# TensorFlow Machine Learning Cookbook - Second Edition

# by Nick McClure

# Published by Packt Publishing, 2018

# Learning to play Tic Tac Toe

To show how adaptable neural networks can be, we will now attempt to use a neural network in order to learn the optimal moves for Tic Tac Toe. We will approach this knowing that Tic Tac Toe is a deterministic game and that the optimal moves are already known.

# Getting ready

To train our model, we will use a list of board positions followed by the best optimal response for a number of different boards. We can reduce the amount of boards to train on by considering only board positions that are different with regard to symmetry. The non-identity transformations of a Tic Tac Toe board are a rotation (in either direction) of 90 degrees, 180 degrees, and 270 degrees, a horizontal reflection, and a vertical reflection. Given this idea, we will use a shortlist of boards with the optimal move, apply two random transformations, and then feed that into out neural network for learning.

*Since Tic Tac Toe is a deterministic game, it is worth noting that whoever goes first should either win or draw. We will hope for a model that can respond to our moves optimally and ultimately result in a draw.*

If we annotate Xs by 1, Os by -1, and empty spaces by 0, then the following diagram illustrates how we can consider a board position and an optimal move as a row of data:

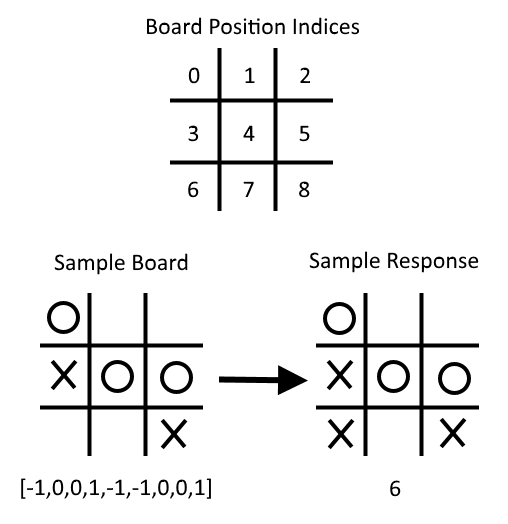


Figure 9: Here, we illustrate how to consider a board and an optimal move as a row of data. Note that X = 1, O = -1, and empty spaces are 0, and we start indexing at 0

In addition to the model loss, to check how our model is performing we will do two things. The first check we will perform is to remove a position and an optimal move row from our training set. This will allow us to see if the neural network model can generalize a move it hasn't seen before. The second method we will take to evaluate our model is to actually play a game against it at the end.

The list of possible boards and optimal moves can be found on the GitHub directory for this recipe here: <https://github.com/nfmcclure/tensorflow_cookbook/tree/master/06_Neural_Networks/08_Learning_Tic_Tac_Toe> and in the Packt repository: <https://github.com/PacktPublishing/TensorFlow-Machine-Learning-Cookbook-Second-Edition>.

# How to do it...

We proceed with the recipe as follows:

1. We need to start by loading the necessary libraries for this script, as follows:

import tensorflow as tf

import matplotlib.pyplot as plt

import csv

import random

import numpy as np

import random

1. Next, we declare the following batch size for training our model:

batch\_size = 50

1. To make visualizing the boards a bit easier, we will create a function that outputs Tic Tac Toe boards with Xs and Os. This is done with the following code:

def print\_board(board):  
 symbols = ['O', ' ', 'X']  
 board\_plus1 = [int(x) + 1 for x in board]  
 board\_line1 = ' {} | {} | {}'.format(symbols[board\_plus1[0]],  
 symbols[board\_plus1[1]],  
 symbols[board\_plus1[2]])  
 board\_line2 = ' {} | {} | {}'.format(symbols[board\_plus1[3]],  
 symbols[board\_plus1[4]],  
 symbols[board\_plus1[5]])  
 board\_line3 = ' {} | {} | {}'.format(symbols[board\_plus1[6]],  
 symbols[board\_plus1[7]],  
 symbols[board\_plus1[8]])  
 print(board\_line1)  
 print('\_\_\_\_\_\_\_\_\_\_\_')  
 print(board\_line2)  
 print('\_\_\_\_\_\_\_\_\_\_\_')  
 print(board\_line3)

1. Now we have to create a function that will return a new board and an optimal response position under a transformation. This is done with the following code:

def get\_symmetry(board, response, transformation):

'''

:param board: list of integers 9 long:

opposing mark = -1

friendly mark = 1

empty space = 0

:param transformation: one of five transformations on a board:

rotate180, rotate90, rotate270, flip\_v, flip\_h

:return: tuple: (new\_board, new\_response)

