Keith Saver

Reinforcement Learning for Games and Simulations

Problem Statement:
The purpose of this project is to analyze the effect of adjusting the learning rate of a neural net used in reinforcement learning. A Deep Q-Network is used here due to its proven ability to be effective in games and simulations as shown by DeepMind. The OpenAI Gym is used to supply a variety of standardized learning environments.

Dataset:
The dataset here is generated by the OpenAI environments as the game is played and the model trains. Three environments are used here: Cartpole-v1, Acrobot-v1, and SpaceInvaders-ram-v0.

High Level Overview of Steps:
Reinforcement learning is powerful due to its ability to learn strategies in many different environments with a standardized initial model. Performance is compared across three different environments. Three different learning rates are used for each environment in order to compare how changing the learning rate affects both the model's scores and convergence. The environments were selected to provide different levels of complexity.

Hardware:
Windows 10 with i7 - 6700k CPU @ 4.00GHz, 32GB RAM

Software:
Python 3.6, Tensorflow 1.13.1, Keras 2.2.4, Gym, atari-py (https://github.com/Kojoley/atari-py)

References:
https://github.com/keon/deep-q-learning,
https://towardsdatascience.com/cartpole-introduction-to-reinforcement-learning-ed0eb5b58288

Lesson Learned and Pros/Cons
For both cartpole and acrobot, the lowest learning rate performed best, and higher learning rates resulted in lower scores. Higher learning rates seem to to overtrain the model at points, not letting it slowly converge on a more effective strategy. The model struggled to learn on space invaders, as the environment is very complex (even with 15+ hours of training).

Installation Instructions:

As outlined above, Python 3.6.8 was used for this project. Python 3.6 can be downloaded here: https://www.python.org/downloads/release/python-360/. Matplotlib, NumPy, Tensorflow, Keras, and Gym are all required for this project. Pip was used to install all of these packages via the command prompt, eg: "pip install keras".

Installing the gym was slightly more complicated, as not all environments and packages associated with the OpenAI Gym are supported for Windows operating systems. First, install gym via "pip install gym", then install packages for Atari by running "pip install gym[atari]". However, not all Gym packages to run Atari simulations are supported on Windows. However, there is an OpenAI atari-py fork for Windows that can be found here: https://github.com/Kojoley/atari-py. This was installed using "pip install atari-py", which allowed for Atari OpenAI environments to be run on Windows 10.

Python and package versions:

```
Python 3.6.8 (tags/v3.6.8:3c6b436a57, Dec 24 2018, 00:16:47) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import numpy as np
>>> print(np.__version__)
1.16.1
>>> import tensorflow as tf
>>> print(tf.__version__)
1.13.1
>>> import keras
Using TensorFlow backend.
>>> print(keras.__version__)
2.2.4
>>> import gym
>>> print(gym.__version__)
0.12.1
>>> import matplotlib
>>> print(matplotlib.__version__)
3.0.2
```

Data/Environments:

For this project, there is not a datafile, as reinforcement learning was instead performed on games and simulations. The OpenAI Gym was used to provide environments: http://gym.openai.com/envs/#classic_control. Three different environments were used: Cartpole-v1, Acrobot-v1, and SpaceInvaders-ram-v0. Cartpole and Acrobot are both environments from the Classic Control section, and are comparatively simple. The Space Invaders environment is a simulation of the Atari game Space Invaders. In this case, the

environment grants the program (in this case, the DQN reinforcement learning model) access to Space Invader's RAM.

There is another Space Invaders environment in which the observation is the game's pixels instead. However, the reason the RAM version was used here was to allow for the exact same reinforcement learning model to be used for Cartpole, Acrobot, and Space Invaders. The version of space invaders that provides access to the game's pixels would likely use a convolutional neural net for training, but this is not compatible with Cartpole and Acrobot. By picking three different environments that are all compatible with the same DQN model, it is possible to isolate varying the learning rate while keeping all other facets of the DQN model the same.

Due to the ease of using the OpenAI Gym and as a result of how the environments were selected, there was really no data wrangling that needed to be done here. The environment returns information about the current state of the game, as well as the reward granted, which is all the information that the reinforcement model needs.

Deep Q-Network Model:

I studied a number of DQN implementations online, and adjusted them slightly to fit my needs for this project. The two DQN implementations I took the most inspiration from can be found here: https://github.com/gsurma/cartpole/blob/master/cartpole.py, https://github.com/keon/deep-q-learning

Note that both of these DQN implementations are designed for the OpenAI Cartpole game. Parameters and portions of the algorithm were adjusted to change how the model learns and interacts with the three selected environments. Changes will be discussed below with code.

