced classification problem through Principal Component Analysis and Neural Networks

Abstract

The goal of this paper is to apply machine learning algorithms to the problem of Chess opening classifications. Representing Chess move sequences as matrices allows for the employment of principal component analysis and neural network classification. After discussing the implementation of these two models, we compare their performance and reach conclusions regarding each model's accuracy and room for improvement.

Introduction & Problem Statement

A traditional Chess game is broken up into three sections: an opening, midgame, and late game. The opening is characterized by initial piece development, where common goals include activating as many pieces as possible by moving them out of their starting positions, controlling the center of the board, and preparing for proper defense, ex. Castling. There are many styles of openings developed by masters of the game throughout history and, although many avid chess players have favorites, there is no one single opening deemed to be the best.

The core focus of this paper is to detail the development of algorithms to classify Chess openings. There are ten openings that the algorithm we develop must successfully classify:

1. Sicilian Defense
2. French Defense
3. Ruy Lopez
4. Italian Game
5. English Opening
6. Queen's Gambit Declined
7. Caro-Kann Defense
8. King's Indian Defense
9. Queen's Pawn Game
10. Nimzo-Indian Defense

To solve this classification problem, we evaluate the performance of PCA and neural networks. The input data of a Chess opening is later discussed in the Data Design & Implementation, but the gist of the problem has been explained as a classification of 10 different chess openings given a move sequence.

Related Work

Machine learning has been applied to the field of chess and brought to market in many ways. Nearly all online chess games feature a mode to play against the computer and forcing the computer to make random moves would make for an extremely easy to beat opponent. To form calculated, thought out moves for the computer to play against its human competitor, it employs machine learning algorithms to discern the most appropriate move of all its options to play. Some artificial intelligence chess algorithms are stronger than others, and there are a select few that hover above the rest as some of the most notable: DeepBlue, which was developed by IBM and remarkably defeated the world chess champion Gary Kasparov over a six-game match, crowning it the first computer to do so. Shredder, a chess software application used by many students, which holds a feature allowing fine tuned chess elo simulation settings for a wide range of potential difficulties.

In his paper “Learning to Play the Game of Chess”, Sebastian Thun from the University of Bonn discusses his implementation of a chess move generating algorithm utilizing neural networks scraping information from end game states of wins, ties, and losses. This was achieved in part by the employment of *temporal difference learning*, or TD, a method of developing evaluation functions by the use of recursion. TD had been previously used in a Backgammon AI written by A. L. Simon, which played Backgammon games at a high level. However, it struggled when it was trained with different games like chess and go. Thun improved the algorithm by enforcing more analytical thinking with chess-specific spatial understanding which influences the AI above Simon’s more simplistic inductive neural network approach. An example he mentions is that of a knight fork between a queen and a king (a simultaneous attack which forces the sacrifice of the queen in defense of the king). A purely inductive neural network will require extremely large amounts of training data and time to discover that it is the position of both the queen and the king in relation to the knight which makes this position most unfavorable in comparison to, like the article suggests, the number of weak pawns. Unfortunately, this algorithm struggles as this neural network does not have the capability to assign unlimited trials of each possible feature combination, which blocks it from finding many, potentially powerful, moves.

One of the main problems chess machine learning algorithms face is the large number of possible moves which all have separate influences on an outcome of a game. It is very hard to tell which move may hold more blame for a loss than another when the correctness of each move is a) only truly discovered at the end of the game and b) is so influenced by the current board state. David E. Fogel et al. attempted to combat these issues in their paper “A Self-Learning Evolutionary Chess Program”, where they used three neural networks in conjunction with a genetic algorithm to compute powerful chess moves. Interestingly, the three neural networks are specialized to compute values in specific areas of the board. The algorithm judges given movesets by genetic algorithm, and generates a future extension of how the game will play out by way of the neural networks. The code can then decide which move will be the best.

To analyze chess algorithms, one can look at the coding implementations and visit how certain techniques can modify move combinations. However, these movesets have stylistic flairs that can be described by chess players utilizing pertinent vocabulary. The powerful Stockfish engine tends to, by the account of Sreerag SR in his paper “Evaluation of Strategy by AlphaZero and StockFish on Chess Game” describes AlphaZero, the power chess engine, to have the following traits:

- Prefers closed pawn systems

- Pawn structure is of extreme importance when deciding to train majors and minors
- Prefers early castling and prioritizes defense on the castling side
- Tries to trade heavily

Giraffe, a chess move algorithm that uses automatic feature extraction, pattern recognition, and parameter tuning on evaluation functions, was built by Matthew Lai in his paper "Giraffe: Using Deep Reinforcement Learning to Play Chess". Using these tools, Giraffe would have equivalent chess elo with a FIDE International Master. The code creates trees of potential moves from multiple given positions. Then, using neural networks, probabilities of winning are broadly generalized by the neural network such that the program can narrow down specific subtrees for the program to further analyze. Rather than encode each board state as a bitmap, which may cause the algorithm to struggle due this method's inability to demonstrate the complex inequality of pieces distance from each other and the nature in which these positions influence how strong the current board state is, Giraffe uses lists of pieces and their corresponding coordinates. This way, pieces that have similar coordinates can have closer evaluations.

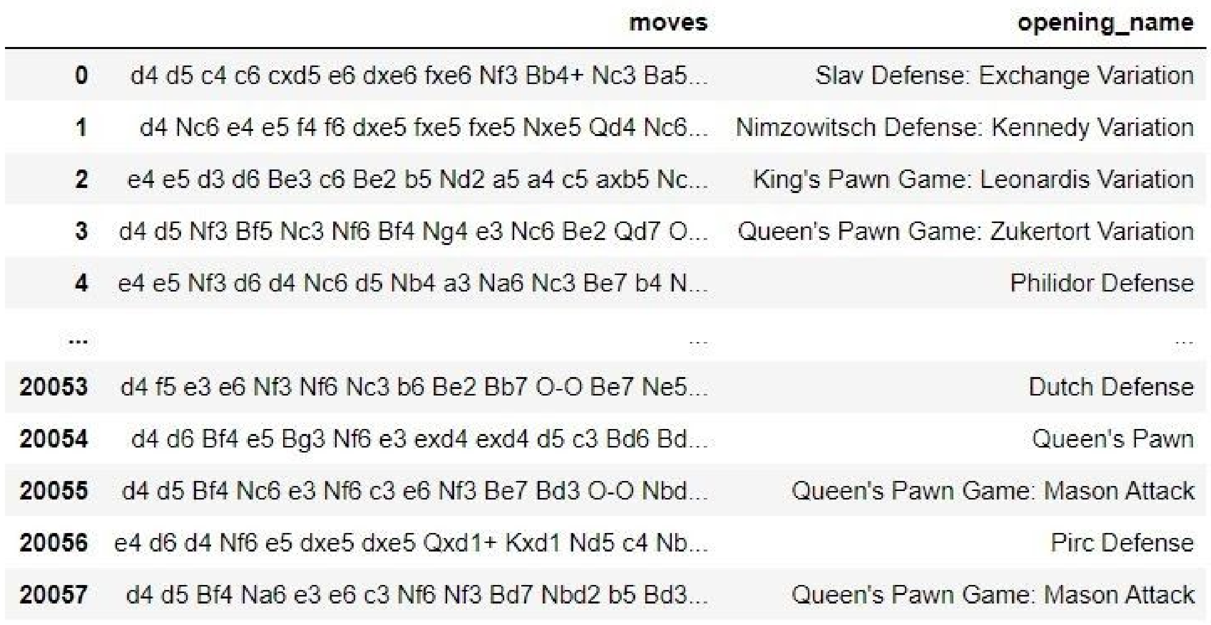
A strong position comparison function is essential to develop a robust chess playing AI. Usually people rate board states with an integer: a large positive rating will imply white is winning, 0 is a tied board state, and negative numbers favor black. One of the most simplistic ways in which we rate boards is by counting pieces remaining on the board for each side. Each piece additionally holds a weight. When white takes a piece with a large weight, like a queen, we rate the board state as a higher integer than when white takes one with a small weight, say a pawn. In addition to this, however, winning percentages are also influenced by board positions and traits of the games in comparison to which remaining pieces are left on each side. If a game is open, meaning there are relatively few pawns blocking piece progression, then bishops are typically favored. Two bishops in the late game, dubbed the bishop pair, are exceptionally powerful in the late game. Other spatial imbalances, like open files (columns where there are no pawns blocking the way of the rooks and queen), the presence of castling, and more also have small but together profound influences on a proper rating of the current boardstate. DeepChess by Eli David et al. determines stronger moves and thus, board states, using supervised learning: it trains over randomly generated input pairs of game states and is told which side has won that game.

