**Process**

We initially met and discussed the project. We decided to work in github and code separately in different branches at first. Some preliminary code was written using just functions to run the training, simulation and generating new generations of ANNs. This code followed the suggestions in the assignment text closely. Eventually a class-based solution was implemented instead, that also diverged somewhat from the suggested approach. After most of the coding work was done, we met to discuss choices made in the code, the different implemented methods, further testing to be done and planning of the presentation and the report.

**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

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

The submitted code consists of three python scripts. Two of them contain a class, the ANN class and a GA class. They are described in more detail below. The last script 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.

**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 4 hidden layer nodes. When initialized, the ANN also partially trains itself using a random action, and the initial 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 obervations 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.

There are three major operations that this class is responsible for, in addition to storing the weights, biases and MLPClassifier. First, this class runs the simulation for for itself on the CartPole environment. Second, its also performs crossover on itself and another ANN, in that case you pick one and pass the other as a parameter. Third, it performs mutations on its weights and biases.

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.

The methods for performing the crossover takes a different ANN object as an input parameter to preform the crossover with. There is an option to unravel any potential matrix and perform the crossover on all the weights, instead of the same crossover on the individual node weights. This is set to False as default.

There are three types of crossover that are implemented, chosen with the crossover\_method parameter. The single-point crossover selects an index and performs the crossover before this point. The two-point crossover selects two indices and performs the crossover on the data between these. The uniform crossover first generates a random number for every datapoint, if the random number is below 0.5, a crossover is performed with the data at the same position in the other array. If it is above 0.5, it is left alone. This is all illustrated in the figure below.

ILLUSTRATION

The methods for performing mutations will change the weights and biases depending on the probability descirbed 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.

**GA Class**

The GA class was made to initialize and hold a generation of ANNs, run the simulation fo 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 performinance 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?

**Parameters and script for running**

The last script defines the parameters. We need to define the number of ANNs in a generation, n, and the number of generations, num\_gen. The number of nodes in the hidden layer, and number of layers, hlayer\_size, can also be set with a tuple. We will only use one hidden layer, and 4 nodes in the hidden layer. A crossover\_method is set with a string and mutation rate can be set with a number or with a callable.

This is where we set the mutation rate, which can either be a floating point number between 0 and 1, or a lambda function. Here we have set it as

which means that the mutation rate is 5 divided by the average reward.

Next, the CartPole-v1 environment is initialized and the maximum number of steps set to infinite.

The GA object is initialized with given parameters and then the ga.run() function is called to train the generations of ANNs.