The script dqn_dev.py runs the DQN algorithm and prints results to csv's, as described below:

First, packages are imported, and parameters and other information is provided:

```
import gym
import numpy as np
import random
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import Adam
from collections import deque
import csv

#List of OpenAI Gym environments for testing
environments = ["CartPole-v1", "Acrobot-v1", "SpaceInvaders-ram-v0"]

discount_rate = 0.95

#List of learning rates to test
learning_rates = [0.001, 0.003, 0.005]

state_mem_max = 1000000
batch_size = 50

init_rand = 1.0
min_rand = 0.01
rand_decay = 0.999

episodes = 500
```

As seen above, the three environments and three learning rates are all defined here. The above linked implementations both used learning rates of 0.001. The higher learning rates here are used in order to test how training is affected by varying the training rate. Reinforcement learning models often struggle to converge (either take a very long time to converge on a strategy, or may not converge onto the correct strategy), so it is likely that altering the learning rate can affect this. Additionally, Acrobot and especially Space Invaders are more complex and more difficult to learn than Cartpole, so the increased learning rate may affect training differently depending on the complexity of the environment.

The discount rate is defined, which is used to value future rewards lower than immediate rewards. A maximum number of states for the model to remember is defined, as well as a batch size, which determines how many of those states are "remembered" in each iteration of training. A batch size of 50 is used here, which is higher than in many models found online. This was done in an attempt to improve learning in the more complex environments. Similarly, the rate of decay of explorations was decreased compared to many other implementations to try to increase early exploration in order to improve training, particularly in the more complex environments.

```python
#Implement class for DQN algorithm
class DQN_Alg:
    #Learning rate must be passed in as a parameter
    def __init__(self, obs_space, action_space, learning_rate):
        self.rand_rate = init_rand
        self.action_space = action_space
        self.learning_rate = learning_rate
        self.memory = deque(maxlen = state_mem_max)
        self.model = Sequential()
        self.model.add(Dense(48, input_shape=(obs_space,), activation="relu"))
        self.model.add(Dense(48, activation="relu"))
        self.model.add(Dense(self.action_space, activation="linear"))
        self.model.compile(loss="mse", optimizer=Adam(lr=learning_rate))

    #Append state to memory
    def remember(self, state, action, reward, next_state, done):
        self.memory.append((state, action, reward, next_state, done))

    #Perform action given a state
    def act(self, state):
        #Perform random actions, depending random rate
        if np.random.rand() < self.rand_rate:
            return random.randrange(self.action_space)
        #Else act as to maximize predicted reward
        q_vals = self.model.predict(state)
        return np.argmax(q_vals[0])

    #Replay batch of history for training
    def replay(self):
        if len(self.memory) < batch_size:
            return
        batch = random.sample(self.memory, batch_size)
        for state, action, reward, next_state, done in batch:
            q = reward
            if not done:
                q = reward + discount_rate * np.amax(self.model.predict(next_state)[0])
            q_vals = self.model.predict(state)
            q_vals[0][action] = q
            self.model.fit(state, q_vals, verbose = 0)
        #Decrease random rate as learning occurs
        self.rand_rate *= rand_decay
        self.rand_rate = max(min_rand, self.rand_rate)
```

The above class defines a DQN model. A few changes have been made in this model from other implementations. A learning rate must now be passed in during the initialization of the DQN solver, as this project aims to analyze the effects of varied learning rates. Additionally, the number of neurons in each dense layer of the model has been increased. During testing, the model struggled to improve performance significantly in both the Acrobot and Space Invaders environments, so more complexity was added in hopes that this could improve training (batch size was increased for the same reason, as described above).

```python
#Perform training on all environments with all learning rates
def play_envs():
    for environment in environments:
        for learning_rate in learning_rates:
            env = gym.make(environment)
            observation_space = env.observation_space.shape[0]
            action_space = env.action_space.n
            dqn = DQN_Alg(observation_space, action_space, learning_rate)
            results = []
            for run_num in range(episodes):
                state = env.reset()
                state = np.reshape(state, [1, observation_space])
                step_count = 0
                score = 0
                done = False
                while not done:
                    #env.render()
                    #Have model perform an action
                    action = dqn.act(state)
                    #Update state based on the action
                    next_state, reward, done, _ = env.step(action)
                    score += reward
                    #Negative reward if game is finished (game loss)
                    if done:
                        reward = -10
                    #Set all positive rewards to 1 and all negative rewards to -1
                    reward = np.sign(reward)
                    next_state = np.reshape(next_state, [1, observation_space])
                    dqn.remember(state, action, reward, next_state, done)
                    state = next_state
                    if done:
                        results.append([run_num + 1, dqn.rand_rate, score])
                        break
                    dqn.replay()
            #Print results of training with a learning rate on a specific environment to a labeled file
            with open(environment+"_" + str(learning_rate) + ".csv", "w", newline = "") as file:
                writer = csv.writer(file)
                writer.writerows(results)

if __name__ == "__main__":
    play_envs()
```

