Comp 6321 - Machine Learning - Project report

Federico O'Reilly Regueiro

December 15^{th}, 2016

1 Introduction and motivation

The current proliferation of readily available data collections that has been spawned by the inetrnet has set the stage for machine learning as a pervasive means to develop different kinds of intelligent systems. It is, however, the heterogeneity and inconsistency of such data that limits what can be done by regular supervised learning. Some of the biggest efforts put forth while tackling a supervised learning problem continue to be collection and curation of data sets.

Additionally, under certain circumstances, the notion of labeling all input instances becomes unwieldy; such as the case of the input frames of an automated vehicle.

The lack of clear or constant labels to inputs gives rise to two forms of learning that are not supervised. Fully unsupervised learning, under which algorithms aim at finding data's underlying structure; and reinforcement learning, which in some ways falls between supervised learning and unsupervised learning. In reinforcement rearning, 'labels' are sparse and time-delayed [4]. This sparsity and delay give rise to what is referred to as 'the credit assignment problem', where given a certain outcome from a series of actions, it is difficult to ascertain which, if any, of the actions leading to the outcome should bear the largest responsibility for said outcome.

There are several approaches to reinforcement learning, we focus on a particular temporal-domain type which tries to assign the credit of a given outcome somewhat evenly among the events leading to it. It does so by supposing that at any point in time the reward (or penalty) is equal to the reward gained at that point plus the sum of possible discounted future rewards. The rationale for discounting rewards over time corresponds to the notion of rewards being more desirable now than later on[1]. Although there are several formulations, the two main approaches to TD differ in what they strive to learn. On the one hand we can learn the intrinsic value or potential reward of a given state, and on the other hand, we can learn the worth of an action given a state; the former being based on value and the latter is known as policy-based or Q, for quality of policy.

[1] As Andrew Ng puts it in his online lecture on the topic, we might be dead tomorrow. [6]

Applications of reinforcement learning are varied and currently under development in fields such as vehicle control, robotics, gaming and prediction of streaming data such as that of financial applications to name a few.

On their paper regarding evolutionary neural networks for playing othello, Moriarty and Miikkulainen state, 'games are an important domain for studying problem-solving strategies [5].' Traditionally, due to their well-defined rules, state-transitions and goals, games have made a good sandbox for the development of any form of intelligent agents.

All approaches to game-playing agents share one goal, to reduce the scope of the state-space in which a search for optimal action takes place. Some approaches use expert-knowledge, such as is the case with minimax using heuristic functions. Statistically informed methods such as the one I have implemented strive to not so much prune the search space but to redistribute the task so that when an evaluation is needed, an exploration o a similar state-space has already been performed. They achieve this by first widely exploring the state-space (train) in order to learn what sort of branches of said space to discard and which sort of branches tend to lead to better outcomes.

For this project, I have chosen othello given its somewhat restricted scope; there is only one sort of token, two players, a limited number of moves per match ($moves \leq 60$), the branching factor is relatively small and the outcome is *zero-sum*. Originally I had planned to train the network with labeled board configurations but finding or creating a sufficient data-set proved to be much more difficult than anticipated; which is the reason for having subsequently chosen RL.

1.a State-of-the-art

Much has been achieved in the field of game-playing agents, recently Google's Alpha-go bested top-ranking go player Lee Sedol. Go has frequently been mentioned as the unattainable goal in computer game-playing given the vastness of its state-space. Alpha-go achieved this outstanding result with the combination of different strategies, via 3 Neural Networks [1]. The networks were trained with different methods, including playing against random agents and each other as well as deterministic expert-knowledge based agents. One of the networks is a fully connected value network with a single output while the other two are policy networks specialized for speed and accuracy respectively, each treating the classification problem as a multi-class classification, outputting values for every square on the board.

Another milestone in the field, is a deep policy network created by DeepMind which has learned, with no human intervention, how to play Atari games at a very high level [4].

2 Description of approach

2.a Model choices: TD - flavor

There are several formulations for TD learning [7], including policy or Q based and value based. I have chosen to use the TD-(λ) formulation from Tesauro's classic work on Backgammon [8] which is well suited for neural networks. In Tesauro's work, given an outcome at the end of the game , updates depend on the difference between predictions at successive states.

$$\Delta_{w_t} = \alpha(P_{t+1} - P_t) \sum_{k=1}^{t} \lambda^{t-k} \nabla_w P_k, \quad 0 \leq \lambda \leq 1$$

Where γ is the discount factor for rewards, λ is the relevance of past actions wrt the weight updates. We can apply an update every turn to minimize the difference between state appraisal at consecutive states s_t, s_{t+1} and as we do this, we keep a cached copy of the gradients for future updates. At end-game, a final update is performed given the difference between the appraisal of the second-to-last state and the final outcome.