'''

if transformation == 'rotate180':

new\_response = 8 - response

return board[::-1], new\_response

elif transformation == 'rotate90':

new\_response = [6, 3, 0, 7, 4, 1, 8, 5, 2].index(response)

tuple\_board = list(zip(\*[board[6:9], board[3:6], board[0:3]]))

return [value for item in tuple\_board for value in item], new\_response

elif transformation == 'rotate270':

new\_response = [2, 5, 8, 1, 4, 7, 0, 3, 6].index(response)

tuple\_board = list(zip(\*[board[0:3], board[3:6], board[6:9]]))[::-1]

return [value for item in tuple\_board for value in item], new\_response

elif transformation == 'flip\_v':

new\_response = [6, 7, 8, 3, 4, 5, 0, 1, 2].index(response)

return board[6:9] + board[3:6] + board[0:3], new\_response

elif transformation == 'flip\_h':

# flip\_h = rotate180, then flip\_v

new\_response = [2, 1, 0, 5, 4, 3, 8, 7, 6].index(response)

new\_board = board[::-1]

return new\_board[6:9] + new\_board[3:6] + new\_board[0:3], new\_response

else:

raise ValueError('Method not implmented.')

1. The list of boards and their optimal responses is in a .csv file in the directory, available at the github repository <https://github.com/nfmcclure/tensorflow_cookbook> or the Packt repository [https://github.com/PacktPublishing/TensorFlow-Machine-Learning-Cookbook-Second-Edition. We will create a function that will load the file with the boards and responses and will store it as a list of tuples, as follows:](https://github.com/PacktPublishing/TensorFlow-Machine-Learning-Cookbook-Second-Edition)

def get\_moves\_from\_csv(csv\_file):

'''

:param csv\_file: csv file location containing the boards w/ responses

:return: moves: list of moves with index of best response

'''

moves = []

with open(csv\_file, 'rt') as csvfile:

reader = csv.reader(csvfile, delimiter=',')

for row in reader:

moves.append(([int(x) for x in row[0:9]],int(row[9])))

return moves

1. Now we need to tie everything together to create a function that will return a randomly-transformed board and response. This is done with the following code:

def get\_rand\_move(moves, rand\_transforms=2):

# This function performs random transformations on a board.

(board, response) = random.choice(moves)

possible\_transforms = ['rotate90', 'rotate180', 'rotate270', 'flip\_v', 'flip\_h']

for i in range(rand\_transforms):

random\_transform = random.choice(possible\_transforms)

(board, response) = get\_symmetry(board, response, random\_transform)

return board, response

1. Next, we need to initialize our graph session, load our data, and create a training set as follows:

sess = tf.Session()

moves = get\_moves\_from\_csv('base\_tic\_tac\_toe\_moves.csv')

# Create a train set:

train\_length = 500

train\_set = []

for t in range(train\_length):

train\_set.append(get\_rand\_move(moves))

1. Remember that we want to remove one board and an optimal response from our training set to see if the model can generalize making the best move. The best move for the following board will be to play at index number 6:

test\_board = [-1, 0, 0, 1, -1, -1, 0, 0, 1]

train\_set = [x for x in train\_set if x[0] != test\_board]

1. We can now create functions to create our model variables and our model operations. Note that we do not include the softmax() activation function in the following model because it is included in the loss function:

def init\_weights(shape):

return tf.Variable(tf.random\_normal(shape))

def model(X, A1, A2, bias1, bias2):

layer1 = tf.nn.sigmoid(tf.add(tf.matmul(X, A1), bias1))

layer2 = tf.add(tf.matmul(layer1, A2), bias2)

return layer2

1. Now we need to declare our placeholders, variables, and model as follows:

X = tf.placeholder(dtype=tf.float32, shape=[None, 9])

Y = tf.placeholder(dtype=tf.int32, shape=[None])

A1 = init\_weights([9, 81])

bias1 = init\_weights([81])

A2 = init\_weights([81, 9])

bias2 = init\_weights([9])

model\_output = model(X, A1, A2, bias1, bias2)

1. Next, we need to declare our loss function, which will be the average softmax of the final output logits (unstandardized output). Then we will declare our training step and optimizer. We also need to create a prediction operation if we want to be able to play against our model in the future, as follows:

loss = tf.reduce\_mean(tf.nn.sparse\_softmax\_cross\_entropy\_with\_logits(logits=model\_output, labels=Y))

train\_step = tf.train.GradientDescentOptimizer(0.025).minimize(loss)

prediction = tf.argmax(model\_output, 1)

1. We can now initialize our variables and loop through the training of our neural network with the following code:

# Initialize variables

init = tf.global\_variables\_initializer()

sess.run(init)

loss\_vec = []

for i in range(10000):

# Select random indices for batch

rand\_indices = np.random.choice(range(len(train\_set)), batch\_size, replace=False)

# Get batch

batch\_data = [train\_set[i] for i in rand\_indices]

x\_input = [x[0] for x in batch\_data]

y\_target = np.array([y[1] for y in batch\_data])

# Run training step

sess.run(train\_step, feed\_dict={X: x\_input, Y: y\_target})

# Get training loss

temp\_loss = sess.run(loss, feed\_dict={X: x\_input, Y: y\_target})

loss\_vec.append(temp\_loss)

if i%500==0:

print('iteration ' + str(i) + ' Loss: ' + str(temp\_loss))