Design & Implementation

Data Collection & Cleaning

To find trends in opening positions in order to classify them, we found an online dataset from Kaggle containing around 20,000 sequences of different chess games along with their associated opening name, player ratings, number of moves and more. To clean the data in preparation for putting it in a more usable form for classification, we removed all columns outside of each game's move list and its respective opening.

Our raw dataset after irrelevant columns were removed

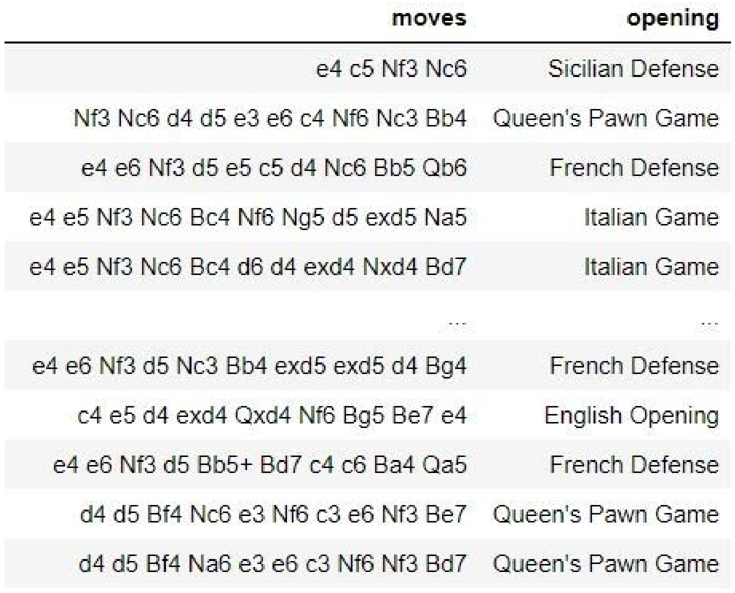


	moves	opening_name
0	d4 d5 c4 c6 cxd5 e6 dxe6 fxe6 Nf3 Bb4+ Nc3 Ba5...	Slav Defense: Exchange Variation
1	d4 Nc6 e4 e5 f4 f6 dxe5 fxe5 fxe5 Nxe5 Qd4 Nc6...	Nimzowitsch Defense: Kennedy Variation
2	e4 e5 d3 d6 Be3 c6 Be2 b5 Nd2 a5 a4 c5 axb5 Nc...	King's Pawn Game: Leonardis Variation
3	d4 d5 Nf3 Bf5 Nc3 Nf6 Bf4 Ng4 e3 Nc6 Be2 Qd7 O...	Queen's Pawn Game: Zukertort Variation
4	e4 e5 Nf3 d6 d4 Nc6 d5 Nb4 a3 Na6 Nc3 Be7 b4 N...	Philidor Defense
...	...	...
20053	d4 f5 e3 e6 Nf3 Nf6 Nc3 b6 Be2 Bb7 O-O Be7 Ne5...	Dutch Defense
20054	d4 d6 Bf4 e5 Bg3 Nf6 e3 exd4 exd4 d5 c3 Bd6 Bd...	Queen's Pawn
20055	d4 d5 Bf4 Nc6 e3 Nf6 c3 e6 Nf3 Be7 Bd3 O-O Nbd...	Queen's Pawn Game: Mason Attack
20056	e4 d6 d4 Nf6 e5 dxe5 dxe5 Qxd1+ Kxd1 Nd5 c4 Nb...	Pirc Defense
20057	d4 d5 Bf4 Na6 e3 e6 c3 Nf6 Nf3 Bd7 Nbd2 b5 Bd3...	Queen's Pawn Game: Mason Attack

Many of the already assigned opening titles had overly specific naming schemes. For example, a Queen's Pawn Game may be further classified as a Mason Attack or the Zukertort Variation. For simplicity, we wrote and used a script to remove all terminology referring to alternate variations, and treated each like they were the main opening of which they were derived from.

A chess opening does not have a set number of moves, but rather it is classified more by the general principle of the period of which the two players initialize their activation of their pieces. Because discerning when an opening has finalized and the midgame has begun is an overly complex feat of itself, we decided to classify based on the first 5 moves of each player, which totals to a set of ten moves for each moveset. These steps lead us to construct a new table of the first ten moves of each move set alongside its simplified opening title.

Our cleaned dataset, with moveset reduced to <= 10 moves and the opening titles simplified



moves	opening
e4 c5 Nf3 Nc6	Sicilian Defense
Nf3 Nc6 d4 d5 e3 e6 c4 Nf6 Nc3 Bb4	Queen's Pawn Game
e4 e6 Nf3 d5 e5 c5 d4 Nc6 Bb5 Qb6	French Defense
e4 e5 Nf3 Nc6 Bc4 Nf6 Ng5 d5 exd5 Na5	Italian Game
e4 e5 Nf3 Nc6 Bc4 d6 d4 exd4 Nxd4 Bd7	Italian Game
...	...
e4 e6 Nf3 d5 Nc3 Bb4 exd5 exd5 d4 Bg4	French Defense
c4 e5 d4 exd4 Qxd4 Nf6 Bg5 Be7 e4	English Opening
e4 e6 Nf3 d5 Bb5+ Bd7 c4 c6 Ba4 Qa5	French Defense
d4 d5 Bf4 Nc6 e3 Nf6 c3 e6 Nf3 Be7	Queen's Pawn Game
d4 d5 Bf4 Na6 e3 e6 c3 Nf6 Nf3 Bd7	Queen's Pawn Game

After consideration with our classification methods in mind, we decided the best form of representation of each opening was a series of 64x1 matrices where each was a model of each board state after each move in the adjusted moveset. Each entry in the matrix corresponded with a specific position on the board. Values were given according to what type of piece occupied that spot: the more valuable a piece was in a given location, the more extreme the entry would be in the matrix.