Much literature can be found recently regarding Q-learning and deep-Q networks, however, I found this approach to be unsuitable for othello since the set of available actions (and therefore of potential policies) depends on the current state and varies greatly from one state to another, therefore finding the right policy for the total potential actions (64, where an action can be seen as laying down a token on the board) is an inadequate choice when each state we will have an average of 7 available actions.

Another interesting optimization algorithm for othello-playing networks is based on genetic-algorithms or evolutionary neural networks, ENNs [2] [5] [3]. This seems like an effective approach but the time needed to spawn multiple generations proved to be far greater than the time before the project's due date.

3 Setup

The chosen environment for implementing the network was TensorFlow, given its current widespread use and rapidly growing user-base. In order to run TensorFlow, an Anaconda/python 3.5 environment was created on windows' included bash/ubuntu. The environment must be active for python to detect the TensorFlow v0.12 module. This makes the code somewhat less portable than what I originally intended.

3.a Network Implementation

A good deal of effort for this project was placed in learning to use the TensorFlow packages from Google and in deciding a proper network configuration. Not much can be written regarding the environment except that it represents a bit of a paradigm shift from the dominant more linear sort of programming. In order to avoid the overhead incurred by interfacing Python code with the more efficient optimized engine, Tensorflow requires defining graphs of operations which are then executed as a batch in the back-end the moment one part of the graph is evaluated (eg the net's output). Additionally, TensorFlow uses tensors or multi-dimensional matrices, which is not unlike Matlab or Numpy and well suited for the task.

The implemented network receives a batch of boards representing the possible resulting configurations or states s_{t+1} given the current state s_t. The expected output is the argument maximizing the value of the successor state over all available actions a:

$$out = \arg\max_a (V(s_t, a))$$

A board is an 8×8 matrix with zero-entries at unoccupied positions, 1 for each position with a black token and -1 for each square with a white token.

Regarding the net architecture, we have chosen to use an informed architecture. In other words, the design of the network is thought out for the particular task at hand, and not necessarily with generalization in mind. The ANN consists of three conv-nets running in parallel and feeding their respective outputs to a fully connected net comprised of two hidden layers of 64 and 32 units respectively as well as one output activation unit at the end. One of the conv-nets receives an 8×1 window sliding across columns and rows of the board, another receives a 3×3 window placed at corners and mid-section of the board. The remaining conv-net receives the board diagonals.

Like many board games, othello has patterns that can be strategically significant yet they are highly sensitive to position. In other words, a certain configuration in a corner has vastly different implications than it would have if it were 1 row or column inwards. Therefore, we use convolutional networks to find these patterns but suppose the weighting of said patterns to be highly dependent on their location on the board. For this reason, we have avoided the pooling layer typically implemented at the end of a convolutional layer. We use three parallel convolution nets for the purpose of discovering features over different views of the board.

We first pre-process our board by generating a $< batchSize > \times 8 \times 8 \times 8$ tensor with all possible flips and rotations of the board. The flips and rotatoions of the board are fed into the nets in the guise of channels, borrowing from the image processing origin of convolution nets. For the conv-net specialized in rows and columns, as well as the one specialized in diagonals, only the four flips, and not the rotations are fed into the net, in an $< batchSize > \times 8 \times 1 \times 4$ tensor. For the 3×3 window, we cut a 5×5 corner-section across all symmetries $< batchSize > \times 8 \times 8 \times 8$ tensor, yielding a $< batchSize > \times 5 \times 5 \times 8$ tensor. The convolutions have the following strides: for the row/column net, strides are 1×1 with no padding, allowing the window to slide over 8 positions in each one of the four symmetries. For the smaller, 3×3 sliding window, the strides are 2×2 with no padding, allowing the window to slide over four positions of the smaller 5×5 sections of the flipped and rotated board. For the diagonals, there is no stride; a conv-net has been used exclusively for harnessing the built-in capabilities of processing different channels.

The three nets output to different feature filters, allowing each net to automatically discover features in the given board configurations. Both the row/column and the 3×3 filters output to 10 feature-maps, while the diagonal, outputs to only 4 features.

All feature-maps are then output to a 64-unit layer which is fully connected to a 32-unit layer and which in turn outputs a single value, per each board in the batch. The entirety of the nets units have a hyperbolic tangent function, allowing for the negative rating of certain configurations and all units have a bias of 1[2]. The weights of the net trained for the current submission were initialized with tensorflow's truncated normal generator with default values of $\mu = 0, \sigma = 1.0$.

The parameters used for the implementation were:

- Optimizer - Gradient Descent
- Learning rate - 0.01
- Discount factor γ - 0.9
- Temporal difference weighting λ - 0.3
- $P(Dropout)$ during training = 0.3[3]

3.b Testing/training routine

In order for the network to learn, it plays against a random player-agent, which plays a randomly chosen move from all available moves. The network can be assigned either black or white at random for each match since opening the game (black) or following (white) have different implications for game-play. This requires multiplying the input matrix representing a board by -1 in the case that the network plays white, in order for the model to fully exploit whatever it has learned while playing either color.

