Comp 6321 - Machine Learning - Project report addendum

Federico O'Reilly Regueiro

December 20^{th}, 2016

1 Some corrections and observations

At the time of submission there was still a bug in the code that greatly hampered performance and thus there had been little chance of simulating enough matches to see a real trend in the network. The bug has now been fixed by changing the way the gradient application functions are appended to the graph (init function starting at line 123), how these gradient applications are called (line 68) and how the gradients are reset(line 48 in the code in section 2.a. The original problem was that every epoch, an operation was being added to the computation graph, quickly consuming system memory. Now the operation is added only once to the graph and invoked by iterating a list of operation handles.

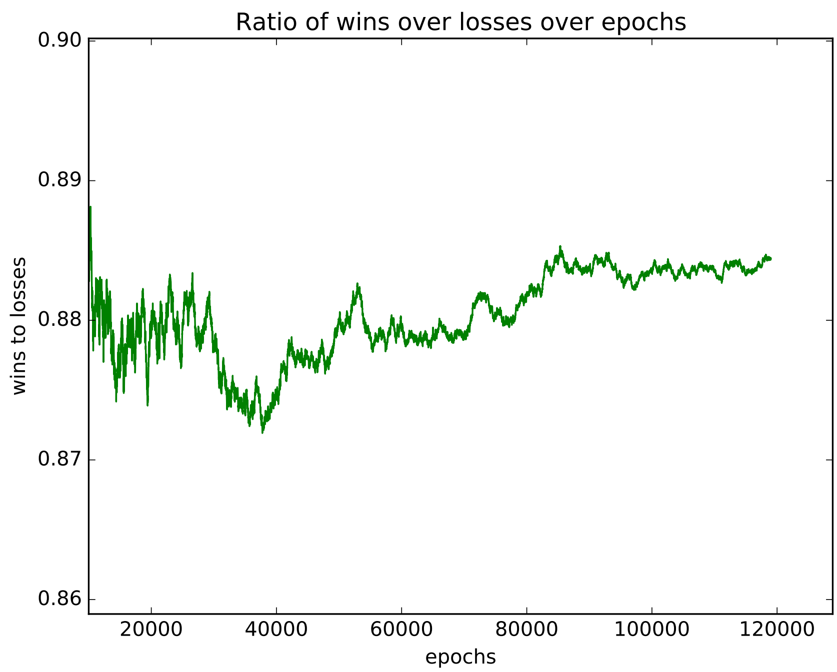

Figure 1: Ratio of wins and losses per epoch, we can see that the ratio is slowly improving but still under the expected ratio 1.0 for two random players.

This has allowed me to run the code for a sufficient number of epochs [4] to start seeing a trend in the network's performance as can be seen in figure **??**. Performance is still far from what I expected and improvement in performance is rather slow. This could be due to a number of reasons. I explore the possible reasons and strategies to be taken to minimize them:

- The network is too big: unlikely since some of the papers [2] [1] [3] report reasonable performance with larger networks.
- The learning rate is too low: possibly, Tesauro [4] reports reasonable performance with a learning rate of 0.1 after 125,000 iterations. The learning rate in the present implementation is one third of that.
- The nature of Othello, where the board configuration and particularly the material count can so quickly change, makes it difficult for the model to converge and it will simply take longer than Backgammon. Leouski and Utgoff [3] indicate similar performance with a NN based on board symmetry and only report better performance with a three-network model where responsibilities for opening, middle and end-game are spread across the three models.

- Either λ or γ are improperly tuned: different λ or γ settings might greatly impact the learning of the model. I have used Tesauro's reported values but given the aforementioned particularities of Othello, training might require different γ or λ

- The model from which we started is inadequate: gradient descent converges towards local minima and the model from which we started might have been close to a shallow local minimum, meaning that the model will never improve much. Try again with a different initial model whose random weights might be closer to a steeper minimum.

- The preprocessing performed, introducing domain-knowledge, is actually of little use to the net. Maybe the choice of rows, columns, 3×3 windows and diagonals is a good human heuristic but harder to learn from for a NN. Try a raw board input or another sort of pre-processing.

1.a Repository

I will most likely continue to work on this code, which has found its home at http://github.com/friketrike/ML-AI-proj-2016 .

2 Appendix i - Code fixes

The memory leak was fixed by avoiding the addition of operations after initialization as described in section 1.

