**Project Title**

Chess program with human pattern

**Team Member Name:**

Yang Seongjin

**Abstract**

I will program a chess program with human pattern. The reason why I choose this topic is because of my interest in Artificial Intelligence (AI). I want to be an AI engineer after I graduate, and this chess program is a good topic for a very first step to take. My result should be a chess program with “human pattern” so, it requires dataset of humans not from logical computers. Since I need data of humans' chess games, it is not going to be a reinforcement learning. It will be unsupervised learning for chess algorithm because I want it to find ideal “human pattern” from actual human games.

**Literature Review**

Chess is a board game. Chess, which has a limited number of possibilities compared to another board games, Go, has long been chosen as the subject of artificial intelligence. The chess program Deep Blue is a good example. Deep Blue was created by IBM in 1996 and is known as the first artificial intelligence to beat the chess master. Another board game, Go, was also defeated by artificial intelligence. Since the appearance of AlphaGo made by Google in 2016, the hegemony of unbeatable has fallen and interest in artificial intelligence has soared. I am going to borrow some of AlphaGo's methodology rather than Deep Blue since the AlphaGo is the new one. AlphaGo can be divided into a value network, a policy network, and a Monte Carlo tree search. To compare each role in chess, it is as follows: The value network is to score the odds of winning in the current situation. The policy network is to score each move which the player can take. Monte Carlo Tree Search is an artificial intelligence that looks at the number of different cases and looks for the best results. It is said that Google has taught the policy network through the game datasets and guidance of numerous masters. They improved the policy network by making them play each other and choose the winners. And they improved the value network by checking their calculations.

AlphaGo fought against Lee Se-dol 9 dan by judging the situation by the policy network, and based on it, the value network calculated the current winning rate, finding different ways by tree-searching, and choosing it. My goal is not a chess program that wins all the way, but a chess program with human patterns, so I would like to implement it only as a policy network that reads the situation and a value network that produces a winning rate except for Monte Carlo Tree search.

Since the policy network is to score each move which the player can take, I will use an algorithm which can calculate every moves all chess pieces can go. I do not think it is necessary to use machine learning. I will objectify chess pieces, so chess pieces have Castling or En Passant special rules in its objects.

The value network will use clustering because it will take results of policy network as inputs which is the number of clusters and calculate which cluster have most density. The cluster have most density is the one that most human pattern-like move.

**Methodology**

**Requirement Analysis**

1. Program must be created base on Python.

2. Program must have GUI.

3. Program must have options: Black, White, CPU vs CPU.

4. Program must give downside of the chessboard to users whether they chose Black or White.

5. When the option is CPU vs CPU, Program must give downside of the chessboard to White.

6. Program must show possible moves by shading blocks when users clicked a chess piece they want to move.

7. Program must present proceed and rewind buttons when user chose the CPU vs CPU option.

**Algorithm**

The policy network does simple thing which calculating possible moves of every chess pieces, so I would not use machine learning but simple Brute Force algorithm.

The value network will choose the densest cluster as the next move by clustering dataset. Candidates for algorithms to be used for clustering include K means, density estimation, and EM algorithms. K means algorithm inputs the number of clusters and data, and it outputs as many clusters as they got. This is appropriate when you know the number of clusters. The advantage of density estimation is that you do not have to enter the number of clusters, but it outputs the clusters on its own. So, this algorithm might not give you the number of clusters. Therefore, the chess program will not take this algorithm because the number of clusters, or the number of movements, is already set and I want to get density of each clusters already set. EM algorithm is expectation maximization algorithm. In other words, it is an algorithm that estimates parameter values based on the data entered and outputs a single value, but it would not be chosen because I wants to choose from various options rather than a single value. Because of these reasons, I will take K means Algorithm for value network.

**Pseudocode**

From http://stanford.edu/~cpiech/cs221/handouts/kmeans.html

# Function: K Means

# -------------

# K-Means is an algorithm that takes in a dataset and a constant

# k and returns k centroids (which define clusters of data in the

# dataset which are similar to one another).

def kmeans(dataSet, k):

# Initialize centroids randomly

numFeatures = dataSet.getNumFeatures()

centroids = getRandomCentroids(numFeatures, k)

# Initialize book keeping vars.

iterations = 0

oldCentroids = None

# Run the main k-means algorithm

while not shouldStop(oldCentroids, centroids, iterations):

# Save old centroids for convergence test. Book keeping.

oldCentroids = centroids

iterations += 1

# Assign labels to each datapoint based on centroids

labels = getLabels(dataSet, centroids)