The use of a random player-agent also palliates one of the biggest issues in reinforcement learning which is the explore exploit dilemma whereby acting on learned ideal actions diminishes the potential to explore the solution-space further or conversely, the earnest exploration of the solution-space implies a much slower convergence towards ideal state-appraisal or policies. The network agent will be drawn to repeat patterns it has discovered to be fruitful while the random player-agent will impose a certain amount of exploration.

[2] actually in the unit trained for this submission, biases were 0.1 due to a typo in the code, but this has been corrected in the script

[3] Dropout was added almost as an afterthought, inspired by Moriarty and Miikkulainen's ENN implementation where reportedly encoding the network configuration as part of its 'genome' gave a significant performance boost to the evolutionary optimization.

Since the network trains against a random agent, there is little risk of it over-specializing or overfitting, as would be the case if it were to train against a deterministic agent. Due to this, there is little distinction between training and testing and the learning process yields a form of test result as it happens. Each move during a match in which learning takes place incurs a TD error, which is then used to update the weights as described in section 2.a. The value of the game's outcome is estimated to be of the form:

$$R\left(s_T\right) = tanh\left(tokens_{network} - tokens_{randomPlayer}1\right)$$

A count of matches won, lost or tied is updated after each iteration and is used to measure the progression of learning.

4 Results

The implementation part of the network has presented a fair set of issues, many of them which simply require more time to find and solve. The performance of the net is currently far from what I expected but the process has been highly educational regarding putting together a network of this nature.

Firstly, the constraints regarding putting together computation graphs, as is required by tensorflow, were difficult to assimilate and still, to the time of writing this report, my code suffers from some form of memory leak that severely impacts performance after many iterations, requiring saving parameters as a checkpoint file, exiting Python[4], reimporting modules, restoring the checkpoint and restarting training. These steps must be performed every 30 to 100 iterations in order to maintain a reasonable pace of learning, otherwise playing a game and performing backpropagating can take up to 10 minutes after training five or six hours. This issue has made it difficult to train the network over a sufficiently large set of iterations in order for it to significantly learn as well as evaluating performance and tuning parameter settings.

Another problem encountered is that the network tends to be overly optimistic[5] outputting values close to 1 for the best-appraised state or board configuration from a given batch even when the network is 1 move away from losing a match.

Additionally, the black-box nature of an ANN has made it somewhat difficult to evaluate the learning process of the net with anything else than its (somewhat feeble) performance.

At the time of writing this report, many such problems had just surfaced so evidence for the previous network's performance is included. I have re-written a good deal of the code, which I include as a reference but unfortunately cannot include results as this would require leaving the network training for quite a few hours still. I have started training it and the memory-leak seems to be at least partially solved and value appraisals seem to be closer to target and improving slightly over epochs, as are errors per prediction.

[4] or deleting modules...

[5] Bah, humbug!

I have attached a long log file for the previous' net training over slightly more than 1000 epochs. (Warning, the file is close to 895,000 lines with Unix line termination -not CR/LF- if opening in a Windows environment use something such as Notepad++ and do not use Notepad since formatting will be broken).

Logging is given in an informative fashion such as:

```
(
    (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, B, W, 0, 0, 0)
    (0, 0, 0, W, B, 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)
)

Black  plays:
max value v at index idx is:  0.770257 2
(
                   ...0, 0, 0, 0)
...
...
                   ...W, B, B, B)
)

White 's turn:
(
    (W, W, W, W, W, B, B, W)
    (W, B, B, B, B, B, B, B)
    (W, W, B, B, B, B, W, W)
    (W, B, B, B, W, B, W, W)
    (W, W, B, B, B, B, W, B)
    (W, W, B, W, W, W, B, B)
    (W, W, W, W, W, B, B, B)
    (B, W, W, W, W, B, B, B)
)

White 's turn:
no moves
White 's turn:
Game Over
Score: Black -  32  White -  32
Loss at end-game:  [[-0.79794914]]
The round took: 9.991903066635132  seconds.
Up to now:  3  wins,  2  losses and  1  ties.
```

A plot paints an eloquent picture regarding the unimpressive performance of the previous version of the network. The previous version is included in the deliverable in a folder labeled 'oldAndBuggy' and the current version is included at the base layer as it seems to perform significantly better.

The code is also printed at the end of the document for the reader's convenience.

