Teaching Machines to Play

Daniel Ahl
Supervisor: Matthew Johnson

ABSTRACT

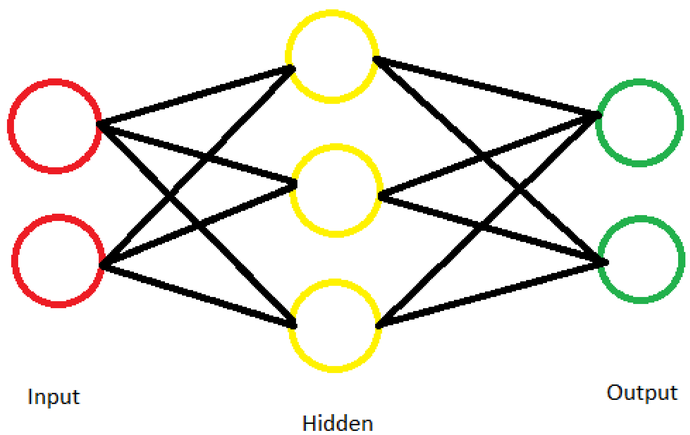
Normally a computer must follow a set of pre-programmed, explicit, and often complex instructions in order to accomplish a task, but with **machine learning**, the computer is programmed to learn how to complete the task on its own.

This project tests a variety of popular machine learning techniques effectiveness and efficiency when applied to a simple game environment. The environment is a small game where the computer must move a **cart** right or left along an axis in order to **balance a pole** on top of it for 200 timesteps.

The findings highlight the most effective methods as well as shortcomings in popular methods when information from the environment is restricted. Interestingly, the computer's learning pattern mimics that of a human, if much slower. For example, it first learns to keep the pole upright, then later finds keeping it in the center of the screen is ideal (going off screen results in death).

BACKGROUND

- The focus of this project is on the **reinforcement learning**, an area within machine learning where the **agent** (computer), takes **actions** within an **environment** and receives **rewards** (good or bad) in response. It should then use this information to figure out what the best course of action is.
- Each technique is implemented using a **neural network**, a method influenced by neurons in the human brain. Data is fed to the network, each number multiplied by **weights**, then either passed to the next neuron or dropped depending on an activation function.
- **OpenAI Gym** is used for the environment, specifically **CartPole-v0**.
  - The library accepts actions a number, then returns a reward and the state of the environment as a series of numbers.
  - The agent gets a reward of 1 for every timestep the pole is balanced, capped at **200 steps**.
- **Tensorflow** is used to construct and run neural networks, taking the state of the environment as input and producing information about possible actions as output.
- All code for this project is written in **Python**.

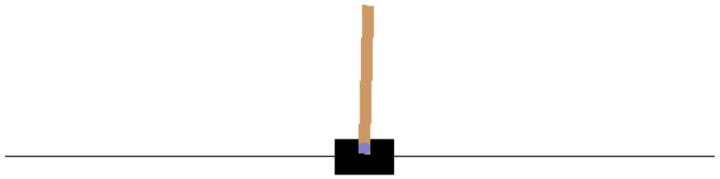


- An example of a simple **neural network**

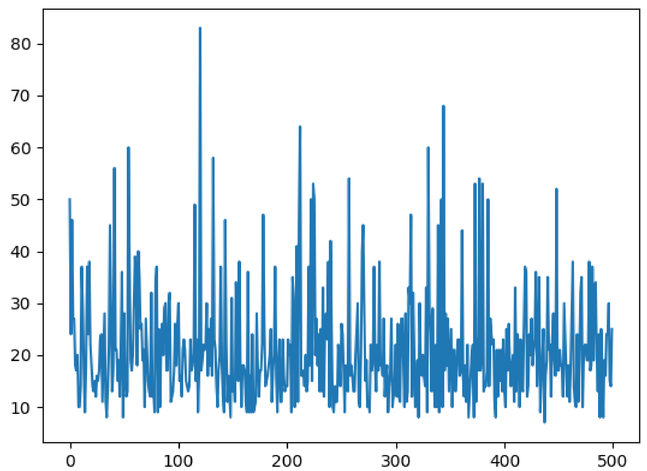
METHODS

- Most of these methods involve updating the **neural network**. The key component here is the **weights**, and that is what will actually be updated in order to improve the accuracy of the network.

