

[Visit MIT Deep Learning](#) [Run in Google Colab](#) [View Source on GitHub](#)

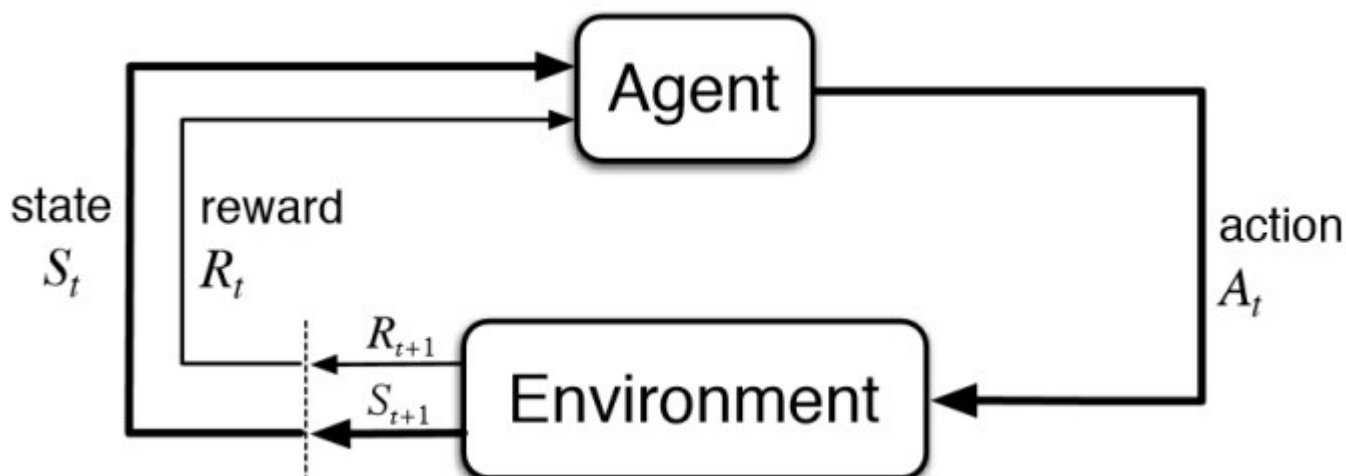
▼ Copyright Information

```
# Copyright 2021 MIT 6.S191 Introduction to Deep Learning. All Rights Reserved.  
#  
# Licensed under the MIT License. You may not use this file except in compliance  
# with the License. Use and/or modification of this code outside of 6.S191 must  
# reference:  
#  
# © MIT 6.S191: Introduction to Deep Learning  
# http://introtodeeplearning.com  
#
```

▼ Laboratory 3: Reinforcement Learning

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 a world, so they must learn to navigate environments by optimizing a reward function. Within an environment, 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 idea of "trial and error", or 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 timestep, receiving a corresponding reward, and updating its state according to what it has "learned".



While the ultimate goal of reinforcement learning is to teach agents to act in the real, physical world, games provide a convenient proving ground for developing RL algorithms and agents. Games have some properties that make them particularly well suited for RL:

1. In many cases, games have perfectly describable environments. For example, all rules of chess can be formally written and programmed into a chess game simulator;
2. Games are massively parallelizable. Since they do not require running in the real world, simultaneous environments can be run on large data clusters;
3. Simpler scenarios in games enable fast prototyping. This speeds up the development of algorithms that could eventually run in the real-world; and
4. ... Games are fun!

In previous labs, we have explored both supervised (with LSTMs, CNNs) and unsupervised / semi-supervised (with VAEs) learning tasks. Reinforcement learning is fundamentally different, in that we are training a deep learning algorithm to govern the actions of our RL agent, that is trying, within its environment, to find the optimal way to achieve a goal. The goal of training an RL agent is to determine the best next step to take to earn the greatest final payoff or return. In this lab, we focus on building a reinforcement learning algorithm to master two different environments with varying complexity.

1. **Cartpole**: Balance a pole, protruding from a cart, in an upright position by only moving the base left or right. Environment with a low-dimensional observation space.
2. **Pong**: Beat your competitors (whether other AI or humans!) at the game of Pong. Environment with a high-dimensional observation space -- learning directly from raw pixels.

Let's get started! First we'll import TensorFlow, the course package, and some dependencies

```
#Install some dependencies for visualizing the agents
```

```
!apt-get install -y xvfb python-opengl x11-utils > /dev/null 2>&1
```

```
!pip install gym pyvirtualdisplay scikit-video > /dev/null 2>&1
```

```
# Import Tensorflow 2.0
```

```
%tensorflow_version 2.x
```

```
import tensorflow as tf
```

```
import numpy as np
```

```
import base64, io, time, gym
```

```
import IPython, functools
```

```
import matplotlib.pyplot as plt
```

```
import time
```

```
from tqdm import tqdm
```

```
# Download and import the MIT 6.S191 package
```

```
!pip install mitdeeplearning
```

```
import mitdeeplearning as mdl
```

Collecting mitdeeplearning

```

Downloading https://files.pythonhosted.org/packages/9d/ad/650eb53c0d9d1213536fe94bc156
|████████████████████████████████████████| 2.1MB 9.8MB/s
Requirement already satisfied: numpy in /usr/local/lib/python3.7/dist-packages (from mitde
Requirement already satisfied: regex in /usr/local/lib/python3.7/dist-packages (from mitde
Requirement already satisfied: tqdm in /usr/local/lib/python3.7/dist-packages (from mitde
Requirement already satisfied: gym in /usr/local/lib/python3.7/dist-packages (from mitde
Requirement already satisfied: cloudpickle<1.7.0,>=1.2.0 in /usr/local/lib/python3.7/dist-pa
Requirement already satisfied: scipy in /usr/local/lib/python3.7/dist-packages (from gym
Requirement already satisfied: pygame<=1.5.0,>=1.4.0 in /usr/local/lib/python3.7/dist-pa
Requirement already satisfied: future in /usr/local/lib/python3.7/dist-packages (from py
Building wheels for collected packages: mitdeeplearning
  Building wheel for mitdeeplearning (setup.py) ... done
  Created wheel for mitdeeplearning: filename=mitdeeplearning-0.2.0-cp37-none-any.whl si
  Stored in directory: /root/.cache/pip/wheels/af/dc/2a/5c3633135e7e4ef4fd31463cfa1942cl
Successfully built mitdeeplearning
Installing collected packages: mitdeeplearning
Successfully installed mitdeeplearning-0.2.0

