Notes for presentation

Here are some suggested notes for the presentation. Feel free to make changes and additions but remember that they expect all group members to talk for about 5 minutes, give or take.

Structure:

Problem definition

- Introduction of project, problem and the Gym environment (Clement)

Solution

- Artificial Neural Networks and how we use them in our project (Andreas)
- Genetic algorithms. What are they and why use them? (Kristian)
- GA – Crossover Algorithm (Ronny)
- GA – Mutation Algorithm (Kristian)
- Short description of code / implementation? (Andreas?)

Results

- Results and demonstration (Ivica)

The requirements for the presentation is that it ***"cover[s] the problem definition, solution and results".*** And so, each section should not be too focused on theory, but be a combination of how the algorithms work in theory, and how we have implemented / used them in this project.

Introduction of project, problem and the Gym environment (Clement)

Keywords

- Using artificial neural network to interact with a simulated environment
- Using genetic algorithms to improve ANNs over multiple generation
- Very short description of OpenAI GYM
- Description of the CartPole environment.
    - Problem to be solved

Suggestion for text

In this project we tackled the problem of training artificial neural networks to perform a task in a simulated environment.

The training happens by using multiple generations of many individual networks. Genetic algorithms are used to develop favorable traits, and the best resulting networks are the outcome of the process. Hopefully, these will be able to perform the task on a satisfactory or high level.

OpenAI Gym is a toolkit for training reinforcement learning algorithms and there are multiple environments and tasks to test an agent's learning ability against. The one we have made use of in this project is called the CartPole environment. This is a 2D game where a cart move along a track. The cart can move to the left or the right on the screen. On top of the cart a pole is placed in a hinge and balanced. The goal of the game is to keep the pole upright by pushing the cart towards the right or towards the left, to stop the pole from falling.

[The slide should contain

a picture/video of CartPole environment

and an illustration of observation array]

The game or environment is at any point described with 4 observations:

- The location of the cart
- The velocity of the cart
- The angle of the pole
- The velocity of the pole at the tip

There is no need to know about what has happened previousy, the agent only needs to know the current state of the enviornment, described by these four numbers.

The artificial neural networks take these four observations as input and then decides on the next action to take, which is to apply a unit force in the left or right direction. The game ends either when the pole falls below a certain angle, when the cart hits the boundaries of the game, or when the maximum number of timesteps have passed. Every timestep the network receives an observation and it must decide on an action. It is not possible to do nothing; a force must be applied in either direction.

Initially, the environment starts at a value close to zero. There is a small random number given to all the four observations to get the game started. We also assume that the first action performed is a random action, so a push either in the left or the right direction, determined by a randomizer function. From there on, it is our agent's task to move the cart to balance the pole.

Originally, the game was meant to stop after simulating 500 timesteps, but to make sure the agents become very good at balancing, we let the simulations run much longer. The limit we set was 10 000 timesteps.

The simulation of this game happens to every artificial neural network in every generation until the desired number of generations have passed. We can choose whether to render the environment, meaning display what happens in every step. We will not do this while training the networks, as this will take too much time, but at the end of the presentation, we will perform a running through the program and display the best network's performance.

Artificial Neural Networks and how we use them in our project (Andreas)

Keywords

- Description of how ANNs work
    - Weights, biases, activation function
- Implementation of ANNs in this project
    - Input nodes
    - Hidden layer
    - Output
    - Specific implementation
        - Class
        - MLPClassifier
        - ReLU
        - Etc.
- Descriptions of generational improvement of ANNs
    - Genetic algorithms explained in detail by others

Suggestion for text

Genetic/Evolutionary Algorithms (Kristian)

The current plan is that you introduce genetic algorithms and talk about why we might want to use it in general and for this project. Then, after Ronny talks about the crossover algorithm, you describe the mutation algorithm.

We divided it like this because the crossover algorithm takes longer to explain, and these two parts should take up about the right amount of time combined.

Keywords

- Short description of theory of genetic algorithms
    - Type of evolutionary algorithm
    - Models the algorithm as a biological entitiy, with genes
- How we use genetic algorithms for this project
    - Individual weights and biases are genes
    - Sets of weights and sets of biases are chromosomes
    - An ANN is seen as an individual
    - New ANNs are made by selecting best ANNs to parent new, child ANNs
        - Probability based on reward
        - Crossover algorithms used to create children ANNs
    - Mutation algorithm used to introduce new information

Suggestion for text

The genetic algorithms are types of evolutionary algorithm. This means they mimick the sort of evolution living beings might go through in many generations. When using a genetic algorithm specifically, we see the network as a biological organism with genes.

