University of Colorado - Boulder

ASEN 5519 Decisionmaking Under Uncertainty

Final Project

Machine Learning for Atari Breakout

Authors:
Jordan Abell
Galen Bascom
Tyler Shea

Ann and H.J. Smead
Aerospace Engineering Sciences
UNIVERSITY OF COLORADO **BOULDER**

Contents

1 Introduction

The goal of this project is to train agents using neural networks to play Atari Breakout. We used the *TensorFlow Agents* library for Python as a way to create and interface with the environments and train our agents. The core idea was to use DQN to play Atari and achieve reasonable rewards–depending on training time. While working on the project, we added an intermediate goal of getting DQN to work on a Cart Pole environment. Ultimately, we were able to compare DQN and categorical DQN in both environments. We were also able to train a REINFORCE agent for the Cart Pole environment.

2 Background and Related Work

Both the Cart Pole environment, a classic controls environment, and the Atari game environment are often used to benchmark performance for reinforcement learning algorithms [8]. This project was inspired by a groundbreaking paper presented in class [1], which showed high performance by DQN on a wide range of Atari games. Breakout was one of the games at which DQN, with proper problem formulation and training time, could excel. Of course, further advances are constantly being made. For example, Deepmind has created Agent 57, which excels at all Atari games, including ones for which the previous paper showed DQN performing below human level [10]. Our plan was not to match the levels of performance currently possible, but to learn how to apply the tools learned in our coursework to this more challenging problem, as well as to learn and apply new tools in a different language (Python).

3 Problem Formulation

3.1 Cart Pole

The Cart Pole problem can be represented as a fully observable Markov Decision Process (MDP). The state space of Cart Pole is a directly observable 4-dimensional vector that contains the cart position, cart velocity, pole angle, and pole angular velocity. The cart position is constrained between $[-4.8 \quad 4.8]$ and the pole angle is constrained to $[-0.418 \quad 0.418]$ [rad]. The action space in the Cart Pole environment is 2-dimensional, push left and push right. You receive a reward of 1.0 for every step taken in the environment that the pole is within the pole angles $[-12^\circ \quad 12^\circ]$, and the environment terminates once the pole angle is greater than $\pm 12^\circ$. A rendering of the Cart Pole environment is included in Appendix I.

3.2 Breakout

Atari Breakout is another classic game used in artificial intelligence benchmarking. When the ball hits the paddle, it bounces off; when it falls past the paddle, a life is lost (you have 5 lives). Reward is accumulated by breaking the bricks at the top of the screen, with more reward for higher layers. A large part of our initial work was in figuring out how to use this environment. We considered implementing our own game engine, but eventually decided to use preexisting tools to represent the environment. The arcade learning environment (ALE) is a C++ implementation of the classic Atari game engine [2], and the *TensorFlow Agents* library provides wrappers to load and interface with that environment. As the project progressed, we learned how to use those functions and leverage the ALE.

The observation that comes natively from the environment corresponds to a frame of the Atari game (see image in Appendix I), of a size (210, 160, 3). Built-in functions exist to rescale and stack those frames, as a single frame does not provide ball velocity. The frames were also converted to grayscale and normalized to enhance learning performance. The action space consisted of four actions: NOOP, FIRE, LEFT, RIGHT. There is also some stochasticity in the ALE/Breakout-v5 environment used, as described in the Gym documentation [11].

4 Solution Approach

4.1 Algorithm Descriptions

DQN

A Deep Q-Network uses a neural network to approximate the Q-function. The Q-function takes inputs of the current state (or observation) and possible action and returns a measure of the immediate reward and discounted future rewards. The input layer of the DQN neural network corresponds to the size of the state (or observation) space of the environment and the output provides the action. The Q-network processes the observation through its hidden layers to create Q-value estimates for each possible action in the environment. Therefore, the output layer of the Q-network corresponds to the action-space of the problem.

The loss function for DQN is given by

$$l(s,a,r,s') = r + \gamma max_{a'} Q_{\theta'}(s',a') - Q_{\theta}(s,a) \tag{1}$$

Categorical DQN

Categorical DQN (C51) is very similar to DQN, except instead of estimating the Q-value directly for a state-action pair, it estimates a Q-value probability distribution [3]. The atoms in the categorical Q network are projected forward, giving a probabilistic representation for the Q value. This structure allows the training to be more stable than for DQN.

REINFORCE

REINFORCE stands for "REward, Increment = Non-negative Factor times Offset Reinforcement times Characteristic Eligibility," and it is an algorithm where weight adjustments of a neural network are made in the direction of immediate reinforcement[6]. The algorithm is considered model-free and no gradient is directly computed. Instead, each output node of every layer in the network is drawn from a distribution that is dependent on the input to each layer multiplied to a weight. To guarantee that the resulting action of the agent statistically follows the gradient of immediate reward, the ith node of a given output layer will have the following change in weight for its jth input [6]:

$$\Delta w_{ij} = \alpha_{ij}(r - b_{ij})e_{ij} \tag{2}$$

The α term is the learning rate factor, b is the reinforcement baseline, e is the characteristic eligibility, and g is the probability mass function of a given output layer's node to the input values and weighting values. These terms are described by the following equations:

$$e_{ij} = \partial ln(g_i)/\partial w_{ij} \tag{3}$$

$$g_i(\zeta, w^i, x^i) = Pr(y_i = \zeta | w^i, x^i) \tag{4}$$

4.2 Cart Pole

We first implemented and trained a DQN agent and a Categorical DQN agent in the Cart Pole environment. Our implementations were based on existing resources from TensorFlow, [4] and [5] for the DQN and Categorical DQN agents, respectively. There were three primary components: setting up the environment, setting up the agent, and training. The environment object that we used was a TFPyEnvironment, which represents a Python environment object as a TensorFlow environment. This allows TensorFlow objects, such as a TF Agent, to interface with the environment. For the Cart Pole environment we used 'CartPole-v0'. We distinguish between the training environment and the evaluation environment to ensure that evaluation does not modify the training environment.

