Evolving Artificial Neural Networks To Play Games

**Preface**

We use Artificial Neural Nets (ANNs) to control a “cart” within a simulated environment. The environment is essentially a game, where the performance of the ANN is scored by how long it can use its control of the cart to balance a pole on top of it. We then use the ANNs with the best scores to make new generations of ANNs and repeat the process in an attempt to evolve the ANNs by way of natural selection.

**Introduction**

An Artificial Neural Net is a collection of nodes, and connections, that are put together to simulate the theoretical workings of neurons found in biological entities. From a programming perspective, an ANN receives a number of inputs from a number of sources that are given different weights, the ANN then decides on an output. In the Simulated environment in this project there are two possible outputs the ANN can output: move to the left or move to the right. For every frame of the “game”, the ANN is given all the parameters of the scene and give an output.

**Our Solution to the Problem – A Brief Description of the Code**

The submitted code consists of four python scripts. Three of them contain a class, the ANN class, DNA class and a GA class. They are described in more detail below and in the code. The last script (run.py) defines all necessary parameters, define the desired class objects and starts the training, simulation and comparison. Wherever applicable, vectorized methods have been preferred to perform the operations needed.

Every artificial neural network in every generation is an object from the ANN class. The GA class at any time holds the current generation of ANNs and has functions to evaluate them and create a new generation of ANNs for as many generations as has been specified. We only need one GA object to complete the task set out in this project.

The reason we choose to split the code in classes is because of reusability. For example, later we might prefer to use a different classifier. In that case all we need to do is change ANN to another classifier class that contains the same required methods and our genetic algorithm would still work without any changes to the code.

**ANN Class**

The ANN class is made as a subclass of the MLPClassifier class.

At initialization, in addition to creating an MLPClassifier object, it is also passed the CartPole environment. The number of hidden layers can be passed as a parameter, or it will be set automatically, according to the function provided in the assignment text. In our implementation, 4 input nodes would give one hidden layer with 4 nodes. When initialized, the ANN also partially trains itself using a random action, and a random state of the CartPole environment. This is to get a starting point, and the correct shape for the weights and biases. The initial state of the environment is returned by env.reset(). Initially all observations are assigned a random value in range [-0.05, 0.05]. This is different from the assignment text suggestions, where a random sample from the observation space was used for partial training.

In addition to storing the weights, biases and MLPClassifier, this class contains a method that calculates the fitness of the classifier, i.e. sets the score gotten while running the environment.

The method for running the simulation will record observations from the environment and predict and perform actions for each timestep until the environment either fails, if the pole falls or the cart hits the boundary, or the maximum amount of iterations/timesteps have passed. The maximum number of iterations is set as default to 10 000. If the render parameter is set to True, the environment is rendered for every timestep. It is set to False as default, because it will take too long to render during the training of the ANNs.

The method also counts repeat actions, and checks whether an action has been repeated more than the desired maximum repetition. If it has, a random action is performed in the next step. The counter is reset if the random action is different from the previous.

If the parameter partial\_fit is set to true, the ANN will do a partial fit at the end of each step, using the current observation of the environment and the predicted action. It is set to False as default, and we did not use this ability. It is briefly discussed in the results section of the report.

At the end, a reward property is given to the ANN object, which is set to +1 for every timestep completed before the simulation ended.

**GA Class**

The GA class was made to initialize and hold a generation of ANNs, run the simulation for the environment for all of them, and then generate a new generation of ANNs based on the results. The environment and needed parameters are held as properties for the objects. The first generation is initialized and trained. The population of ANNs at any generation is held in the property self.population.

After initializing a GA object, the run() method is used to start the process. The best agent from the generation is stored, fitness of the current/first generation is printed and stored. A new generation is created with parents chosen with probabilities based on the ANNs rewards. Again, the best agent from the generation is stored, fitness of the current/first generation is printed and stored. This is repeated for the specified number of generations. There is an if statement that is in place that can be commented that will stop the running when an ANN has reached the max iterations and then also a reward equal to the max iterations, without failing.

In the method that creates the child ANNs, the breed() method, the mutation rate is either given as a float, or it can be given as a callable function. If a lambda function is passed, the function will take the average reward of the parent ANNs and calculate a mutation rate based on this. This way, we can make a lambda function that gives a smaller mutation rate for high performance ANNs and a higher mutation rate for low performance ANNs.

After running all the generations, we can render the best ANN from the latest generation using the render\_best() method. This will

Method for rendering average?