The play_envs() function carries out all simulations. It loops through each environment and learning rate for a total of 9 runs of 500 episodes each. The OpenAI environments make it very easy to pass information to and from the DQL model. The observation and action space variables are used to pass information about the environment to the DQN, and to allow the DQN to pass an action back into the environment. Nothing specific needs to be known about the observation space, allowing for the same DQN model to be run on all three environments. The environment also returns a reward from the current state, as well as a boolean flag that returns true if the simulation has finished (eg: the model has lost the game).

Here I decided to pass all positive rewards back to the DQN as 1, and all negative rewards as -1, unless the game was over, in which case -10 was returned. The reason is that in some cases, very large rewards may be passed back that are not very consistent or reliable, and thus may lead to

ineffective training. For example, in Space Invaders, there is occasionally a bonus that if shot, grants the player a large number of bonus points. However, this bonus appears randomly and rarely, so the model may train badly if it over values shooting this bonus.

For each iteration of running 500 episodes on a specific environment with a given learning rate, the run numbers, random action rates, and scores are recorded. This list of 500 rows is then printed to a CSV that is named according to learning rate and environment.

A second script data.py is used to read the data from the CSV's generated by dqn_dev.py and print charts using matplotlib.

```
import csv
import matplotlib.pyplot as plt
import matplotlib.ticker as ticker


filenames = ["CartPole-v1_0.001.csv", "CartPole-v1_0.003.csv", "CartPole-v1_0.005.csv", \
             "Acrobot-v1_0.001.csv", "Acrobot-v1_0.003.csv", "Acrobot-v1_0.005.csv", \
             "SpaceInvaders-ram-v0_0.001.csv", "SpaceInvaders-ram-v0_0.003.csv", "SpaceInvaders-ram-v0_0.005.csv"]

envs = ["Cartpole", "Acrobot", "Space Invaders"]

learning_rates = ["0.001", "0.003", "0.005"]

#Read data from CSV to list of lists
def csv_to_list(file):
    data = []
    with open(file, newline = "") as f:
        reader = csv.reader(f)
        for row in reader:
            data.append(row)
    return data
```

The filenames are defined, matching the CSV's created previously. Lists of the environments and learning rates are defined for use in labeling charts. Then, a function is defined to read the CSV's data to a list of lists.

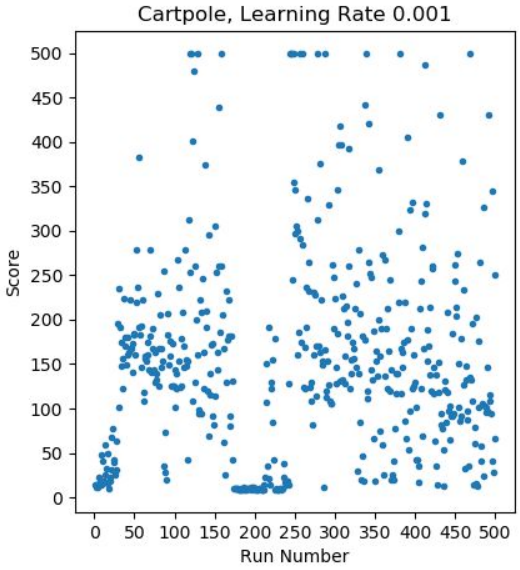
```
#Calculate running averages, append to each row
def running_avgs(data):
    sum_scores = 0
    for row_num in range(len(data)):
        sum_scores += float(data[row_num][2])
        data[row_num].append(sum_scores/(row_num + 1))
    return data
```

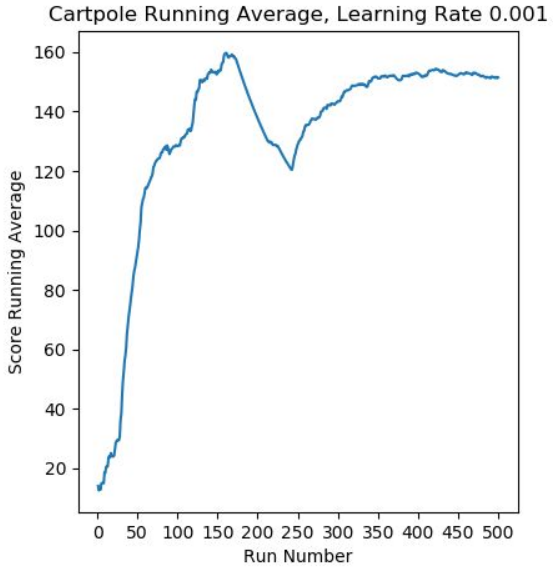
A function is defined to calculate the running average of scores, and appends this data to the existing data for each row.