```

Before we dive in, let's take a step back and outline our approach, which is generally applicable to reinforcement learning problems in general:

1. **Initialize our environment and our agent:** here we will describe the different observations and actions the agent can make in the environment.
2. **Define our agent's memory:** this will enable the agent to remember its past actions, observations, and rewards.
3. **Define a reward function:** describes the reward associated with an action or sequence of actions.
4. **Define the learning algorithm:** this will be used to reinforce the agent's good behaviors and discourage bad behaviors.

▼ Part 1: Cartpole

3.1 Define the Cartpole environment and agent

Environment

In order to model the environment for both the Cartpole and Pong tasks, we'll be using a toolkit developed by OpenAI called [OpenAI Gym](https://gymnasium.openai.com/). 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 Cartpole environment, we can use `env = gym.make("CartPole-v0")`, which we gained access to when we imported the `gym` package. We can instantiate different [environments](#) by passing the environment name to the `make` function.

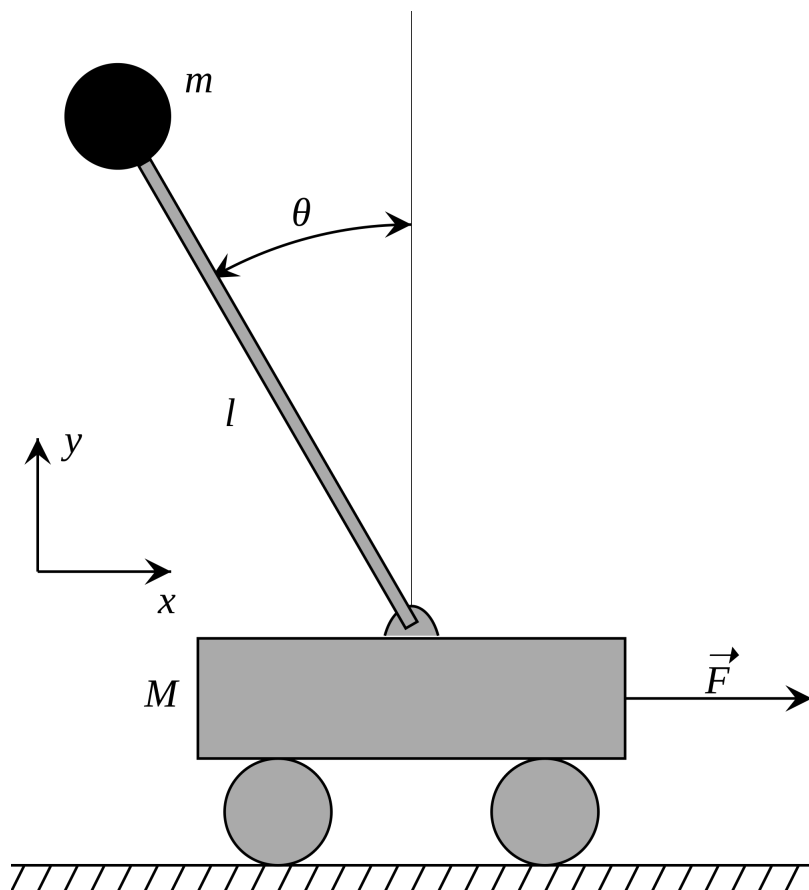
One issue we might experience when developing RL algorithms is that many aspects of the learning process are inherently random: initializing game states, changes in the environment, and the agent's actions. As such, it can be helpful to set a initial "seed" for the environment to ensure some level of reproducibility. 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.

```
### Instantiate the Cartpole environment ###
```

```
env = gym.make("CartPole-v0")
env.seed(1)
```

```
[1]
```

In Cartpole, a pole is attached by an un-actuated joint to a cart, which moves along a frictionless track. The pole starts upright, and the goal is to prevent it from falling over. The system is controlled by applying a force of +1 or -1 to the cart. A reward of +1 is provided for every timestep 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 of the track. A visual summary of the cartpole environment is depicted below:



Given this setup for the environment and the objective of the game, we can think about: 1) what observations help define the environment's state; 2) what actions the agent can take.

First, let's consider the observation space. In this Cartpole environment our observations are:

1. Cart position
2. Cart velocity
3. Pole angle
4. Pole rotation rate

We can confirm the size of the space by querying the environment's observation space:

```
n_observations = env.observation_space
print("Environment has observation space =", n_observations)

Environment has observation space = Box(-3.4028234663852886e+38, 3.4028234663852886e+38,
```

Second, we consider the action space. At every time step, the agent can move either right or left. Again we can confirm the size of the action space by querying the environment:

```
n_actions = env.action_space.n
print("Number of possible actions that the agent can choose from =", n_actions)

Number of possible actions that the agent can choose from = 2
```