1. **Q-Value:** At each timestep, the state is fed into the neural network to produce a q-value, representing the expected future reward, for every possible action at the moment. After the action is taken, the network weights are updated to move closer to the real q-value.
2. **Policy Gradient:** The network receives the state and produces the probabilities of choosing each action. At the end of each **episode** (completed individual game), every action made is assigned a value equal to the total number of rewards received after that point (earlier moves have a higher value). A batch of episodes are run and all their data is used to update the network to favor better earlier choices.
3. **Actor-Critic:** A combination of the last two methods. First, a network produces **q-values** to estimate the value of all moves at each timestep, which is used to update a **policy network** that produces the action probabilities. This uses the value network to bootstrap the policy network, so we don't have to wait for batches to complete to make changes.

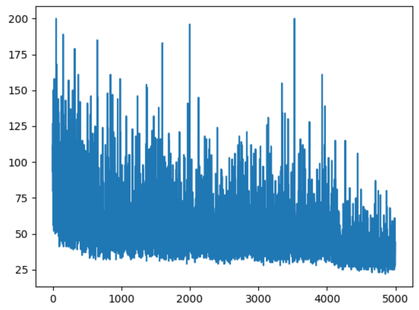


- A picture of the rendered **CartPole environment**

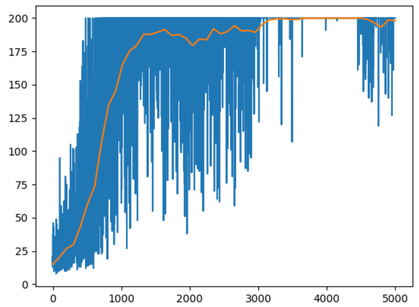


- The rewards received over 500 episodes of **randomly selecting actions** at each timestep.

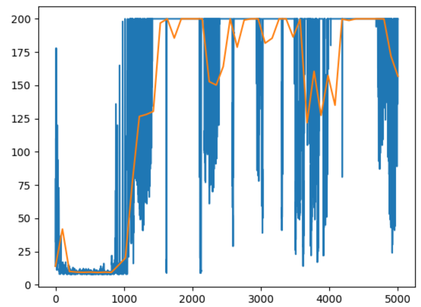
RESULTS



- **Q-Value:** We can see while this approach performs better than randomly selecting actions, it does not outperform it by much. Furthermore, the performance is very noisy and there appears to be a slight downward trend. Likely due to overly optimistic predictions from the network.



- **Policy Gradient:** Here we have our most effective, if not efficient, approach. It takes about 1000 episodes to learn, then another 2000 to become stable. The success is likely due to the approach favoring strong choices early on, important in a game about balancing.



- **Actor-Critic:** For the first 1000 episodes, this agent does terribly, likely due to awful value predictions from an inexperienced value network. Afterwards, it quickly learns to choose correctly. However, the network is not stable, possibly due to the value network continuing to adjust and becoming overly optimistic about suboptimal actions.

CONCLUSIONS

- Parameter tuning is incredibly important to the success of an agent.
- Policy Gradient is an effective method for solving environments where early moves matter more than later moves.
- Diminishing the learning rate over time is important. There are many cases, such as in Actor-Critic, where the optimal solution is found, yet the network continuously updates in the name of exploration and inadvertently loses the solution it had.
- Reward/Value-driven techniques struggle with environments that have a lack of variety in the rewards they return. These value-based approaches try and steer agents away from negative rewards and towards larger positive rewards, but when every timestep returns a reward of 1, the network can get overly optimistic about the expected future reward.