The DQN agent requires a representation for the Q-Network. For the Q-Network, we used a dense neural network, defined with a single hidden layer with 100 nodes. The network accepted a 4-dimensional input vector that corresponds to the 4-dimensional state/observation vector for the Cart Pole environment. The Q-Network has a 2-dimensional output vector, which corresponds to the size of the action space. We also used the Adam optimizer, as for the DQN implementation used in the homework. The categorical DQN agent uses a categorical Q-Network, with the difference that it has an an parameter for the number of atoms, the number of probability distribution elements used to approximate the Q probabilities.

The training process uses a replay buffer, which stores experience data to train the network on. We ran simulations in the environment and stored the trajectories in the buffer. We also used a driver, which acts as a wrapper to collect the training data. The driver takes an environment, a policy, a replay buffer, and the number of steps to take. To collect training data we use an epsilon-greedy collect policy. Finally, each training iteration a batch random trajectories are pulled from the replay buffer and used to train the network. The process is identical for DQN and categorical DQN.

Finally, for the REINFORCE algorithm, the implementation for the Cart Pole problem was likewise drawn from the example provided on TensorFlow [7]. The implementation is similar to those for DQN and categorical DQN, using the same neural network structure, environment, and optimizer. Unlike the DQN and Categorical DQN, the REINFORCE code utilizes a different buffer implementation. The library `reverb` is used. It serves the same function as the replay buffer in the DQN and Categorical DQN implementations but can only be used in Linux. The driver used is the TensorFlow PyDriver.

4.3 Atari

For the Atari implementation, we were able to find a resource that shows how to use the *TensorFlow Agents* library and its functions as applied to a different Atari game, Pong [12]. With this starting point, we were able to use the *Tensorflow Agents* library on a new environment, and we wrote code that used both agents to compare their performance.

In other respects, our implementation for the Breakout environment was similar to the implementation for the Cart Pole environment. We used the functions discussed in section 3.2 for wrapping and stacking the frames, as well as a uniform replay buffer and and Adam optimizer. The neural network structure was based directly on the structure used in the Nature paper, and it used layers of convolutional neural networks followed by a dense layer and the output layer [1]. Our learning rate was $2.5e-3$, and the other selected hyperparameters (which were largely based off the Pong example) can be found in the code in Appendix II. Every 250 training iterations, we evaluated how well the agents were performing, and saved off the best performing agent.

5 Results

5.1 Cart Pole

In the Cart Pole environment, both our DQN and Categorical DQN agents were able to learn policies to achieve the maximum possible average reward of 200, averaged over 100 episodes. Figures 1 and 2 show the learning curves for the DQN agent and the Categorical DQN agent, respectively. Both agents here were trained over 20,000 training iterations.

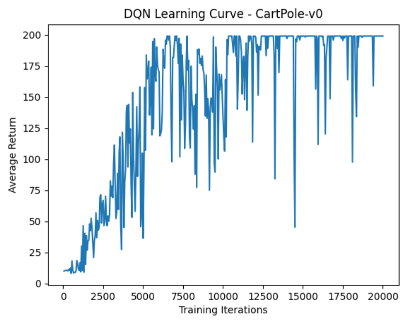

Figure 1: DQN Learning Curve, $\alpha = 1e-3$

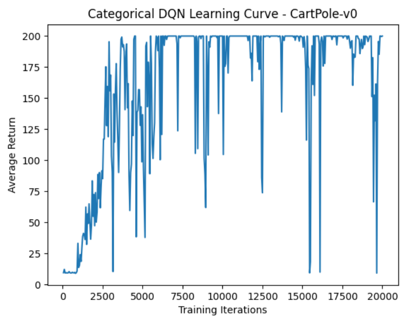

Figure 2: Categorical DQN Learning Curve, $\alpha = 1e-3$

Both agents have very noisy average returns during the training process. The DQN agent first reaches a maximum average return after approximately 6,000 iterations while the Categorical DQN agent first reaches an average return around 200 after approximately 3,000 iterations. By the end of the 20,000 training iterations both agents are able to achieve an average return around 200 fairly consistently. To improve the stability of the agents' training, we tried decreasing the learning rates to $\alpha = 5e-4$.

