Assignment 6
Reinforcement Learning

1 Introduction

Reinforcement learning (RL) is a subset of machine learning which poses learning problems as interactions between agents and environments. It often assumes agents have no prior knowledge of the given world, so they must learn to navigate environments by optimizing some provided reward function. Within a world, an agent can take certain actions and receive feedback–in the form of positive or negative rewards–with respect to their decision. As such, an agent's feedback loop is somewhat akin to the manner in which a child might learn to distinguish between "good" and "bad" actions. In practical terms, our RL agent will interact with the environment by taking an action at each time step, receiving a corresponding reward, and updating its state according to what it's "learned".

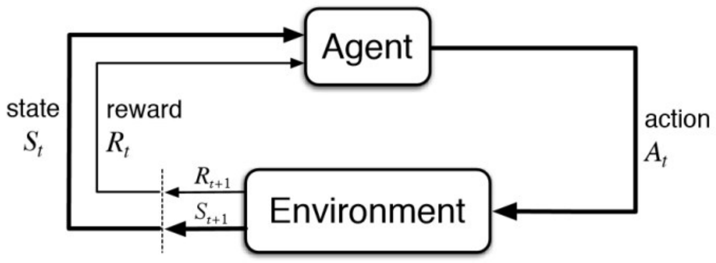

Figure 1

In this lab, we focus on building a model-free reinforcement learning algorithm to master two different environments with varying complexity.

2 Cartpole

2.1 Define and inspect the environment

In order to model our environment, we'll be using a toolkit developed by OpenAI, OpenAI Gym. It provides several pre-defined environments for training and testing reinforcement learning agents, including those for classic physics control tasks, Atari video games, and robotic simulations. To access the basic version of a control task, "Cart Pole", we can use env = gym.make("CartPole-v0"). When we imported gym, we gained access to higher level functions in the package, including creating virtual worlds. Each environment has a specific identifier (for which you can read through here) which is accessed by passing the environment name as a string variable. One issue we might experience when developing RL algorithms is that many aspects of the learning process are inherently random: initializing game states, changes in the environment, and the agent's actions. As such, it can be helpful to set a random "seed" for one of these variables to ensure some level of reproduciblity. Much like you might use numpy.random.seed, we can call the comparable function in gym, seed, with our defined environment to ensure the environment's random variables are initialized the same each time.

CartPole Environment:

A pole is attached by an un-actuated joint to a cart, which moves along a frictionless track. The system is controlled by applying a force of +1 or -1 to the cart. The pendulum starts upright, and the goal is to prevent it from falling over. A reward of +1 is provided for every time step that the pole remains upright. The episode ends when the pole is more than 15 degrees from vertical, or the cart moves more than 2.4 units from the center.

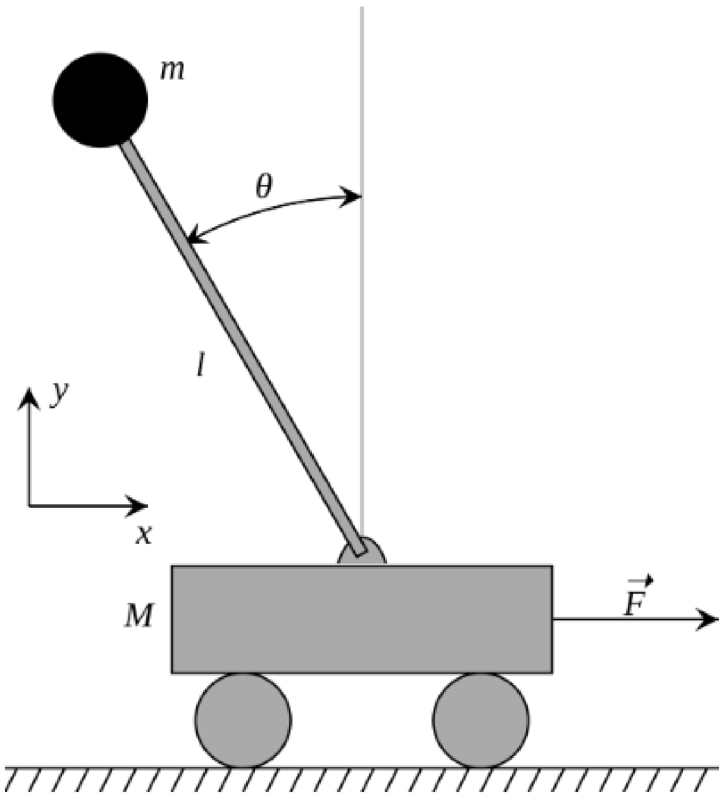

Figure 2

Observations:

1. position of cart.
2. velocity of cart.
3. angle of pole.

4. rotation rate of pole

We can confirm the size of the space by querying the observation space.
At every time step, the agent can move either right or left. Confirm the size of the action space by querying the environment.

2.2 Define the Agent

Let's define our agent, which is simply a deep neural network which takes as input an observation of the environment and outputs the probability of taking each of the possible actions.
Define the action function that executes a forward pass through the network and samples from the output. Take special note of the output activation of the model.