▼ Cartpole agent

Now that we have instantiated the environment and understood the dimensionality of the observation and action spaces, we are ready to define our agent. In deep reinforcement learning, a deep neural network defines the agent. This network will take as input an observation of the environment and output the probability of taking each of the possible actions. Since Cartpole is defined by a low-dimensional observation space, a simple feed-forward neural network should work well for our agent. We will define this using the `Sequential` API.

```
### Define the Cartpole agent ###
```

```
# Defines a feed-forward neural network
```

```
def create_cartpole_model():
```

```
    model = tf.keras.models.Sequential([
```

```
        # First Dense layer
```

```
        tf.keras.layers.Dense(units=32, activation='relu'),
```

```
        # TODO: Define the last Dense layer, which will provide the network's output
```

```
# TODO: Define the last dense layer, which will provide the network's output.
# Think about the space the agent needs to act in!'''
# [TODO Dense layer to output action probabilities]
#tf.keras.layers.Dense(units=2, activation='softmax') # DONE (my bad)
tf.keras.layers.Dense(units=n_actions, activation=None) # DONE (corrected)

])
return model
```

```
cartpole_model = create_cartpole_model()
```

Now that we have defined the core network architecture, we will define an *action function* that executes a forward pass through the network, given a set of observations, and samples from the output. This sampling from the output probabilities will be used to select the next action for the agent. We will also add support so that the `choose_action` function can handle either a single observation or a batch of observations.

Critically, this action function is totally general -- we will use this function for both Cartpole and Pong, and it is applicable to other RL tasks, as well!

```
### Define the agent's action function ###
```

```
# Function that takes observations as input, executes a forward pass through model,
# and outputs a sampled action.
# Arguments:
# model: the network that defines our agent
# observation: observation(s) which is/are fed as input to the model
# single: flag as to whether we are handling a single observation or batch of
# observations, provided as an np.array
# Returns:
# action: choice of agent action
def choose_action(model, observation, single=True):
    # add batch dimension to the observation if only a single example was provided
    observation = np.expand_dims(observation, axis=0) if single else observation #compressed wa

    '''TODO: feed the observations through the model to predict the log
    probabilities of each possible action.'''
    #logits = model.predict(''TODO'')
    logits = model.predict(observation) #DONE

    '''TODO: Choose an action from the categorical distribution defined by the log
    probabilities of each possible action.'''
    #action = [''TODO'']
    #action = [logits[0] if single else logits] #it seems it is a list comprehension (MISTAKEN)
    action = tf.random.categorical(logits, num_samples=1) # DONE (CORRECTED)

    action = action.numpy().flatten()

    return action[0] if single else action #compressed way to do it!
```

▼ 3.2 Define the agent's memory

Now that we have instantiated the environment and defined the agent network architecture and action function, we are ready to move on to the next step in our RL workflow:

1. **Initialize our environment and our agent:** here we will describe the different observations and actions the agent can make in the environment.
2. **Define our agent's memory:** this will enable the agent to remember its past actions, observations, and rewards.
3. **Define the learning algorithm:** this will be used to reinforce the agent's good behaviors and discourage bad behaviors.

In reinforcement learning, training occurs alongside the agent's acting in the environment; an *episode* refers to a sequence of actions that ends in some terminal state, such as the pole falling down or the cart crashing. The agent will need to remember all of its observations and actions, such that once an episode ends, it can learn to "reinforce" the good actions and punish the undesirable actions via training. Our first step is to define a simple `Memory` buffer that contains the agent's observations, actions, and received rewards from a given episode. We will also add support to combine a list of `Memory` objects into a single `Memory`. This will be very useful for batching, which will help you accelerate training later on in the lab.

Once again, note the modularity of this memory buffer -- it can and will be applied to other RL tasks as well!

```
### Agent Memory ###
```

```
class Memory:
    def __init__(self):
        self.clear()

    # Resets/restarts the memory buffer
    def clear(self):
        self.observations = []
        self.actions = []
        self.rewards = []

    # Add observations, actions, rewards to memory
    def add_to_memory(self, new_observation, new_action, new_reward):
        self.observations.append(new_observation)

        '''TODO: update the list of actions with new action'''
        # TODO: your update code here
        self.actions.append(new_action) #DONE
```

```

    # TODO: update the list of rewards with new reward
    # TODO: your update code here
    self.rewards.append(new_reward) #DONE

# Helper function to combine a list of Memory objects into a single Memory.
# This will be very useful for batching.
def aggregate_memories(memories):
    batch_memory = Memory()

    for memory in memories:
        for step in zip(memory.observations, memory.actions, memory.rewards):
            batch_memory.add_to_memory(*step)

    return batch_memory

# Instantiate a single Memory buffer
memory = Memory()

```

▼ 3.3 Reward function

We're almost ready to begin the learning algorithm for our agent! The next step is to compute the rewards of our agent as it acts in the environment. Since we (and the agent) is uncertain about if and when the game or task will end (i.e., when the pole will fall), it is useful to emphasize getting rewards **now** rather than later in the future – this is the idea of discounting. This is a similar concept to discounting money in the case of interest. 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.

To compute the expected cumulative reward, known as the **return**, at a given timestep in a learning episode, we sum the discounted rewards expected at that time step t , within a learning episode, and projecting into the future. We define the return (cumulative reward) at a time step t , R_t as:

$$R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$$

where $0 < \gamma < 1$ is the discount factor and r_t is the reward at time step t , and the index k increments projection into the future within a single learning episode. Intuitively, you can think of this function as depreciating any rewards received at later time steps, which will force the agent prioritize getting rewards now. Since we can't extend episodes to infinity, in practice the computation will be limited to the number of timesteps in an episode – after that the reward is assumed to be zero.