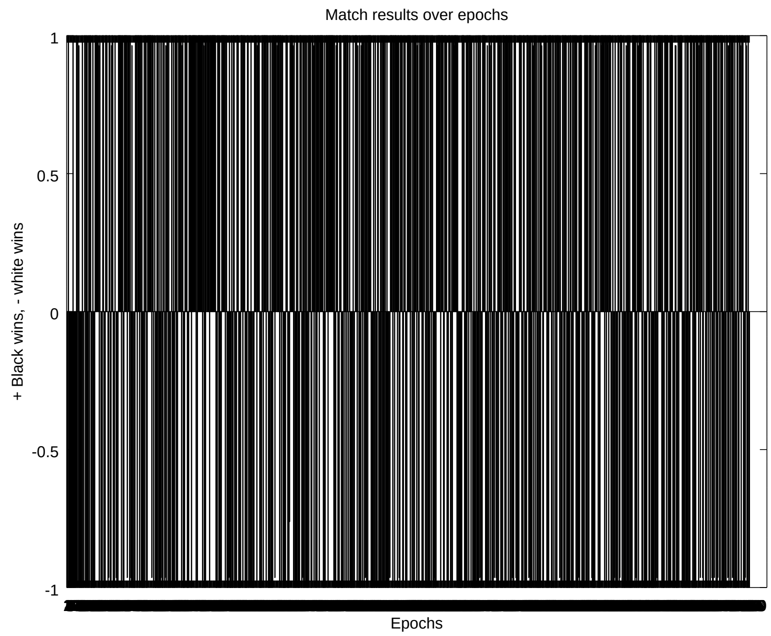

Figure 1: Wins and losses per epoch, black bars indicate a win, positive wins are wins by the model, negative wins are matches won by the random player-agent.

5 Future work

The project has been very satisfactory in so far as I have gained experience designing and implementing a network of non-trivial proportions, yet there are several avenues that are left open for future work. Further fixing the memory leak in my code is a necessary precondition for any one of them. I list the three most important ones in what seems to me to be order of relevance.

5.a Cross-validate different parameter settings

Choosing one model, perform training against a random player agent with different values for the learning rate, γ, λ and dropout.

5.b Merge with principles of GAs

Supposing that the network were to achieve a reasonable level of performance playing against a random player-agent, a logical next step would be to have a network learn from playing another network. However, if the same model plays against itself, the risk of overfitting is very high, where the network will most likely learn given a very narrow set of responses to its own actions (namely, its own responses). One simple way to avoid this would be to train several nets against random player-agents up to a reasonable performance level and then have them all play/train against each other choosing pairs randomly for every match. This would require a true division of training/testing, potentially against well established deterministic agents. Potentially, implement ENNs such as the ones proposed in Chelapilla [2], Moriarty and Miikkulainen [5] or Chong [3] and combine both approaches.

5.c Introduce a small noise component

If overfitting is the Achilles' tendon of Neural Nets, one simple solution to this problem is the addition of jitter or a small noise component to the inputs. Since our inputs have only three possible values per square, adding jitter might yield some benefits by smoothing the output function slightly. There is very little implementation and computational overhead needed in order to test this idea; it's probably worth it.

