Evolving Artificial Neural Networks to Play Games

Problem Definition

Use genetic algorithms to gradually improve generations of artificial neural networks

Introduction

In this project we make use of artificial neural networks and genetic algorithms to develop a network that can successfully play a simple 2D game. The game is called CartPole and revolves around a simulation of a cart on a track with a pole balanced on top of it. The goal of the game is to keep the pole upright by moving the cart. The longer it stays upright, the higher your score. The network takes four observations from the environment at any time and must decide whether to push the cart in the left or the right direction. The observations describe the carts position, the carts velocity and the poles angle and the velocity of the pole at the tip. The game ends when the pole falls below a certain angle, the cart reaches the game boundaries, or the maximum number of steps have been reached.

We use multiple generations of many artificial neural networks. New generations are made using crossover and mutation algorithms and careful selection of parent networks. We must choose the parameters and method implementations that efficiently performs this generational evolutionary learning process. Hopefully, the networks will increase in ability to play the game and the outcome will be a high-performance network.

Participation in Project

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
Discussion of final code and choice of methods
Preparation for presentation

Ivica Kostric
Initial discussions and planning of project
Wrote and implemented final version of code with classes and methods
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
Discussion of final code and choice of methods
Report writing
Planning of presentation and preparing structure of presentation

Clement Couronne
Review of code
Preparation for presentation

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. The code was further updated, testing was carried out and the report was written, describing the project and summarizing the results.

Included in Project Submission

3 scripts holding the necessary classes to run the project

- genetic_algorithm.py
- neural_network.py
- dna.py

1 script for defining parameters and running the entire project

- run.py

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

There are three scripts containing the classes necessary to run this project. GA is a class that stores all networks in one generation, simulates them, stores rewards and the best networks for each generation, chooses parents and creates new child generations. The classes Entities and ANN initialize the individual artificial neural network objects. The DNA creates an object with the necessary genetic algorithm methods we will use to create child networks from the chosen parents. In the run.py script, there is a main function for running the entire project, and all the parameters are defined here. The descriptions of the classes and their methods are not exhaustive. For more information, the methods in the scripts also contains docstrings explaining their use and abilities.

The GA class creates an object that contains everything we need to run the multiple generation learning process. It contains every artificial neural network in a generation at any time, holds the object containing the genetic algorithm methods, stores the fitness measures over time and the best networks for every generation. It has methods of its own to create new generations, run simulations and calculate the fitness for the ANNs in every generation. It also has functions for reporting results, running and rendering simulations for the best resulting network and the average network for one run of the multigenerational process.

The ANN class is a subclass of the MLPClassifier SciKit-Learn class. In addition to the features of MLPClassifier, it also stores the parameters passed to it through an Entities object. The MLPClassifier is partially trained using a random output from the CartPole environment. It has methods for returning its weights and biases, cloning itself, useful when creating new child ANNs. Lastly, the method run_simulation() will perform the simulation itself, by receiving observations for every step, and predicting the next step. This simulation can be rendered if that is desirable and is limited by the maximum number of iterations passed to it. It will also end if the network fails to keep the pole upright or the cart hits the boundaries.

The Entities class is a helper class holding the parameters common to all the networks that we have defined from the start. It also has a method for creating the ANNs that pass the properties of the Entities object to the new ANN object.

The DNA class contains the genetic algorithms we use in the project. It stores parameters related to method choice and configurations and mutation rate. These are the functions used to create two new children ANNs from two parent ANNs. The combine()-method oversees the crossover and mutation algorithms and returns the new child weights and biases.
The methods handling the mutation use the mutation rate as a probability that any weight or bias will mutate. The mutation itself can be done in multiple ways. Our implementation can set the mutated weight or bias to zero, make a new permutation of the weights and biases to be mutated, add a uniformly distributed number between in the range [-1, 1], or add a normally distributed number, with expectation 0 and standard deviation 1.
The methods handling the crossover also has multiple opportunities for different configurations. The crossover algorithm performs the operations on two sets of weights and biases from two parent ANNs and produces a new set for the child ANNs. There are three versions of the crossover algorithm implemented:

- One-point crossover
  On each row of correlating weights or biases for both ANNs, there is a split set at a random index, and the weights before this switches place between the two ANNs.
- Two-point crossover
  Similar as one-point crossover, but two splits are made and the data between these points is exchanged between the ANNs.
- Uniform crossover
  This uses a random number to give a 50% chance for any weight or bias to switch place with its corresponding weight or bias in the other ANN.

While the weights and biases belong to ANNs, the crossover and mutation algorithms themselves only act on lists of arrays containing the weights and biases belonging to ANNs.