# Assign centroids based on datapoint labels

centroids = getCentroids(dataSet, labels, k)

# We can get the labels too by calling getLabels(dataSet, centroids)

return centroids

# Function: Should Stop

# -------------

# Returns True or False if k-means is done. K-means terminates either

# because it has run a maximum number of iterations OR the centroids

# stop changing.

def shouldStop(oldCentroids, centroids, iterations):

if iterations > MAX\_ITERATIONS: return True

return oldCentroids == centroids

# Function: Get Labels

# -------------

# Returns a label for each piece of data in the dataset.

def getLabels(dataSet, centroids):

# For each element in the dataset, chose the closest centroid.

# Make that centroid the element's label.

# Function: Get Centroids

# -------------

# Returns k random centroids, each of dimension n.

def getCentroids(dataSet, labels, k):

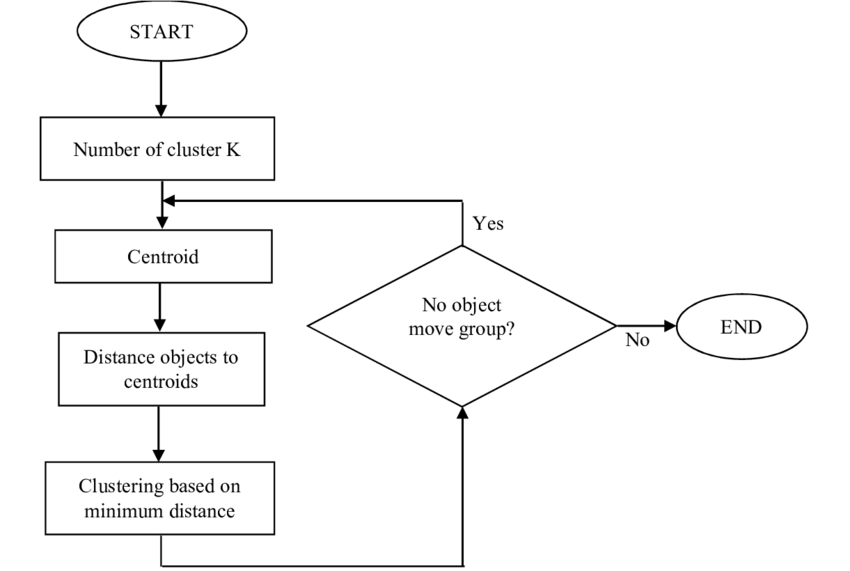
# Each centroid is the geometric mean of the points that

# have that centroid's label. Important: If a centroid is empty (no points have

# that centroid's label) you should randomly re-initialize it.

**Flowchart**

From <https://www.researchgate.net/figure/Flowchart-of-k-means-clustering-algorithm_fig1_318341309>

****

**Project Implementation**

Coding is attached in zip file also.

1. Kmeansclusttryout.ipynb

import chess

import chess.svg

from IPython.display import SVG

board = chess.Board()

SVG(chess.svg.board(board=board,size=400))

import torch

import numpy as np

from kmeans\_pytorch import kmeans

import pandas as pd

chess\_game = pd.read\_csv("dataset/games.csv", index\_col=0)

len(chess\_game)

# data

k = board.legal\_moves.count()

print(k)

data\_size, dims, num\_clusters = 1000, 4, k

x = np.random.randn(data\_size, dims) / 6

x = torch.from\_numpy(x)

# kmeans

cluster\_ids\_x, cluster\_centers = kmeans(

X=x, num\_clusters=num\_clusters, distance='euclidean', device=torch.device('cuda:0')

)

# board.push(g1f3) => how to move

x = chess\_game.iloc[0][11]

print(x)

string\_game = ''.join(str(ord(c)) for c in x)

print(string\_game)

cluster\_ids\_x, cluster\_centers = kmeans(

X=string\_game, num\_clusters=num\_clusters, distance='euclidean', device=torch.device('cuda:0')

)

''.join(str(ord(c)) for c in x)

chess.BaseBoard.piece\_map(board)

board.push\_san("Bf4")

SVG(chess.svg.board(board=board,size=400))

x = chess\_game.iloc[100][11]

print(x)

list(x)

print(x)

type(x)

list\_x = x.split()

type(list\_x)

list\_x

board = chess.Board()

SVG(chess.svg.board(board=board,size=400))

for i in list\_x:

board.push\_san(i)