1. The following is the code needed to plot the loss over the model training:

plt.plot(loss\_vec, 'k-', label='Loss')

plt.title('Loss (MSE) per Generation')

plt.xlabel('Generation')

plt.ylabel('Loss')

plt.show()

We should get the following plot for the loss per generation:

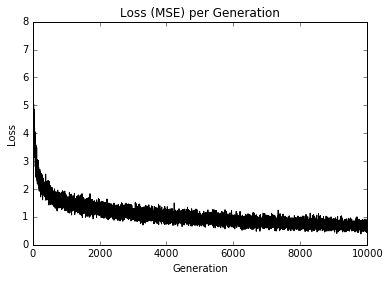


Figure 10: A Tic-Tac-Toe train set loss over 10,000 iterations

In the preceding diagram, we have plotted the loss over the training steps.

1. To test the model, we need to see how it performs on the test board that we removed from the training set. We are hoping that the model can generalize and predict the optimal index for moving, which will be index number 6. Most of the time the model will succeed, shown as follows:

test\_boards = [test\_board]

feed\_dict = {X: test\_boards}

logits = sess.run(model\_output, feed\_dict=feed\_dict)

predictions = sess.run(prediction, feed\_dict=feed\_dict)

print(predictions)

1. The preceding step should result in the following output:

**[6]**

1. In order to evaluate our model, we need to play against our trained model. To do this, we have to create a function that will check for a win. This way, our program will know when to stop asking for more moves. This is done with the following code:

def check(board):

wins = [[0,1,2], [3,4,5], [6,7,8], [0,3,6], [1,4,7], [2,5,8], [0,4,8], [2,4,6]]

for i in range(len(wins)):

if board[wins[i][0]]==board[wins[i][1]]==board[wins[i][2]]==1.:

return 1

elif board[wins[i][0]]==board[wins[i][1]]==board[wins[i][2]]==-1.:

return 1

return 0

1. Now we can loop through and play a game with our model. We start with a blank board (all zeros), we ask the user to input an index (0-8) of where to play, and we then feed that into the model for a prediction. For the model's move, we take the largest available prediction that is also an open space. From this game, we can see that our model is not perfect, as follows:

game\_tracker = [0., 0., 0., 0., 0., 0., 0., 0., 0.]

win\_logical = False

num\_moves = 0

while not win\_logical:

player\_index = input('Input index of your move (0-8): ')

num\_moves += 1

# Add player move to game

game\_tracker[int(player\_index)] = 1.

# Get model's move by first getting all the logits for each index

[potential\_moves] = sess.run(model\_output, feed\_dict={X: [game\_tracker]})

# Now find allowed moves (where game tracker values = 0.0)

allowed\_moves = [ix for ix,x in enumerate(game\_tracker) if x==0.0]

# Find best move by taking argmax of logits if they are in allowed moves

model\_move = np.argmax([x if ix in allowed\_moves else -999.0 for ix,x in enumerate(potential\_moves)])

# Add model move to game

game\_tracker[int(model\_move)] = -1.

print('Model has moved')

print\_board(game\_tracker)

# Now check for win or too many moves

if check(game\_tracker)==1 or num\_moves>=5:

print('Game Over!')

win\_logical = True

1. The preceding step should result in the following interactive output:

**Input index of your move (0-8): 4  
Model has moved  
 O | |  
\_\_\_\_\_\_\_\_\_\_\_   
 | X |   
\_\_\_\_\_\_\_\_\_\_\_   
 | |   
Input index of your move (0-8): 6   
Model has moved   
O | |   
\_\_\_\_\_\_\_\_\_\_\_   
 | X |   
\_\_\_\_\_\_\_\_\_\_\_   
 X | | O   
Input index of your move (0-8): 2   
Model has moved   
O | | X   
\_\_\_\_\_\_\_\_\_\_\_   
O | X |   
\_\_\_\_\_\_\_\_\_\_\_   
X | | O   
Game Over!**

# How it works...

In this section, we trained a neural network to play Tic Tac Toe by feeding in board positions and a nine-dimensional vector, and predicted the optimal response. We only had to feed in a few possible Tic Tac Toe boards and apply random transformations to each board to increase the training set size.

To test our algorithm, we removed all instances of one specific board and saw whether our model could generalize to predict the optimal response. Finally, we played a sample game against our model. While it is not perfect yet, there are still different architectures and training procedures that can be applied to improve it.

# [Artificial Intelligence with Python](https://www.safaribooksonline.com/library/view/artificial-intelligence-with/9781786464392/)

# by Prateek Joshi

# *Published by [Packt Publishing](https://www.safaribooksonline.com/library/publisher/packt-publishing/" \t "_self), 2017*

# Building a bot to play Tic-Tac-Toe

Tic-Tac-Toe (Noughts and Crosses) is probably one of the most famous games. Let's see how to build a game where the computer can play against the user. This is a minor variant of the Tic-Tac-Toe recipe given in the easyAI library.