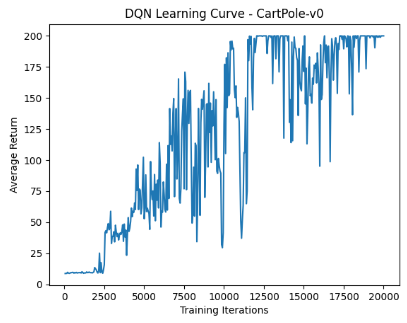

Figure 3: DQN Learning Curve, $\alpha = 5e-4$

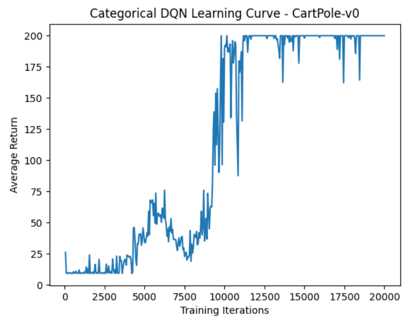

Figure 4: Categorical DQN Learning Curve, $\alpha = 5e-4$

As expected, the learning curves increase much slower. For the DQN agent the training does not appear to be significantly more stable, although the agent does still reach the maximum reward. In contrast, the lower learning rate dramatically improved the stability of the categorical DQN agent. The training takes longer, reaching the maximum reward around 10,000 training iterations, but the noise is significantly reduced. After about 11,000 iterations, this agent's average return stays very close to 200.

Overall, our two agents were very successful in training in the Cart Pole environment. While the training was not always completely stable, each one consistently reached the maximum reward of 200 by the end of 20,000 training iterations.

For the REINFORCE agent, an $\alpha = 1e-3$ was used and 250 training iterations (2 episodes per iteration) were investigated. The resulting average return (each evaluation done over 100 training episodes) against the number of episodes used for training is provided below.

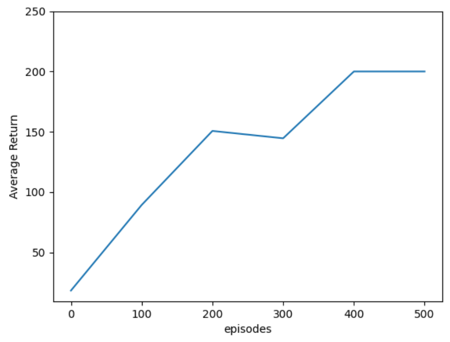

Figure 5: REINFORCE Learning Curve, $\alpha = 1e-3$

The learning curve for REINFORCE approaches the optimal value of 200 far more quickly than DQN and categorical DQN. A possible reason for this could be the fact that REINFORCE does not consider future rewards in its computation of its training loss value. Rather, it only considers the immediate reward in a way where its gradient is followed.

5.2 Breakout

Main Results

We first trained our DQN and categorical DQN agents over 200,000 iterations. Each of these training sessions took over 15 hours, and we were able to train an agent to do significantly better than a random policy. A random policy had an average return of 1.8 calculated over 10 attempts, whereas the best DQN policy had a return of 16.9 and the best categorical DQN policy had a return of 21.6.

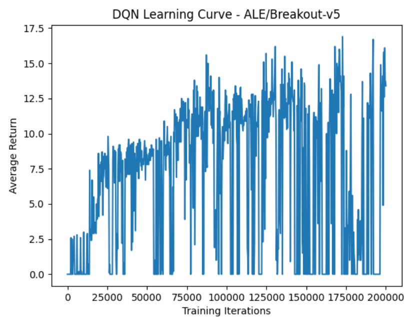

Figure 6: Results for DQN at 200,000 iterations.

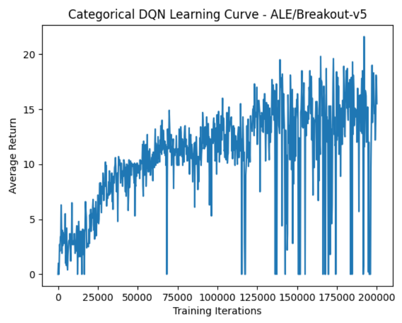

Figure 7: Results for C51 at 200,000 iterations.

Longer Training Time

We concluded that more training time was necessary. We trained the categorical DQN for 500,000 iterations, and we increased the batch size from 32 to 64 to try to get more stability.

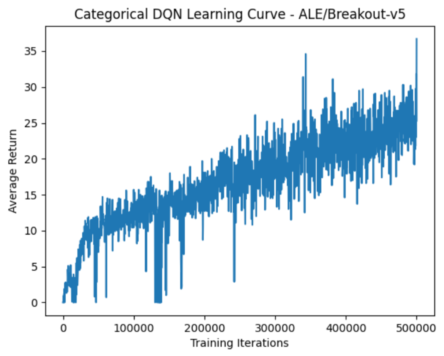

Figure 8: Learning curve for Atari C51 agent trained for 500,000 iterations.

Algorithm Error

After the bulk of the training was completed, we noticed that we had inadvertently introduced a bug that gathered experience to add to the replay buffer using the `policy` (instead of the `collect_policy`). While training, the epsilon greedy exploration did not happen as intended. It remains unclear to us why we got as promising of results as we did. We hypothesized that the training would have been faster with this error corrected. We reran the 200,000 iteration DQN and categorical DQN with this fixed, and with batch size 64.

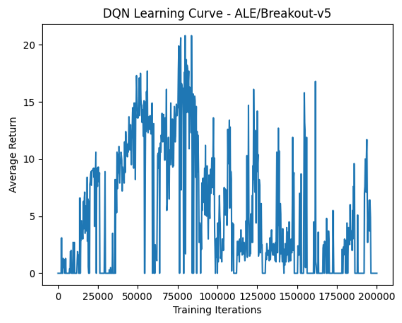

Figure 9: Results for DQN with collect bug fixed.