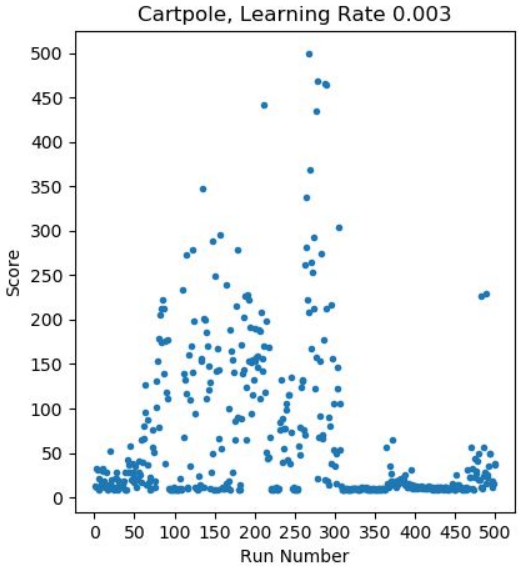
```
#Print scores and running average per run for each environment learning rate pair
def print_scores(data, num):
    run_num = [float(row[0]) for row in data]
    scores = [float(row[2]) for row in data]
    avgs = [row[3] for row in data]

    env = envs[int(num/3)]
    lrate = learning_rates[num%3]

    f, ax = plt.subplots(1, 2, figsize = (10, 5))
    ax[0].scatter(run_num, scores, marker = ".")
    ax[0].set_title(env +', Learning Rate ' + lrate)
    ax[0].xaxis.set_major_locator(ticker.MultipleLocator(50))
    ax[0].yaxis.set_major_locator(ticker.MultipleLocator(50))
    ax[0].set_xlabel('Run Number')
    ax[0].set_ylabel('Score')
    ax[1].plot(run_num, avgs)
    ax[1].set_title(env +' Running Average, Learning Rate ' + lrate)
    ax[1].xaxis.set_major_locator(ticker.MultipleLocator(50))
    ax[1].yaxis.set_major_locator(ticker.MultipleLocator(20))
    ax[1].set_xlabel('Run Number')
    ax[1].set_ylabel('Score Running Average')
    #plt.show()
    plt.savefig(env + lrate + "_graphs.png")

#Run for all CSV's
for i in range(len(filenames)):
    data = csv_to_list(filenames[i])
    data = running_avgs(data)
    print_scores(data, i)
```

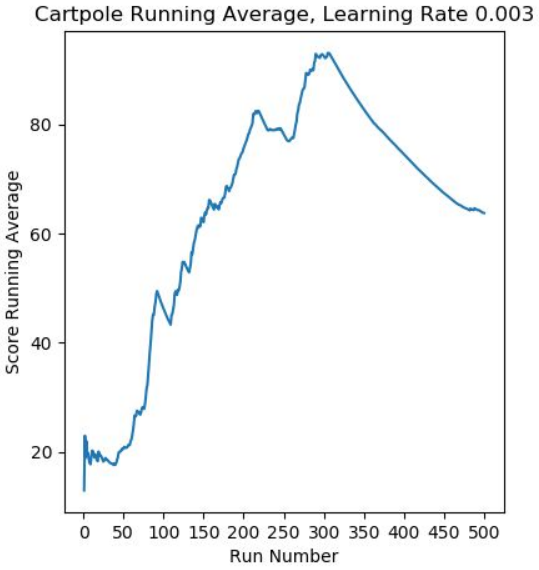
A function is defined that takes in data and an identifier for the file, and uses the matplotlib library to print charts of scores and running average scores by run number. The graphs are then saved to a file, named according to environment and learning rate.