Take note of the form of this sum – we'll have to be clever about how we implement this function. Specifically, we'll need to initialize an array of zeros, with length of the number of time steps, and fill it with the real discounted reward values as we loop through the rewards from the episode, which will have been saved in the agents memory. What we ultimately care about is which actions are

better relative to other actions taken in that episode -- so, we'll normalize our computed rewards, using the mean and standard deviation of the rewards across the learning episode.

```

#### Reward function ####

# Helper function that normalizes an np.array x
def normalize(x):
    x -= np.mean(x)
    x /= np.std(x)
    return x.astype(np.float32)

# Compute normalized, discounted, cumulative rewards (i.e., return)
# Arguments:
#   rewards: reward at timesteps in episode
#   gamma: discounting factor
# Returns:
#   normalized discounted reward
def discount_rewards(rewards, gamma=0.95):
    discounted_rewards = np.zeros_like(rewards)
    R = 0
    for t in reversed(range(0, len(rewards))):
        # update the total discounted reward
        R = R * gamma + rewards[t]
        discounted_rewards[t] = R

    return normalize(discounted_rewards)

```

▼ 3.4 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. In this lab, we will focus 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. We achieve this by simply **scaling** the probabilities by their associated rewards -- effectively amplifying the likelihood of actions that result in large rewards.

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](#). Like in supervised learning, we can use stochastic gradient descent methods to achieve the desired minimization.

Let's begin by defining the loss function.

```

#### Loss function ####

```

```

# Arguments:
#   logits: network's predictions for actions to take
#   actions: the actions the agent took in an episode
#   rewards: the rewards the agent received in an episode
# Returns:
#   loss
def compute_loss(logits, actions, rewards):
    '''TODO: complete the function call to compute the negative log probabilities'''
    #neg_logprob = tf.nn.sparse_softmax_cross_entropy_with_logits(
    #   logits='''TODO''', labels='''TODO''')
    neg_logprob = tf.nn.sparse_softmax_cross_entropy_with_logits(logits=logits, labels=actions)
    #tf.nn.sparse_softmax_cross_entropy_with_logits(labels, logits, name=None) # the order of t

    '''TODO: scale the negative log probability by the rewards'''
    #loss = tf.reduce_mean('''TODO''')
    loss = tf.reduce_mean(neg_logprob * rewards) # DONE

    return loss

```

Now let's use the loss function to define a training step of our learning algorithm:

```

### Training step (forward and backpropagation) ###

def train_step(model, optimizer, observations, actions, discounted_rewards):
    with tf.GradientTape() as tape:
        # Forward propagate through the agent network
        logits = model(observations)

        '''TODO: call the compute_loss function to compute the loss'''
        #loss = compute_loss('''TODO''', '''TODO''', '''TODO''')
        loss = compute_loss(logits, actions, discounted_rewards) # DONE

        '''TODO: run backpropagation to minimize the loss using the tape.gradient method'''
        #grads = tape.gradient('''TODO''', model.trainable_variables)
        grads = tape.gradient(loss, model.trainable_variables) #DONE
        optimizer.apply_gradients(zip(grads, model.trainable_variables))

```

▼ 3.5 Run cartpole!

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. To do this,

we'll track how the rewards evolve as a function of training -- how should the rewards change as

```
### Cartpole training! ###
```

```
# Learning rate and optimizer
```

```
learning_rate = 1e-3
```

```
optimizer = tf.keras.optimizers.Adam(learning_rate)
```

```
# instantiate cartpole agent
```

```
cartpole_model = create_cartpole_model()
```

```
# to track our progress
```

```
smoothed_reward = mdl.util.LossHistory(smoothing_factor=0.9)
```

```
plotter = mdl.util.PeriodicPlotter(sec=2, xlabel='Iterations', ylabel='Rewards')
```

```
if hasattr(tqdm, '_instances'): tqdm._instances.clear() # clear if it exists
```

```
for i_episode in range(500):
```

```
    plotter.plot(smoothed_reward.get())
```

```
    # Restart the environment
```

```
    observation = env.reset()
```

```
    memory.clear()
```

```
    while True:
```

```
        # using our observation, choose an action and take it in the environment
```

```
        action = choose_action(cartpole_model, observation)
```

```
        next_observation, reward, done, info = env.step(action)
```

```
        # add to memory
```

```
        memory.add_to_memory(observation, action, reward)
```

```
    # is the episode over? did you crash or do so well that you're done?
```

```
    if done:
```

```
        # determine total reward and keep a record of this
```

```
        total_reward = sum(memory.rewards)
```

```
        smoothed_reward.append(total_reward)
```

```
    # initiate training - remember we don't know anything about how the
```

```
    # agent is doing until it has crashed!
```

```
    train_step(cartpole_model, optimizer,
```

```
                observations=np.vstack(memory.observations),
```

```
                actions=np.array(memory.actions),
```

```
                discounted_rewards = discount_rewards(memory.rewards))
```

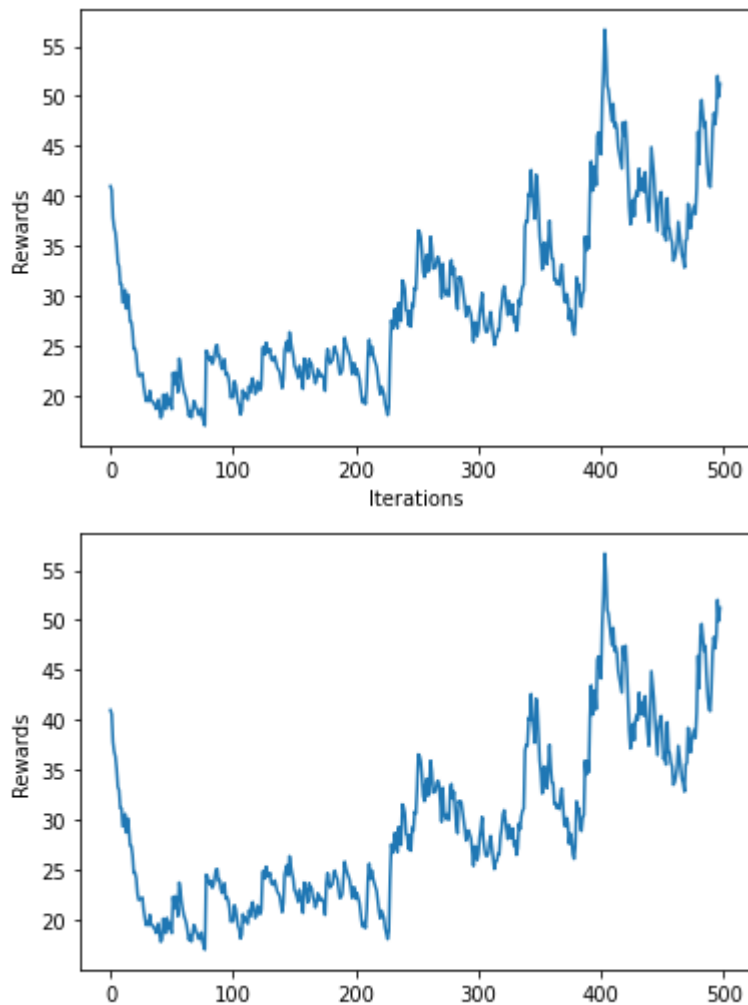
```
    # reset the memory
```

```
    memory.clear()
```

```
    break
```

```
# update our observatons
```

```
observation = next_observation
```



