Teaching Machines To Play

How do you program a machine to act a certain way without explicitly telling it what to do? Is it possible for a machine to learn from its experiences and improve itself over time? The answer is yes and the key is machine learning. Machine learning is a broad field in computer science concerned with programming the capability for learning into machines. The concept that I will be exploring in this paper is how to get an individual computer program, or agent, to learn to play a simple game without the program having any prior knowledge about the game environment. The paper will start with some general background and then cover each method I attempted over the semester, my implementation of it, and my findings and conclusions after testing and tweaking. OpenAI Gym was used for several game environments and Tensorflow was used to construct neural networks (more on these topics later). The entire project is coded in the Python programming language and hosted on GitHub.

There are several subfields within machine learning that work differently to accomplish different types of goals. One such subfield is called supervised learning. This is where a computer is given a set of training data, such as pictures and labels, and learns to map the input to output. For example, a supervised learning problem may have a large set of pictures of dogs and cats, labeled accordingly, and would have the machine iterate through the pictures and adjust its predictions of whether a picture contains a dog or a cat based on the correct answer found in the training data. Given a large enough data set and enough iterations, the machine could then be taken out into the real world and be able to identify dogs and cats in new pictures it had never encountered before. This is a particularly narrow example, but the principle can be scaled up given a complex enough machine and robust enough data set, and can be applied to real-life problems such as recognizing characters in people’s handwriting and or automatically recognizing and filtering spam email.

The field that we will be exclusively working within for our project is called reinforcement learning. This field is related to supervised learning as we are training our agent based on data received but different in that instead of iterating through pre-set training data before being released into an environment, the training data is the feedback the agent gets from interacting with the environment. In reinforcement learning, the agent is given the state of the environment represented in some arbitrary format (such as the raw pixel data of the screen, or a series of numbers describing the state, for example), takes an action based on this state, and receives a numerical reward representing how good or bad of a move that was. The logic behind this is that the agent will gather information about the environment by interacting with it, then over time favor actions that net bigger rewards and avoid actions that do not, maximizing overall score and ideally finding the best possible playstyle. This is a good choice for our problem because it fits so neatly into the issue of playing a game. It’s also interesting because it mimics how a human would learn to play a game for the first time: press different buttons to see what happens, then figure out what to do in each situation to get the best score possible.

In order to test different reinforcement learning techniques, we need an environment to test within. Luckily we don’t need to reinvent the wheel, as the previously mentioned OpenAI Gym has a large number of game environments freely available and rigged up for reinforcement learning. The way to use it is to simply import the Gym module into a python script and then create an object containing any environment that Gym supports. The environment object has a reset function that resets the environment and returns information about the initial state of the game, usually in the form of an array of numbers. The other key function is the environment’s step function, which takes in an action (usually a single integer representing a button press), executes the selected action, and returns the game state, a reward, and whether or not the game has ended.

The first environment I looked at was a simple game called Frozen Lake. It runs in the command line and represents the game visually using text. The game state is given by a single integer representing the number tile the player is currently in. The player starts in the top right corner of a “frozen lake” and must traverse through the grid of tiles to the bottom right corner to escape. Reaching the end tile gives a reward of 1 while falling in a hole along the way gives a reward of -1 and ends the game. All other actions receive a reward of 0. However, the “lake” is slippery and so there is a random chance that when you attempt to move in one direction, the environment will move you one tile in a completely different direction. The first reinforcement learning technique I try in this environment is called Q-Learning.

Q-Learning uses something q-values, a number representing the value of taking a specific action given a specific state. Determining a state-action pair’s q-value makes use of the Bellman equation, which describes the value of an action as the immediate reward received combined with the discounted total future reward. In Q-Learning it looks something like this:

Q(state, action) = reward + discount\_factor \* max(Q(nextState))

where max(Q(nextState)) refers to the maximum possible q-value within the new resulting state. For this first attempt we will store our q-values in a q-matrix where the rows are all possible game states (0-15 for the 16 tiles that the player can be in), and columns are all possible actions (0-3 for the 4 possible directions), so that Q(1, 3) refers to the q-value of taking action 3 in state 1. To update our table of values we’ll use the equation:

