CPSC 371 Vincent Tennant

Artificial Intelligence 230099844

Phase 2

Mapping of Rubix to Neural Network Input layer

The training for the rubix cube is encoded in a way that easily allows for the cube state to be input into the neural network. Since my original training data for a 2x2x2 rubix cube was incorrect some modification was required. The training data consists for 80 lines with each line representing a cube state, and the next optimal move. Below is an example of one such line:

```
-1.0 1.0 -1.0 1.0 -1.0 1.0 -1.0 -1.0 -1.0 1.0 -1.0 -1.0 -1.0 1.0 1.0 -1.0 -1.0 1.0 -1.0 -1.0 1.0 -1.0
1.0 1.0 -1.0 -1.0 -1.0 1.0 -1.0 -1.0 1.0 -1.0 1.0 -1.0 1.0 -1.0 -1.0 1.0 -1.0 1.0 -1.0 1.0 -1.0 -1.0
-1.0 1.0 -1.0 -1.0 -1.0 1.0 1.0 -1.0 -1.0 1.0 -1.0 -1.0 1.0 -1.0 1.0 1.0 -1.0 -1.0 -1.0 1.0 -1.0 -1.0
1.0 -1.0 1.0 -1.0 1.0 -1.0 -1.0 -1.0 -1.0 1.0
```

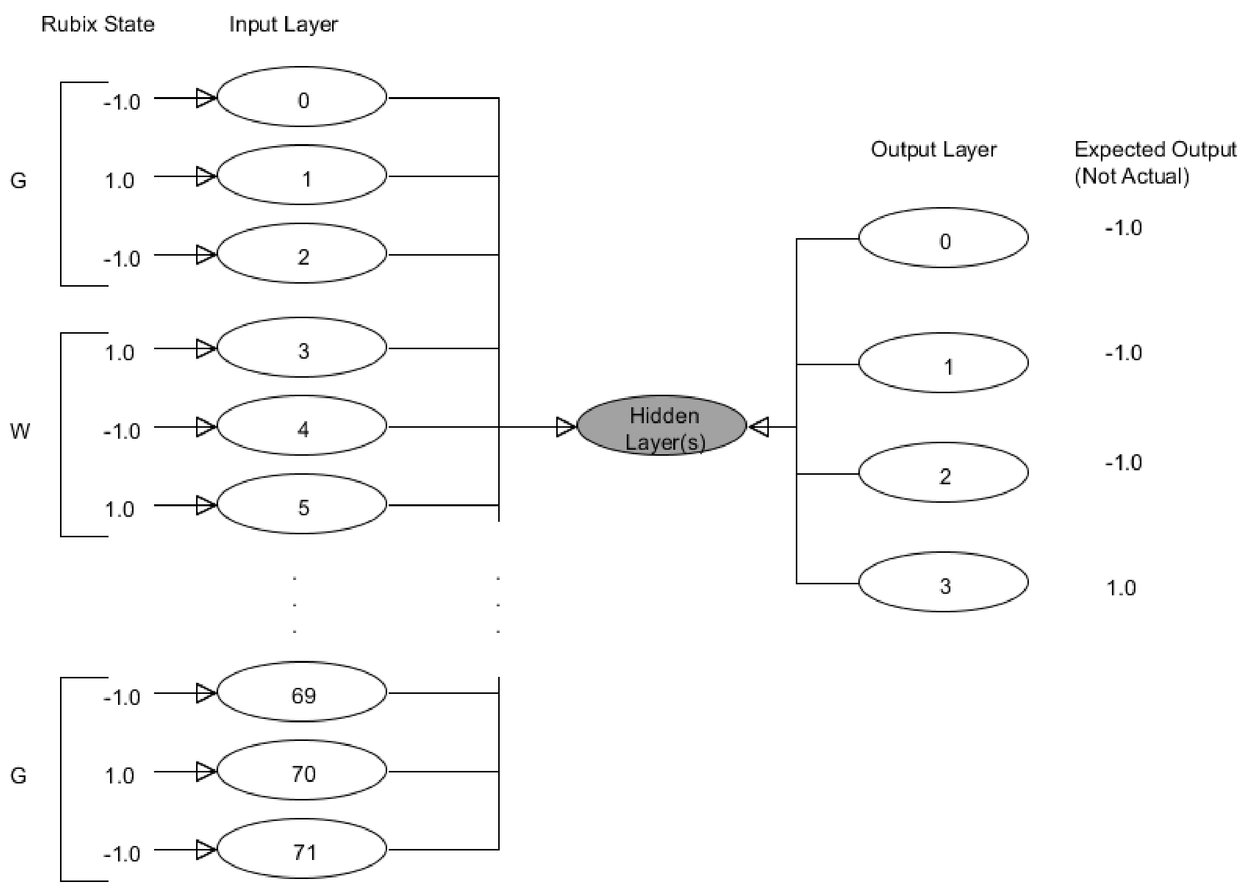
While quite long, each number has a purpose. The state of the cube is encoded as a series of 1's and -1's. Before being translated into these series of 1's and 0's the cube is represented as a simple string. Below is the string representation of the same cube state as above as well as a table showing which colours belong to which encoding.