Create a new Python file and import the following packages:

from easyAI import TwoPlayersGame, AI\_Player, Negamax

from easyAI.Player import Human\_Player

Define a class that contains all the methods to play the game. Start by defining the players and who starts the game:

class GameController(TwoPlayersGame):

def \_\_init\_\_(self, players):

# Define the players

self.players = players

# Define who starts the game

self.nplayer = 1

We will be using a 3x3 board numbered from one to nine row-wise:

# Define the board

self.board = [0] \* 9

Define a method to compute all the possible moves:

# Define possible moves

def possible\_moves(self):

return [a + 1 for a, b in enumerate(self.board) if b == 0]

Define a method to update the board after making a move:

# Make a move

def make\_move(self, move):

self.board[int(move) - 1] = self.nplayer

Define a method to see if somebody has lost the game. We will be checking if somebody has three in a row:

# Does the opponent have three in a line?

def loss\_condition(self):

possible\_combinations = [[1,2,3], [4,5,6], [7,8,9],

[1,4,7], [2,5,8], [3,6,9], [1,5,9], [3,5,7]]

return any([all([(self.board[i-1] == self.nopponent)

for i in combination]) for combination in possible\_combinations])

Check if the game is over using the loss\_condition method:

# Check if the game is over

def is\_over(self):

return (self.possible\_moves() == []) or self.loss\_condition()

Define a method to show the current progress:

# Show current position

def show(self):

print('\n'+'\n'.join([' '.join([['. ', 'O', 'X'][self.board[3\*j + i]]

for i in range(3)]) for j in range(3)]))

Compute the score using the loss\_condition method:

# Compute the score

def scoring(self):

return -100 if self.loss\_condition() else 0

Define the main function and start by defining the algorithm. We will be using Negamax as the AI algorithm for this game. We can specify the number of steps in advance that the algorithm should think. In this case, let's choose 7:

if \_\_name\_\_ == "\_\_main\_\_":

# Define the algorithm

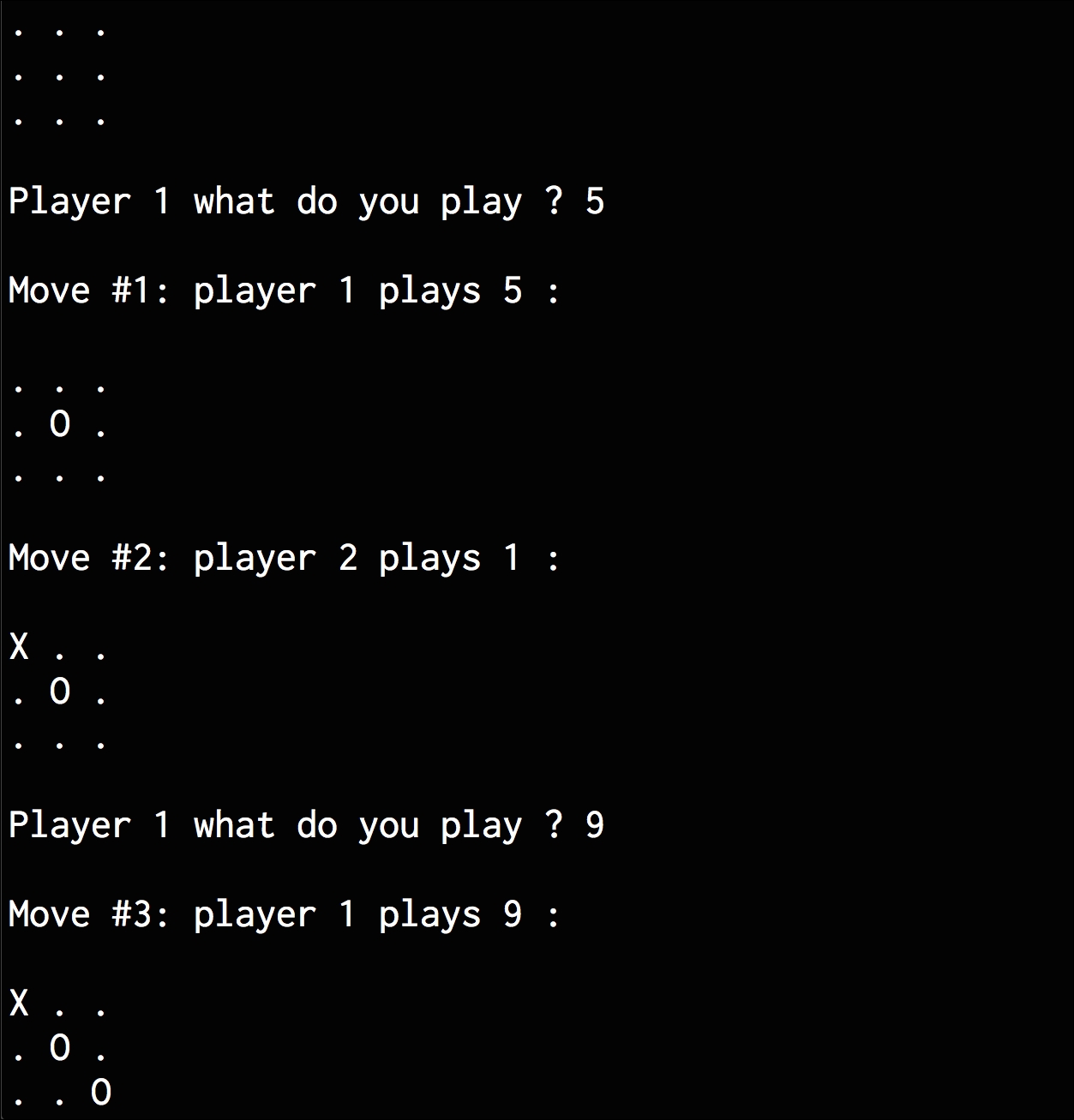
algorithm = Negamax(7)