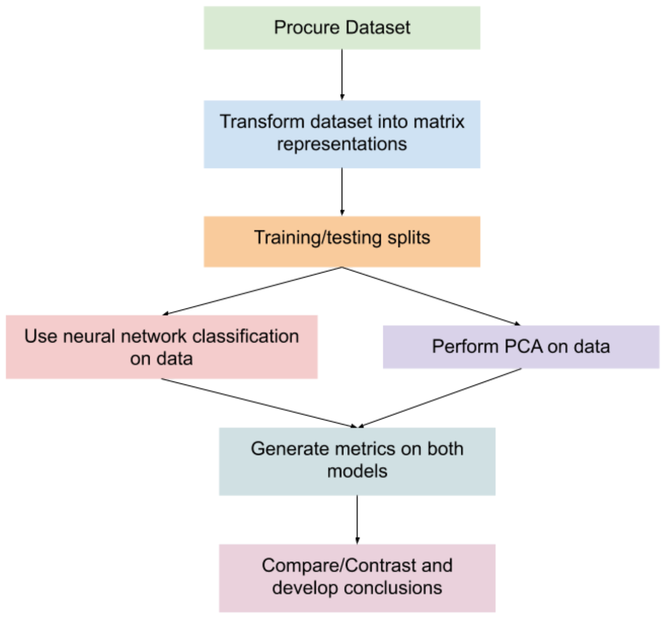
To build these matrices from strings of movesets, we had to work backwards. Viewing the first move, we analyzed from where that piece could have moved to end up in that location. Once we found these locations, we then visited them in the base state of the chess board. Any piece that matched the piece type of the moved piece must have been the piece that was moved, due to the non-ambiguous nature of chess notation. This process was repeated with the next move in series in reference to the previously generated board state, again and again until all ten moves were represented as matrices.

We also included support for understanding matrix representations of both types of castling as well as ambiguous piece movement (which is not truly ambitious, as the factors clearing up the ambiguity are explained in the notation. EX: Nfxe4. This is somewhat confusing due to the fact that multiple knights may have the opportunity to take on e4, but the information ‘f’ stating the knight in question originated from the f file should, in the game’s context, give the player full information in order to understand from where the knight came from).

The corresponding matrix for the base boardstate

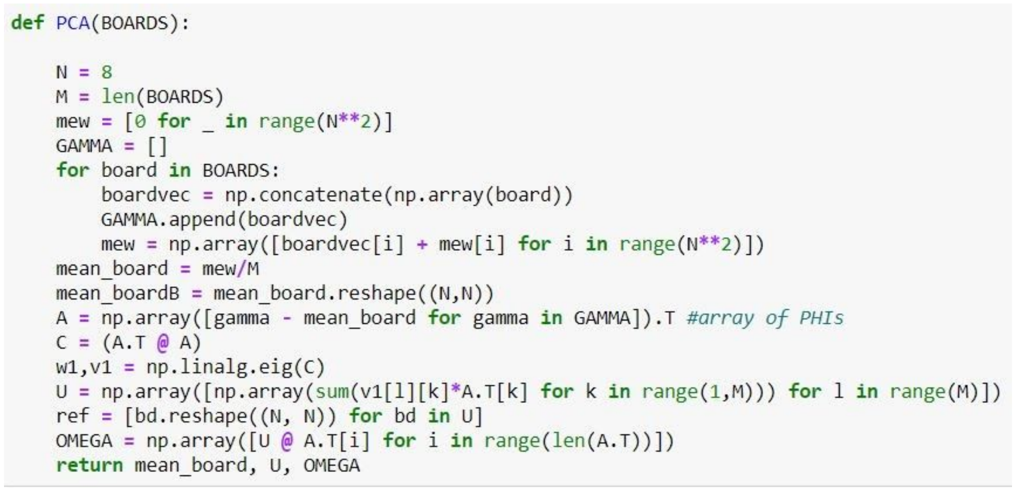
	a	b	c	d	e	f	g	h
0	5000	2000	3000	9000	12000	3000	2000	5000
1	1000	1000	1000	1000	1000	1000	1000	1000
2	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0
6	-1000	-1000	-1000	-1000	-1000	-1000	-1000	-1000
7	-5000	-2000	-3000	-9000	-12000	-3000	-2000	-5000

Here is a general pipeline of our steps taken to classify Chess Openings


Procure Dataset
Transform dataset into matrix representations
Training/testing splits
Use neural network classification on data
Perform PCA on data
Generate metrics on both models
Compare/Contrast and develop conclusions

