Chess Openings

Solving a nuanced classification problem through PCA and Neural Networks

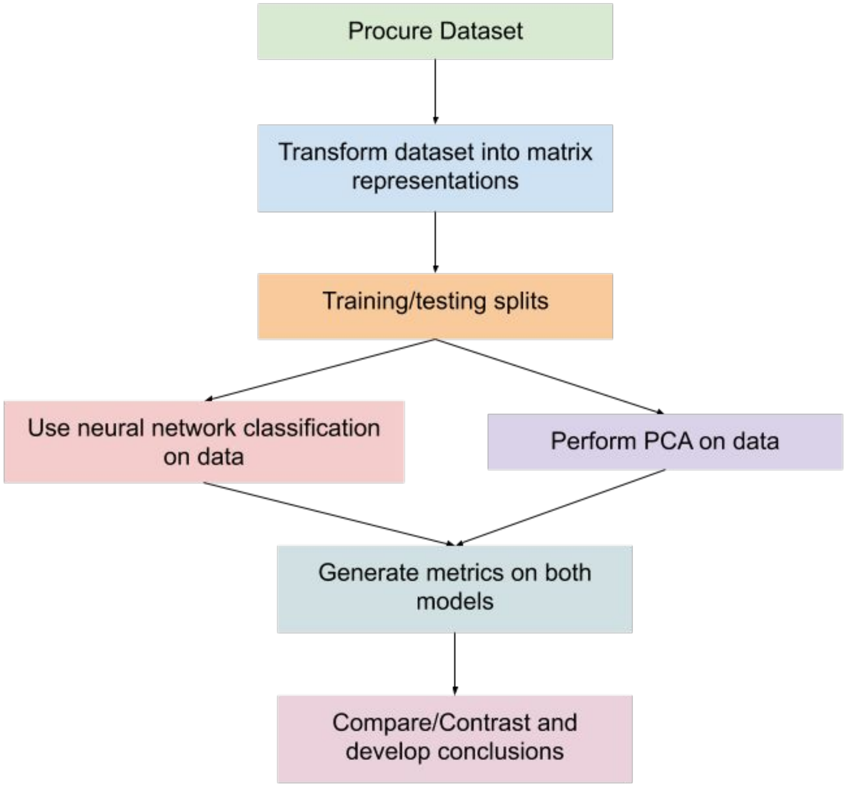
Problem Statement

Given the first ten moves in a Chess game, how can we properly classify them into ten different Chess openings?

Sicilian Defense, French Defense, Ruy Lopez, Italian Game, English Opening, Queen's Gambit Declined, Caro-Kann Defense, King's Indian Defense, Queen's Pawn Game, Nimzo-Indian Defense

Design & Implementation

Process


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

Data Pipelines

Convert Chess moves into a numeric matrix, vectorize into 64 x 1 vector for classification purposes.

a	b	c	d	e	f	g	h
5000	2000	3000	9000	12000	3000	2000	5000
1000	1000	1000	1000	1000	1000	1000	1000
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
-1000	-1000	-1000	-1000	-1000	-1000	-1000	-1000
-5000	-2000	-3000	-9000	-12000	-3000	-2000	-5000

“e4 c5 Bc4 Nf6 Nc3 d6 Nf3 g6 Ng5 e6”

a	b	c	d	e	f	g	h
5000.0	2000.0	2400.0	9000.0	12000.0	3000.0	800.0	5000.0
1000.0	1200.0	1000.0	1000.0	200.0	0.0	1000.0	1000.0
0.0	400.0	0.0	0.0	800.0	1200.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	1000.0	0.0	0.0
0.0	0.0	0.0	1000.0	0.0	0.0	0.0	0.0
0.0	0.0	800.0	0.0	800.0	1200.0	0.0	0.0
1000.0	1000.0	1000.0	0.0	800.0	1000.0	1000.0	1000.0
5000.0	1200.0	3000.0	9000.0	12000.0	2400.0	800.0	5000.0

PCA

Calculate Mean Boards and Eigen-Openings to allow rapid classification of input openings.

```
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
```

Neural Networks

Algorithmically simulate neuron activity similar to that of the human brain.

Use a 3-layer network to train a model that classifies vectorized openings into ten output classes.