GWROYBBYROWGGWROYBBYROWG

R:	-1.0 -1.0 -1.0
B:	-1.0 -1.0 1.0
G:	-1.0 1.0 -1.0
Y:	-1.0 1.0 1.0
O:	1.0 -1.0 -1.0
W:	1.0 -1.0 1.0

Note that the last four values in the training data are considered the optimal move to be taken, given this specific state.

In order for a neural network to accept the training data for a rubix cube there must be 72 input neurons. Since each colour in the cubes state is encoded as three values, and there are 24 total coloured squares on a 2x2x2 cube. This means that each coloured square on the rubix rube requires 3 total input neurons.



The last four number in the training data represent the next move to be taken. With the axis to rotate about encoded as 2 bits, or two values. The direction, clockwise or counter-clockwise, encoded as 1 and -1 respectively. Finally the slice to rotate. Since a 2x2x2 cube only has two slices to rotate on any given axis slices 0, and 1 and encoded as to values, -1 and 1 respectively.

The Program

The main class presents the gui and allows the user to perform the unit tests, as well as perform the XOR problem and run an example rubix cube problem. Neural network class is the main portion for the neural networks. The feed forward operation, and the save and load operations. Each layer is a simple layer object, with the exception of the hidden layers being an array of layers, to allow for multiple hidden layers. The layers class is used to represent the layers within the neural network. Layers class contains operations for creating the synapses between layers, as well as changing the neuron values themselves. The neurons for each layer are stored within a simple array, with the neurons being arranged in the same order they are arranged in the array. The synapses are stored in a 2D array, with the first dimension corresponding to the neuron that the the synapse is going from, and the second dimension containing the synapse weight for the corresponding synapse.

Stochastic Back-propagation and the Back-propagation interface are strongly related. The back-propagation interface is used to allow for multiple classes to use stochastic back-propagation with coupling themselves to the algorithm. Stochastic back-propagation itself trains neural networks, rather any object that implements the interface. Stochastic back-propagation is a faily simple class, with the SBP method being the backbone of the class. All other methods are there to help SBP perform its training. An array list is used to contain the current tuple being used for the training data, this tuple contains the appropriate input to be used, as well as the expected output. Other arrays are used to contain information like Dwkj, the weight updates for the hidden to output synapses, Dwji, the weight updates for the input to hidden synapses, and the error at the hidden neurons.

XOR Results

The results of the XOR problem were satisfactory. Given 20000 training iterations, learning rate of 0.05, and the standard XOR neural network topology of 2-2-1 the problem was solvable. An issue encounter during the training of the XOR neural network was the total error of the neural network could be well below the error threshold, but the actual output would not be correct. When given enough training though the network was easily trained.

Problems Encountered

Ensuring all of the math was correct throughout stochastic back-propagation proved to be tricky. Double and triple checking each function was beneficial. As well, reorganizing and rewriting the training data was time consuming. The heat maps were difficult to figure out how to create, as well as time consuming.