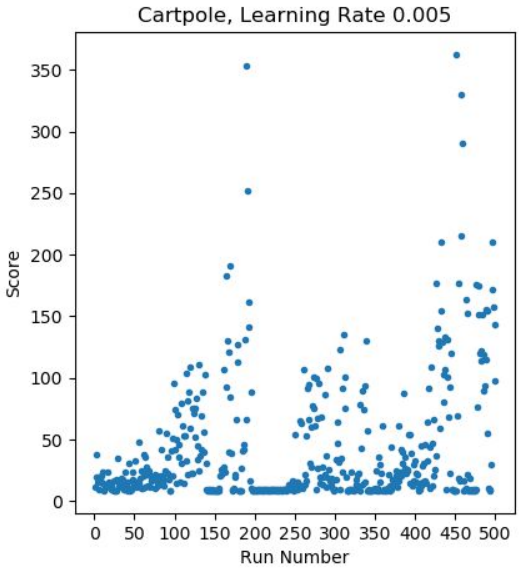
Results:

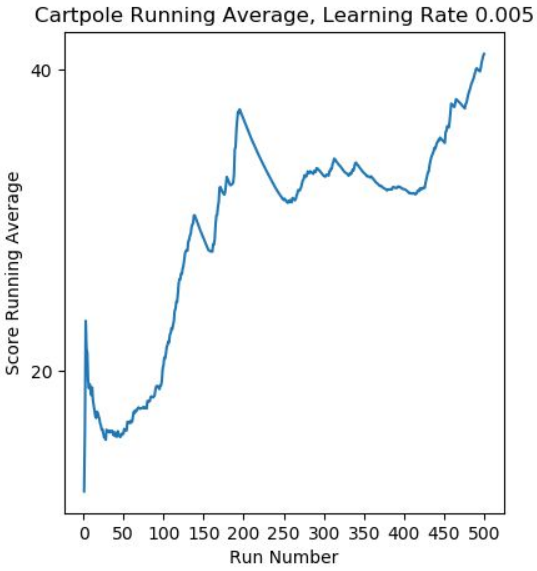

Cartpole, Learning Rate 0.001
Score
Run Number

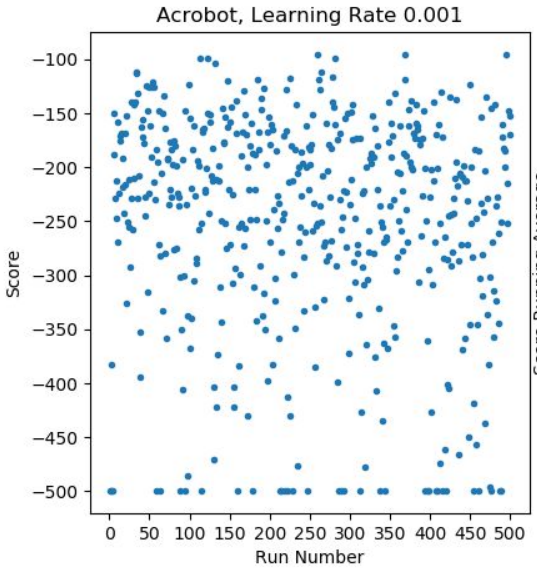

Cartpole Running Average, Learning Rate 0.001
Score Running Average
Run Number

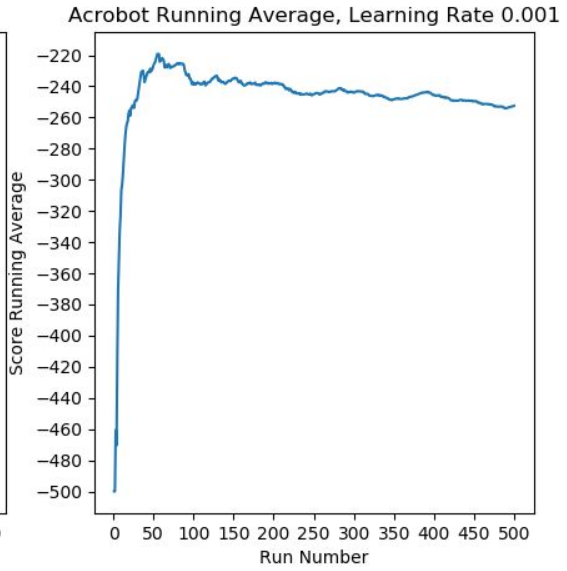

Cartpole, Learning Rate 0.003
Score
Run Number