SVG(chess.svg.board(board=board,size=400))

SVG(chess.svg.board(board=board,size=400))

board = chess.Board()

x = chess.Move("d2","d4")

x.from\_square

board = chess.Board()

x = board.push\_san("d4")

SVG(chess.svg.board(board=board,size=400))

x.from\_square

x.to\_square

2. multivarregtryout

import chess

import chess.svg

from IPython.display import SVG

board = chess.Board()

SVG(chess.svg.board(board=board,size=400)) #show board at jupyter notebook

import pandas as pd

chess\_game = pd.read\_csv("dataset/games.csv", index\_col=0)

dataset = []

#for i in range(1000):

for i in range(len(chess\_game)):

x = chess\_game.iloc[i][11]

x = x.split()

array\_move = []

for j in range(349):

list\_square = []

if len(x) < j+1:

array\_move.append([-1,-1])

else:

a = board.push\_san(x[j])

list\_square.append(a.from\_square)

list\_square.append(a.to\_square)

array\_move.append(list\_square)

dataset.append(array\_move)

board = chess.Board()

#print("%d 번째"%i, "게임 코드", array\_move)

import torch

import torch.nn as nn

import torch.nn.functional as F

import torch.optim as optim

torch.manual\_seed(1)

x1\_train = torch.FloatTensor(dataset[0])

x2\_train = torch.FloatTensor(dataset[1])

x3\_train = torch.FloatTensor(dataset[2])

y\_train = torch.FloatTensor(dataset[3])

w1 = torch.zeros(1, requires\_grad=True)

w2 = torch.zeros(1, requires\_grad=True)

w3 = torch.zeros(1, requires\_grad=True)

b = torch.zeros(1, requires\_grad=True)

optimizer = optim.SGD([w1,w2,w3,b], lr=1e-5)

nb\_epochs = 1000

for epoch in range(nb\_epochs + 1):

hypothesis = x1\_train \* w1 + x2\_train \* w2 + x3\_train \* w3 + b

cost = torch.mean((hypothesis - y\_train) \*\* 2)

optimizer.zero\_grad()

cost.backward()

optimizer.step()

if epoch % 100 == 0:

print('Epoch {:4d}/{} w1: {:.3f} w2: {:.3f} w3: {:.3f} b: {:.3f} Cost: {:.6f}'.format(

epoch, nb\_epochs, w1.item(), w2.item(), w3.item(), b.item(), cost.item()

))

import numpy as np

x\_raw = np.array(dataset[0:1000])

#x\_raw = x\_raw.transpose([1,2,0])

y\_raw = [np.array(dataset[11554])]

x\_train = torch.FloatTensor(x\_raw)

y\_train = torch.FloatTensor(y\_raw)# which has full m349 moves, 13859 as well

print(x\_train.shape)

print(y\_train.shape)

W = torch.zeros((1000, 1,1), requires\_grad=True)

b = torch.zeros(1, requires\_grad=True)

optimizer = optim.SGD([W, b], lr=1e-5)

nb\_epochs = 20

for epoch in range(nb\_epochs + 1):

hypothesis = x\_train.matmul(W) + b

cost = torch.mean((hypothesis - y\_train) \*\* 2)

optimizer.zero\_grad()

cost.backward()

optimizer.step()

print('Epoch {:4d}/{} hypothesis: {} Cost: {:.6f}'.format(

epoch, nb\_epochs, hypothesis.squeeze().detach(), cost.item()

))

3. LSTMtryout.ipynb

import chess

import chess.svg

from IPython.display import SVG

board = chess.Board()

SVG(chess.svg.board(board=board,size=400)) #show board at jupyter notebook

import pandas as pd

chess\_game = pd.read\_csv("dataset/games.csv", index\_col=0)

dataset = []

#for i in range(1000):

for i in range(len(chess\_game)):

x = chess\_game.iloc[i][11]

x = x.split()

array\_move = []

for j in x:

list\_square = []

a = board.push\_san(j)

list\_square.append(a.from\_square)

list\_square.append(a.to\_square)

array\_move.append(list\_square)

dataset.append(array\_move)

board = chess.Board()

#print("%d 번째"%i, "게임 코드", array\_move)

import torch

import torch.nn as nn

import numpy as np

import matplotlib.pyplot as plt

%matplotlib inline

chess\_game.head()

chess\_game.shape #(20058, 15)

dpp\_ds = [] #datapreprocessing dataset

for i in dataset:

a = []

for j in i:

for k in j:

a.append(k)

dpp\_ds.append(a)