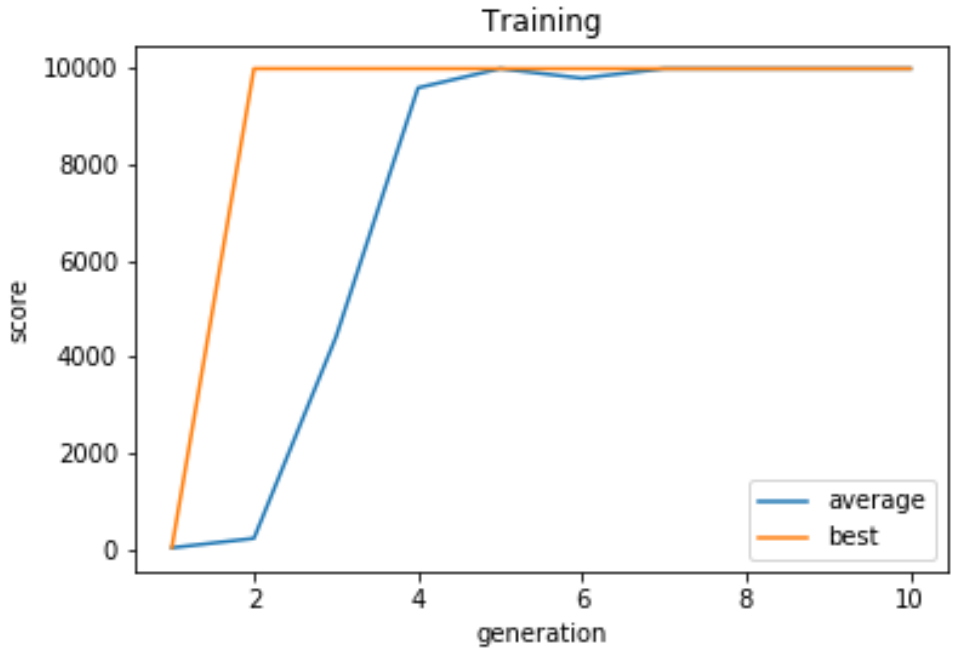
There exists an option for unravelling the weights and the biases between/on a layer and making a crossover on the entire set of weights or biases. The parameter indicating this is stored as a property of the DNA object and is called crossover_ravel and set to False as default.

The run script contains the parameters and a function for running the project. The environment is initialized here, and you can choose whether to render the game during the training.
In the run script, parameters can be changed to run the procedure in many different configurations, but in the result section we will discuss our choices for the methods we ended up setting in the final version of the run script. This becomes our preferred configuration.

Results

We wrote a script for testing our algorithm with many different configurations of parameters and choices of methods. We ran the entire process using 10 generations of 50 networks per generation. From this we could see that especially the activation function and the network structure play a large role in how the algorithm will perform. We settled on using a different activation function than the rectifier function (ReLU) that was recommended in the assignment text. We found that using the hyperbolic tangent function (tanh) gave the best results in combination with other sound parameter choices. This gave us consistent results and quick convergence.
We also found that the network seemed to perform slightly better with a reduced number of nodes in the hidden layer. In fact, having a single node in the hidden layer seemed to work just as well even though this becomes less of a network. We still decided to go with using one node because it is simpler. We also noticed that doing a partial fit after every step was unnecessary and slow. It can be turned on with a parameter.

The choices and parameters for the crossover and mutation methods are also important. The mutation algorithm seems to have a bigger impact. This could be due to the small size of our network and the small number of coefficients to change. All the implemented mutation methods work, but the best result seems to be by adding a small number to the weight or bias that is to be mutated. This number is normally distributed $\sim N(0,1)$. As far as the crossover, we use single-point crossover, and we do not unravel the weights and biases, but make the divide on each row of the matrix, if there exists one.
The mutation rate has a bigger impact than the method itself. If the rate is to low, the solution space is poorly explored. With a mutation rate too high, we might undo any advances we have done. Because of this we settled on using a function that sets the mutation rate according to the average score of the parent networks. That way a network with low scoring parents is changed more drastically, and we get to explore the solution space. The children of better scoring networks are more protected from mutations that might ruin a good solution.
To summarize, we include a graph showing a typical result from using these parameters.



Parameters
Activation function: tanh
Hidden layer size: 1
Crossover method: Single point
Mutation method: Adding normally distributed number $\sim N(0,1)$.
Mutation rate: 5/average parental score
No partial training after every step

Sources:
Sources used to gather information on methods implemented in our code

- Davidrajuh, R. “Genetic Algorithms - I”, Lecture slides, Spring 2015
- Chakravorty, A. “Genetic Algorithms #L13”, Lecture slides, 23.03.2017
- “Genetic algorithm” Wikipedia.org https://en.wikipedia.org/wiki/Genetic_algorithm
- “Crossover (genetic algorithm)” Wikipedia.org https://en.wikipedia.org/wiki/Crossover_(genetic_algorithm)
- “Mutation (genetic algorithm)” Wikipedia.org https://en.wikipedia.org/wiki/Mutation_(genetic_algorithm)
- “Activation function” Wikipedia.org https://en.wikipedia.org/wiki/Activation_function