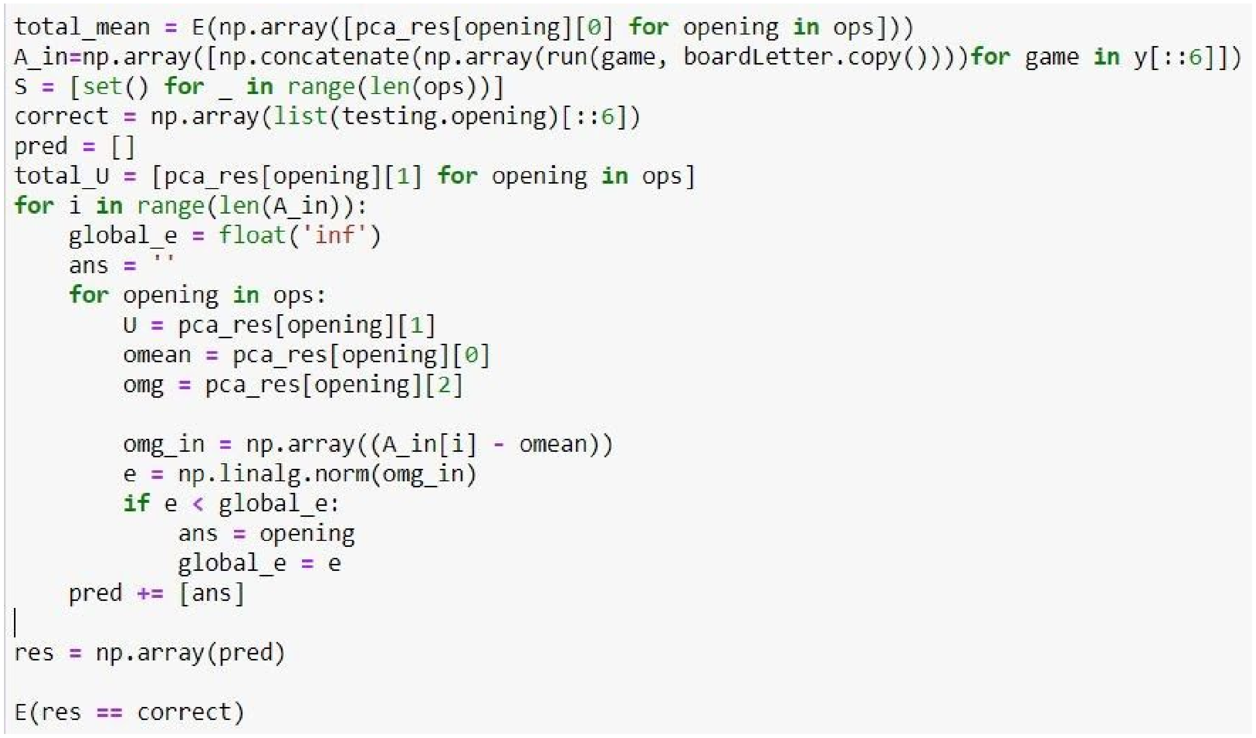
PCA Design & Implementation

The principal component analysis used in Chess opening classification is similar to that of EigenFace recognition. For a mathematical overview, please refer to the previously referenced paper on EigenFace generation with PCA. In essence, PCA reduces dimensionality in data by using eigenvectors to represent certain board openings.



```python
def PCA(BOARDS):

    N = 8
    M = len(BOARDS)
    mew = [0 for _ in range(N**2)]
    GAMMA = []
    for board in BOARDS:
        boardvec = np.concatenate(np.array(board))
        GAMMA.append(boardvec)
        mew = np.array([boardvec[i] + mew[i] for i in range(N**2)])
    mean_board = mew/M
    mean_boardB = mean_board.reshape((N,N))
    A = np.array([gamma - mean_board for gamma in GAMMA]).T #array of PHIs
    C = (A.T @ A)
    w1,v1 = np.linalg.eig(C)
    U = np.array([np.array(sum(v1[l][k]*A.T[k] for k in range(1,M))) for l in range(M)])
    ref = [bd.reshape((N, N)) for bd in U]
    OMEGA = np.array([U @ A.T[i] for i in range(len(A.T))])
    return mean_board, U, OMEGA
```

The function above was written to serve as a PCA pipeline to generate EigenOpenings for each opening. Represented by each opening is a mean board as well as an eigenvector matrix that can later be used to compute differences between input images.

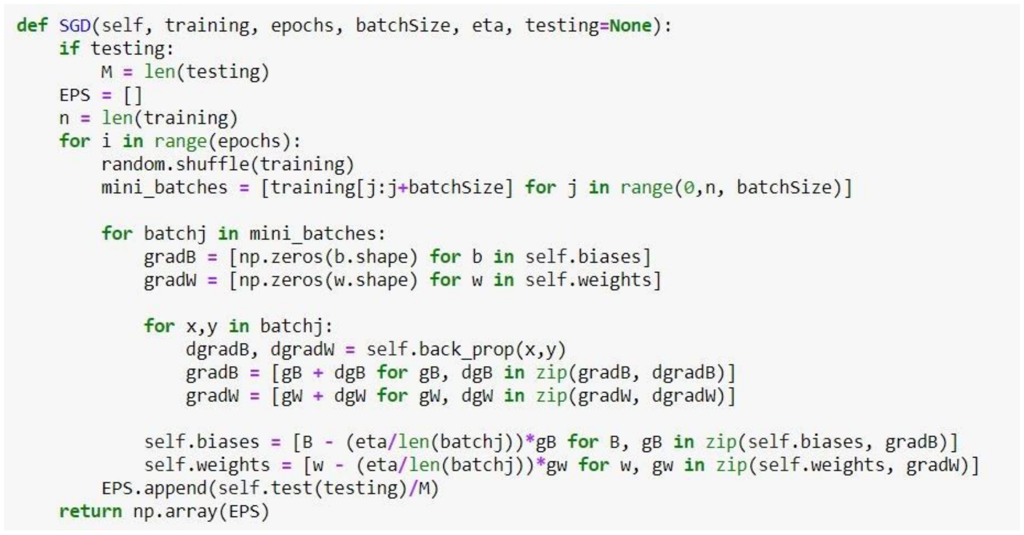


```python
total_mean = E(np.array([pca_res[opening][0] for opening in ops]))
A_in=np.array([np.concatenate(np.array(run(game, boardLetter.copy())))for game in y[::6]])
S = [set() for _ in range(len(ops))]
correct = np.array(list(testing.opening)[::6])
pred = []
total_U = [pca_res[opening][1] for opening in ops]
for i in range(len(A_in)):
    global_e = float('inf')
    ans = ''
    for opening in ops:
        U = pca_res[opening][1]
        omean = pca_res[opening][0]
        omg = pca_res[opening][2]

        omg_in = np.array((A_in[i] - omean))
        e = np.linalg.norm(omg_in)
        if e < global_e:
            ans = opening
            global_e = e
    pred += [ans]

res = np.array(pred)

E(res == correct)
```

The code above is for classifying input images from PCA. The algorithm performs the classification, then takes the NumPy array of the predictions and the NumPy array of the actual classes to generate an array of the indicator for if the prediction is correct. If one takes the expected value of that array, we should get the overall accuracy of the model, which is discussed in the results section.

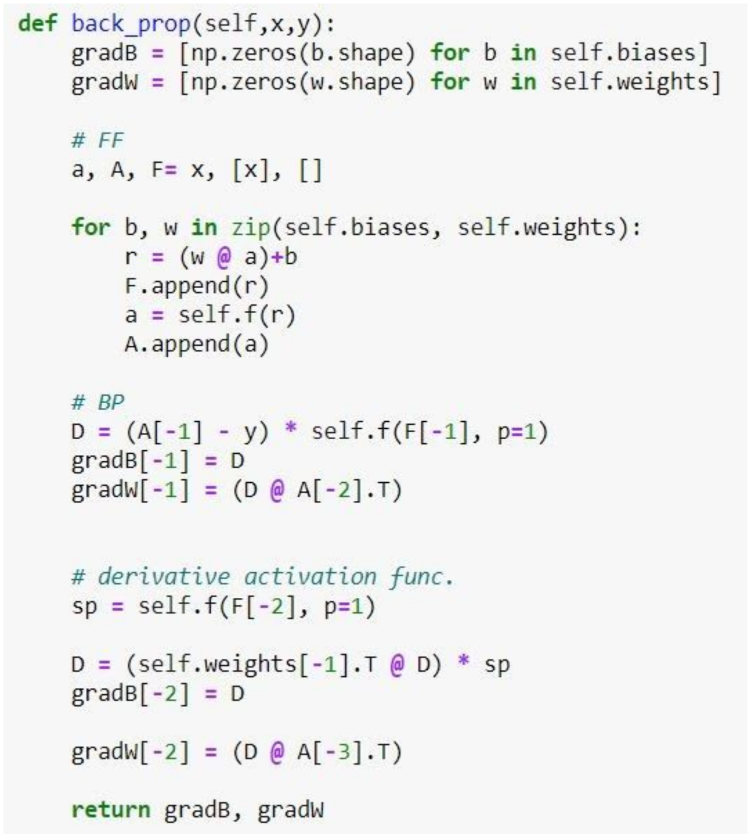
Neural Network Design & Implementation

Our classification network utilizes a three-layer network to perform Chess opening classification. There is one hidden layer. The neural network was manually implemented as a Python class to allow for maximum customizability. The neural network is initialized with random weights and biases. The main methods of the *NeuralNetwork* class in Python are stochastic gradient descent, backwards propagation, feed forward activation, and a method for testing and classification.