Heat Maps

Tuples chosen are formatted below as (Hidden Layer Size, Number of Training Iterations, Learning Rate). The top 5 tuples chosen are as follows:

Tuple 1: (5, 30000, 0.05)

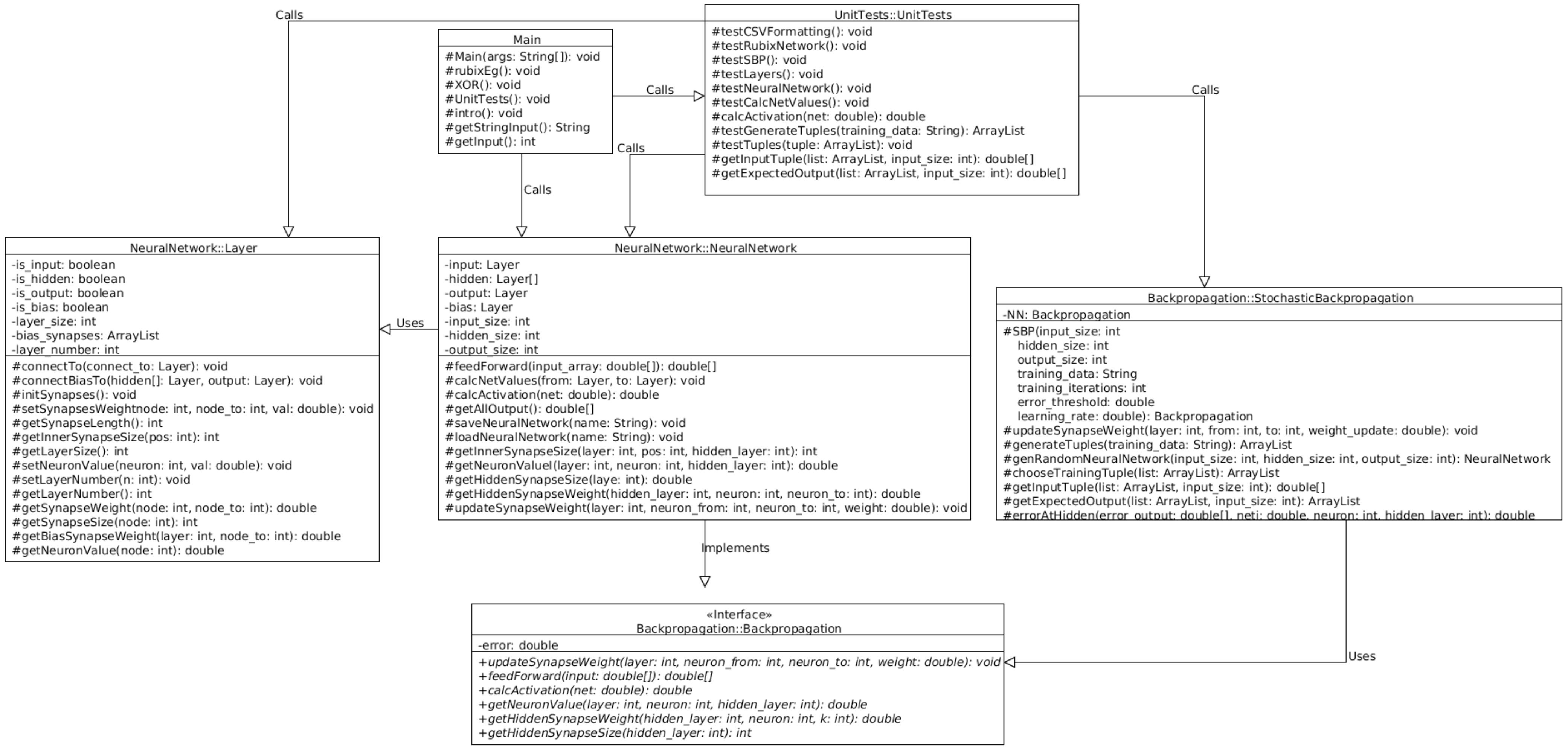
Tuple 2: (6, 25000. 0.05)

Tuple 3: (7, 25000, 0.05)

Tuple 4: (8, 30000, 0.02)

Tuple 5: (9, 30000, 0.02)

All with errors of 0.0. due to the size and organization of the heat maps they can be found in the heat map directory.