6 Appendix i - Code

6.a ANN

```
import tensorflow as tf
import numpy as np

class othello_net():
    @staticmethod
    def weight_variables(shape, name=None):
      initial = tf.random_normal(shape, stddev=0.03, name=name)
      return tf.Variable(initial)

    @staticmethod
    def bias_variables(shape, name=None):
      initial = tf.constant(1.0, shape=shape, name=name)
      return tf.Variable(initial)

    def __init__(self, session, parent=None):
        if parent:
            pass # TODO for GAs some form of deepcopy + modif
        else:
            self.initialize_conv1_weights()
            self.initialize_fc_weights()
            self.initialize_board_placeholders()
            self.initialize_training_placeholders()
            self.initialize_turn_boards()
            self.initialize_train_vars()
            self.initialize_convs()
            self.initialize_ff()
            self.accum_grads = []
            self.discount_factor = 0.9
            self.lambdaa = 0.7
            self.opt = tf.train.GradientDescentOptimizer(3e-3)
            self.vars_list = [self.conv1_weights, self.conv1_bias,
                              self.conv2_weights, self.conv2_bias,
                              self.conv_diag_weights, self.conv_diag_bias,
                              self.fc1_weights, self.fc1_bias, self.fc2_weights,
                              self.fc2_bias, self.out_weights, self.out_bias]
            self.grad_var_list = self.opt.compute_gradients(self.h_out, self.vars_list)
            self.values_history = []
            session.run(tf.initialize_all_variables())

    # TODO if we ever get to do the evolutionary variant, tic toc... time is pressing
```

```python
    @classmethod
    def spawn(cls, parent):
        return cls(parent)

    # When we want to start a new match we need to clear some variables
    def reset_for_game(self):
        self.accum_grads = []
        self.accum_grad = []
        self.values_history = []

    # give an appraisal given a board configuration, train if told to do so
    def evaluate(self, boards, session, train=False):
        boards = np.asarray(boards)
        if boards.ndim == 2:
            boards = np.expand_dims(boards, axis=0)
        batch_size = boards.shape[0]
        diag = self.get_diagonal(boards)
        if train:
            v, grad_var_list = session.run([self.h_out, self.grad_var_list],
                                           feed_dict={self.boards: boards,
                                                      self.boards_diag: diag,
                                                      self.keep_prob: 0.7,
                                                      self.batch_size: batch_size})
            idx = np.argmax(v, 0)
            self.values_history.append(v[idx])
            if self.values_history.__len__() > 1:
                loss = self.values_history[-1] - self.values_history[-2]
                print('loss: ', loss[0][0])
                for outer_index, vars in enumerate(self.vars_list):
                    self.opt.apply_gradients([(loss * self.accum_grads[outer_index], vars)])
            _ = [(self.update_lambda_grads(g_v[0], index), g_v[1]) for index, g_v in enumerate(grad_var_list)]
        else:
            v, grad_var_list = session.run(self.h_out,
                                           feed_dict={self.boards: boards,
                                                        self.boards_diag: diag,
                                                        self.keep_prob: 1,
                                                        self.batch_size: batch_size})
            idx = np.argmax(v, 0)
        print('max value v at index idx is: ', v[idx][0][0], idx[0])
        return idx[0], v[idx][0][0] # TODO this looks ugly fix upstream

    # NOTE if training, this should be called at the end of a match
    def learn_from_outcome(self, tally):
        loss = np.tanh(tally) - self.values_history[-1]
```

```
        print('Loss at end-game: ', loss)
        for outer_index, vars in enumerate(self.vars_list):
            self.opt.apply_gradients([(loss * self.accum_grads[outer_index], vars)])

    # TODO fix this, for some reason it's broken on the tf side
    def set_epochs(self, epochs):
        self.epochs.assign(epochs)


    def initialize_conv1_weights(self):
        # create 8 by 1 filter - row/col
        self.conv1_weights = self.weight_variables([1, 8, 4, 10])
        self.conv1_bias = self.bias_variables([1, 1, 1, 10])

        # Chop 5x5 part of the board and slide a 3x3 window over it
        self.conv2_weights = self.weight_variables([3, 3, 8, 10])
        self.conv2_bias = self.bias_variables([1, 1, 1, 10])

        # a bit of a mis-use of conv-nets but pass the diagonals into a single input
        # the useful bits of input channels and features are the reason for this
        self.conv_diag_weights = self.weight_variables([1, 1, 4, 4])
        self.conv_diag_bias = self.bias_variables([1, 1, 1, 4])

    # Try with 64/32 1st layer, second layer neurons
    def initialize_fc_weights(self):
        self.fc1_weights = self.weight_variables([174, 64])
        self.fc1_bias = self.bias_variables([1,64])

        self.fc2_weights = self.weight_variables([64, 32])
        self.fc2_bias = self.bias_variables([1, 32])

        self.out_weights = self.weight_variables([32,1])
        self.out_bias = self.bias_variables([1,1])

    def initialize_train_vars(self):
        self.epochs = tf.Variable(0)

    def initialize_board_placeholders(self):
        self.boards = tf.placeholder(tf.float32, shape=[None, 8, 8])
        self.boards_diag = tf.placeholder(tf.float32, shape=[None, 8, 1, 4])

    def initialize_training_placeholders(self):
        self.batch_size = tf.placeholder(tf.int32)
        self.keep_prob = tf.placeholder(tf.float32)
        self.outcome_val = tf.placeholder(tf.float32)