Start the game:

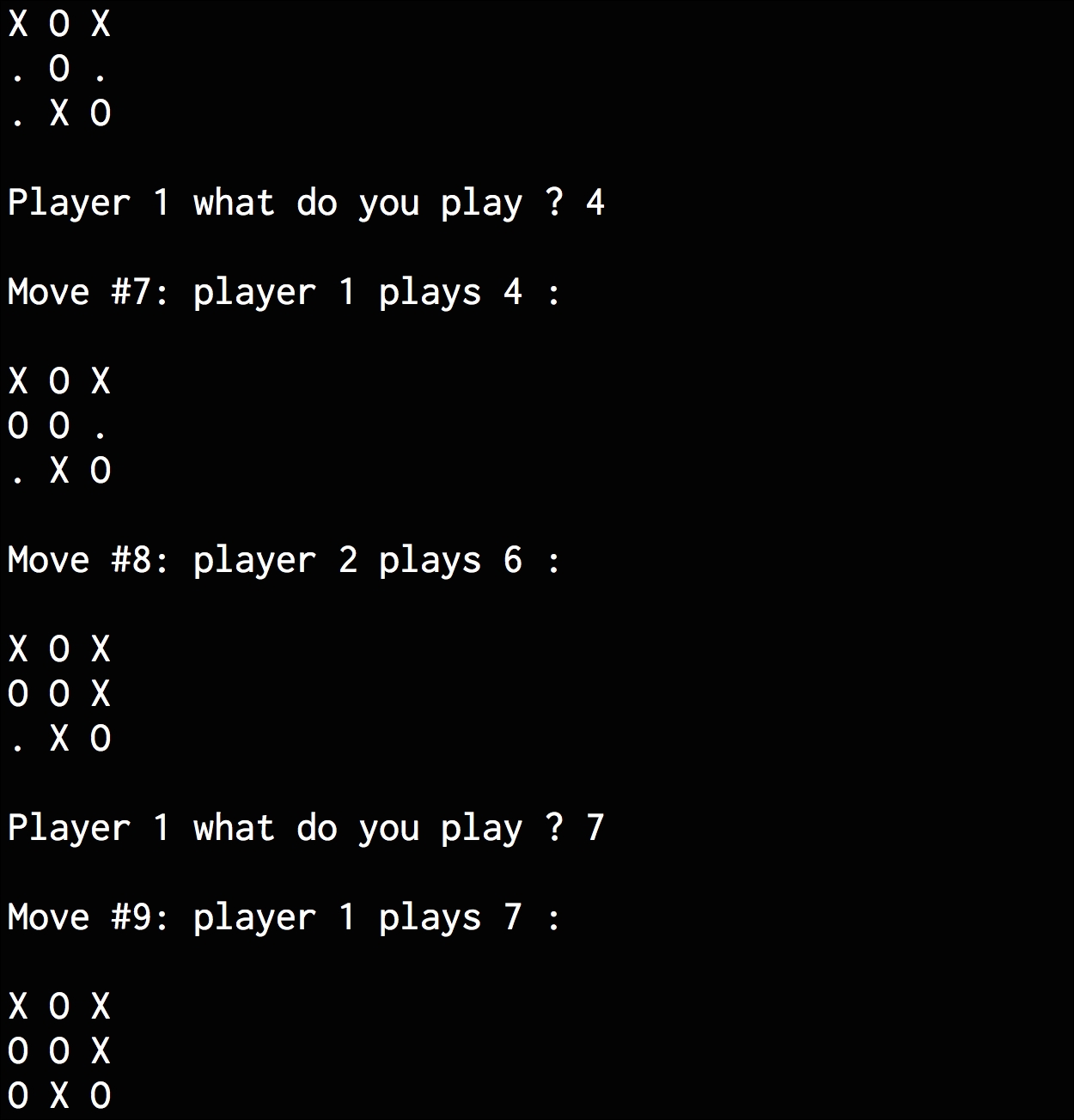
# Start the game

GameController([Human\_Player(), AI\_Player(algorithm)]).play()

The full code is given in the file tic\_tac\_toe.py. It's an interactive game where you play against the computer. If you run the code, you will get the following output on your Terminal at the beginning:



If you scroll down, you will see the following printed on your Terminal once it finishes executing the code:



As we can see, the game ends in a draw.

