CS-362 HW 4:

Jacob Hurst

April 30, 2019

1 Introduction

Nim is a game in which two players alternately take one or more objects from one of a number of heaps, each trying to take, or to compel the other to take, the last remaining object.

2 Discussion

The model for input and outputs I selected for my neural network matches that of my initial network, 18 inputs representing 3, 6 bit, pile sizes in binary and 9 outputs, 3 denoting the selected pile and 6 denoting the remaining pile size in binary. labeled training data was generated all at once at the start of the program by generating the set of pile configurations then evaluating each and storing the evaluation (see strategy and xor_solver functions). I first trained the network on a smaller training set (max pile size of 8) and then ramped my way up to piles of size 64. The training of the network on the 64^3 training data was a little slow so I opted to focus on training the network with a max pile size of 32. The current version for this submission reflects that decision. I also added functionalities to my network to enable autosaving and autoloading training progress using cPickle.

In developing the network, I experimented a lot with various batch sizes and learning rates. I found that smaller batches of my training data and a learning rate close to 0.01 resulted in a faster convergence of error to 0.25. After some time, most of the training variants I had completed appeared to converge to this error of 0.25. Further training from this point appeared to offer insignificant improvements or worse, would result in an increase in error - possibly from overfitting.

When playing against the trained neural network, optimal opening moves are chosen relatively consistently. I noticed the network consistently balances piles - reflecting optimal strategy - but then may struggle slightly to carry on from there. The network takes wins relatively consistently when it is presented with opportunities.

3 Read Me

```
# CS-362 HW4: Neural Network Nim Player
Nim is a mathematical game of strategy in which two players take turns removing objects from piles. The
    ↪  player that empties the last pile is declared the winner.
This neural network has been trained to play Nim. The network currently plays at 75% accuracy.

## Getting Started
* Code for the project can be found in the 'code' subdirectory.
* Logs and figures for the project can be found in the 'log' subdirectory.
* Discussion offers further insights on this project

## Usage
To run, 'cd' into the 'code' subdirectory and run 'python nim_network.py'.

The code will automatically load 'nim_network.pkl' which contains the training data.

User will be prompted to train or play a game.
To train the network, type 'T' (to save and quit training progress 'Ctrl-C', to quit training progress
    ↪ 'Ctrl-Z').
To play a game, type 'P'. Then provide the starting pile sizes 'size_1, size_2, size_3'. The game will
    ↪ open on the computers turn and alternate thereafter.

### Author
Jacob Hurst
Jhurst@cs.unm.edu
```