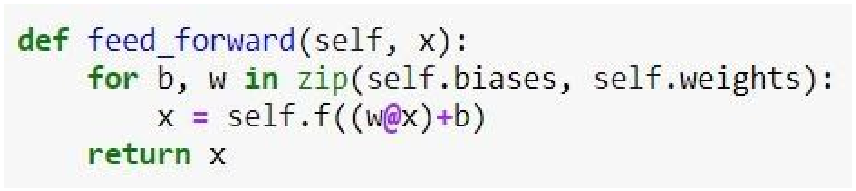
```python
def SGD(self, training, epochs, batchSize, eta, testing=None):
    if testing:
        M = len(testing)
    EPS = []
    n = len(training)
    for i in range(epochs):
        random.shuffle(training)
        mini_batches = [training[j:j+batchSize] for j in range(0,n, batchSize)]

        for batchj in mini_batches:
            gradB = [np.zeros(b.shape) for b in self.biases]
            gradW = [np.zeros(w.shape) for w in self.weights]

            for x,y in batchj:
                dgradB, dgradW = self.back_prop(x,y)
                gradB = [gB + dgB for gB, dgB in zip(gradB, dgradB)]
                gradW = [gW + dgW for gW, dgW in zip(gradW, dgradW)]

            self.biases = [B - (eta/len(batchj))*gB for B, gB in zip(self.biases, gradB)]
            self.weights = [w - (eta/len(batchj))*gw for w, gw in zip(self.weights, gradW)]
        EPS.append(self.test(testing)/M)
    return np.array(EPS)
```

Stochastic gradient descent is where the learning occurs over many epochs. This method splits the training data into mini batches to use for learning, and then adjusts the weights to minimize a loss function through back propagation. Then, it tests the current iteration of the network to check its accuracy, then moves onto the next epoch.



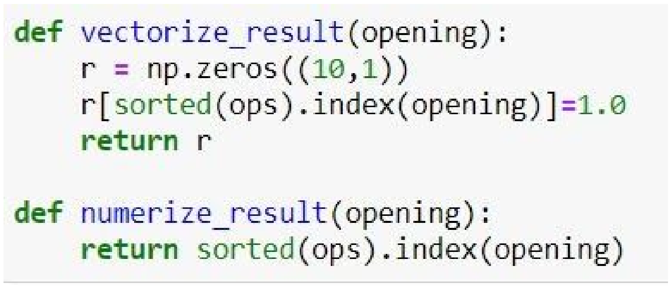
```python
def back_prop(self,x,y):
    gradB = [np.zeros(b.shape) for b in self.biases]
    gradW = [np.zeros(w.shape) for w in self.weights]

    # FF
    a, A, F= x, [x], []

    for b, w in zip(self.biases, self.weights):
        r = (w @ a)+b
        F.append(r)
        a = self.f(r)
        A.append(a)

    # BP
    D = (A[-1] - y) * self.f(F[-1], p=1)
    gradB[-1] = D
    gradW[-1] = (D @ A[-2].T)


    # derivative activation func.
    sp = self.f(F[-2], p=1)

    D = (self.weights[-1].T @ D) * sp
    gradB[-2] = D

    gradW[-2] = (D @ A[-3].T)

    return gradB, gradW
```

The back-propagation function is where the neural network calculates the gradients of the weights and biases in order to properly adjust weights towards a minimal loss.



```python
def feed_forward(self, x):
    for b, w in zip(self.biases, self.weights):
        x = self.f((w@x)+b)
    return x
```

The feed-forward function applies the sigmoid activation function to the weights, input, and bias. This is used to calculate probability of being in a certain class, which is later used by softmax activation to classify an opening by maximum probability. To format the training and testing data for the network, we employ vectorization of the openings and move sequences. These functions can be seen below.



```python
def vectorize_result(opening):
    r = np.zeros((10,1))
    r[sorted(ops).index(opening)]=1.0
    return r

def numerize_result(opening):
    return sorted(ops).index(opening)
```

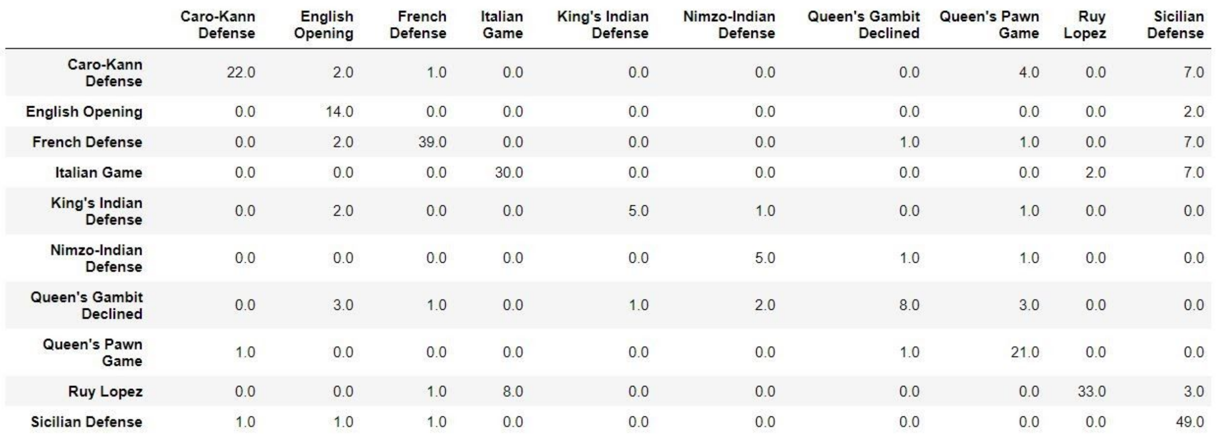
The vectorization can be seen as an indexer for each opening, associated with a certain index in a 10-vector. The numerize_result function is then used to convert the vectorized opening back into a text opening. With the design and implementation explained, we can now observe model results.

Results

Principal Component Analysis Results

The implementation of PCA for classifying Chess openings yielded interesting results. Let us first look at the confusion matrix.