# [Python Deep Learning](https://www.safaribooksonline.com/library/view/python-deep-learning/9781786464453/)

# by Peter Roelants; Valentino Zocca; Daniel Slater; Gianmario Spacagna

# *Published by [Packt Publishing](https://www.safaribooksonline.com/library/publisher/packt-publishing/), 2017*

# Implementing a Python Tic-Tac-Toe game

Let's build a basic implementation of Tic-Tac-Toe so we can see what an implementation of the min-max algorithm looks like. If you do not feel like copying all of this, you can find the full code in the GitHub repository <https://github.com/DanielSlater/PythonDeepLearningSamples> in the tic\_tac\_toe.py file.

In the game board, we will be represented by a 3 x 3 tuple of integers. Tuples are used instead of lists so that later on, we can get equality between matching board states. In this case, **0** represents a square that has not been played in. The two players will be marked **1** and **-1**. If player one makes a move in a square, that square will be marked with their number. So here we go:

def new\_board():

return ((0,0,0),

(0,0,0),

(0,0,0))

The new\_board method will be called before the play for a fresh board, ready for the players to make their moves on:

def **apply\_move**(**board\_state**, **move**, **side**):

move\_x, move\_y = move

state\_list = list(list(s) for s in board\_state)

state\_list[move\_x][move\_y] = side

return tuple(tuple(s) for s in state\_list)

The apply\_move method takes one of the 3 x 3 tuples for board\_state and returns a new board\_state with the move by the given side applied. A move will be a tuple of length 2, containing the coordinate of the space that we want to move to as two integers. Side will an integer representing the player who is playing the move, either 1 or -1:

import itertools

def available\_moves(**board\_state**):

for x, y in itertools.product(range(3), range(3)):

if board\_state[x][y] == 0:

yield (x, y)

This method gives us the list of legal moves for a given 3 x 3 board\_state, which is simply all the non-zero squares. Now we just need a method to determine whether a player has the three winning marks in a row:

def **has\_3\_in\_a\_line**(line):

return all(x==-1 for x in line) | all(x==1 for x in line)

The has\_3\_in\_a\_line takes a sequence of three squares from the board. If all are either 1 or -1, it means one of the players has gotten three in a row and has won. We then need to run this method against each possible line on the Tic-Tac-Toe board to determine whether a player has won:

def **has\_winner**(board\_state):

# check rows

for x in range(3):

if has\_3\_in\_a\_line (board\_state[x]):

return board\_state[x][0]

# check columns

for y in range(3):

if has\_3\_in\_a\_line([i[y] for i in board\_state]):

return board\_state[0][y]

# check diagonals

if has\_3\_in\_a\_line([board\_state[i][i] for i in range(3)]):

return board\_state[0][0]

if has\_3\_in\_a\_line([board\_state[2 - i][i] for i in range(3)]):

return board\_state[0][2]

return 0 # no one has won

With just these few functions, you can now play a game of Tic-Tac-Toe. Simply start by getting a new board, then have the players successively choose moves and apply those moves to board\_state. If we find that there are no available moves left, the game is a draw. Otherwise, if has\_winner returns either 1 or -1, it means one of the players has won. Let's write a simple function for running a Tic-Tac-Toe game with the moves decided by methods that we pass in, which will be the control policies of the different AI players that we will try out:

def play\_game(**plus\_player\_func**, **minus\_player\_func**):

**board\_state** = new\_board()

**player\_turn** = 1

We declare the method and take it to the function that will choose the action for each player. Each player\_func will take two arguments: the first being the current board\_state and the second being the side that the player is playing, 1 or -1. The player\_turn variable will keep track of this for us:

while True:

\_available\_moves = list(available\_moves(board\_state))

if len(\_available\_moves) == 0:

print("no moves left, game ended a draw")

return 0.

This is the main loop of the game. First we have to check whether there are any available moves left on board\_state; if there are, the game is not over and it is a draw:

if player\_turn > 0:

move = plus\_player\_func(board\_state, 1)

else:

move = minus\_player\_func(board\_state, -1)

Run the function associated with whichever player's turn it is to decide a move:

if move not in \_avialable\_moves:

# if a player makes an invalid move the other player wins

print("illegal move ", move)

return -player\_turn

If either player makes an illegal move, that is an automatic loss. Agents should know better:

board\_state = apply\_move(board\_state, move, player\_turn)

print(board\_state)

winner = has\_winner(board\_state)

if winner != 0:

print("we have a winner, side: %s" % player\_turn)

return winner

player\_turn = -player\_turn

Apply the move to board\_state and check whether we have a winner. If we do, end the game; if we don't, switch player\_turn to the other player and loop back around.