Q(s,a) = Q(s,a) + learning\_rate \* (r + discount\*max(Q(nextS) - Q(s,a))

Where learning\_rate is a number between 1 and 0 that determines how quickly we replace old values with new ones in our q-matrix. A rate of 1 will completely replace an existing q-value with the newly calculated one and a rate of 0 will never replace any values. The discount refers to the discount factor between 0 and 1. A rate of 0 completely disregards all future rewards in the calculation, and a rate of 1 treats the possibility of future rewards with the same weight as the actual received reward. For each timestep, the agent looks at q-values of each action in the current state and chooses the action that has the highest q-value, with an epsilon percent chance of picking an action randomly decreasing over time.

Initially, I set the discount rate to 0.97, the learning rate to 0.9. After running the agent for 1000 games, or “episodes”, it was only able to win the game 23% of the time. I believe this is because the learning rate is too high. If the agent tries to move right and slips left into a hole, all it knows is that it tried to move right and received a negative reward, when in fact moving right could be the correct move. This conflicting information is problematic, so I lowered the learning rate to 0.2 in order to make the agent more cautious about new information. I left the discount rate at 0.97 because the only time the agent receives substantial rewards is when winning or losing the game, and as such it should heavily consider future rewards when making decisions, as the winning set of actions may only produce a good reward 6 moves in the future, for example. After lowering the learning rate, the agent was able to complete 55-70% of 1000 episodes successfully. This was much better, but not ideal. I assumed the random slipping was the key component to failed episodes.

Following my hunch, I wrote my own version of Frozen Lake, referred to from this point on as Unfrozen Lake, which mimicked exactly how the Frozen Lake environment worked but allowed me to control the rate of slipping. If I eliminate the chance of slipping and set my Q-Learning agent loose in this environment, about 99% of the 1000 episodes it completes are successful. Often it found the solution within the first 10 episodes, and I never saw it lose a game after episode 20. As we can see this approach works very well for a game like Frozen Lake with discrete states because it is easy to transform each game state to a distinct row within a table, but what about a game that has continuous states?

CartPole is the next OpenAI Gym game environment I looked at. The goal of the game is to balance a pole vertically on a cart for 200 timesteps. Each timestep you must move either left or right across a straight line. The game state is given as an array of 4 floating point numbers. For each timestep that the pole is balanced, you get a reward of 1. If the pole tilts too much in either direction or if your cart moves off-screen, you get a reward of 0 and the game is over. Because of the continuous nature of this environment, as the cart can be at any x coordinate on the screen and the pole can be at any angle, as well as the fact that the game state is given as 4 floating point numbers, the q-matrix approach will not be applicable here. In order to handle these continuous game states, we will need to use a neural network.

A neural network is a computational model inspired by the human brain. It mimics the way the brain processes information through firing signals between neurons. Each neuron is represented by a layer of nodes, and each node holds a number. As information is passed from one layer to the next, the numbers contained within the nodes of the last layer are multiplied by a single weight for each number and added together to create the value of a single node in the next layer. The process is repeated for each node in the new layer. Once the new values have populated the new layer, an activation function is applied to each node to determine whether or not to pass its data to the next layer, and how to manipulate it beforehand. There are many types of possible activation functions, and one of the simplest and most widely used is a Rectified Linear Unit, or ReLU. For each node that ReLU is applied, if the value contained is negative, ReLU passes a 0 along, effectively not firing this neuron. If the value is 0 or above, the number is passed along to the next layer unchanged.

As our goal is to just replace our q-matrix with a neural network, we will create an extremely simple network. We will have an input layer of 4 nodes, for each of the 4 floating point numbers in our game state, and multiply them by a 4 by 2 matrix of weights to produce an output layer of 2 nodes, containing the q-values for moving left or right. Note that in this simple 2-layer network, we are not applying an activation function, and instead only doing matrix multiplication.  
 The next major aspect of a neural network is how to update our weights to improve our predictions. This is accomplished using something called gradient descent. First, we must define a loss function, a differential equation describing how far our network’s prediction was from the correct answer. An example of such an equation that we can use for our problem is

(realQVal - predictedQVal) ^ 2

which describes the difference between the predicted q-value of an action, referring to the output of the network, and the actual value we calculate after taking said action. After we define a loss function, we find its derivative, or “gradient”, and “descend” it by adjusting the network’s weights in the direction opposite the gradient, moving towards a lower point in the loss function and effectively minimizing loss. The amount to descend the gradient is determined by the step size, which should be relatively small. A smaller step size will take a longer time to train, but a larger size may step right over the optimum solution. As mentioned before, the neural network is built using TensorFlow, which is a python library that handles the backend of updating neural networks, such as calculating the gradient and adjusting the step size.