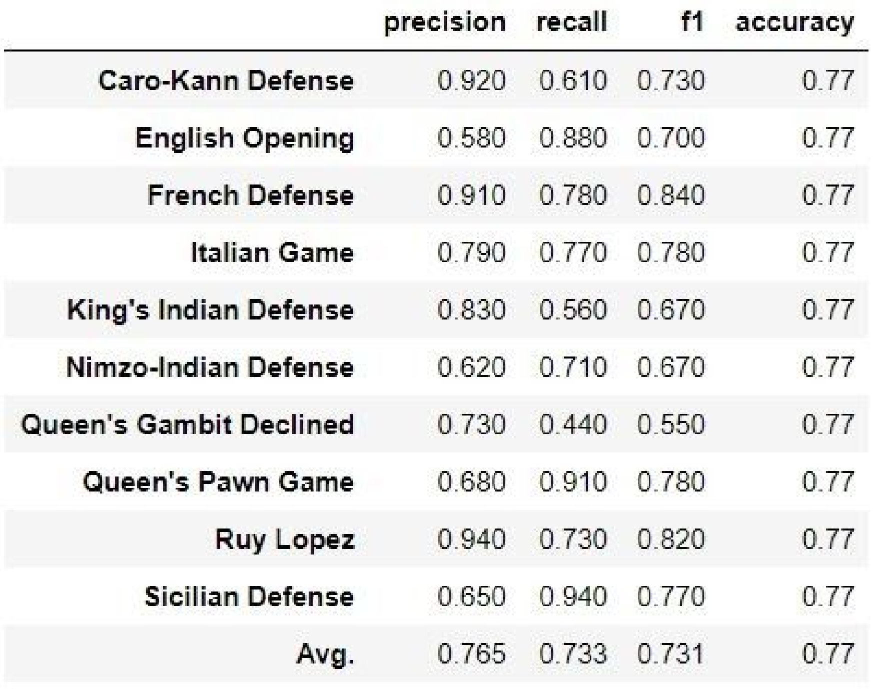
Confusion Matrix for PCA Model



	Caro-Kann Defense	English Opening	French Defense	Italian Game	King's Indian Defense	Nimzo-Indian Defense	Queen's Gambit Declined	Queen's Pawn Game	Ruy Lopez	Sicilian Defense
Caro-Kann Defense	22.0	2.0	1.0	0.0	0.0	0.0	0.0	4.0	0.0	7.0
English Opening	0.0	14.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	2.0
French Defense	0.0	2.0	39.0	0.0	0.0	0.0	1.0	1.0	0.0	7.0
Italian Game	0.0	0.0	0.0	30.0	0.0	0.0	0.0	0.0	2.0	7.0
King's Indian Defense	0.0	2.0	0.0	0.0	5.0	1.0	0.0	1.0	0.0	0.0
Nimzo-Indian Defense	0.0	0.0	0.0	0.0	0.0	5.0	1.0	1.0	0.0	0.0
Queen's Gambit Declined	0.0	3.0	1.0	0.0	1.0	2.0	8.0	3.0	0.0	0.0
Queen's Pawn Game	1.0	0.0	0.0	0.0	0.0	0.0	1.0	21.0	0.0	0.0
Ruy Lopez	0.0	0.0	1.0	8.0	0.0	0.0	0.0	0.0	33.0	3.0
Sicilian Defense	1.0	1.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	49.0

Along the diagonals are the test cases that the PCA algorithm was able to correctly classify. The columns represent what the actual class label is, and the rows represent the predicted class. For example, looking at the entry in the row labeled “Caro-Kann Defense” and the column labeled “English Opening,” we can see that the PCA algorithm incorrectly labeled an English Opening as a Caro-Kann Defense twice. This confusion matrix allows for an insightful analysis of Chess openings. For instance, the frequent misclassification of the Sicilian Defense opening could suggest that the Sicilian Defense is not as unique as the other openings. In addition, a high misclassification count could suggest similarity between two openings. From the given confusion matrix, we can generate model metrics to gain further information on the performance of PCA in this classification problem.

Metrics for PCA Model



	precision	recall	f1	accuracy
Caro-Kann Defense	0.920	0.610	0.730	0.77
English Opening	0.580	0.880	0.700	0.77
French Defense	0.910	0.780	0.840	0.77
Italian Game	0.790	0.770	0.780	0.77
King's Indian Defense	0.830	0.560	0.670	0.77
Nimzo-Indian Defense	0.620	0.710	0.670	0.77
Queen's Gambit Declined	0.730	0.440	0.550	0.77
Queen's Pawn Game	0.680	0.910	0.780	0.77
Ruy Lopez	0.940	0.730	0.820	0.77
Sicilian Defense	0.650	0.940	0.770	0.77
Avg.	0.765	0.733	0.731	0.77

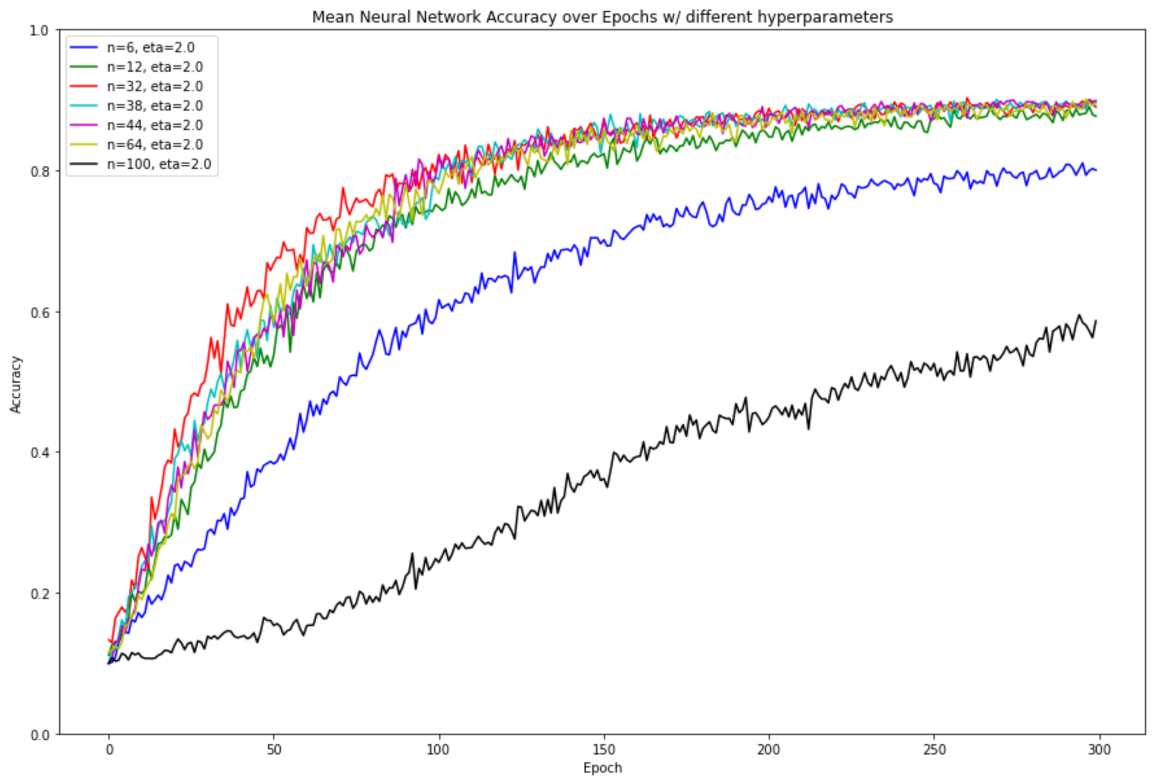
Immediately, it can be seen that the overall model accuracy was 0.77. The accuracy metric means that the PCA model was able to correctly classify 77% of the testing data. This performance is markedly exceptional, as random choosing would yield an expected accuracy of 0.10, since there are 10 classes and assigning a uniformly random class will mean around 10% will be correctly classified. The average precision of the PCA model was 0.77, which is also the average. The average f1 score was 0.731, which means the model performed decently. There is a noticeable variance in the metrics of the model with respect to each class's metric. To reduce this, a proposed solution would be to gather more training data. This idea is supported by the Law of Large numbers, the average metrics would converge to the true metrics of the model as our data size grows. Mathematically,

$$\lim_{n\to\infty} P[\,|\,(\tfrac{1}{n}\sum_{i=1}^{n} X_i) - \mu\,| < \varepsilon\,] = 0.$$