Here is how we could write a method for a control policy that would choose actions completely at random out of the available legal moves:

def random\_player(board\_state, side):

moves = list(available\_moves(board\_state))

return random.choice(moves)

Let's run two random players against each other and check whether the output might look something like this:

**play\_game(random\_player, random\_player)**

**((0, 0, 0), (0, 0, 0), [1, 0, 0])**

**([0, -1, 0], (0, 0, 0), [1, 0, 0])**

**([0, -1, 0], [0, 1, 0], [1, 0, 0])**

**([0, -1, 0], [0, 1, 0], [1, -1, 0])**

**([0, -1, 0], [0, 1, 1], [1, -1, 0])**

**([0, -1, 0], [0, 1, 1], [1, -1, -1])**

**([0, -1, 1], [0, 1, 1], [1, -1, -1])**

**we have a winner, side: 1**

Now we have a good way of trying out different control policies on a board game, so let's go about writing something a bit better. We can start with a min-max function that should play at a much higher standard than our current random players. The full code for the min-max function is also available in the GitHub repo in the min\_max.py file.

Tic-tac-toe is a game with a small space of possibilities, so we could simply run a min-max for the whole game from the board's starting position until we have gone through every possible move for every player. But it is good practice to still use an evaluation function, as for most other games we might play, this will not be the case. The evaluation function here will give us one point for getting two in a line if the third space is empty; it'll be the opposite if our opponent achieves this. First, we will need a method for scoring each individual line that we might make. The score\_line will take sequences of length 3 and score them:

def **score\_line**(line):

minus\_count = line.count(-1)

plus\_count = line.count(1)

if plus\_count == 2 and minus\_count == 0:

return 1

elif minus\_count == 2 and plus\_count == 0:

return -1

return 0

Then the evaluate method simply runs through each possible line on the tic-tac-toe board and sums them up:

def **evaluate**(board\_state):

score = 0

for x in range(3):

score += score\_line(board\_state[x])

for y in range(3):

score += score\_line([i[y] for i in board\_state])

#diagonals

score += score\_line([board\_state[i][i] for i in range(3)])

score += score\_line([board\_state[2-i][i] for i in range(3)])

return score

Then, we come to the actual min\_max algorithm method:

def **min\_max**(board\_state, side, **max\_depth**):

best\_score = None

best\_score\_move = None

The first two arguments to the method, which we are already familiar with, are board\_state and side; however, max\_depth is new. Min-max is a recursive algorithm, and max\_depth will be the maximum number of recursive calls we will make before we stop going down the tree and just evaluate it to get the result. Each time we call min\_max recursively, we will reduce max\_depth by 1, stopping to evaluate when we hit 0:

moves = list(available\_moves(board\_state))

if not moves:

return 0, None

If there are no moves to make, then there is no need to evaluate anything; it is a draw, so let's return with a score of 0:

for **move** in moves:

**new\_board\_state** = apply\_move(board\_state, **move**, side)

Now we will run through each legal move and create a new\_board\_state with that move applied:

**winner** = has\_winner(**new\_board\_state**)

if **winner** != 0:

return **winner** \* **10000**, move

Check whether the game is already won in this new\_board\_state. There is no need to do any more recursive calling if the game is already won. Here, we are multiplying the winner's score by 1,000; this is just an arbitrary large number so that an actual win or loss is always considered better/worse than the most extreme result we might get from a call to evaluate:

else:

if **max\_depth** <= 1:

score = evaluate(**new\_board\_state**)

else:

score, \_ = **min\_max**(**new\_board\_state**, -side, **max\_depth** - 1)

If you don't already have a winning position, then the real meat of the algorithm starts. If you have reached max\_depth, then now is the time to evaluate the current board\_state to get our heuristic for how favorable the current position is to the first player. If you haven't reached max\_depth, then recursively call min\_max with a lower max\_depth until you hit the bottom:

if side > 0:

if best\_score is None or score > best\_score:

best\_score = score

best\_score\_move = move

else:

if best\_score is None or score < best\_score:

best\_score = score

best\_score\_move = move

return best\_score, best\_score\_move

Now that we have our evaluation for the score in new\_board\_state, we want either the best or worst scoring position depending on which side we are. We keep track of which move leads to this in the best\_score\_move variable, which we return to with the score at the end of the method.

A min\_max\_player method can now be created to go to our earlier play\_game method:

def **min\_max\_player**(board\_state, side):

return min\_max(board\_state, side, 5)[1]

Now if we run a series of games with random\_player against a min\_max player, we will find that the min\_max player wins almost every time.

The min max algorithm, though important to understand, is never used in practice because there is a better version of it: min max with alpha beta pruning. This takes advantage of the fact that certain branches of the tree can be ignored or pruned, without needing to be fully evaluated. Alpha beta pruning will produce the same result as min max but with, on average, half as much search time.

To explain the idea behind alpha beta pruning, let's consider that while building our min-max tree, half of the nodes are trying to make decisions to maximize the score and the other half to minimize it. As we start evaluating some of the leaves, we get results that are good for both min and max decisions. If taking a certain path through the tree scores, say -6, the min branch knows it can get this score by following the branch. The thing that stops it from using this score is that max decisions has to make the decisions, and it cannot choose a leaf favorable to the min node.