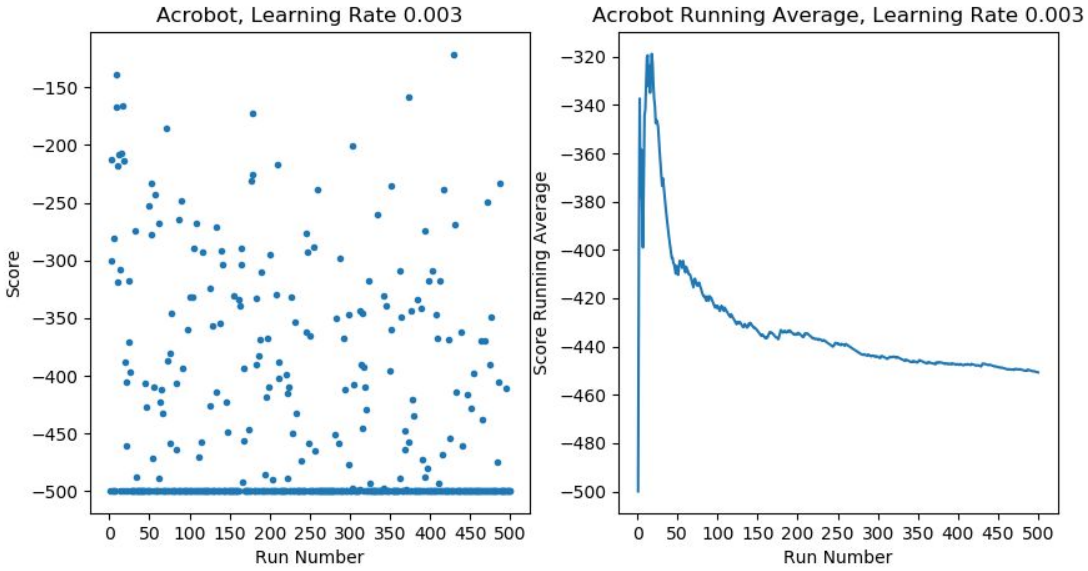

Cartpole Running Average, Learning Rate 0.003
Score Running Average
Run Number

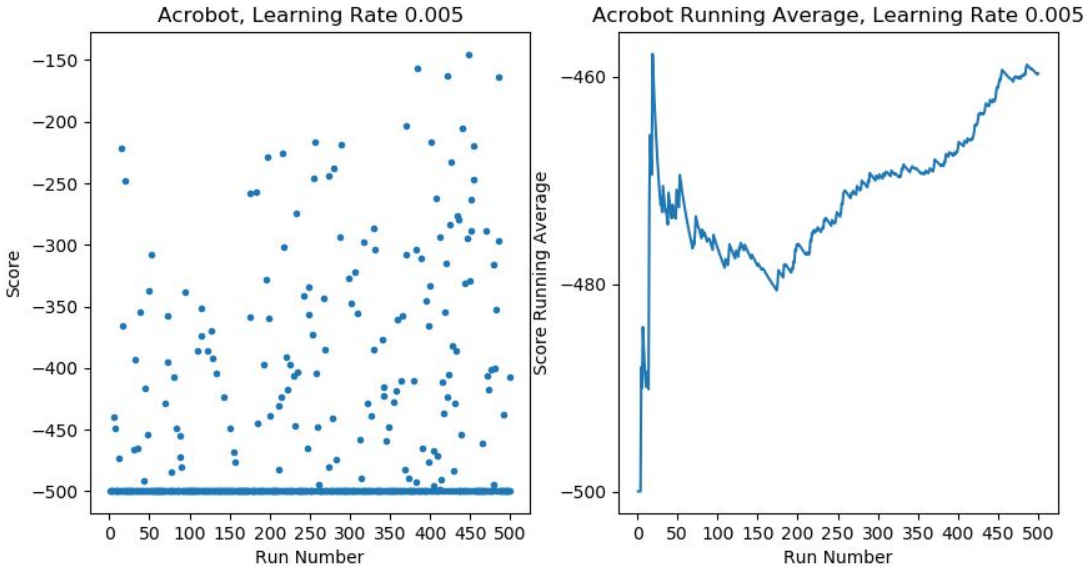

Cartpole, Learning Rate 0.005
Score
Run Number

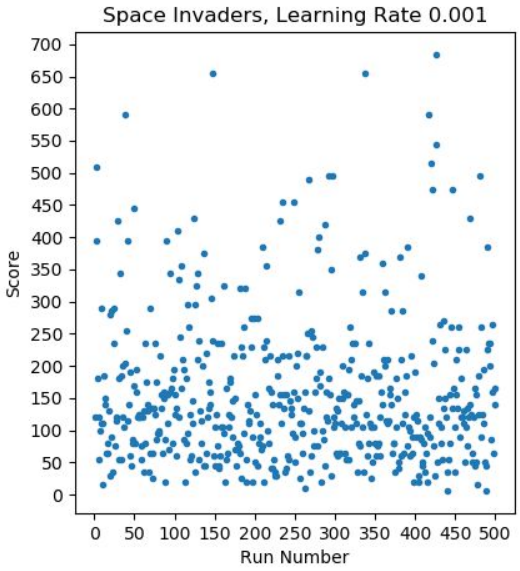

Cartpole Running Average, Learning Rate 0.005
Score Running Average
Run Number

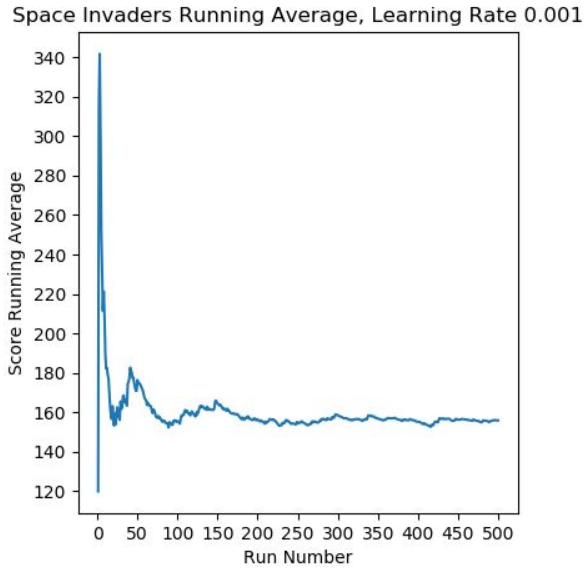