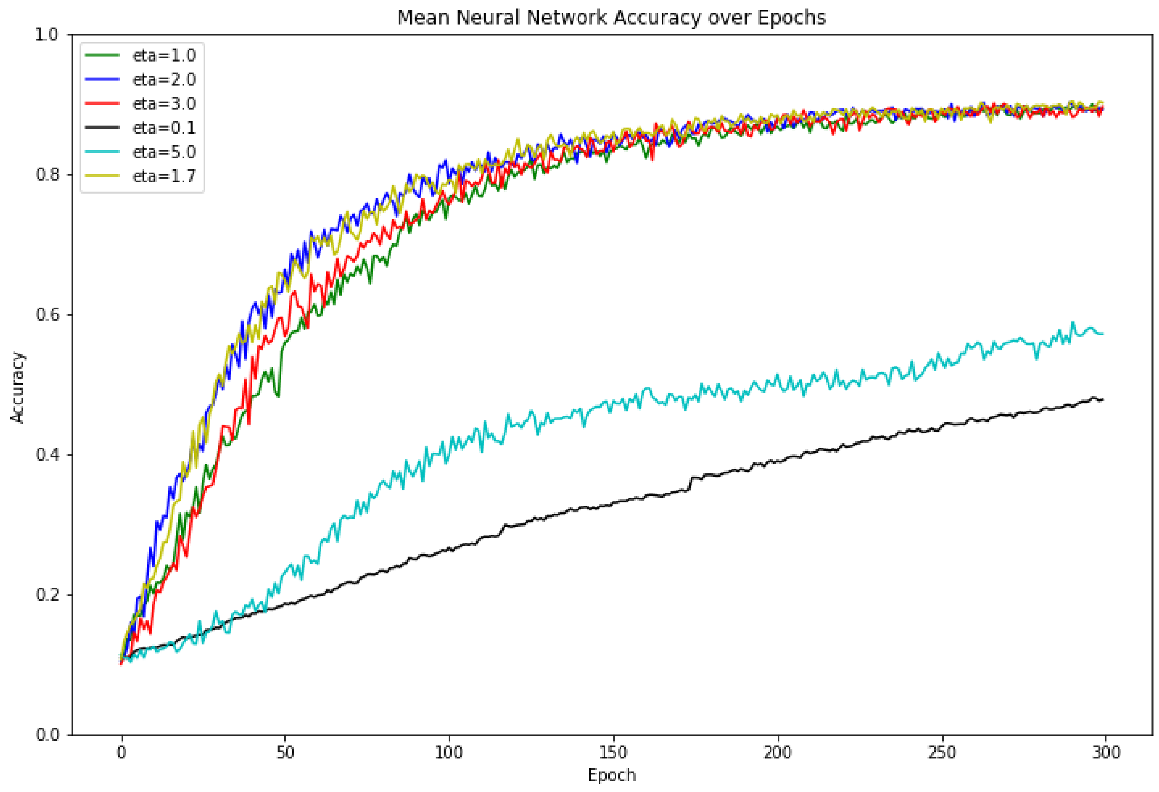
Where X_i is a sample metric and μ is the true metric. In other words, our sample mean for each metric will converge to the true metric as our sample size grows towards infinity. To improve the model, one could employ stratified k-fold cross validation in order to preserve class proportions. A striking example of how stratification could improve the model can be seen in the Sicilian Defense. The PCA model incorrectly classifies many other openings as a sicilian defense, which may be due to the disproportionate amount of training data from the Sicilian Defense opening. Overall, it is safe to say that the PCA classification was generally successful in properly classifying Chess openings.

Neural Network Results

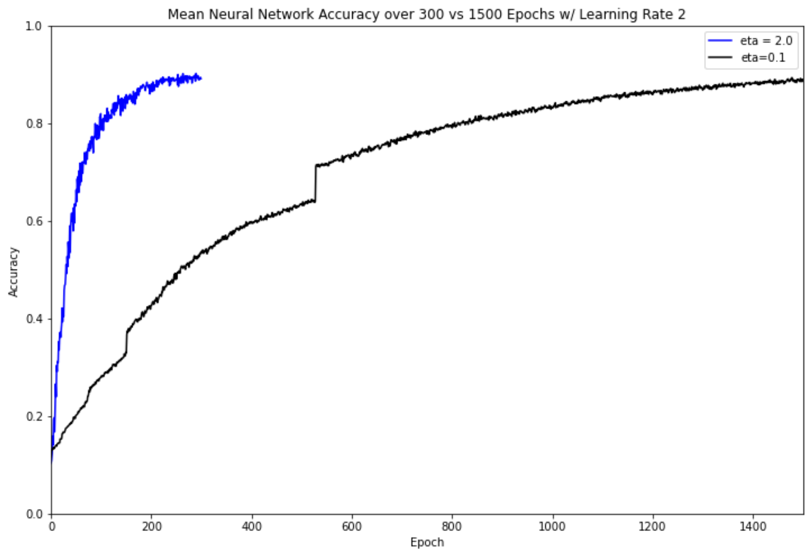
The results of neural network classification are more nuanced and delicate compared to that of PCA. This is due to the extensive hyperparameter optimization that is required to gain optimal accuracy. The hyperparameters the come into play are neurons (in the hidden layer) and learning rate, denoted by η (eta). Tuning these hyperparameters can yield amazing results in the neural network performance. Let us observe how the neural network performance changes when using different neuron counts.



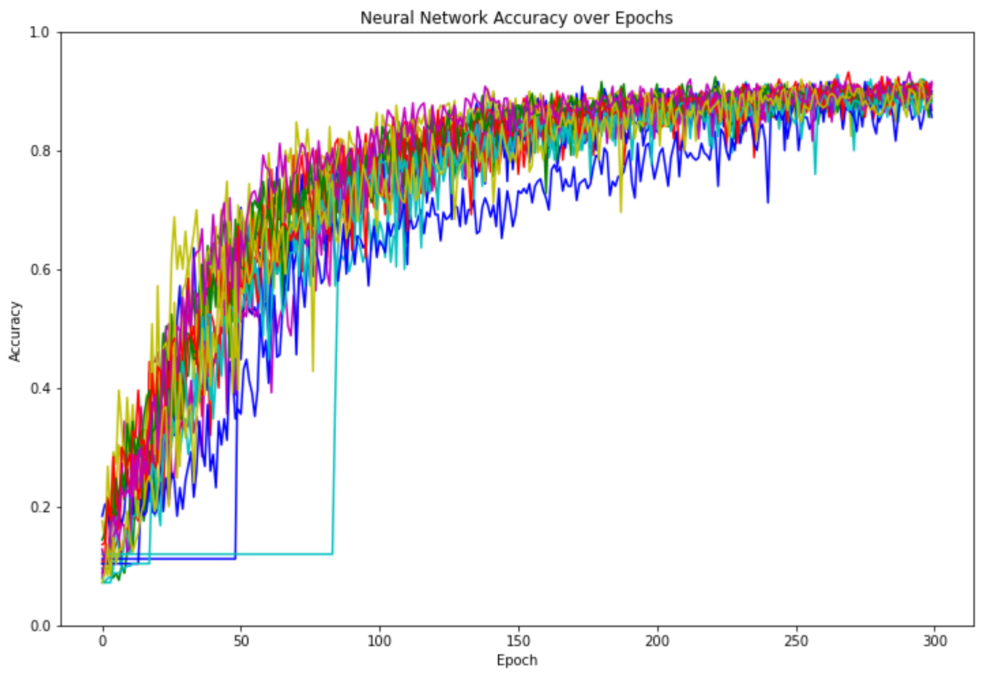
As can be seen, using a small amount of neurons yields suboptimal results, and using too many neurons also yields suboptimal results. From this graph, it can be seen that using a range of 32-44 neurons yields the most optimal results. Luckily, all networks converge and none of them destabilize. We can now look at how changing the learning rate impacts the model's performance, with a fixed neuron count of 44 in the hidden layer.