2.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=0.1, 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_train_vars()
            self.initialize_convs()
            self.initialize_ff()
            self.accum_grads = []
            self.discount_factor = 0.9
            self.lambdaa = 0.75
            self.opt = tf.train.GradientDescentOptimizer(3e-2)
            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.initialize_accum_grad_vars()
            self.append_lambda_op()
            self.values_history = []
```

```python
        session.run(tf.initialize_all_variables())

    # TODO if we ever get to do the evolutionary variant, tic toc... time is pressing
    @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.lambda_resets[idx] for idx in range(self.gaccum_list.__len__())]
        self.values_history = []

    # give an appraisal given a board configuration, train if told to do so
    def evaluate(self, boards, session, train=False, verbose=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)
        half_sym, chopped = self.turn_boards(boards)
        if train:
            v, grad_var_list = session.run([self.h_out, self.grad_var_list],
                                           feed_dict={self.boards_half_sym: half_sym,
                                                      self.boards_chopped: chopped,
                                                      self.boards_diag: diag,
                                                      self.keep_prob: 0.7,
                                                      self.batch_size: batch_size}) # self.keep_prob: 1,
            idx = np.argmax(v, 0)
            self.values_history.append(v[idx])
            _ = [self.lambda_updates[idx] for idx in range(self.gaccum_list.__len__())]
        else:
            v = session.run(self.h_out, feed_dict={self.boards_half_sym: half_sym,
                                                   self.boards_chopped: chopped,
                                                   self.boards_diag: diag,
                                                   self.keep_prob: 1,
                                                   self.batch_size: batch_size}) #self.keep_prob: 1,
            idx = np.argmax(v, 0)
```

```
        if verbose:
            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, session, verbose=False):
        error = self.values_history[-1] - np.tanh(tally/32)
        if verbose:
            print('Loss at end-game: ', loss)
        for idx, vars in enumerate(self.vars_list):
            session.run(self.gradient_applications[idx], feed_dict={self.error: error})

    # 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_accum_grad_vars(self):
        self.gaccum_list = []
        for gv in self.grad_var_list:
            g = tf.Variable(tf.zeros(gv[0].get_shape()))
            self.gaccum_list.append(g)
```

```
    def append_lambda_op(self):
        self.lambda_updates = []
        self.lambda_resets = []
        self.gradient_applications = []
        for idx, gv in enumerate(self.grad_var_list):
            self.lambda_updates.append(
                tf.add(self.gaccum_list[idx].__mul__(self.lambdaa),
                       gv[0]))
            self.lambda_resets.append(self.gaccum_list[idx].__mul__(0))
            self.gradient_applications.append(
                self.opt.apply_gradients(
                    [(self.gaccum_list[idx].__mul__(self.error), gv[1])]))

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

    def initialize_board_placeholders(self):
        self.boards_half_sym = tf.placeholder(tf.float32, shape=[None,
            8, 8, 4])
        self.boards_chopped = tf.placeholder(tf.float32, shape=[None, 5,
            5, 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.error = tf.placeholder(tf.float32)

    @staticmethod
    def turn_boards(boards):
        boards = np.asarray(boards)
        sym1 = boards[:,::-1,::]
        sym2 = boards[:,:,::-1]
        sym3 = sym2[:,::-1, ::]
        sym4 = np.transpose(boards, (0, 2, 1))
        sym5 = np.transpose(sym1, (0, 2, 1))
        sym6 = np.transpose(sym2, (0, 2, 1))
        sym7 = np.transpose(sym3, (0, 2, 1))
        full_sym = np.stack([boards, sym1, sym2, sym3, sym4, sym5, sym6,
            sym7], axis=3)
        half_sym = full_sym[:, :, :, 0:4]
        chopped = full_sym[:, 0:5, 0:5, :]
        return half_sym, chopped

    @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)
```

2.b interface

The code was modified in order to accomodate the net playing black or white with no change in its reading of the boards configuration (eg. lines 26, 32 and 64). Printing messages has been suppressed unless calling the interface with a boolean setting 'verbose' (line 31) which is false by default.

```
1 # COMP 6321 Machine Learning, Fall 2016
2 # Federico O'Reilly Regueiro - 40012304
3 # Final project - othello with neural nets
4
5 import othello as o
6 import position as p
7 import board as b
8 import tensorflow as tf
9 import othelloNetV2 as otnet
10 import time
11 import random
12
13 session = tf.Session()
14 game = o.game()
15 on = otnet.othello_net(session)
16 score_series = []
17
18
19 def batch(color):
20     board_now = game.board
21     possible_moves = board_now.get_valid_moves(color)
22     boards = []
23     for m in possible_moves:
24         new_board = b.Board(board_now)
25         new_board.do_move(m, color)
26         new_board.relativize(color)
27         boards.append(new_board.squares)
28     return boards, possible_moves
29
30
31 def play_net(train=False, verbose=False):
32     color = random.choice((b.BLACK, b.WHITE))
33     done = False
34     tic = time.time()
35     global score_series
36     if verbose:
37         print(game.board.to_string())
38     if color == b.WHITE:
39         done = not game.play_random_turn(b.opposite(color), verbose)
40     while not done:
41         boards, moves = batch(color)
42         if boards:
43             if verbose:
```