Uses **sigmoid** and **softmax** activation functions.

```python
class NeuralNetwork():
    def __init__(self, neurons, f=sigmoid):

        self.biases = [np.random.randn(d, 1) for d in neurons[1:]]
        self.weights = [np.random.randn(d1, d2) for d2,d1 in zip(neurons[:-1], neurons[1:])]

        self.layers = len(neurons)
        self.neurons = neurons

        # random initialization
        self.f = f

    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)
```

```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

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

    def test(self, testing):
          for(x,y) in testing:
              print(x)
              print("-"*30)
              print(y)
              print("*"*30)
              break
        res = [(softmax(self.feed_forward(x)), y) for (x, y) in testing]
        return sum(int(x == y) for (x, y) in res)

    def classify(self, game: str):
        B=np.concatenate(np.array(run(game, boardLetter.copy()))/10000).reshape(64, 1)
        res = softmax(self.feed_forward(B))
        return sorted(ops)[res]
```

Results

PCA Results

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

PCA Results (cont.)

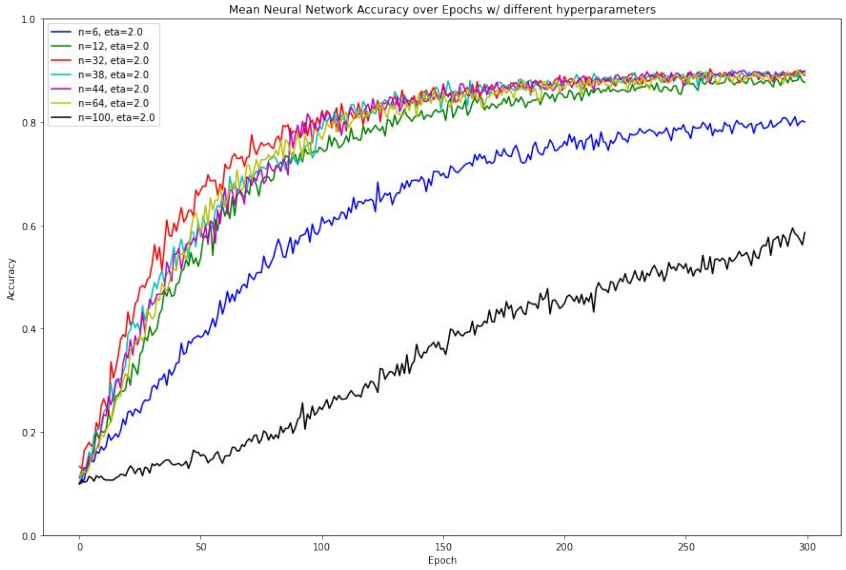
Metrics for PCA Model

	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

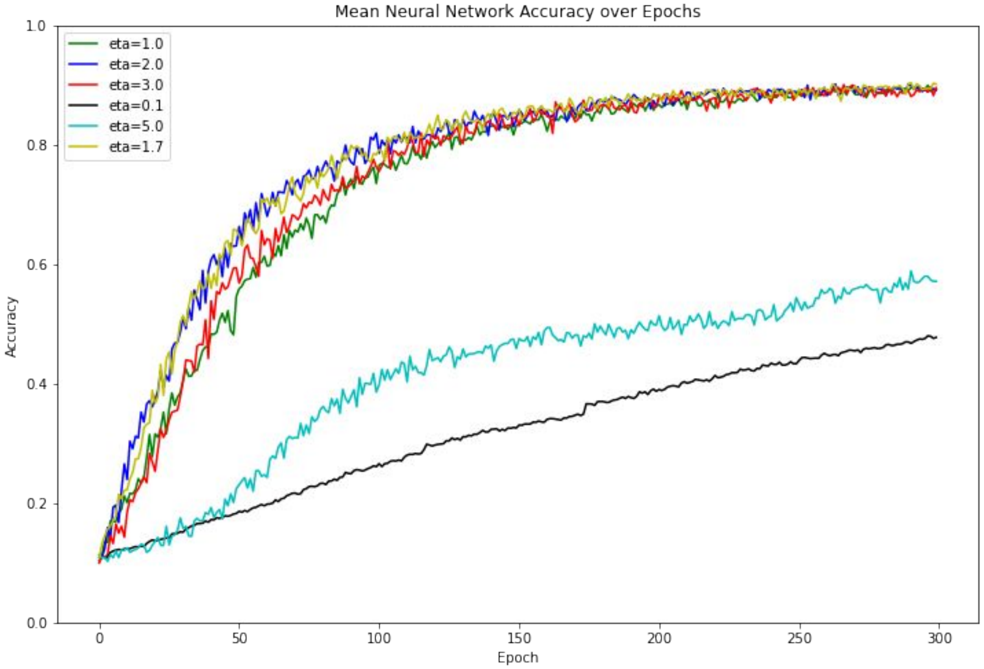
Neural Network Hyperparameter Optimization

Optimize learning rate and neurons in hidden layer to achieve maximum accuracy.

Best combination: **44 neurons** and **2.0 learning rate**.