[Slide should contain illustration of how weights are seen as genes

Illustration can be same as in ANN part of presentation]

In our case, we can view the individual weight between two nodes as one gene. All the weights and biases in the artificial neural network are genes that make up the individual network.

In every generation there will be networks that perform better and networks that perform worse. Using genetic algorithms, we choose parent networks from a generation to create new pairs of children in the next generation. The parents are chosen according to their fitness or performance.

The hope is that by combining the features of high-performance parents, we will create even better children networks. We create children using a crossover algorithm, that will be explained in more detail later.

In each generation, the networks are scored on how long they manage to play the game without failing. The parents are then chosen with a probability based on the score they received.

This means that an agent that manages to play the game for 300 timesteps, will be much more likely to be involved in creating children in the next generation than an agent that only manages to play the game for 20 steps before failing.

Depending on how much computing power and time we have, we can either limit the number of generations, or let the genetic algorithm run until the individual networks become very similar, meaning they converge towards a solution.

In addition to the crossover algorithm creating new children networks, we can also perform mutations in the weights and biases of the network. The mutations can be done in different ways, but in our case, we will make a small increase or decrease in the weight or bias being mutated. This introduces new information into the system, and helps avoid the network from getting stuck with a local solution that is much worse than the globally best solution.

The likelihood of a mutation is described by the mutation rate, a number between 0 and 1. Let’s look closer at the crossover algorithm:

GA - The Crossover Algorithm (Ronny)

The current plan is that you talk about the crossover algorithm after Kristian introduces genetic algorithms, but before he describes the mutation algorithm.

Keywords

- Uses the weights and biases from the parent ANNs
    - Generates child ANNs
        - New weight and bias configuration
- Our implementations
    - Single-point crossover
    - Two-point crossover
    - Uniform
    - Unravel matrix or keep shape?

GA - The Mutation Algorithm (Kristian)

The current plan is that you talk about the crossover algorithm after Kristian introduces genetic algorithms, but before he describes the mutation algorithm.

Keywords

- Our implementation:
  - Generate a random number for every weight and bias
    - If the number is below the mutation rate, perform mutation
    - If it is above, leave alone
  - For mutation, add a small random number to the weight or bias
    - Normally distributed with expectation 0, and standard deviation 1
  - Discuss the alternative implementations in our code:
    - Set the weight/bias to 0
    - Add a uniformly distributed number between -1 and 1
- Takes a mutation rate, either a number or a function
  - We have used an adaptive mutation rate
    - More mutations for bad networks
    - Less for good networks
    -

Suggestion for text

There are many ways to implement a mutation algorithm, but first we must figure out whether to perform a mutation, based on the mutation rate as the probability.

We generate an array of random numbers, uniformly distributed between 0 and 1, for all the weights and biases. If this number is smaller then the mutation rate, we perform a mutation, if not, we leave that weight or bias alone.

There are many ways of mutating, depending on the model and learning agent we are using. Mutation involves changing the properties of the network and introduce new information. This can help the network not to get stuck in local solutions that are much worse than the globally best solution. We have a few different ways of mutating the weights and biases in our code, but the one we ended up using is adding a small, normally distributed number to the weight or bias. In practice, this means we increase or decrease the number by a random, small amount.

The mutation rate can either be set as a number, for example 0.01, a 1% chance, but we also have the option in our solution, to pass a function as the mutation rate. We ended up using a function that decreases the mutation rate, the better the average scores of the parents. This is to avoid ruining good networks. For networks with a low scoring parents, the mutation rate then becomes higher, to give more drastic change.

Short description of code / implementation? (Andreas?)

We did not decide on this in our discussion, but I thought I (or maybe someone else, if their part is too short) could give a quick description of the code and how it runs. This would literally be 1 minute or less.

Results and demonstration (Ivica)

You can decide what you want to say and show for the results like we discussed, but we should probably cover:

- Short description of why we chose the different methods / implementations in the algorithms – if you don't do any testing for one of the choices, maybe say a word or two about why you prefer one over the others
- Why we chose to partial_train using env.reset() and avoided partial_training for every step
    - Nice if you say a sentence on why you think env.observation_space.sample() gives bad results, like you explained at the meeting
- Results
    - How many generations until solution, on average, or something like that
    - Other metrics of your choosing
    - Remember to mention that even a good ANN can still fail
- Comparing best ANN, average ANN – seems like a demand in the assignment text?
- Live demonstration running code