When applying this method to the CartPole, we get underwhelming results. At first, it appears the agent does learn, scoring in the 40 to 100 range, out of a possible 200 points. Comparatively, taking random actions will score between 10 and 80, usually below 40. After the first 50 to 100 episodes, however, this Q-Network approach curves back down, quickly falling into a 20 to 40 scoring range and staying there. I believe this is because of how the CartPole environment works and the fact that we must rely on our own network’s prediction of future q-values in order to calculate the current q-value. The CartPole environment gives the agent a reward of 1 for each timestep the pole is balanced and ends the game otherwise. The neural network has no concept of a game over or the fact that the maximum possible reward is 200 for an episode. Due to this, the agent is essentially receiving a reward of 1 regardless of the action it takes for every moment it interacts with the environment, and so our network’s initially modest predictions for future rewards quickly balloons far past the maximum of 200 and becomes overly optimistic about every action it takes.

It’s clear a different approach is required for this environment. The next method we will look at is called policy gradient. With policy gradient, instead of having our network estimate a value function, like in the q-value approach, and then choosing an action based on value, we will reconfigure it to produce a policy, or what action to take given a specific game state. Given a game state as input, our network will output a probability distribution for all available actions. Our agent will then select an action from that probability distribution. This approach is compelling because it introduces a random component more naturally. Instead of occasionally ignoring a value in order to choose a random action, if a policy says the agent should move left 70% of the time in a specific state, the agent will still move right 30% of the time. This allows for exploration at any point in the training process, and if the agent finds that moving left is a better move in that situation, it will increase its likeliness to move that way in the future. Also, though not pertinent to the CartPole environment, this technique should perform better in dynamic environments where similar or identical game states may require different actions at different times.

To construct our policy gradient network, we will add a hidden layer, comprised of 10 nodes, in between the existing input and output layer, with a ReLU activation function applied to it. Layers between the input and output layers in a neural network are called “hidden layers” because of their obscure nature. Unlike the input layer (representing the game state) and the output layer (representing the probability of choosing each action), we can’t truly understand the meaning of the values produced in the hidden layer as data passes through it. As the network trains, each node is extracting some type of information from the input layer that makes sense to the computer, but whose meaning is “hidden” from us. The last modification we will make is to apply a softmax function to the output layer, which simply takes the output values and converts them into probabilities summing to 1.0.

In order to update the network, we must find the gradient of the policy, hence the name. This is described as log(policy) \* advantage where the advantage is defined as reward associated with that action and the policy is defined as the weights responsible for producing the probability of the chosen action. The other tweak in updating our network with this approach is that we are actually going to ascend the gradient instead of descending it, in order to maximize our policy with regards to how much of an advantage it had, defined by the rewards it produced.

In order for the advantage to be more insightful than a reward of 1 each timestep, we will allow an experience buffer to build. We will collect the game state, action taken, and reward received, then at the end of each episode we will iterate backwards through the rewards and add each received reward to a running tally, discounting it along the way. In this way, we are essentially retroactively converting each reward into a q-value, an estimate of the given reward combined with discounted future rewards. The result would be, with a theoretical perfect 200 point run, an initial value close to 200 that decreases with each move over the course of the episode. This makes sense as it would be more critical of early moves, as they are more important to successfully balance the pole for a complete run than moves made just before failing. I apply this transformation at the end of each episode and add that episode’s history to an experience buffer. I then use the entire experience buffer to update the network and clear the buffer. This allows for a smoother learning curve. When the network is trained from a large batch of data instead of a single episode, it is less likely to make drastic changes in the wrong direction. The larger the batch size, the slower but more consistent the rate of improvement.