4 Sample Output After 512 Small Batch Iterations

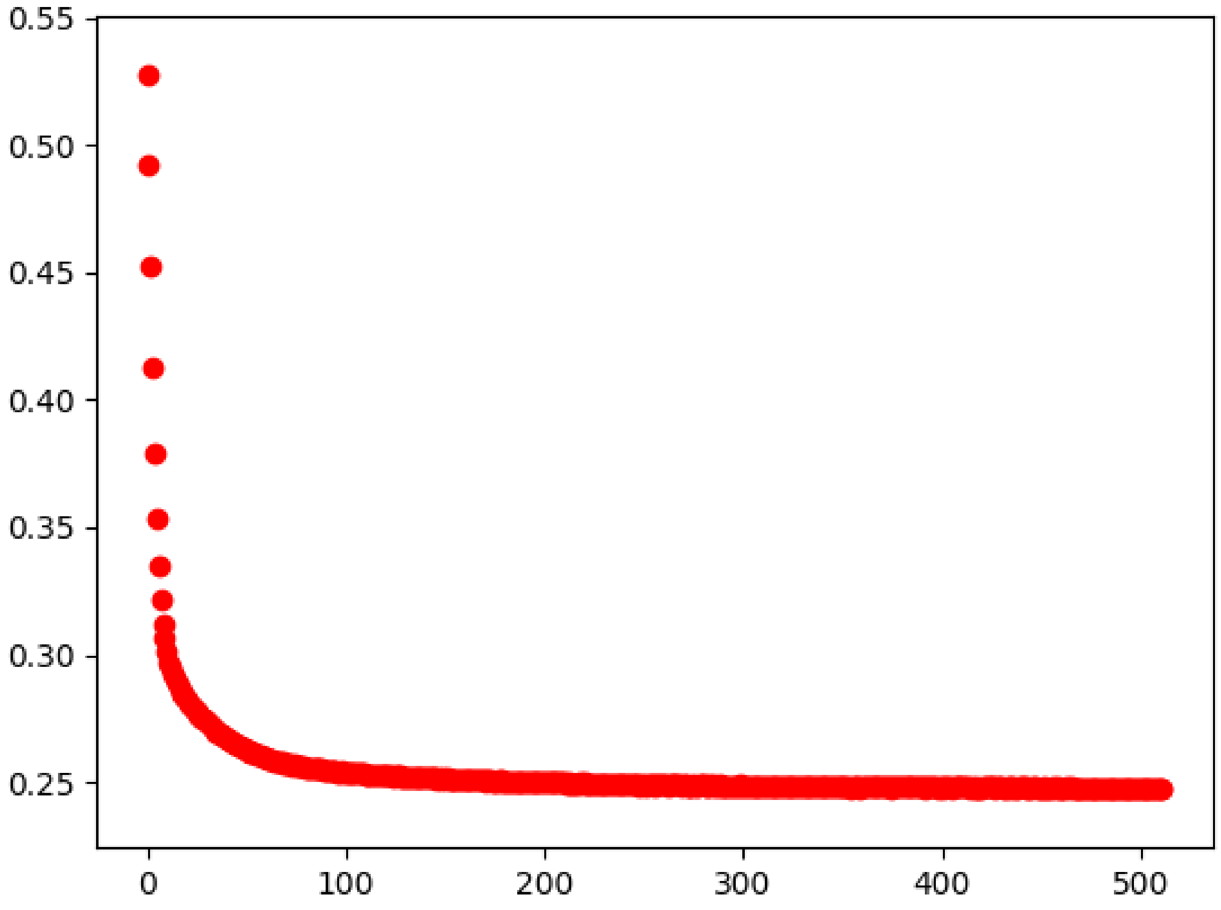

Figure 1: Convergence of small batch neural network after 512 iterations.

```
Training loaded.
Would you like to train (T) or play (P)? P
Input 3 pile dimensions: 15,15,14
Pile dimensions: 15, 15, 14.
***************************
Evaluation:
[[0.14196746]
 [0.13339138]
 [0.71609746]
 [0.00455191]
 [0.15150659]
 [0.4137397 ]
 [0.40730321]
 [0.45356518]
 [0.4943386 ]]
Selection: 2
Remaining: 0
Strategy: (take, selection) (14, 2)
***************************
Computer took 14 from pile 2!
```

```
Pile dimensions: 15, 15, 0.
Human turn | provide your move 'amount,index': 15,0
Human took 15 from pile 0!
Pile dimensions: 0, 15, 0.
***************************
Evaluation:
[[0.10347911]
 [0.8555042 ]
 [0.04695787]
 [0.01599541]
 [0.19629058]
 [0.28857622]
 [0.45285667]
 [0.4411775 ]
 [0.51235731]]
Selection: 1
Remaining: 0
Strategy: (take, selection) (15, 1)
***************************
Computer took 15 from pile 1!
Computer won!
```