```

```python
    def initialize_turn_boards(self):
        self.sym1 = tf.reverse(self.boards, [False, True, False])
        self.sym2 = tf.reverse(self.boards, [False, False, True])
        self.sym3 = tf.reverse(self.sym2, [False, True, False])
        self.sym4 = tf.transpose(self.boards, perm=[0, 2, 1])
        self.sym5 = tf.transpose(self.sym1, perm=[0, 2, 1])
        self.sym6 = tf.transpose(self.sym2, perm=[0, 2, 1])
        self.sym7 = tf.transpose(self.sym3, perm=[0, 2, 1])
        self.board_sym_tensor = tf.transpose([self.boards, self.sym1, self.sym2, self.sym3, self.sym4,
            self.sym5, self.sym6, self.sym7], [1,2,3,0])
        self.boards_half_sym = tf.slice(self.board_sym_tensor, [0, 0, 0, 0], [-1, -1, -1, 4])
        self.boards_chopped = tf.slice(self.board_sym_tensor, [0, 0, 0, 0], [-1, 5, 5, -1])

    @staticmethod
    def get_diagonal(boards):
        boards = np.asarray(boards)
        diags1 = np.expand_dims(boards.diagonal(0, 1, 2), axis=2)
        diags2 = np.expand_dims(np.fliplr(boards).diagonal(0,2,1), axis=2)
        # Yes, lr does the trick, flipud actually reverses batches (dimension 0)
        diags3 = np.fliplr(diags1)
        diags4 = np.fliplr(diags2)
        return np.stack([diags1, diags2, diags3, diags4], axis=3)

    def initialize_convs(self):
        self.h_conv1 = tf.nn.tanh(tf.nn.conv2d(self.boards_half_sym,
            self.conv1_weights, strides=[1, 1, 1, 1], padding='VALID') + self.conv1_bias)

        self.h_conv2 = tf.nn.tanh(tf.nn.conv2d(self.boards_chopped,
            self.conv2_weights, strides=[1, 1, 1, 1], padding='VALID') + self.conv2_bias)

        self.h_conv_diag = tf.nn.tanh(tf.nn.conv2d(self.boards_diag, self.conv_diag_weights,
            strides = [1, 8, 1, 1], padding='VALID') + self.conv_diag_bias)

    def initialize_ff(self):
        conv1_flat = tf.reshape(self.h_conv1, [-1, 1*8*10])
        conv2_flat = tf.reshape(self.h_conv2, [-1, 3*3*10])
        conv_diag_flat = tf.reshape(self.h_conv_diag, [-1, 1*1*4])
        conv_out = tf.concat(1, [conv1_flat, conv2_flat, conv_diag_flat])
        self.h_fc1 = tf.nn.tanh(tf.matmul(conv_out, self.fc1_weights)+self.fc1_bias)
        self.h_fc1_drop = tf.nn.dropout(self.h_fc1, self.keep_prob)
```

```
        self.h_fc2 = tf.nn.tanh(tf.matmul(self.h_fc1_drop,
            self.fc2_weights) + self.fc2_bias)
        self.h_fc2_drop = tf.nn.dropout(self.h_fc2, self.keep_prob)
        self.h_out = tf.nn.tanh(tf.matmul(self.h_fc2_drop,
            self.out_weights) + self.out_bias)

    def update_lambda_grads(self, grad, idx):
        if not self.accum_grads or idx == self.accum_grads.__len__():
            self.accum_grads.append(grad)
        else:
            self.accum_grads[idx].__mul__(self.lambdaa)
            self.accum_grads[idx] = tf.add(self.accum_grads[idx], grad)
```

6.b interface

```
# COMP 6321 Machine Learning, Fall 2016
# Federico O'Reilly Regueiro - 40012304
# Final project - othello with neural nets

import othello as o
import position as p
import board as b
import tensorflow as tf
import othelloNetV2 as otnet
import time
import random

session = tf.Session()
game = o.game()
on = otnet.othello_net(session)
score_series = []
wins = 0
losses = 0
ties = 0

def batch(color):
    board_now = game.board
    possible_moves = board_now.get_valid_moves(color)
    boards = []
    for m in possible_moves:
        new_board = b.Board(board_now)
        new_board.do_move(m, color)
        new_board.relativize(color)
        boards.append(new_board.squares)
    return boards, possible_moves

def play_net(train=False):
    color = random.choice((b.BLACK, b.WHITE))
    done = False
    tic = time.time()
    global wins
    global losses
    global ties
    global score_series
    print(game.board.to_string())
    if color == b.WHITE:
        done = not game.play_random_turn(b.opposite(color))
    while not done:
        boards, moves = batch(color)
        if boards:
            print(game.turn_to_string(color), ' plays:')
            idx, v = on.evaluate(boards, session, train)
```

```python
            game.play_move(moves[idx], color)
            print(game.board.to_string())
        else:
            game.pass_moves += 1
        print(game.turn_to_string(b.opposite(color)),'\'s turn:')
        done = not game.play_random_turn(b.opposite(color))
    outcome = game.board.get_score()
    score_series.append(outcome)
    if train:
        on.learn_from_outcome(outcome[game.turn_to_string(color)] - outcome[game.turn_to_string(b.opposite(color))])
    if score_series[-1][game.turn_to_string(color)] > score_series[-1][game.turn_to_string(b.opposite(color))]:
        wins += 1
    elif score_series[-1][game.turn_to_string(color)] < score_series[-1][game.turn_to_string(b.opposite(color))]:
        losses += 1
    else:
        ties += 1
    game.reset()
    on.reset_for_game()
    print('The round took:', time.time()-tic, ' seconds.')
    print('Up to now: ', wins, ' wins, ', losses, ' losses and ', ties, ' ties.')
    return outcome


def save_checkpoint(path="./otnet_v2.ckpt"):
    saver = tf.train.Saver()
    saver.save(session, path)
    print('Saved checkpoint')

def restore_checkpoint(path="./otnet_v2.ckpt"):
    saver = tf.train.Saver()
    saver.restore(session, path)
    print("Model restored.")
```