To get a sense of how our agent did, we can save a video of the trained model working on balancing the pole. Realize that this is a brand new environment that the agent has not seen before!

Let's display the saved video to watch how our agent did!

```
saved_cartpole = mdl.lab3.save_video_of_model(cartpole_model, "CartPole-v0")  
mdl.lab3.play_video(saved_cartpole)
```

Successfully saved 109 frames into CartPole-v0.mp4!

How does the agent perform? Could you train it for shorter amounts of time and still perform well? Do you think that training longer would help even more?

▼ Part 2: Pong

In Cartpole, 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 the 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. RL provides a powerful framework for training AI systems with the ability to handle and interact with dynamic, unpredictable environments. In this part of the lab, we'll use the tools and workflow we explored in Part 1 to build an RL agent capable of playing the game of Pong.

▼ 3.6 Define and inspect the Pong environment

As with Cartpole, we'll instantiate the Pong environment in the OpenAI gym, using a seed of 1.

```
def create_pong_env():  
    return gym.make("Pong-v0", frameskip=5)  
env = create_pong_env()  
env.seed(1); # for reproducibility
```

Let's next consider the observation space for the Pong environment. Instead of four physical descriptors of the cart-pole setup, in the case of Pong our observations are the individual video frames (i.e., images) that depict the state of the board. Thus, the observations are 210x160 RGB images (arrays of shape (210,160,3)).

We can again confirm the size of the observation space by query:

```
print("Environment has observation space =", env.observation_space)  
  
Environment has observation space = Box(0, 255, (210, 160, 3), uint8)
```

In Pong, at every time step, the agent (which controls the paddle) has six actions to choose from: no-op (no operation), move right, move left, fire, fire right, and fire left. Let's confirm the size of the action space by querying the environment:

```
n_actions = env.action_space.n
print("Number of possible actions that the agent can choose from =", n_actions)
```

```
Number of possible actions that the agent can choose from = 6
```

▼ 3.7 Define the Pong agent

As before, we'll use a neural network to define our agent. What network architecture do you think would be especially well suited to this game? Since our observations are now in the form of images, we'll add convolutional layers to the network to increase the learning capacity of our network. Note that you will be tasked with completing a template CNN architecture for the Pong agent -- but you should certainly experiment beyond this template to try to optimize performance!

```
### Define the Pong agent ###

# Functionally define layers for convenience
# All convolutional layers will have ReLu activation
Conv2D = functools.partial(tf.keras.layers.Conv2D, padding='same', activation='relu')
Flatten = tf.keras.layers.Flatten
Dense = tf.keras.layers.Dense

# Defines a CNN for the Pong agent
def create_pong_model():
    model = tf.keras.models.Sequential([
        # Convolutional layers
        # First, 32 5x5 filters and 2x2 stride
        Conv2D(filters=32, kernel_size=5, strides=2),

        #'''TODO: define convolutional layers with 48 5x5 filters and 2x2 stride'''
        # [your Conv layer here]
        Conv2D(filters=48, kernel_size=5, strides=2), #DONE

        #'''TODO: define two convolutional layers with 64 3x3 filters and 2x2 stride'''
        # [your Conv layers here]
        Conv2D(filters=64, kernel_size=3, strides=2),#DONE
        Conv2D(filters=64, kernel_size=3, strides=2),#DONE (it says TWO conv layers)

        Flatten(),

        # Fully connected layer and output
        Dense(units=128, activation='relu'),#DONE
```

```

'''TODO: define the output dimension of the last Dense layer.
#Pay attention to the space the agent needs to act in'''
# [TODO: your Dense layer here]
# Dense(units=6, activation=None) #DONE (it is better to use the var for units that has 6
Dense(units=n_actions, activation=None) #DONE (mistaken)

])
return model

pong_model = create_pong_model()

```

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`. This is awesome because our action function provides a modular and generalizable method for all sorts of RL agents!

▼ 3.8 Pong-specific functions

In Part 1 (Cartpole), we implemented some key functions and classes to build and train our RL agent -- `choose_action(model, observation)` and the `Memory` class, for example. However, in getting ready to apply these to a new game like Pong, we might need to make some slight modifications.