**Crossover**

When the GA class defined in *Genetic\_algoithm.py* calls the *new\_generation(self)* method, it will call breed(self,parent1,parent2) and *select\_n\_from\_distribution(2)*. *select\_n\_from\_distribution(2)* randomly picks two ANNs as parents from the current generation from the population with probability defined by calling *get\_relative\_fitness(self).* *breed(self,parent1,parent2)* is run *self.n*/2 times where n is the size of the generation set when the GA was constructed.

Parent1 and parent2 are cloned to child1 and child2. Child1 is then used to call *crossover(self, other, \*args)* from the ANN class defined in *neural\_network.py.* The *coefs* and *intercepts* defined by the *MLPClassifier,* representing all weights and biases, are merged into rows, one from each parent. *crossover(self, other, \*args)* then calls *\_crossover\_rows(self, row1, row2, crossover\_method, ravel)* to do one of three defined crossover operations. The input ravel is a Boolean that should be set to true if input parameters *row1* and *row2* are matrices.

The code has three crossover methods available.

one-point:

The two rows are split in two somewhere at random, and two new rows are made using these four parts. One part is made with the first part of row 1 and the second part of row 2, the other is made with the opposite.

two-point:

Much like "one-point”, except the rows are divided into three parts at random. One of the new rows would be made from the first part of row 2, the second part of row 1 and the third part of row 2. Vice versa for the other new row.

uniform:

Uniform crossover assigns a 50% chance to each variable in the row that they be one of two parts used to make the new rows. The new rows are then made much like how the one-point split is made.

In our code, these methods are realized by using a mask of index numbers using slice() for one or two point crossover, and using an array of Boolean values for the uniform crossover.

This is all illustrated in the figure below.

ILLUSTRATION

**Mutation method**

The methods for performing mutations will change the weights and biases depending on the probability described by the mutation rate. There is a possibility to pass a lambda function as a mutation rate, which is described in the GA class, where this is handled.

The way the mutation is done differs from the suggestion in the assignment. Here, the probability for every weight to mutate is set equal to the mutation rate by making a mask of boolean values. If a random number representing a weight is below the mutation rate. The mutation will be performed.

Unlike the suggestion, a mutation is not performed by switching weights between children ANNs. A few different options for mutation have been implemented and can be chosen by commenting/uncommenting the desired option. The option being used in the code is one which adds a small number to the weight to be mutated. The number added is normally distributed with expectation 0 and standard deviation 1, .

There is also a clone function, that makes a new ANN with deep copies of the same weights and biases. This is used to initially make the child ANNs from the parents before crossover and mutation.

Results

After running the algorithm with many different configurations, we found that there are several factors that influence the ability and effectiveness of the algorithm. When it comes to the parameter choice for the neural network it seems that activation function and network structure play the biggest role. Activation we found to be the best is ‘tanh’. When we use it in conjunction with other sound parameter choices, we get consistent results and quick convergence. In addition, when it comes to neural network structure, bigger is not always better. Is seems that our algorithm converges just as fast when we use fewer nodes in hidden layer, even when it is only one.

The other important parameter choices are crossover and mutation methods. Mutation seems to be more important of the two. This is probably due to small structure of our network and very few coefficients to swap around. Almost any mutation method works, but the one we got the best results with was adding random value from a normal distribution ~N(0,1) to elements. More important was the choice of mutation rate. This is because if the rate is too low, mutation rarely happens so our algorithm doesn’t get to properly explore the solution space. On the other hand if the rate it too large we might easily lose a good solution since mutation happens too frequently. We settled on a function that sets the mutation rate according to the average score of networks parents. This way, if we have a good solution mutation will happen infrequently so we wont lose it, but if our parents have low scores we get to explore other parts of solution space.

**Ronny Wathne**

Discussion of final code and choice of methods

Report writing

Planning of presentation

Preparation for presentation

**Kristian Fredrik Molina**

Initial discussions and planning of project

Organising group communication

Review of code

Preparation for presentation

**Ivica Kostric**

Initial discussions and planning of project

Wrote and implemented final version of code

Classes for ANNs and generations of ANNs with necessary functions

Discussion of final code and choice of methods

Testing different implemented genetic algorithms

Planning of presentation

**Andreas Nesse**

Initial discussions and planning of project

Wrote preliminary code script - following assignment suggestions

Functions for simulating ANNs and creating new generations

Discussion of final code and choice of methods

Report writing

Making presentation slides and structure

**Clement Couronne**

Review of code

Preparation for presentation