Acrobot, Learning Rate 0.001
Score
Run Number

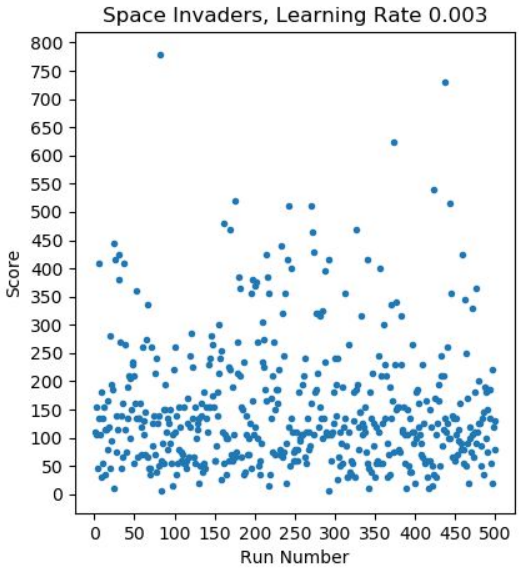

Acrobot Running Average, Learning Rate 0.001
Score Running Average
Run Number

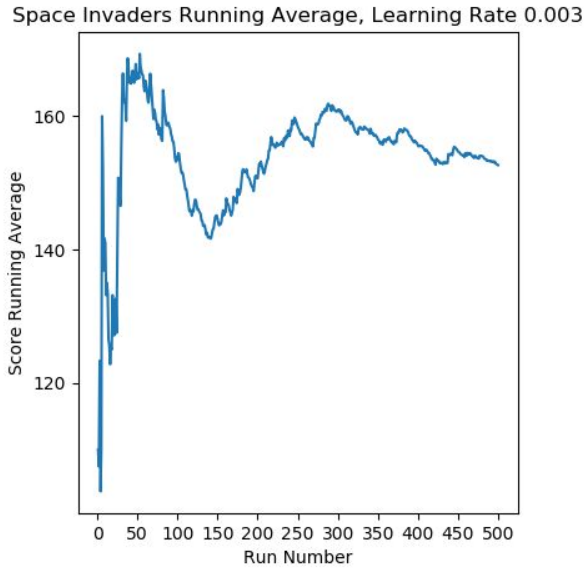

Acrobot, Learning Rate 0.003
Score
Run Number
Acrobot Running Average, Learning Rate 0.003
Score Running Average
Run Number