Running this approach proved successful. Setting the episode batch size to 10, the agent average score rose from 20 to 175 in the first 1000 episodes and hit an average score of a perfect 200 around 3000 episodes into training. There is some noise in the data depending on factors like the initial weights but the agent often performed relatively close to the described conditions. An interesting detail is that, while the agent’s average score usually reached or got close to 200, the individual game scores still remained noisy. The agent became more consistent over time but still made errors resulting in low performance runs late into training. Sometimes it would do perfect for hundreds of episodes but ultimately fall back into inconsistent results. I believe this has to do with the step size. The agent has no idea that 200 is the maximum reward achievable, so it’s possible it’s still trying to “learn” a better theoretical policy. This means that as the agent gets better, we should shrink the step size and lessen the rate at which it learns, as we don’t want it to adjust its weights out of the optimal solution in the name of exploration.

The next method I tested is called Actor-Critic. The way it works by having a policy network, or “actor”, choose actions based on the game state and then have a value-function, like the q-value network, “critique” those actions. The value-function network learns the value of a game state based on the rewards it nets. I update my value network at the end of each episode by discounting my episode’s history of rewards, like in the policy gradient approach. Once the critic learns the value of each state, we use it to update the actor, or policy network. Each timestep the actor takes an action, and if the value-function says that it moved into a more valuable state, it is positively-critiqued and we update the policy network to favor that state-action pair in the future. The opposite happens if the critic claims the policy moved the agent into a less valuable state.

This method is functional with our CartPole environment but less consistent. Interestingly, the agent does abysmally for the first thousand episodes, scoring about 10 out of 200, much worse than simply choosing actions randomly. This is likely due to the fact that the value network has not yet learned how to predict the value of a state accurately, causing it to advise the policy network poorly at this stage. After these first 1000 episodes, the agent averages a score of 200 in about 500 episodes, beating out the policy gradient learning curve by about half the time. However, this success is not consistent and the average score bounces up and down during training, ranging from about 125 to 200. This is possibly due to the fact that the critic is still adjusting its predictions as the training continues. Once the critic learns to predict state values accurately, we should freeze or greatly reduce its learning rate so that it consistently offers good advice to the actor network.

The last method we will look at is a bit of a departure from the previous ones. This method uses genetic algorithms and instead of updating one individual to learn from its experiences, it creates multiple individuals that follow their instincts then uses survival of the fittest to improve them. Each individual agent is defined by its genome. For the purposes of our project, we will use the weights of our policy network as the genome, with each individual weight being an individual gene. This creates a genome of 60 genes of floating points ranging from 0 to 100. First, we create a population of individuals. Then we run them through a number of episodes each, where each agent uses its policy network to choose what actions to take. We take the average of the agent’s scores across all episodes and use that as its fitness score. Next, we take a number of the most fit individuals from the population, pair them off, and create 2 offspring, mixing their genes with a 50/50 probability of one parent’s gene going to either offspring. Each of the children’s genes has a 1% probability of mutating, meaning we pick a new random number to be the gene. The children replace the weakest individuals in the population and the next generation begins.

Applying this method to CartPole proved successful, even when restricting the population to a tiny 10 individuals with 4 parents per generation. Depending on how good the initial fitness scores are, the genetic algorithm approach was able to produce a 200-scoring individual within just a few generations and almost always less than 10. Sometimes, we would see a population where every individual achieved a fitness score of 200 by the end of ten generations. We tried other approaches that narrowed the number of genes down to 14 and 10 in each genome, but they performed about the same. This points to the idea that CartPole might be an overly simplistic environment for this method.

After testing all these methods, I found that all require tweaking their parameters and some methods overall work better than others for specific problems. I would assert policy gradient works best for training an individual to learn to play CartPole due to the nature of the environment, while q-learning is effective for games with distinct environment states such as the Frozen and Unfrozen Lake environments. Genetic algorithms that use network weights as genes show promise for more complex problems. There are also environments for which none of these methods were successful as well as methods that proved too resource intensive to be practical, which I left out for the sake of clarity. Hopefully, this paper proved as informative to you as my own research was to me.