Namely, we need to think about what happens 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 -- the players (or agents) are just playing each other. So, after a game ends, we will need to reset the reward to zero when a game ends. This will result in a modified reward function.

```

### Pong reward function ###

# Compute normalized, discounted rewards for Pong (i.e., return)
# Arguments:
#   rewards: reward at timesteps in episode
#   gamma: discounting factor. Note increase to 0.99 -- rate of depreciation will be slower.
# Returns:
#   normalized discounted reward
def discount_rewards(rewards, gamma=0.99):
    discounted_rewards = np.zeros_like(rewards)
    R = 0
    for t in reversed(range(0, len(rewards))):
        # NEW: Reset the sum if the reward is not 0 (the game has ended!)
        if rewards[t] != 0:
            R = 0

```

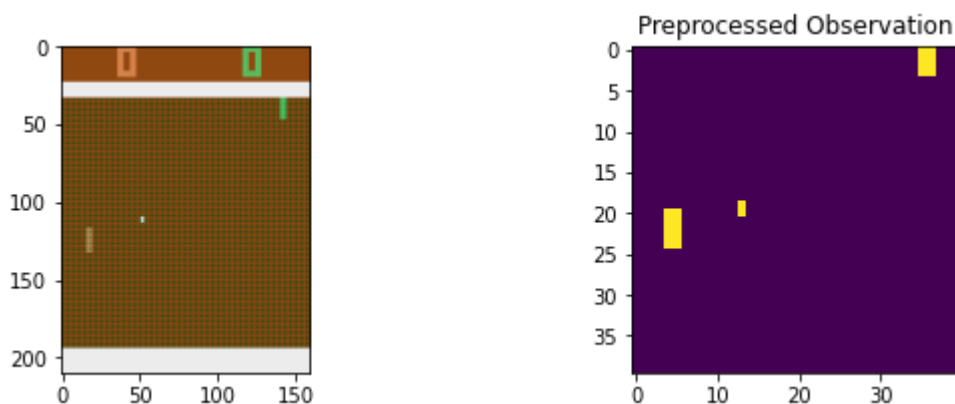
```
# update the total discounted reward as before
R = R * gamma + rewards[t]
discounted_rewards[t] = R
```

```
return normalize(discounted_rewards)
```

Additionally, we have to consider the nature of the observations in the Pong environment, and how they will be fed into our network. Our observations in this case are images. Before we input an image into our network, we'll do a bit of pre-processing to crop and scale, clean up the background colors to a single color, and set the important game elements to a single color. Let's use this function to visualize what a single observation might look like before and after pre-processing.

```
observation = env.reset()
for i in range(30):
    action = np.random.choice(n_actions)
    observation, _, _ = env.step(action)
observation_pp = mdl.lab3.preprocess_pong(observation)

f = plt.figure(figsize=(10,3))
ax = f.add_subplot(121)
ax2 = f.add_subplot(122)
ax.imshow(observation); ax.grid(False);
ax2.imshow(np.squeeze(observation_pp)); ax2.grid(False); plt.title('Preprocessed Observation')
```



Let's also consider the fact that, unlike CartPole, the Pong environment has an additional element of uncertainty -- regardless of what action the agent takes, we don't know how the opponent will play. That is, the environment is changing over time, based on *both* the actions we take and the actions of the opponent, which result in motion of the ball and motion of the paddles.

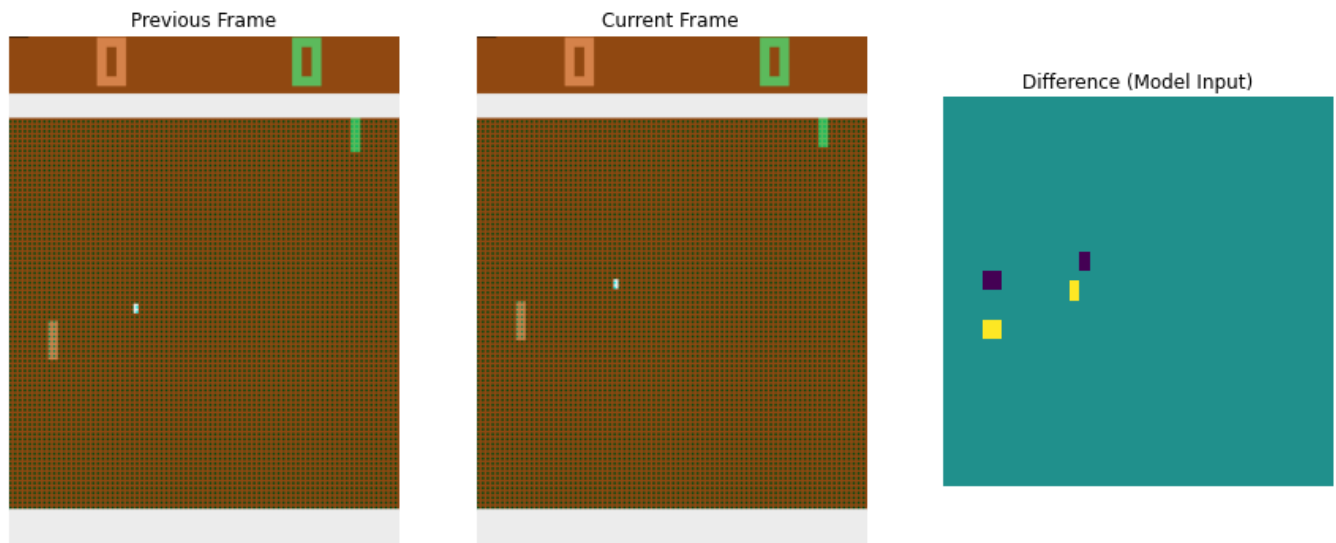
Therefore, to capture the dynamics, we also consider how the environment changes by looking at the difference between a previous observation (image frame) and the current observation (image frame). We've implemented a helper function, `pong_change`, that pre-processes two frames, calculates the change between the two, and then re-normalizes the values. Let's inspect this to visualize how the environment can change:


```

next_observation, __, __ = env.step(np.random.choice(n_actions))
diff = mdl.lab3.pong_change(observation, next_observation)

f, ax = plt.subplots(1, 3, figsize=(15,15))
for a in ax:
    a.grid(False)
    a.axis("off")
ax[0].imshow(observation); ax[0].set_title('Previous Frame');
ax[1].imshow(next_observation); ax[1].set_title('Current Frame');
ax[2].imshow(np.squeeze(diff)); ax[2].set_title('Difference (Model Input)');

```



What do you notice? How and why might these pre-processing changes be important for training our RL algorithm? How and why might consideration of the difference between frames be important for training and performance?