#dpp\_ds = np.array(dpp\_ds, dtype = object)

#all\_data = np.array([])

all\_data = []

print(np.array(dpp\_ds[0]))

for i in dpp\_ds:

a = np.array(i)

all\_data = np.append(all\_data, a)

all\_data = all\_data[:1000]

test\_data\_size = 500

train\_data = all\_data[:-test\_data\_size]

test\_data = all\_data[-test\_data\_size:]

print(len(train\_data))

print(len(test\_data))

from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler(feature\_range=(-1,1))

train\_data\_normalized = scaler.fit\_transform(train\_data .reshape(-1,1))

train\_data\_normalized = torch.FloatTensor(train\_data\_normalized).view(-1)

print(train\_data\_normalized[:5])

print(train\_data\_normalized[-5:])

print(len(train\_data\_normalized))

train\_window = 20

def create\_inout\_sequences(input\_data, tw):

inout\_seq = []

L = len(input\_data)

for i in range(L-tw):

train\_seq = input\_data[i:i+tw]

train\_label = input\_data[i+tw:i+tw+1]

inout\_seq.append((train\_seq ,train\_label))

return inout\_seq

train\_inout\_seq = create\_inout\_sequences(train\_data\_normalized, train\_window)

train\_inout\_seq[:5]

class LSTM(nn.Module):

def \_\_init\_\_(self, input\_size=1, hidden\_layer\_size=100, output\_size=1):

super().\_\_init\_\_()

self.hidden\_layer\_size = hidden\_layer\_size

self.lstm = nn.LSTM(input\_size, hidden\_layer\_size)

self.linear = nn.Linear(hidden\_layer\_size, output\_size)

self.hidden\_cell = (torch.zeros(1,1,self.hidden\_layer\_size),

torch.zeros(1,1,self.hidden\_layer\_size))

def forward(self, input\_seq):

lstm\_out, self.hidden\_cell = self.lstm(input\_seq.view(len(input\_seq) ,1, -1), self.hidden\_cell)

predictions = self.linear(lstm\_out.view(len(input\_seq), -1))

return predictions[-1]

model = LSTM()

loss\_function = nn.MSELoss()

optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

USE\_CUDA = torch.cuda.is\_available()

print(USE\_CUDA)

device = torch.device('cuda:0' if USE\_CUDA else 'cpu')

print('학습을 진행하는 기기:',device)

print('cuda index:', torch.cuda.current\_device())

print('gpu 개수:', torch.cuda.device\_count())

print('graphic name:', torch.cuda.get\_device\_name())

cuda = torch.device('cuda')

print(cuda)

epochs = 150

for i in range(epochs):

for seq, labels in train\_inout\_seq:

optimizer.zero\_grad()

model.hidden\_cell = (torch.zeros(1, 1, model.hidden\_layer\_size),

torch.zeros(1, 1, model.hidden\_layer\_size))

y\_pred = model(seq)

single\_loss = loss\_function(y\_pred, labels).to(device)

single\_loss.backward()

optimizer.step()

if i%15 == 1:

print(f'epoch: {i:3} loss: {single\_loss.item():10.8f}')

print(f'epoch: {i:3} loss: {single\_loss.item():10.10f}')

fut\_pred = 100

test\_inputs = train\_data\_normalized[-train\_window:].tolist()

print(test\_inputs)

model.eval()

for i in range(fut\_pred):

seq = torch.FloatTensor(test\_inputs[-train\_window:])

with torch.no\_grad():

model.hidden = (torch.zeros(1, 1, model.hidden\_layer\_size),

torch.zeros(1, 1, model.hidden\_layer\_size))

test\_inputs.append(model(seq).item())

test\_inputs[fut\_pred:]

actual\_predictions = scaler.inverse\_transform(np.array(test\_inputs[train\_window:] ).reshape(-1, 1))

print(actual\_predictions)

move\_pred = []

for i in actual\_predictions:

a = round(float(i))

move\_pred.append(a)

print(round(float(i)))

move\_pred

board = chess.Board()

SVG(chess.svg.board(board=board,size=400))

chess.Move(8, 24)

SVG(chess.svg.board(board=board,size=400))

**Project Results**

First, I was followed K-means clustering algorithm which is the first option for me. It is unsupervised learning, and I thought it is easy because I do not have to give labels for each data. But after I implemented, I noticed that each cluster does not related with each other. Because each cluster are independent trial, so it cannot be the one game.