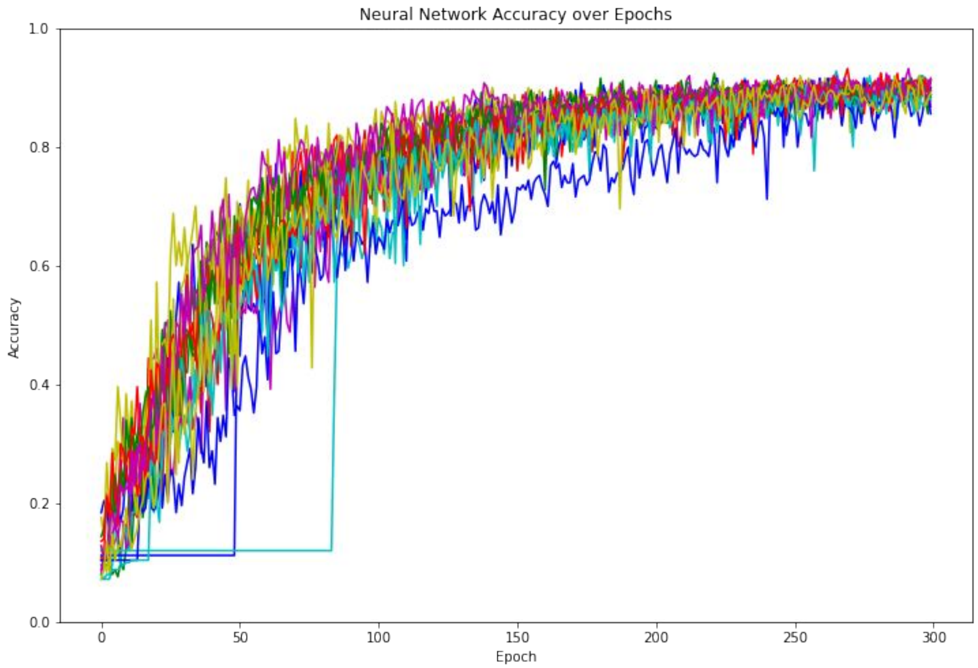


Neural Network Hyperparameter Optimization (cont.)



Neural Network Results (cont.)

Multiple Simulations of the Same Network

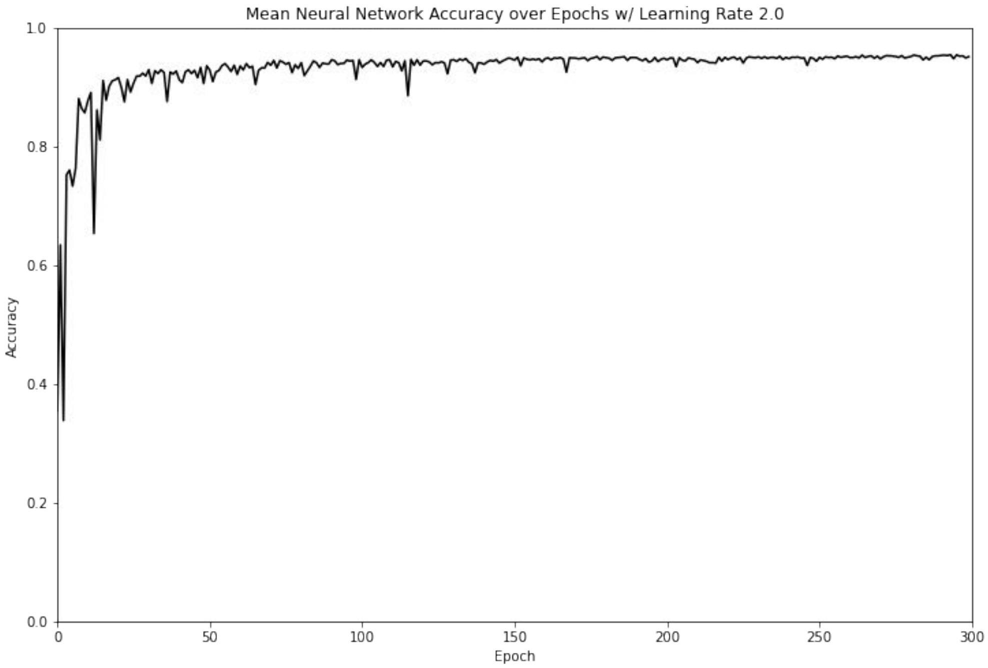


Neural Network Results (cont.)

Training size: **2000** openings

Parameters used: **2.0** learning rate, **44** neurons in hidden layer, **300** epochs.

Final accuracy achieved: **0.95**



PCA vs. Neural Networks Conclusion

Time vs Performance Tradeoff

PCA - **faster, less accurate**

NN - **longer to train, very accurate**