But as more leaves are evaluated, another might be good for the max node, with a score of +5. The max node will never choose a worse outcome than this. But now that we have a score for both min and max, we know if we start going down a branch where the best score for min is worse than -6 and the best score for max is worse than +5, then neither min nor max will choose this branch, and we can save on the evaluation of that whole branch.

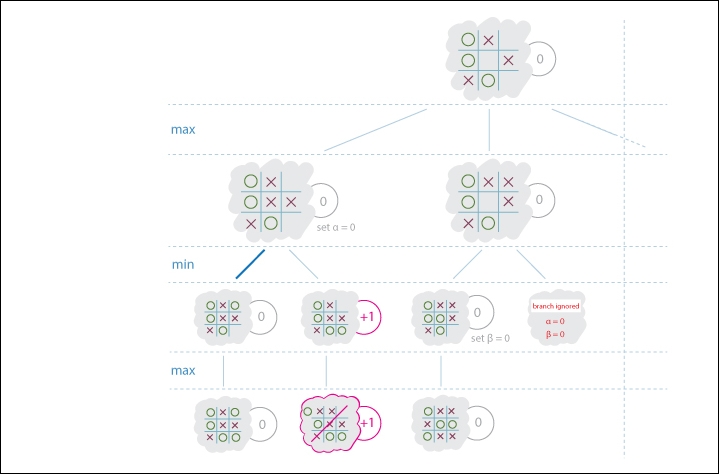
The alpha in alpha beta pruning stores the best result that the max decisions can achieve. The beta stores the best result (lowest score) that the min decisions can achieve. If alpha is ever greater than or equal to beta, we know we can skip further evaluation of the current branch we are on. This is because both the decisions already have better options.

Figure 4 gives an example of this. Here see that from the very first leaf itself, we can set an alpha value of 0. This is because once the max player has found a score of 0 in a branch, they need never choose a lower score. Next, in the third leaf across, the score is 0 again, so the min player can set their beta score to 0. The branch that reads branch ignored no longer needs to be evaluated because both alpha and beta are 0.

To understand this, consider all the possible results that we could get from evaluating the branch. If it were to result in a score of +1, then the min player would simply choose an already existing branch where it had scored 0. In this case, the branch to the ignored branches left. If the score results in -1, then the max player would simply choose the left most branch in the image where they can get 0. Finally, if it results in a score of 0, it means no one has improved, so the evaluation of our position remains unchanged. You will never get a result where evaluating a branch would change the overall evaluation of the position. Here is an example of the min max method modified to use alpha beta pruning:

import sys

def



*Figure 4: Min max method with alpha beta pruning*

min\_max\_alpha\_beta(board\_state, side, max\_depth,

alpha=-sys.float\_info.max,

beta=sys.float\_info.max):

We now pass in both alpha and beta as parameters; we stop searching through the branches that are either less than alpha or more than beta:

best\_score\_move = None

moves = list(available\_moves(board\_state))

if not moves:

return 0, None

for move in moves:

new\_board\_state = apply\_move(board\_state, move, side)

winner = has\_winner(new\_board\_state)

if winner != 0:

return winner \* 10000, move

else:

if max\_depth <= 1:

score = evaluate(new\_board\_state)

else:

score, \_ = min\_max\_alpha\_beta(new\_board\_state, -side, max\_depth - 1, alpha, beta)

Now when we recursively call min\_max\_alpha\_beta, we pass in our new alpha and beta values that may have been updated as part of the search:

if side > 0:

if score > alpha:

alpha = score

best\_score\_move = move

The side > 0 expression means that we are looking to maximize our score, so we will store the score in the alpha variable if it's better than our current alpha:

else:

if score < beta:

beta = score

best\_score\_move = move

If side is < 0 we are minimizing, so store the lowest scores in the beta variable:

if alpha >= beta:

break

If alpha is greater than beta, then this branch cannot improve the current score, so we stop searching it:

return alpha if side > 0 else beta, best\_score\_move

In 1997, IBM created a chess program called Deep Blue. It was the first to beat the reigning world chess champion Garry Kasparov. While an amazing achievement, it would be hard to call Deep Blue intelligent. Though, it has huge computational power, and its underlying algorithm is just the same min-max algorithm from the 50's. The only major difference is that Deep Blue took advantage of the opening theory in chess.

The opening theory comprises of a sequences of moves that are from the starting position and are known to lead to favorable or unfavorable positions. For example, if white starts with the move pawn e4 (the pawn in front of the king moved forward by two spaces), then black responds with pawn c5; this is known as the Sicilian defense, and there are many books written on the sequences of play that could follow from this position. Deep Blue was programmed to simply follow the best moves recommended from these opening books and only start calculating the best min-max move once the opening line of play reaches its end. In this way, it saves on computational time, but it also takes advantage of the vast human research that has gone into the working out of the best positions in the opening stages of chess.