▼ Rollout function

We're now set up to define our key action algorithm for the game of Pong, which will ultimately be used to train our Pong agent. This function can be thought of as a "rollout", where the agent will 1) make an observation of the environment, 2) select an action based on its state in the environment, 3) execute a policy based on that action, resulting in some reward and a change to the environment, and 4) finally add memory of that action-reward to its `Memory` buffer. We will define this algorithm in the `collect_rollout` function below, and use it soon within a training block.

Earlier you visually inspected the raw environment frames, the pre-processed frames, and the difference between previous and current frames. As you may have gathered, in a dynamic game like

Pong, it can actually be helpful to consider the difference between two consecutive observations. This gives us information about the movement between frames -- how the game is changing. We will do this using the `pong_change` function we explored above (which also pre-processes frames for us).

We will use differences between frames as the input on which actions will be selected. These observation changes will be forward propagated through our Pong agent, the CNN network model, which will then predict the next action to take based on this observation. The raw reward will be computed. The observation, action, and reward will be recorded into memory. This will loop until a particular game ends -- the rollout is completed.

For now, we will define `collect_rollout` such that a batch of observations (i.e., from a batch of agent-environment worlds) can be processed serially (i.e., one at a time, in sequence). We will later utilize a parallelized version of this function that will parallelize batch processing to help speed up training! Let's get to it.

```
### Rollout function ###

# Key steps for agent's operation in the environment, until completion of a rollout.
# An observation is drawn; the agent (controlled by model) selects an action;
# the agent executes that action in the environment and collects rewards;
# information is added to memory.
# This is repeated until the completion of the rollout -- the Pong game ends.
# Processes multiple batches serially.
#
# Arguments:
# batch_size: number of batches, to be processed serially
# env: environment
# model: Pong agent model
# choose_action: choose_action function
# Returns:
# memories: array of Memory buffers, of length batch_size, corresponding to the
# episode executions from the rollout
def collect_rollout(batch_size, env, model, choose_action):

    # Holder array for the Memory buffers
    memories = []

    # Process batches serially by iterating through them
    for b in range(batch_size):

        # Instantiate Memory buffer, restart the environment
        memory = Memory()
        next_observation = env.reset()
        previous_frame = next_observation
        done = False # tracks whether the episode (game) is done or not
```

```
while not done:
```

```
    current_frame = next_observation
```

```
    '''TODO: determine the observation change.
```

```
    Hint: this is the difference between the past two frames'''
```

```
    #frame_diff = # TODO
```

```
    #frame_diff = current_frame - previous_frame # DONE (my bad, because that is performed  
    frame_diff = mdl.lab3.pong_change(previous_frame, current_frame) # TODO (corrected)
```

```
    '''TODO: choose an action for the pong model, using the frame difference, and evaluate'
```

```
    #action = # TODO
```

```
    #action = choose_action (pong_model, frame_diff) # DONE (my bad because model is the ge  
    action = choose_action(model, frame_diff) # DONE (corrected)
```

```
    # Take the chosen action
```

```
    next_observation, reward, done, info = env.step(action)
```

```
    '''TODO: save the observed frame difference, the action that was taken, and the resulti
```

```
    #memory.add_to_memory(''''TODO''', ''''TODO''', ''''TODO''')
```

```
    #memory.add_to_memory(next_observation, action, reward) # DONE (my bad because the obvs  
    memory.add_to_memory(frame_diff, action, reward) # DONE (CORRECTED)
```

```
    previous_frame = current_frame
```

```
    # Add the memory from this batch to the array of all Memory buffers
```

```
    memories.append(memory)
```

```
return memories
```

To get a sense of what is encapsulated by `collect_rollout`, we will instantiate an *untrained* Pong model, run a single rollout using this model, save the memory, and play back the observations the model sees. Note that these will be frame *differences*.

```
### Rollout with untrained Pong model ###
```

```
# Model
```

```
test_model = create_pong_model()
```

```
# Rollout with single batch
```

```
single_batch_size = 1
```

```
memories = collect_rollout(single_batch_size, env, test_model, choose_action)
```

```
rollout_video = mdl.lab3.save_video_of_memory(memories[0], "Pong-Random-Agent.mp4")
```

```
# Play back video of memories
```

```
mdl.lab3.play_video(rollout_video)
```



0:29 / 0:29

▼ 3.9 Training Pong

We're now all set up to start training our RL algorithm and agent for the game of Pong! We've already defined the following:

1. Loss function, `compute_loss`, and backpropagation step, `train_step`. Our loss function employs policy gradient learning. `train_step` executes a single forward pass and backpropagation gradient update.
2. RL agent algorithm: `collect_rollout`. Serially processes batches of episodes, executing actions in the environment, collecting rewards, and saving these to `Memory`.

We will use these functions to train the Pong agent.

In the training block, episodes will be executed by agents in the environment via the RL algorithm defined in the `collect_rollout` function. Since RL agents start off with literally zero knowledge of their environment, it can often take a long time to train them and achieve stable behavior. To alleviate this, we have implemented a parallelized version of the RL algorithm, `parallelized_collect_rollout`, which you can use to accelerate training across multiple parallel batches.

For training, information in the `Memory` buffer from all these batches will be aggregated (after all episodes, i.e., games, end). Discounted rewards will be computed, and this information will be used to execute a training step. Memory will be cleared, and we will do it all over again!