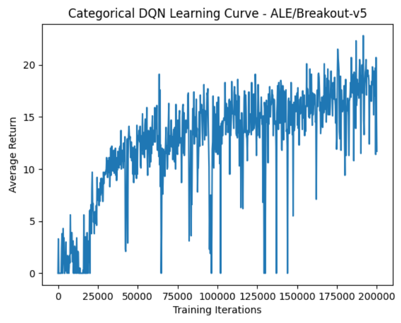

Figure 10: Results for C51 with collect bug fixed.

The performance was not much different, and DQN appears as unstable as before. The epsilon greedy exploration may have valuable effects not seen in these results, but our training was relatively robust to this mistake.

Video Comparison

As a final (enjoyable) way to compare our trained agent to an agent taking random actions, we produced short videos that render the environment while using three different policies, (1) a random policy, (2) a policy extracted from the categorical DQN agent trained over 200,000 iterations, and (3) a policy extracted from the categorical DQN agent trained over 500,000 iterations. Those videos are provided via email attachment. As can be seen in the videos, the random agent does occasionally strike the ball, getting some serendipitous reward. The trained agents, in contrast, execute responses to the observations that involve moving the paddle towards the moving ball, ultimately gaining significantly better rewards.

REINFORCE

The REINFORCE algorithm was applied to the Atari Breakout environment (Breakout-v5) over 400,000 training episodes. However, it was found that the agent was unable to achieve a nonzero reward. Various values of the learning rate α were tried and an example learning curve is provided in Appendix I.

To try to gain a non-zero reward, we used many approaches including modifying of the discount factor the training agent was using, using of the same learning buffer as the DQN and

Categorical DQN algorithms, and variations of the neural network. None of these modifications resulted in a non-zero reward. Consulting the literature, we learned that REINFORCE is an algorithm that incorporates immediate reinforcement [6]. It is very limited in cases where delayed reinforcement is important. In contrast, DQN and categorical DQN incorporate temporal difference in their loss functions [9]. This is critical in Breakout because the immediate reward at the first two states of the game is zero. Therefore, we determined that temporal difference is a key requirement in developing algorithms in environments where the starting state leads to no gradient in immediate reward.

6 Conclusion

We were able to learn how to create and interface with the Atari Environment, satisfying our first tier goal. We were also able to satisfy the second tier goal, by training an agent to outperform a random policy. We did not satisfy the final tier goal (clearing a screen of bricks), which we determined would have required much more extensive training time of the agent. However, we expanded our set of goals; not only training a DQN agent but also learning about and comparing that agent with the categorical DQN and REINFORCE agents. As expected, the categorical DQN agent showed more stability in training and somewhat outperformed the DQN agent. The REINFORCE agent ended up only being applicable to the Cart Pole problem, but for that problem we were able to get it to outperform the other two.

In general, a big challenge with the selected project was that it was not easy to iterate. For example, we would have wanted to change the network structures and many of the other hyperparameters to determine the sensitivity. But when each training took around 15 hours (for the ones we ran at 200,000 iterations) or over 30 hours (for the ones we ran at 500,000), this made it difficult to make small changes and see many different results. Smaller training times often showed very little in the way of results, so it made for difficulty in comparing.

Contributions and Release

Jordan was primarily responsible for applying the DQN and C51 algorithms to the Cart Pole problem and training the associated networks. Galen was primarily responsible for applying the DQN and C51 algorithms to the Atari problem and training the associated networks. Tyler was primarily responsible for applying the REINFORCE algorithm to both problems and making the videos. All members participated in brainstorming, debugging, documenting, and report writing.

The authors grant permission for this report to be posted publicly.

References

[1] Mnih, V. et al., “Human-level control through deep reinforcement learning,” *Nature* No. 518, 529–542, 2015. 1, 4

[2] Bellemare, M.G. et al, “The Arcade Learning Environment: An Evaluation Platform for General Agents,” *Journal of Artificial Intelligence Research* No. 47, 253–279, 2012. 2

[3] Bellemare, Dabney, Munos. “A Distributional Perspective on Reinforcement Learning,” 2017, https://doi.org/10.48550/arXiv.1707.06887. 2

[4] “Train a deep Q network with TF-Agents : tensorflow agents,” TensorFlow, retrieved May 3, 2022, from https://www.tensorflow.org/agents/tutorials/1_dqn_tutorial. 3

[5] “DQN C51/rainbow : tensorflow agents,” TensorFlow, retrieved May 3, 2022, from https://www.tensorflow.org/agents/tutorials/9_c51_tutorial. 3

[6] Williams, R. J, “Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning,” retrieved May 3, 2022, from https://people.cs.umass.edu/ barto/-courses/cs687/williams92simple.pdf 2, 3, 8

[7] “Reinforce agent,” TensorFlow, retrieved May 3, 2022, from https://www.tensorflow.org/agents/tutorials/6_reinforce_tutorial 3

[8] Nagendra, S. et al., “Comparison of Reinforcement Learning algorithms applied to the Cart Pole problem,” 2017 International Conference on Advances in Computing, Communications and Informatics (ICACCI), https://doi.org/10.48550/arXiv.1810.01940. 1

[9] Kochenderfer, M. J., et al. *Algorithms for decision making*, Massachusetts Institute of Technology, 2022. 8