5 Code

```python
import copy
import random
import cPickle
import numpy as np
import matplotlib.pyplot as plt
import signal

debug = True

'''
Class to contain the neural network. Represents weights as matrices.
'''
class NeuralNetwork(object):
    '''
    Default construct, takes dimensions and initializes values to random.
    Attempts to load any training data from file.
    '''
    def __init__(self, dimensions):
        self.num_layers = len(dimensions)
        self.dimensions = dimensions
        self.biases = [np.random.randn(y, 1) for y in dimensions[1:]]
        self.weights = [np.random.randn(y, x) for x, y in zip(dimensions[:-1], dimensions[1:])]
        self.errors = []

        signal.signal(signal.SIGINT, self.signal_handler)
        try:
            self.load()
            print("Training␣loaded.")
        except:
            print("Failed␣to␣load␣training␣data,␣training␣required.")

    '''
    Method to train the network. Defaults to 128 iterations, 32 batch size, and a learning rate of
        ↪ 0.01.
    Prints progress at the end of each iteration. Progress is mean of the mean error among the
        ↪ activations.
    '''
    def train(self, training_data, iterations=128, batch_size=32, learning_rate=0.001):
        n = len(training_data)
        for j in range(iterations):
            random.shuffle(training_data)
            mini_batches = [training_data[k:k+batch_size] for k in range(0, n, batch_size)]
            total = 0.0
            count = 0
            for mini_batch in mini_batches:
                total += self.back_propogation(mini_batch, learning_rate)
                count += 1
            self.errors.append(np.mean(total/count))
            print("Error␣=␣" + str(np.mean(total/count)) + ".")
            print("Iteration␣" + str(len(self.errors)) + "␣complete.")
        print("Training␣saved.")
        self.save()

    '''
    Method to evaluate inputs, propogates forward the given value.
    '''
    def forward_propogation(self, a):
        for b, w in zip(self.biases, self.weights):
            a = self.sigmoid(np.dot(w, a)+b)
        return a

    '''
    Method to calculate the gradient and train the network based on that gradient.
    '''
    def back_propogation(self, mini_batch, learning_rate):
        gradient_b = [np.zeros(b.shape) for b in self.biases]
        gradient_w = [np.zeros(w.shape) for w in self.weights]
        count, total = 0, 0.0
        for x, y in mini_batch:
            delta_gradient_b, delta_gradient_w, activations = self.back_pass(x, y)
            gradient_b = [gb+dgb for gb, dgb in zip(gradient_b, delta_gradient_b)]
            gradient_w = [gw+dgw for gw, dgw in zip(gradient_w, delta_gradient_w)]
            total += self.cost(activations[-1], y)
            count += 1

        self.weights = [w-(learning_rate/len(mini_batch))*gw for w, gw in zip(self.weights, gradient_w)]
        self.biases = [b-(learning_rate/len(mini_batch))*gb for b, gb in zip(self.biases, gradient_b)]
        return total / count

    '''
    Helper for back_propogation, calculates and returns the gradient.
    '''
    def back_pass(self, x, y):
        gradient_b = [np.zeros(b.shape) for b in self.biases]
        gradient_w = [np.zeros(w.shape) for w in self.weights]

        activation = x
        activations = [x]
        zs = []
        for b, w in zip(self.biases, self.weights):
            z = np.dot(w, activation)+b
            zs.append(z)
            activation = self.sigmoid(z)
            activations.append(activation)
        delta = self.cost_derivative(activations[-1], y) * self.sigmoid_derivative(zs[-1])
        gradient_b[-1] = delta
        gradient_w[-1] = np.dot(delta, activations[-2].transpose())

        for l in range(2, self.num_layers):
            z = zs[-l]
            sp = self.sigmoid_derivative(z)
            delta = np.dot(self.weights[-l+1].transpose(), delta) * sp
            gradient_b[-l] = delta
            gradient_w[-l] = np.dot(delta, activations[-l-1].transpose())
        return (gradient_b, gradient_w, activations)

    '''
    Square loss cost function.
    '''
    def cost(self, activations, y):
        return (activations[-1] - y)**2

    '''
    Derivative square loss cost function.
    '''
    def cost_derivative(self, activations, y):
        return 2*(activations-y)

    '''
    Sigmoid function.
    '''
    def sigmoid(self, z):
        return 1.0/(1.0+np.exp(-z))

    '''
    Derivative sigmoid function.
    '''
    def sigmoid_derivative(self, z):
        return self.sigmoid(z)*(1-self.sigmoid(z))

    '''
    Method to save the training progress.
    '''
    def save(self, filename='nim_network.pkl'):
        with open(filename, 'w') as fd:
            network_state = {
                "num_layers":self.num_layers,
                "dimensions":self.dimensions,
                "weights":self.weights,
                "biases":self.biases,
                "errors":self.errors
            }
            cPickle.dump(network_state, fd, 2)

    '''
    Method to load the training progress from file.
    '''
    def load(self, filename='nim_network.pkl'):
        with open(filename, 'r') as fd:
            network_state = cPickle.load(fd)
            self.num_layers = network_state["num_layers"]
            self.dimensions = network_state["dimensions"]
            self.weights = network_state["weights"]
            self.biases = network_state["biases"]
            self.errors = network_state["errors"]