After that, I tried supervised learning method. It needs feature and label to train them, and my dataset is all labeled as “human pattern” since it is done by human. So, I thought classification is not the one that I should implement, so I chose regression, for ideal goal: “human pattern” which will be regressed to. My implementation was multivariable regression.

After I preprocessed data, a move has string type. But training model takes only integers as inputs. One string was changed into two integers which are “to\_square” and “from\_square”. After that, it was created some errors because I need to bind them as one move, but it divided in two, and it creates 3-dimensional metrics. And I could not fit them into the training model.

After that I moved to Long Short Term Memory, which is the kind of deep learning. I flattened the integer data which I transformed before, and it printed out the results, but it was not a valid move, so it was failed.

**Demonstration**

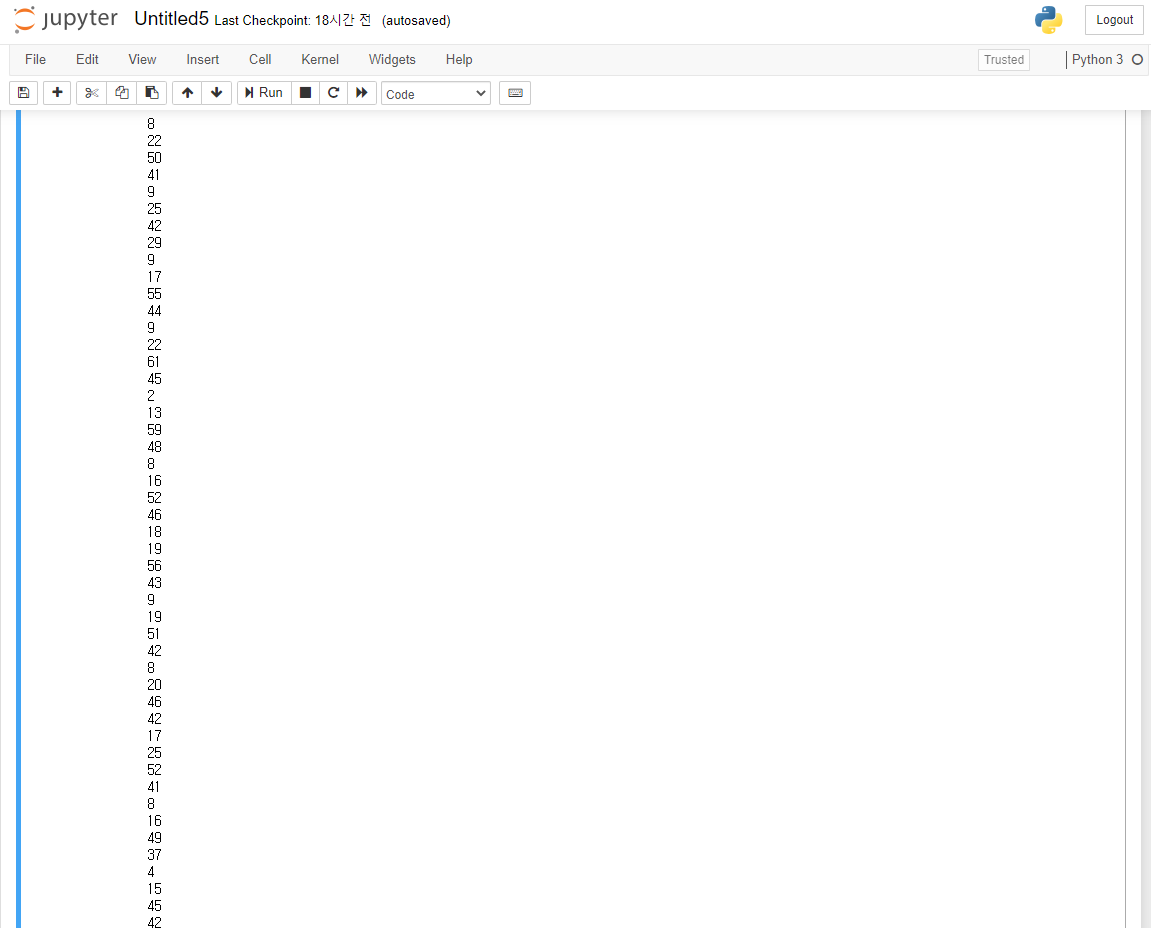
****

Figure 1. Series of From\_square and To\_square by using LSTM. It was not a valid move.