[10] Badia, A.P. et al., “Agent57: Outperforming the Atari Human Benchmark,” 2022, https://doi.org/10.48550/arXiv.2003.13350. 1

[11] Gym Documentation: Breakout, retrieved May 3, 2022 from https://www.gymlibrary.ml/environments/atari/breakout/. 2

[12] Heaton, J., “Applications of Deep Neural Networks”, retrieved May 3, 2022 from https://github.com/jeffheaton/t81_558_deep_learning/blob/master/t81_558_class_12_04_atari.ipynb. 4

7 Appendix I: Additional Figures

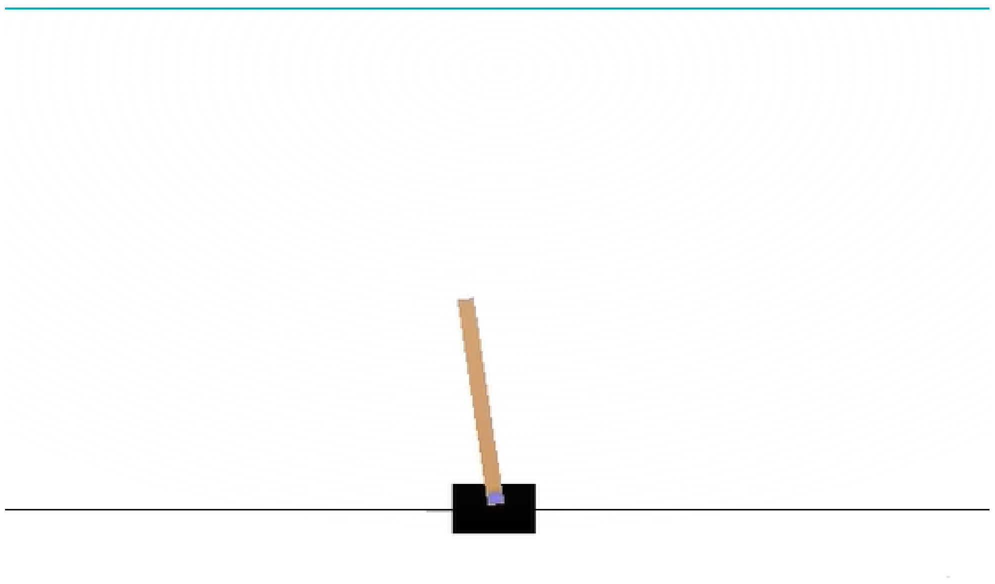

Figure 11: Classic controls Cart Pole environment rendering.