        return self

    '''
    Method to plot the error versus iterations of training.
    '''
    def plot(self):
        plt.scatter(range(len(self.errors)), self.errors, color='red', marker='o')
        plt.show()

    '''
    Method to handle Ctrl-C exit, saves training.
    '''
    def signal_handler(self, signal, frame):
        print('\nTraining saved.')
        self.save()
        exit(0)

'''
Class to hold the Nim game as well as the network.
'''
class Nim(object):
    '''
    Default constructor, initializes network and prompts user to train or play.
    '''
    def __init__(self):
        self.nim_network = NeuralNetwork([18, 14, 9])

        select = raw_input("Would you like to train (T) or play (P)? ")
        if select == 'T' or select == 't':
            training_data = self.XOR_solver(0, 32)
            self.nim_network.train(training_data)
            self.nim_network.plot()
        else:
            piles = [int(x) for x in input("Input 3 pile dimensions: ")]
            self.play(piles)

    '''
    Method to handle game playing logic.
    '''
    def play(self, piles):
        turn = False

        while(piles[0] > 0 or piles[1] > 0 or piles[2] > 0):
            print("Pile dimensions: "+str(piles[0])+", "+str(piles[1])+", "+str(piles[2])+".")

            if(turn):
                move = [int(x) for x in input("Human turn | provide your move \'amount,index\': ")]
                print("Human took "+str(move[0])+" from pile "+str(move[1])+"!")
                piles[move[1]] = piles[move[1]]-move[0]
            else:
                state = self.to_bin(piles[0])
                state += self.to_bin(piles[1])
                state += self.to_bin(piles[2])
                move = self.nim_network.forward_propogation(np.array(state).reshape((18,1)))
                selection = self.which(move[:3])
                remaining = self.from_bin(move[3:])
                if debug:
                    print("***************************")
                print("Evaluation:␣\n" + str(move))
                print("Selection:␣" + str(selection))
                print("Remaining:␣" + str(remaining))
                print("Strategy:␣(take,␣selection)␣" + str(self.strategy({0:piles[0], 1:piles[1], 2:
                    ↪ piles[2]})))
                print("**************************")
            print("Computer␣took␣"+str(piles[selection]-remaining)+"␣from␣pile␣"+str(selection)+"!")
            piles[selection] = remaining

        turn = not turn

    if(turn): print("Computer␣won!")
    else: print("Human␣won!")

'''
Helper for strategy method, returns binary string representation of z.
'''
def binstr(self, n):
    return "".join([str((n >> y) & 1) for y in range(23, -1, -1)])

'''
Method to calculate binary representation of pile as a list.
'''
def to_bin(self, pile):
    binstr = "{0:06b}".format(pile)
    return [int(x) for x in binstr]

'''
Method to convert list into binary representation of network evaluation.
'''
def from_bin(self, pile):
    threshold = 0.75
    amount, power = 0, 0
    for bit in reversed(pile):
        if(bit >= threshold): bit = 1
        else: bit = 0
        amount += bit*(2**power)
        power += 1
    return amount

'''
Method to convert networks position analysis into an int, returns index of most activated neuron.
'''
def which(self, selection):
    max_pile, max_index = -1, -1
    for i in range(len(selection)):
        if(selection[i] > max_pile):
            max_pile = selection[i]
            max_index = i
    return max_index

'''
Method to generate training data on pile sizes between lower and upper.
'''
def XOR_solver(self, lower=0, upper=64):
    training_data = []
    for i in range(lower, upper):
        for j in range(lower, upper):
            for k in range(lower, upper):
                if(i == j and j == k and k == 0): continue
```

```
                    state = {0:i,1:j,2:k}
                    s = self.strategy(state)

                    if s[0] == -1: continue
                    elif s[1] == 0: selection = [1,0,0]
                    elif s[1] == 1: selection = [0,1,0]
                    elif s[1] == 2: selection = [0,0,1]
                    remaining = state[s[1]] - s[0]

                    inputs = self.to_bin(i)
                    inputs += self.to_bin(j)
                    inputs += self.to_bin(k)
                    targets = selection
                    targets += self.to_bin(remaining)

                    training_data.append((np.reshape(np.array(inputs), (18,1)), np.reshape(np.array(
                        ↪ targets), (9,1))))
        return training_data

    '''
    Helper for generating training data, takes a game state and returns the optimal move.
    '''
    def strategy(self, state):
        state = copy.deepcopy(state)
        z = 0
        for pile in state.values():
            z = z ^ pile
        if z == 0: return (-1, -1)
        else:
            z_bin = self.binstr(z)
            z_bin_str = str(z_bin).lstrip('0')
            msb = len(z_bin_str)
            upper = 2**(msb-1)
            which = ''
            remove = -1
            while remove < 1:
                for pile in state.keys():
                    if state[pile] >= upper:
                        which = pile
                        pile_new = z ^ int(state[which])
                remove = state[which]-pile_new
                del state[which]
            return (remove, which)

nim = Nim()
```