2.3 Create the agent's memory

During training, the agent will need to remember all of its observations, actions so that once the episode ends, it can "reinforce" the good actions and punish the undesirable actions. Let's do this by defining a simple memory buffer that contains the agent's observations, actions, and received rewards from a given episode.
At this state, we should be almost ready to begin the learning algorithm for our agent! The final step is to compute the discounted rewards of our agent. Recall from lecture, we use reward discount to give more preference at getting rewards now rather than later in the future. The idea of discounting rewards is similar to discounting money in the case of interest and can be defined as:
$R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$
where γ is the discount factor. In other words, at the end of an episode, we'll want to depreciate any rewards received at later time steps. Since we can't play an infinite number of games, we'll be limited to the number of time steps in an episode. When implementing the function, you can initialize a numpy array of zeros (with length of the number of time steps) and fill it with the real discounted reward values as you loop through the saved rewards from the episode. We'll also want to normalize our output, which you can do using information about the mean and standard deviation of the discounted rewards.

2.4 Define the learning algorithm

Now we can start to define the learning algorithm which will be used to reinforce good behaviors of the agent and discourage bad behaviours. Start by defining the optimizer we want to use.
Then let's define the loss function. In this lab we are focusing on policy gradient methods which aim to maximize the likelihood of actions that result in large rewards. Equivalently, this means that we want to minimize the negative likelihood of these same actions. Like in supervised learning, we can use stochastic gradient descent methods to achieve this minimization.

Since the log function is monotonically increasing, this means that minimizing negative likelihood is equivalent to minimizing negative log-likelihood. Recall that we can easily compute the negative log-likelihood of a discrete action by evaluating its softmax cross entropy https://www.tensorflow.org/api$_docs/python/tf/nn/sparse\_softmax\_cross\_entropy\_with\_logits$
Then let's use the loss function to define a backpropogation step of our learning algorithm.

2.5 Watching the learning process

Having had no prior knowledge of the environment, the agent will begin to learn how to balance the pole on the cart based only on the feedback received from the environment! Having defined how our agent can move, how it takes in new observations, and how it updates its state, we'll see how it gradually learns a policy of actions to optimize balancing the pole as long as possible. You will find a part in the starter code that save a video of the trained model while it is balancing the pole, you can display it to see how your model learn.

2.6 Requirements

- Complete "create_cartpole_model" and "choose_action" to define the agent as described in section 2.2
- Complete "Memory.add_to_memory" to create the agent's memory as described in section 2.3
- Complete "compute_loss" and "train_step" to define the learning algorithm as described in section 2.4.
- You should try altering the model, and see the effect on accuracy and convergence time. Examples for the experiments(different number of layers, adding regularization, changing architecture (e.g. using CNNs)).

3 Pong (Optional)

In Cart Pole, we dealt with an environment that was static–in other words, it didn't change over time. What happens if our environment is dynamic and unpredictable? Well that's exactly the case in Pong, since part of the environment is our opposing player. We don't know how our opponent will act or react to our actions, so the complexity of our problem increases. It also becomes much more interesting, since we can compete to beat our opponent.

3.1 Define and inspect the environment

Similar to the cartpole we will define our environment, but this time the Observations are RGB image of shape (210, 160, 3).

We can again confirm the size of the observation space.
At every time step, the agent has six actions to choose from: noop, fire, move right, move left, fire right, and fire left. you can confirm the size of the action space by querying the environment.

3.2 Define the Agent

We'll define our agent again, but this time, we'll add convolutional layers to the network to increase the learning capacity of our network.
Since we've already defined the action function, choose_action(model, observation), we don't need to define it again. Instead, we'll be able to reuse it later on by passing in our new model we've just created, pong_model.

3.3 Helper Functions

We've already implemented some functions in Cartpole section, so we won't need to recreate them in this section. However, we might need to make some slight modifications. For example, we need to reset the reward to zero when a game ends. In Pong, we know a game has ended if the reward is +1 (we won!) or -1 (we lost unfortunately). Otherwise, we expect the reward at a timestep to be zero. Also, note that we've increased gamma from 0.95 to 0.99, so the rate of decay will be even more rapid.
The updated functions are:

- discount_rewards
- pre_process

Before we input an image into our network, we'll need to pre-process it by converting it into a 1D array of floating point numbers.
Then we will use the "pre_process" function to visualize what an observation might look like before and after pre-processing.

3.4 Training

We've already defined our loss function with compute_loss. If we want to use a different learning rate, though, we can reinitialize the optimizer.
We can also implement a very simple variant of plot_progress. In Pong, rather than feeding our network one image at a time, it can actually improve performance to input the difference between two consecutive observations, which really gives us information about the movement between frames. We'll first pre-process the raw observation, x, and then we'll compute the difference with the image frame we saw one timestep before. We'll also increase the number of maximum iterations from 1000 to 10000, since we expect it to take many more iterations to learn a more complex game.

3.5 Save and display video of training

After finishing the previous changes, We can now save the video of our model learning, and display the result.

3.6 Requirements

- Complete "create_pong_model" to define the agent as described in section 3.2.
- Complete the training loop to start training as described in section 3.4.
- You should try altering the model, and see the effect on accuracy and convergence time. Examples for the experiments(different number of layers, adding regularization, changing architecture).

4 Notes

- Note that it is better to use Python 2 notebook in this assignment.
- Parts of this assignment are based on MIT deep learning course.
- The starter code for the assignment can be found in the resources section in Piazza.
- You should deliver a report explaining all your work.
- Cheating will be severely penalized (for both parties). So, it is better to deliver nothing than deliver a copy. Any online resources used must be clearly identified.