6.c The game

```python
# COMP 6321 Machine Learning, Fall 2016
# Federico O'Reilly Regueiro - 40012304
# Final project - othello with neural nets

import random
import board as b

BLACK = b.BLACK
WHITE = b.WHITE

class game():
    def __init__(self, board=None, turn=None, pass_moves=None):
        self.board = board if board else b.Board()
        self.turn = turn if turn else BLACK
        self.pass_moves = pass_moves if pass_moves else 0

    def reset(self):
        self.board = b.Board()
        self.turn = BLACK
        self.pass_moves = 0

    @classmethod
    def started(cls, board, turn, pass_moves=None):
        p_m = pass_moves if pass_moves else 0
        return cls(board, turn, p_m)

    def play_random_turn(self, turn):
        if self.pass_moves >= 2:
            score = self.board.get_score()
            print('Game Over')
            print('Score: Black - ', score['Black'], ' White - ',
                score['White'])
            return False
        moves = self.board.get_valid_moves(turn)
        if not moves:
            print('no moves')
            self.pass_moves += 1
            return True
        self.pass_moves = 0
        move = random.choice(moves)
        self.board.do_move(move, turn)
        print(self.board.to_string())
        return True

    def play_move(self, move, turn):
        self.board.do_move(move, turn)

```

```python
    def show_available_moves(self, turn):
        moves = self.board.get_valid_moves(turn)
        board_copy = b.Board(self.board)
        for move in moves:
            board_copy.place_token(move, 2)
        print(board_copy.to_string())

    def turn_to_string(self, turn):
        return 'Black' if (turn == b.BLACK) else 'White'
```

6.d The board

```
# COMP 6321 Machine Learning, Fall 2016
# Federico O'Reilly Regueiro - 40012304
# Final project - othello with neural nets

import copy
import position as p

BOARD_SIZE = 8
BLACK = 1
WHITE = -1

def opposite(turn):
    return turn * -1

class Board:
    directions = {'up': p.Pos.up, 'left': p.Pos.left, 'up_left': p.Pos.up_left, 'up_right': p.Pos.up_right,
                  'down_left': p.Pos.down_left, 'down_right': p.Pos.down_right, 'right': p.Pos.right, 'down': p.Pos.down}

    def __init__(self, prev_board=None, single_list_board=None):
        if prev_board:
            self.squares = copy.deepcopy(prev_board.squares)
        elif single_list_board:
            assert single_list_board.__len__() == BOARD_SIZE * BOARD_SIZE
            self.squares = [[0] * BOARD_SIZE for _ in range(BOARD_SIZE)]
            for row in range(BOARD_SIZE):
                for col in range(BOARD_SIZE):
                    self.squares[row][col] = single_list_board[(row*BOARD_SIZE)+col]
        else:
            self.squares = [[0]*BOARD_SIZE for _ in range(BOARD_SIZE)]
            s = int(BOARD_SIZE/2)
            self.squares[s][s] = BLACK
            self.squares[s-1][s-1] = BLACK
            self.squares[s-1][s] = WHITE
            self.squares[s][s-1] = WHITE

    def relativize(self, color):
        for row in self.squares:
            for square in row:
                square *= color

    def to_string(self):
        the_str = "(\n"
        char_map = {BLACK: 'B', 0: 'O', WHITE: 'W', 2: '*'}
```

```
        for idx_v, row in enumerate(self.squares):
            for idx_h, token in enumerate(row):
                if idx_h == 0:
                    the_str += "   ("
                the_str += char_map[token]
                if idx_h == 7:
                    the_str += ")" + '\n'
                else:
                    the_str += ", "
        the_str += ")" + '\n'
        return the_str

    def do_move(self, pos, turn):
        flips = self.get_flips(pos, turn)
        self.do_flips(flips)
        self.place_token(pos, turn)

    def do_flips(self, flips):
        for flip in flips:
            self.place_token(flip, opposite(self.get_token(flip)))

    def place_token(self, pos, token):
        self.squares[pos.v][pos.h] = token

    def get_token(self, pos):
        if not pos.is_valid():
            raise ValueError
        return self.squares[pos.v][pos.h]

    def get_valid_moves(self, turn):
        emptys = self.get_empty_squares()
        valid_moves = []
        for pos in emptys:
            for k in self.directions:
                new_pos = copy.deepcopy(pos)
                direc = self.directions[k]
                if direc(new_pos) and self.is_dir_valid(new_pos, direc, turn) \
                        and (valid_moves.count(pos) == 0):
                    valid_moves.append(pos)
        return valid_moves

    def get_flips(self, pos, turn):
        flips = []
        for k in self.directions:
            direc = self.directions[k]
            new_pos = copy.deepcopy(pos)
            direc(new_pos)
            while self.is_dir_valid(new_pos, direc, turn):
                flips.append(new_pos)
```

```python
                new_pos = copy.deepcopy(new_pos)
                direc(new_pos)
        return flips

    def is_dir_valid(self, cur_pos, direc, turn):
        new_pos = copy.deepcopy(cur_pos)
        while new_pos.is_valid() and self.get_token(new_pos) ==
             opposite(turn):
            if not direc(new_pos):
                break
        return new_pos.is_valid() and new_pos != cur_pos and
             self.get_token(new_pos) == turn

    def get_empty_squares(self):
        empty_positions = []
        for v, row in enumerate(self.squares):
            for h, square in enumerate(row):
                if self.squares[v][h] == 0:
                    pos = p.Pos(h, v)
                    empty_positions.append(pos)
        return empty_positions

    def get_score(self):
        black_count = 0
        white_count = 0
        for row in self.squares:
            for h in row:
                if h == BLACK:
                    black_count += 1
                elif h == WHITE:
                    white_count += 1
        return{'Black': black_count, 'White': white_count}
```