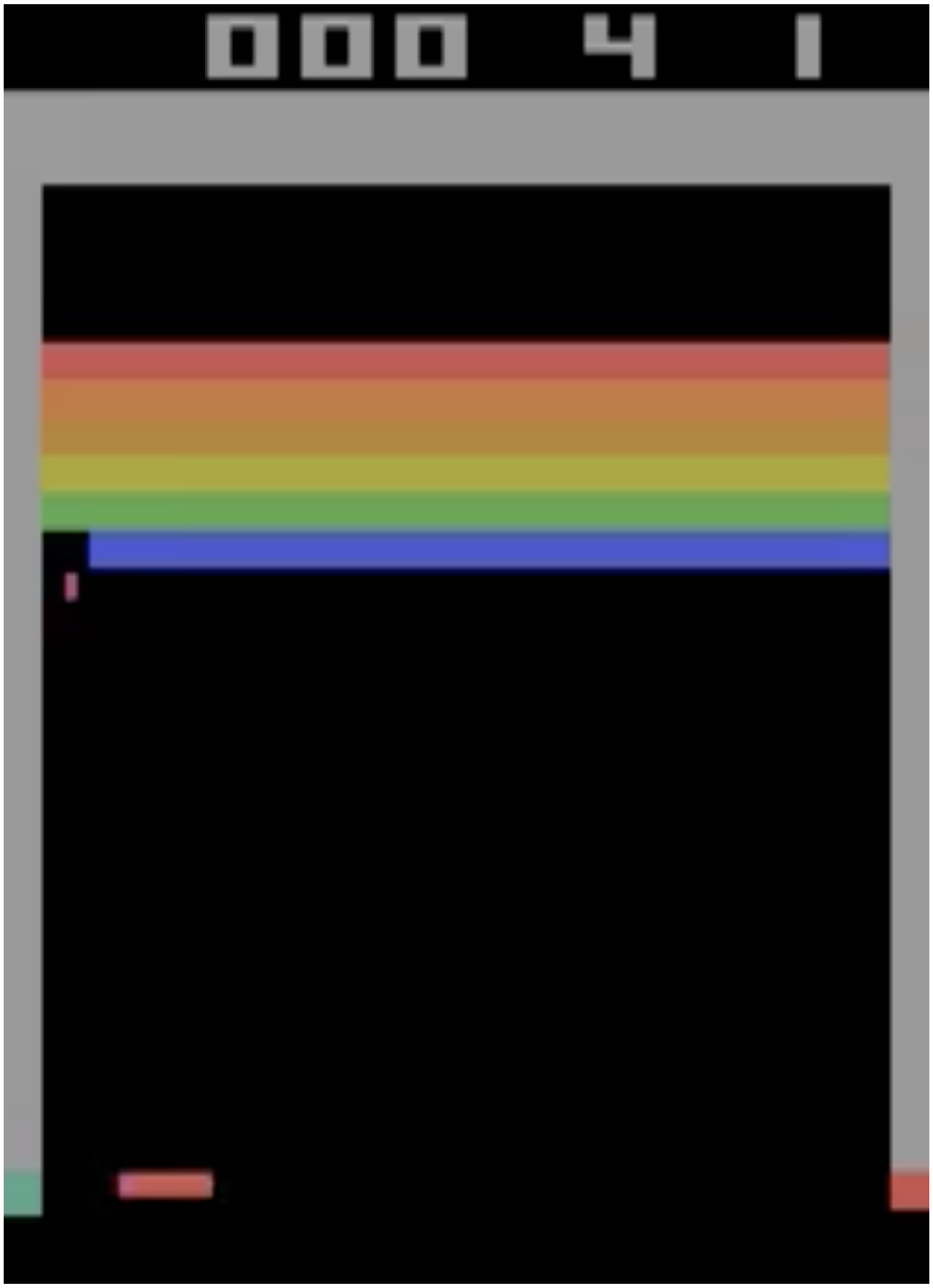

Figure 12: Atari Breakout frame.

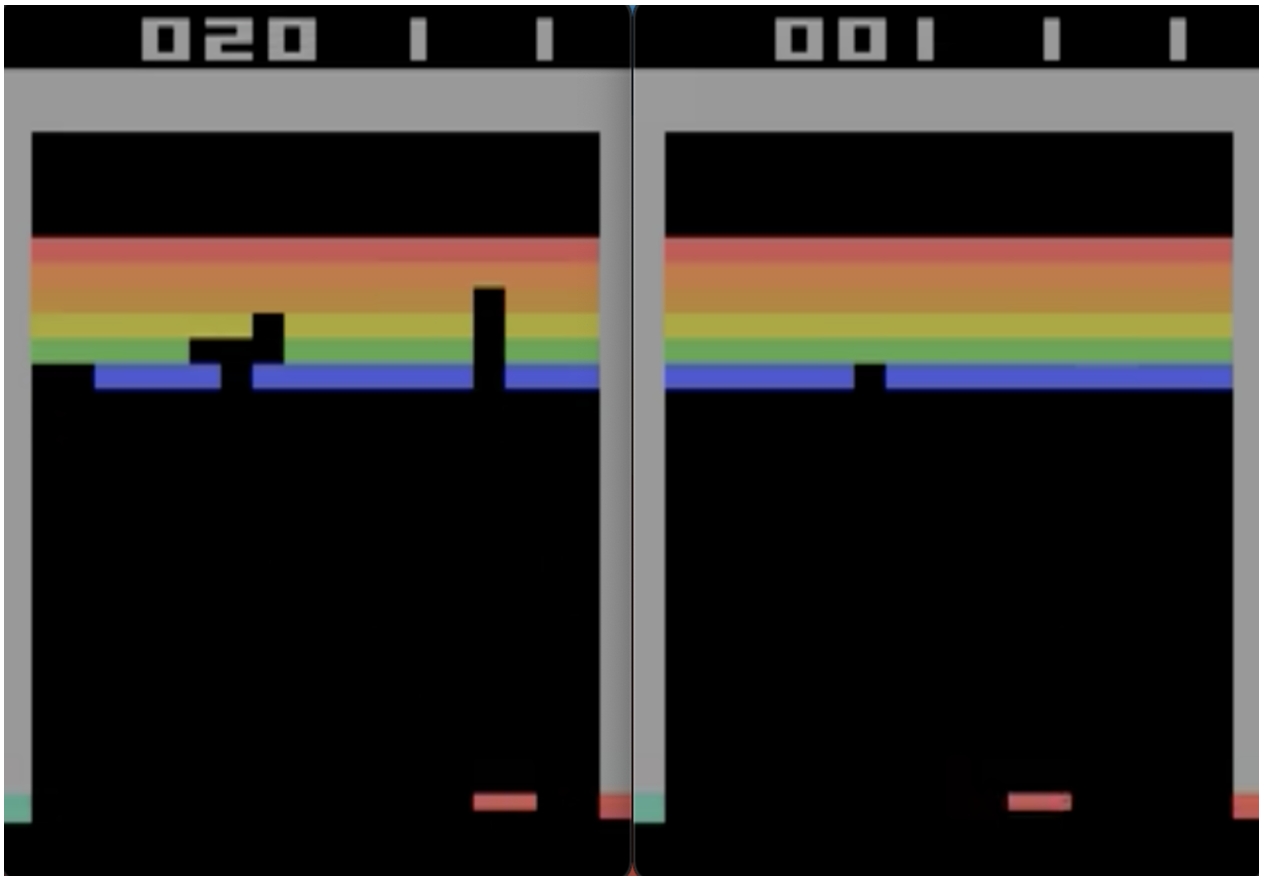

Figure 13: Comparison between the final frames of a representative game of Breakout, played by the C51 agent trained for 200,000 iterations (left) and an agent implementing a random policy (right).