```
                print(game.turn_to_string(color), ' plays:')
            idx, v = on.evaluate(boards, session, train)
            game.play_move(moves[idx], color)
            if verbose:
                print(game.board.to_string())
        else:
            game.pass_moves += 1
        if verbose:
            print(game.turn_to_string(b.opposite(color)), '\'s turn:')
        done = not game.play_random_turn(b.opposite(color), verbose)
    outcome = game.board.get_score()
    color_blind_outcome = {'net': outcome['Black'], 'opponent': outcome['White']}
    if train:
        score_series.append(color_blind_outcome)
        on.learn_from_outcome(color_blind_outcome['net'] - color_blind_outcome['opponent'], session)
    game.reset()
    on.reset_for_game()
    if verbose:
        print('The round took:', time.time()-tic, ' seconds.')
        # print('Up to now: ', wins, ' wins, ', losses, ' losses and ', ties, ' ties.')
    return color_blind_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.")
```

2.c Driver for training the model, storing the model and a history of scores from simulated matches

The script facilitates training the model while saving both the model and the history of scores every 1000 epochs.

```python
import pickle
import time
import othello_interface_v2 as oi2
import os.path

ckpt_fname = 'otnet_v2.ckpt'
pckl_fname = 'net_other.pckl'

if os.path.isfile('otnet_v2.ckpt'):
    oi2.restore_checkpoint()
    if os.path.isfile(pckl_fname):
        f = open(pckl_fname, 'rb')
        x = pickle.load(f)
        f.close()
        oi2.score_series = x

for batch in range(200):
    tic = time.time()
    for _ in range(1000):
        oi2.play_net(True)
    print('batch ', batch, ' took ', (time.time()-tic)/60, ' minutes')
    tl = time.localtime()
    print('finished at: ', tl.tm_hour, ':',tl.tm_min)
    oi2.save_checkpoint()
    oi2.save_checkpoint('SavedModels/otnetv2'+str(oi2.score_series.__len__())+'.ckpt')
    f = open(pckl_fname, 'wb')
    pickle.dump(oi2.score_series, f)
    f.close()
```

2.d Displaying results

Simple routine for loading and displaying results in another python interpreter while the model is still training.

```python
import matplotlib.pyplot as plt
import numpy as np
import pickle
import os


pckl_fname = 'net_other.pckl'


def moving_average(a, n=3):
    ret = np.cumsum(a, dtype=float)
    ret[n:] = ret[n:] - ret[:-n]
    return ret[n-1:]/n


def load_and_display():
    if not os.path.isfile(pckl_fname):
        raise FileNotFoundError
    f = open(pckl_fname, 'rb')
    x = pickle.load(f)
    f.close()
    n = []
    o = []
    for sc in x:
        n.append(sc['net'])
        o.append(sc['opponent'])

    n = np.asarray(n)
    o = np.asarray(o)

    window_length = n.__len__() / 10
    mav = moving_average((n-o), window_length)
    plt.plot(mav, 'b', np.zeros(mav.__len__()), 'k--')
    plt.title('scores-moving average, rectangular window, n='+str(window_length))
    plt.xlabel('epochs')
    plt.ylabel('score moving average')
    plt.show()

    mav2 = moving_average((n-o) > 0, window_length)
    plt.plot(mav2, 'b', 0.5 * np.ones(mav2.__len__()), 'k--')
    plt.title('games won moving average, rectangular window, n='+str(window_length))
    plt.xlabel('epochs')
    plt.ylabel('wins moving average')
```

```
    plt.show()

    # plt.plot(np.cumsum((n-o) > 0), 'g', np.cumsum((n-o) == 0), 'b',
        np.cumsum((n-o) < 0), 'r')
    # plt.legend(['wins', 'ties', 'losses'], loc='upper left')
    # plt.title('Progression of outcomes over epochs')
    # plt.xlabel('epochs')
    # plt.ylabel('accumulated outcomes')
    # plt.show()

    plt.plot(np.divide((np.cumsum((n - o) > 0))+1, np.cumsum((n-o) <
        0)+1), 'g', np.ones(n.__len__()), 'k--')
    plt.ylim([0.85, 1.02])
    plt.title('Ratio of wins over losses over epochs')
    plt.xlabel('epochs')
    plt.ylabel('wins to losses')
    plt.show()
```

References

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

[2] 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.

[3] Anton V. Leouski and Paul E. Utgoff. What a neural network can learn about othello, 1996.

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