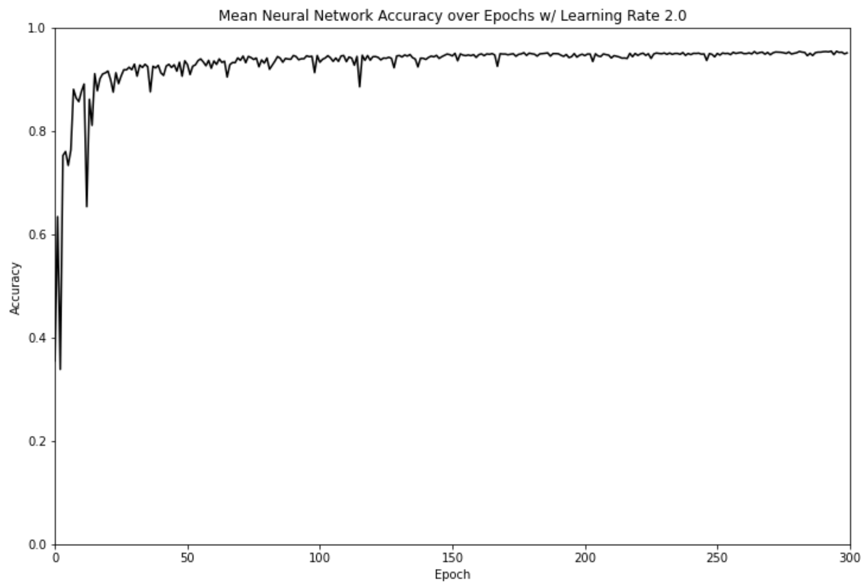
This graph shows a trend that is similar to what was seen in the neurons hyperparameter. Too low of a learning rate will mean the network takes too long to converge, and a very high learning rate means the network will achieve suboptimal performance. It can be seen that optimal, fast convergence can still be achieved with a learning rate around 2.0. The importance of using a proper learning rate for fast convergence can be seen in the following graph.



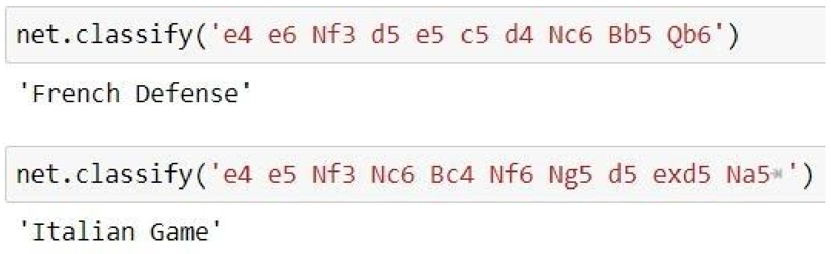
We can observe that similar convergence can be achieved in much less time using a learning rate of 2.0 versus 0.1. Using these optimized hyperparameters, we now analyze performance of the optimal model. Let us observe many observations of the same network over 300 epochs.



Evidently, there is quite a bit of variance among performance, but all networks eventually converge to around 90-95% accuracy. This variance is due to the random initialization of the weights and biases. An accuracy of 95% can be achieved when using 2000 pieces of training data. Let us observe this convergence below.



As can be seen, convergence is achieved much faster with an increase in training data, and also results in a better accuracy of 0.95. This is significant because it gives us confidence in the exceptional performance of the neural network classification. Here is an example of an input and output classification that provides a useful view of how well the network performs.

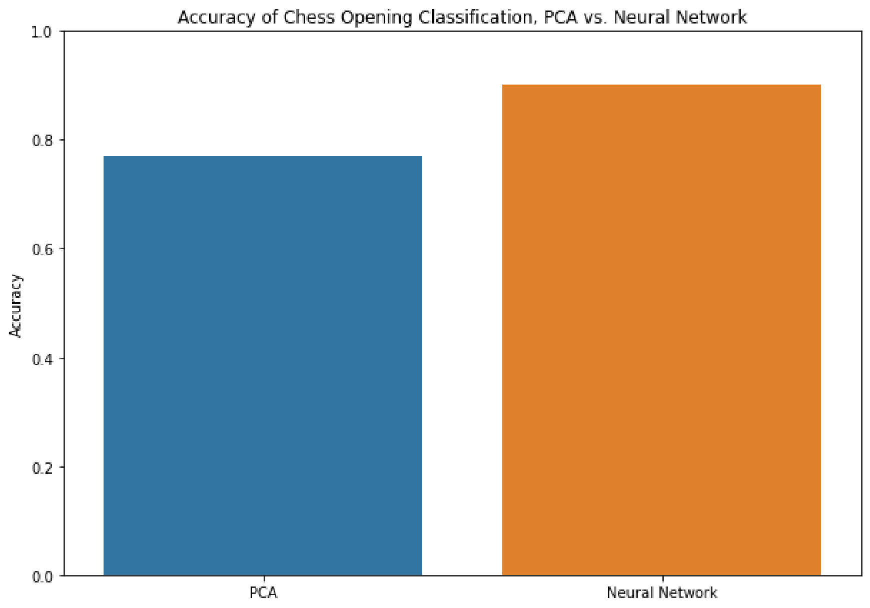


```
net.classify('e4 e6 Nf3 d5 e5 c5 d4 Nc6 Bb5 Qb6')

'French Defense'

net.classify('e4 e5 Nf3 Nc6 Bc4 Nf6 Ng5 d5 exd5 Na5*')

'Italian Game'
```

The move sequences are French Defense and Italian Game, respectively. The neural network can be seen accurately classifying the move sequences which is astonishing, as it can infer everything just from the move sequence thanks to our data vectorization tools previously discussed. There is room for many applications of this network, as it can not only be used to classify Chess openings, but also to classify any vectorized data. In addition, there lies potential for using genetic algorithms to further optimize the model. Overall, the neural network was very successful in accurately classifying the testing data.

Comparison & Conclusion

When considering PCA and neural networks in comparison to each other, the benefits and drawbacks become clear. With PCA one can quickly create an understandable low-dimensional model for classification, but with the tradeoff of lower accuracy.



With neural networks, training the network takes a long time for optimal results (2 minutes for 1 network), but can achieve 95% accuracy over PCA's 77%. This 95% accuracy is such a significant jump that the time cost is a justifiable sacrifice. One can also consider how the Chess board is already low-dimensional (8x8 grid), so PCA may not necessarily be required anyways, and may have a difficult time separating eigenvectors of different openings. Therefore, neural networks do beat out PCA in the context of this specific problem of Chess opening classification.

References

Thrun, Sebastian. "Learning to play the game of chess." *Advances in neural information processing systems*. 1995.

Fogel, David B., et al. "A self-learning evolutionary chess program." *Proceedings of the IEEE* 92.12 (2004): 1947-1954.

Sreerag, S. R. "Evaluation of Strategy by AlphaZero and StockFish on Chess Game."

Lai, Matthew. "Giraffe: Using deep reinforcement learning to play chess." *arXiv preprint arXiv:1509.01549* (2015).

David, Omid E., Nathan S. Netanyahu, and Lior Wolf. "Deepchess: End-to-end deep neural network for automatic learning in chess." *International Conference on Artificial Neural Networks*. Springer, Cham, 2016.