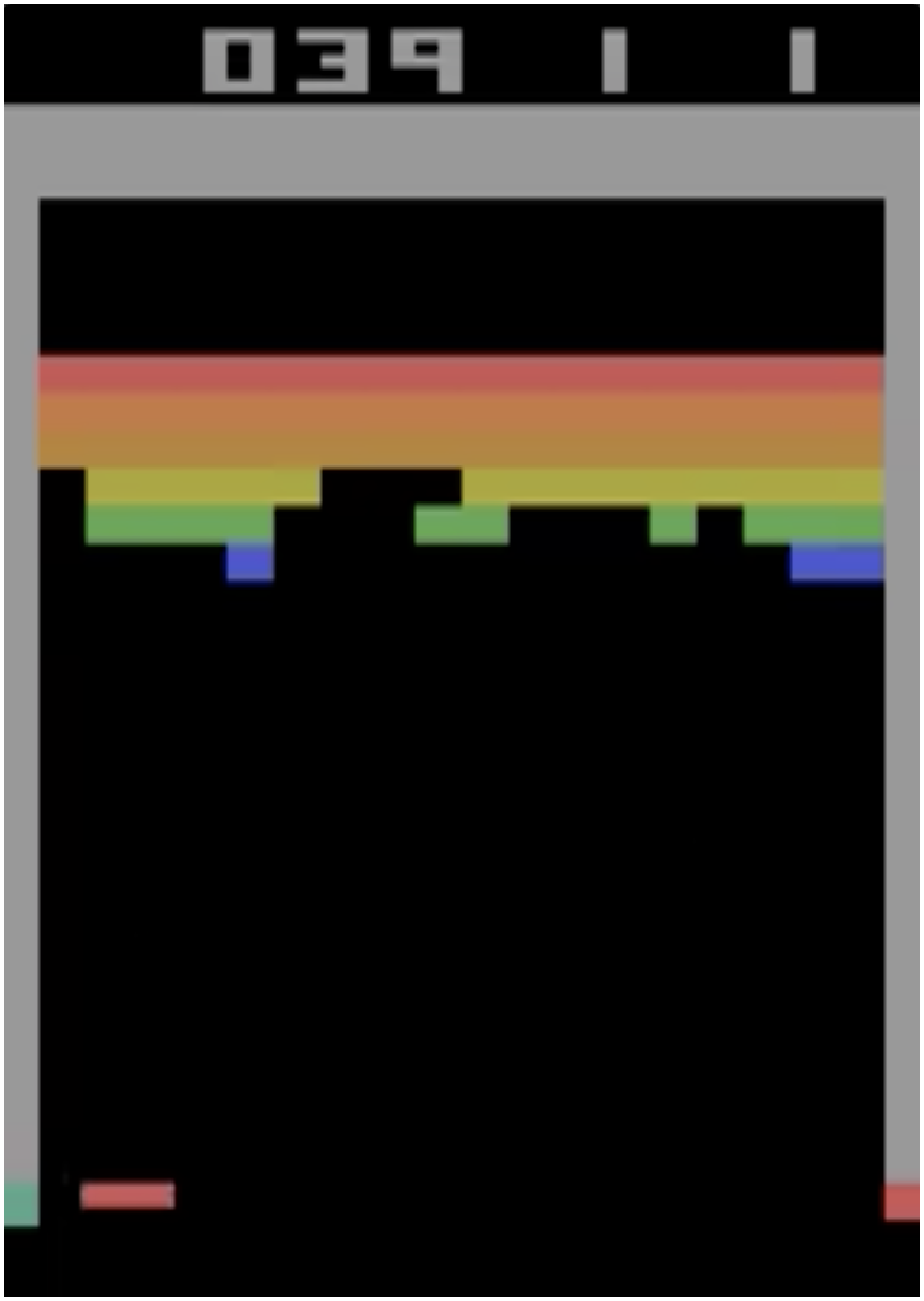

Figure 14: The final frame of a representative game of Breakout, played by the C51 agent trained for 500,000 iterations.