UnitTests::UnitTests
#testCSVFormatting(): void
#testRubixNetwork(): void
#testSBP(): void
#testLayers(): void
#testNeuralNetwork(): void
#testCalcNetValues(): void
#calcActivation(net: double): double
#testGenerateTuples(training_data: String): ArrayList
#testTuples(tuple: ArrayList): void
#getInputTuple(list: ArrayList, input_size: int): double[]
#getExpectedOutput(list: ArrayList, input_size: int): double[]
Main
#Main(args: String[]): void
#rubixEg(): void
#XOR(): void
#UnitTests(): void
#intro(): void
#getStringInput(): String
#getInput(): int
Calls
Calls
Calls
Calls
Calls
NeuralNetwork::Layer
-is_input: boolean
-is_hidden: boolean
-is_output: boolean
-is_bias: boolean
-layer_size: int
-bias_synapses: ArrayList
-layer_number: int
#connectTo(connect_to: Layer): void
#connectBiasTo(hidden[]: Layer, output: Layer): void
#initSynapses(): void
#setSynapsesWeightnode: int, node_to: int, val: double): void
#getSynapseLength(): int
#getInnerSynapseSize(pos: int): int
#getLayerSize(): int
#setNeuronValue(neuron: int, val: double): void
#setLayerNumber(n: int): void
#getLayerNumber(): int
#getSynapseWeight(node: int, node_to: int): double
#getSynapseSize(node: int): int
#getBiasSynapseWeight(layer: int, node_to: int): double
#getNeuronValue(node: int): double
Uses
NeuralNetwork::NeuralNetwork
-input: Layer
-hidden: Layer[]
-output: Layer
-bias: Layer
-input_size: int
-hidden_size: int
-output_size: int
#feedForward(input_array: double[]): double[]
#calcNetValues(from: Layer, to: Layer): void
#calcActivation(net: double): double
#getAllOutput(): double[]
#saveNeuralNetwork(name: String): void
#loadNeuralNetwork(name: String): void
#getInnerSynapseSize(layer: int, pos: int, hidden_layer: int): int
#getNeuronValuel(layer: int, neuron: int, hidden_layer: int): double
#getHiddenSynapseSize(laye: int): double
#getHiddenSynapseWeight(hidden_layer: int, neuron: int, neuron_to: int): double
#updateSynapseWeight(layer: int, neuron_from: int, neuron_to: int, weight: double): void
Backpropagation::StochasticBackpropagation
-NN: Backpropagation
#SBP(input_size: int
hidden_size: int
output_size: int
training_data: String
training_iterations: int
error_threshold: double
learning_rate: double): Backpropagation
#updateSynapseWeight(layer: int, from: int, to: int, weight_update: double): void
#generateTuples(training_data: String): ArrayList
#genRandomNeuralNetwork(input_size: int, hidden_size: int, output_size: int): NeuralNetwork
#chooseTrainingTuple(list: ArrayList): ArrayList
#getInputTuple(list: ArrayList, input_size: int): double[]
#getExpectedOutput(list: ArrayList, input_size: int): ArrayList
#errorAtHidden(error_output: double[], neti: double, neuron: int, hidden_layer: int): double
Implements
«Interface»
Backpropagation::Backpropagation
-error: double
+updateSynapseWeight(layer: int, neuron_from: int, neuron_to: int, weight: double): void
+feedForward(input: double[]): double[]
+calcActivation(net: double): double
+getNeuronValue(layer: int, neuron: int, hidden_layer: int): double
+getHiddenSynapseWeight(hidden_layer: int, neuron: int, k: int): double
+getHiddenSynapseSize(hidden_layer: int): int
Uses