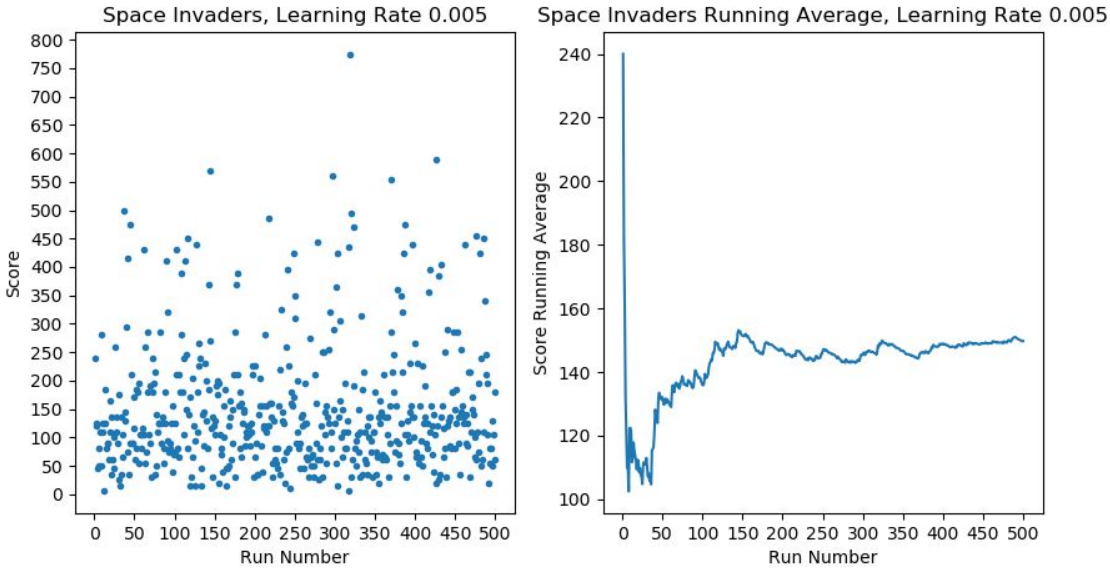

Acrobot, Learning Rate 0.005
Score
Run Number
Acrobot Running Average, Learning Rate 0.005
Score Running Average
Run Number


Space Invaders, Learning Rate 0.001
Score
Run Number


Space Invaders Running Average, Learning Rate 0.001
Score Running Average
Run Number


Space Invaders, Learning Rate 0.003
Score
Run Number


Space Invaders Running Average, Learning Rate 0.003
Score Running Average
Run Number



For each environment learning rate pair, a scatter plot of scores by run number is printed, as well as the running average over the entire 500 runs.

For the Cartpole runs, the lowest learning (0.001) clearly has the best scores, with a running average hovering around 150 from around 150 runs forward. The DQN clearly learns quickly, and generally maintains a high average score. However, there is still a section from around runs 160-220 with very low scores. Reinforcement learning algorithms can occasionally reach areas where they have converged on an incorrect strategy, and take a while to adjust back. This is clearly seen here, as the model performs very badly for that period of time, but then recovers and performs very well again. As the learning rate is increased, the running average scores decrease. Additionally, as the learning rate is increased, the prevalence of sections with consistently low scores also increases.

Training on Acrobot shows similar patterns as Cartpole, with the lowest learning rate having by far the best results. The scores decrease as the learning rate increases, and there are again many areas with consistently lower scores with the higher learning rates. The model reached its highest scores more quickly (in terms of run number) with Acrobot than Cartpole, which is likely because the Acrobot environment tends to take slightly longer to run than Cartpole, especially at first. Thus, the model sees more states per run early on with Acrobot, resulting in more training and fewer random actions in earlier run numbers than with Cartpole.

Space Invaders did not seem to learn in any significant manner. There does not seem to be noticeable or consistent improvement above random actions for any of the learning rates

throughout the entire training periods. As discussed earlier, Space Invaders is a far more complex environment than either Cartpole or Acrobot, so it does make sense that training struggles here. Given that the DQN model did improve on Cartpole and Acrobot, it is likely that it would also improve performance in Space Invaders given enough time, but 500 episodes does not seem to be enough.

Further Comments and Conclusions:

From the results, it seems that lower learning rates for neural nets tends to result in improved training for reinforcement learning. Due to the tendency of reinforcement learning models to struggle to converge, this makes sense, as a lower learning rate may then lead to a slow but steady path to convergence, whereas higher learning rates may lead to the neural net skipping over potential convergence areas. Along this line of thought, it would also be interesting to run similar experiments but with even lower learning rates. At some point, there is also a tradeoff with increasing training time.

Another initial goal of this experiment was to analyze the effects of increasing learning rates on training with more and less complex environments, and with environments with and without randomness. The thought here was that higher learning rates may be effective in relatively simple and/or deterministic environments, as it is likely easier for the model to converge correctly under these conditions. However, in more complex environments or in environments with significant randomness, high learning rates may lead to over adjusting the model to rarer events, or to random events, resulting in the model struggling to converge. This was one of the initial reasons for including training on the Space Invaders environment, but training was taking too long with the given hardware to produce significant results in this case.

References:

OpenAI Gym (data source): https://gym.openai.com/
Example DQN Models: https://github.com/keon/deep-q-learning,
https://github.com/gsurma/cartpole

Youtube URLs:

2 minute: https://youtu.be/6qTyTspKE74
15 minute: https://youtu.be/jxCofJAp_es