Let's run the code block to train our Pong agent. Note that, even with parallelization, completing training and getting stable behavior will take quite a bit of time (estimated at least a couple of hours). We will again visualize the evolution of the total reward as a function of training to get a sense of how the agent is learning.

```
### Hyperparameters and setup for training ###
# Rerun this cell if you want to re-initialize the training process
# (i.e., create new model, reset loss, etc)

# Hyperparameters
learning_rate = 1e-3
MAX_ITERS = 1000 # increase the maximum to train longer
batch_size = 5 # number of batches to run

# Model, optimizer
pong_model = create_pong_model()
optimizer = tf.keras.optimizers.Adam(learning_rate)
iteration = 0 # counter for training steps

# Plotting
smoothed_reward = mdl.util.LossHistory(smoothing_factor=0.9)
smoothed_reward.append(0) # start the reward at zero for baseline comparison
plotter = mdl.util.PeriodicPlotter(sec=15, xlabel='Iterations', ylabel='Win Percentage (%)')

# Batches and environment
# To parallelize batches, we need to make multiple copies of the environment.
envs = [create_pong_env() for _ in range(batch_size)] # For parallelization

### Training Pong ###
# You can run this cell and stop it anytime in the middle of training to save
# a progress video (see next codeblock). To continue training, simply run this
# cell again, your model will pick up right where it left off. To reset training,
# you need to run the cell above.

games_to_win_episode = 21 # this is set by OpenAI gym and cannot be changed.

# Main training loop
while iteration < MAX_ITERS:

    plotter.plot(smoothed_reward.get())

    tic = time.time()
    # RL agent algorithm. By default, uses serial batch processing.
    # memories = collect_rollout(batch_size, env, pong_model, choose_action)
```

```

# Parallelized version. Uncomment line below (and comment out line above) to parallelize
memories = mdl.lab3.parallelized_collect_rollout(batch_size, envs, pong_model, choose_actio
print(time.time()-tic)

# Aggregate memories from multiple batches
batch_memory = aggregate_memories(memories)

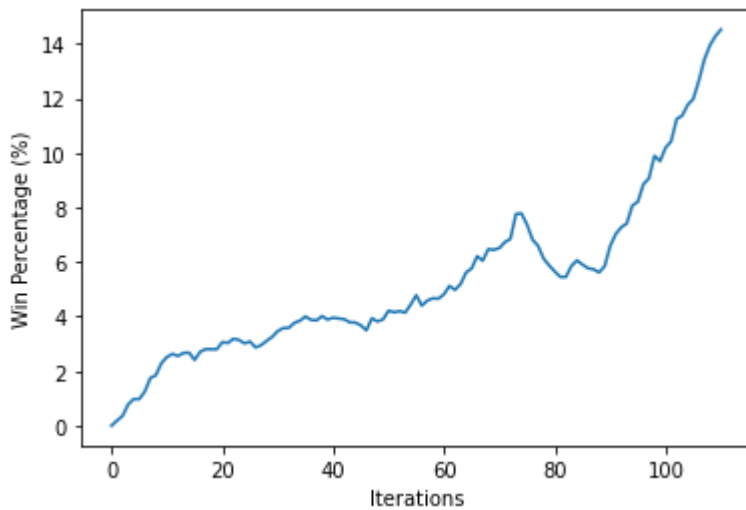
# Track performance based on win percentage (calculated from rewards)
total_wins = sum(np.array(batch_memory.rewards) == 1)
total_games = sum(np.abs(np.array(batch_memory.rewards)))
win_rate = total_wins / total_games
smoothed_reward.append(100 * win_rate)

# Training!
train_step(
    pong_model,
    optimizer,
    observations = np.stack(batch_memory.observations, 0),
    actions = np.array(batch_memory.actions),
    discounted_rewards = discount_rewards(batch_memory.rewards)
)

# Save a video of progress -- this can be played back later
if iteration % 100 == 0:
    mdl.lab3.save_video_of_model(pong_model, "Pong-v0",
                                suffix="_"+str(iteration))

iteration += 1 # Mark next episode

```



```

KeyboardInterrupt                                Traceback (most recent call last)
<ipython-input-33-e13200705475> in <module>()
    17
    18 # Parallelized version. Uncomment line below (and comment out line above) to
parallelize
--> 19 memories = mdl.lab3.parallelized_collect_rollout(batch_size, envs,
pong_model, choose_action)
    20 print(time.time()-tic)
    21

```

30 frames

`/usr/local/lib/python3.7/dist-packages/tensorflow/python/framework/ops.py` in

Finally we can put our trained agent to the test! It will play in a newly instantiated Pong environment against the "computer", a base AI system for Pong. Your agent plays as the green paddle. Let's watch the match instant replay!

```

1794     for k, v in six.iteritems(attrs):

latest_pong = mdl.lab3.save_video_of_model(
    pong_model, "Pong-v0", suffix="_latest")
mdl.lab3.play_video(latest_pong, width=400)

```

Successfully saved 2503 frames into Pong-v0_latest.mp4!

▼ 3.10 Conclusion and submission information

That's it! Congratulations on training two RL agents and putting them to the test! We encourage you to consider the following:

- How does the agent perform?
- Could you train it for shorter amounts of time and still perform well?
- What are some limitations of the current representation i.e., difference of current and previous frames? How is this reflected in the agent's behavior? What could be done to generate an improved representation?
- How does the complexity of Pong relative to CartPole alter the rate at which the agent learns and its performance?
- What are some things you could change about the agent or the learning process to potentially improve performance?

Try to optimize your **Pong** model and algorithm to achieve improved performance. **MIT students and affiliates will be eligible for prizes during the IAP offering.** To enter the competition, MIT students and affiliates should upload the following to the course Canvas:

- Jupyter notebook with the code you used to generate your results, **with the Pong training executed**;
- saved video of your Pong agent competing;
- a description and/or diagram of the architecture, settings, and hyperparameters you used to generate your result -- if there are any additional or interesting modifications you made to the template code, please include these in your description;
- discussion of why these modifications helped improve performance.

Good luck!

Finished on March-18th-2021

(Student: *Andrés Felipe Escallón Portilla*)

✓ 13 s completado a las 17:46 ● ✕