6.e Positions on the board

```python
# COMP 6321 Machine Learning, Fall 2016
# Federico O'Reilly Regueiro - 40012304
# Final project - othello with neural nets

import board as b


class Pos:
    def __init__(self, horizontal, vertical):
            self.h = horizontal
            self.v = vertical

    def __eq__(self, other):
        if type(other) is type(self):
            return self.__dict__ == other.__dict__
        return False

    def __ne__(self, other):
        """Define a non-equality test"""
        if isinstance(other, self.__class__):
            return not self.__eq__(other)
        return NotImplemented

    def __hash__(self):
        hash((tuple(self.h), tuple(self.v)))

    @classmethod
    def from_string(cls, string):
        h = ord(string[0]) - ord('a')
        v = ord(string[2]) - ord('1')
        pos = cls(h, v)
        if not pos.is_valid():
            raise ValueError
        return pos

    def is_valid(self):
        return (0 <= self.h < b.BOARD_SIZE and 0 <= self.v <
            b.BOARD_SIZE )

    def to_string(self):
        horizontal = chr(ord('a') + self.h)
        vertical = chr(ord('1') + self.v)
        return horizontal + ', ' + vertical

    @staticmethod
    def down(pos):
        if pos.v < b.BOARD_SIZE-1:
```

```
            pos.v += 1
            return True
        else:
            return False

    @staticmethod
    def right(pos):
        if pos.h < b.BOARD_SIZE-1:
            pos.h += 1
            return True
        else:
            return False

    @staticmethod
    def up(pos):
        if pos.v > 0:
            pos.v -= 1
            return True
        else:
            return False

    @staticmethod
    def left(pos):
        if pos.h > 0:
            pos.h -= 1
            return True
        else:
            return False

    @staticmethod
    def down_right(pos):
        if (pos.h < b.BOARD_SIZE-1) and (pos.v < b.BOARD_SIZE-1):
            pos.h += 1
            pos.v += 1
            return True
        else:
            return False

    @staticmethod
    def down_left(pos):
        if (pos.h > 0) and (pos.v < b.BOARD_SIZE-1):
            pos.h -= 1
            pos.v += 1
            return True
        else:
            return False

    @staticmethod
    def up_right(pos):
        if (pos.h < b.BOARD_SIZE-1) and (pos.v > 0):
```

```python
            pos.h += 1
            pos.v -= 1
            return True
        else:
            return False

    @staticmethod
    def up_left(pos):
        if (pos.h > 0) and (pos.v > 0):
            pos.h -= 1
            pos.v -= 1
            return True
        else:
            return False
```

References

[1] Christopher Burger. Google deepmind's alphago: How it works, 2016.

[2] Kumar Chellapilla and David B Fogel. Evolution, neural networks, games, and intelligence. *Proceedings of the IEEE*, 87(9):1471–1496, 1999.

[3] Siang Y. Chong, Mei K. Tan, and Jonathon D. White. Observing the evolution of neural networks learning to play the game of othello. *IEEE Transactions on Evolutionary Computation*, 9:240–251, 2005.

[4] Tambet Matiisen. Demystifying deep reinforcement learning, 2015.

[5] David Moriarty and Risto Miikkulainen. Evolving complex othello strategies using marker-based genetic encoding of neural networks. Technical report, 1993.

[6] Andrew Ng. Lecture 16 - applications of reinforcement learning, April 2009.

[7] Brian Tanner and Richard S. Sutton. Td(Lambda) networks: Temporal-difference networks with eligibility traces. In *Proceedings of the 22Nd International Conference on Machine Learning*, ICML '05, pages 888–895, New York, NY, USA, 2005. ACM.

[8] Gerald Tesauro. Practical issues in temporal difference learning. In *Machine Learning*, pages 257–277, 1992.