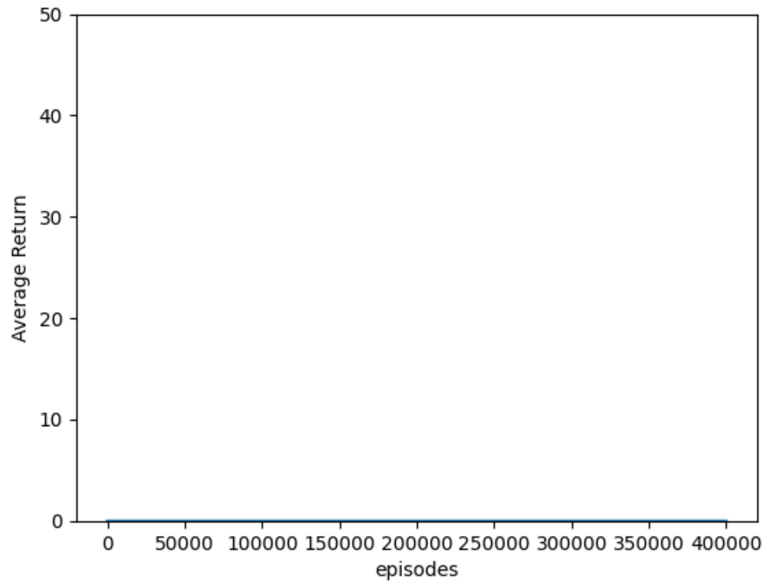

Figure 15: REINFORCE Learning Curve, $\alpha = 2.5e - 3$

8 Appendix II: Code

DQN for Atari

```
import matplotlib.pyplot as plt
import numpy as np
import time
import tensorflow as tf

from tf_agents.environments import suite_atari
from tf_agents.environments import tf_py_environment
from tf_agents.networks import q_network, network
from tf_agents.policies import random_tf_policy
from tf_agents.replay_buffers import tf_uniform_replay_buffer
from tf_agents.trajectories import trajectory
from tf_agents.utils import common
from tf_agents.agents.dqn import dqn_agent
from tf_agents.specs import tensor_spec
from tf_agents.trajectories import time_step as ts

from utilities import plot_learning_curve, save_data

begin_time = time.time()
num_iterations = 200000

initial_collect_steps = 200
collect_steps_per_iteration = 10
replay_buffer_max_length = 100000

batch_size =   32
learning_rate = 2.5e-3
log_interval =   100
num_eval_episodes = 10
eval_interval = 250

env_name = 'ALE/Breakout-v5'
max_episode_frames=27000

env = suite_atari.load(
    env_name,
    max_episode_steps=max_episode_frames,
    gym_env_wrappers=suite_atari.DEFAULT_ATARI_GYM_WRAPPERS_WITH_STACKING)


train_py_env = suite_atari.load(
    env_name,
    max_episode_steps=max_episode_frames,
    gym_env_wrappers=suite_atari.DEFAULT_ATARI_GYM_WRAPPERS_WITH_STACKING)

eval_py_env = suite_atari.load(
    env_name,
    max_episode_steps=max_episode_frames,
    gym_env_wrappers=suite_atari.DEFAULT_ATARI_GYM_WRAPPERS_WITH_STACKING)

train_env = tf_py_environment.TFPyEnvironment(train_py_env)
eval_env = tf_py_environment.TFPyEnvironment(eval_py_env)

class AtariQNetwork(network.Network):
  """QNetwork subclass that divides observations by 255."""

  def __init__(self, input_tensor_spec, action_spec, **kwargs):
    super(AtariQNetwork, self).__init__(
        input_tensor_spec, state_spec=())
    input_tensor_spec = tf.TensorSpec(
        dtype=tf.float32, shape=input_tensor_spec.shape)
    self._q_network = \
        q_network.QNetwork(
        input_tensor_spec, action_spec, **kwargs)

  def call(self, observation, step_type=None, network_state=()):
    state = tf.cast(observation, tf.float32)
    state = state / 255
    return self._q_network(
        state, step_type=step_type, network_state=network_state)


fc_layer_params = (512,)
conv_layer_params=((32, (8, 8), 4), (64, (4, 4), 2), (64, (3, 3), 1))

q_net = AtariQNetwork(
            train_env.observation_spec(),
            train_env.action_spec(),
            conv_layer_params=conv_layer_params,
            fc_layer_params=fc_layer_params)

optimizer = tf.compat.v1.train.RMSPropOptimizer(
    learning_rate=learning_rate,
    decay=0.95,
    momentum=0.0,
    epsilon=0.00001,
    centered=True)

train_step_counter = tf.Variable(0)

observation_spec = tensor_spec.from_spec(train_env.observation_spec())
time_step_spec = ts.time_step_spec(observation_spec)

action_spec = tensor_spec.from_spec(train_env.action_spec())
target_update_period=2000

agent = dqn_agent.DqnAgent(
    time_step_spec,
    action_spec,
    q_network=q_net,
    optimizer=optimizer,
    n_step_update=1.0,
    target_update_tau=1.0,
    target_update_period=target_update_period,
    gamma=0.99,
    reward_scale_factor=1.0,
    gradient_clipping=None,
    debug_summaries=False,
    summarize_grads_and_vars=False)

agent.initialize()
best_agent = agent

def compute_avg_return(environment, policy, num_episodes=10, max_steps=1000):
  total_return = 0.0
  for _ in range(num_episodes):
    step_counter=0
    time_step = environment.reset()
    episode_return = 0.0

    while not time_step.is_last() and step_counter < max_steps:
      action_step = policy.action(time_step)
      time_step = environment.step(action_step.action)
      episode_return += time_step.reward
      step_counter+=1
    total_return += episode_return

  avg_return = total_return / num_episodes
  return avg_return.numpy()[0]


replay_buffer = tf_uniform_replay_buffer.TFUniformReplayBuffer(
    data_spec=agent.collect_data_spec,
    batch_size=train_env.batch_size,
    max_length=replay_buffer_max_length)

dataset = replay_buffer.as_dataset(
    num_parallel_calls=3,
    sample_batch_size=batch_size,
    num_steps=2).prefetch(3)

random_policy = random_tf_policy.RandomTFPolicy(train_env.time_step_spec(),
                                                train_env.action_spec())

def collect_step(environment, policy, buffer):
  time_step = environment.current_time_step()
  action_step = policy.action(time_step)
  next_time_step = environment.step(action_step.action)
  traj = trajectory.from_transition(time_step, action_step, next_time_step)
  buffer.add_batch(traj)
```

```
def collect_data(env, policy, buffer, steps):
  for _ in range(steps):
    collect_step(env, policy, buffer)

collect_data(train_env, random_policy, replay_buffer, \
             steps=initial_collect_steps)

iterator = iter(dataset)

agent.train = common.function(agent.train)
agent.train_step_counter.assign(0)
avg_return = compute_avg_return(eval_env, agent.policy, num_eval_episodes)
returns = [avg_return]
max_return = returns[0]

begin_train_time = time.time()
initialize_time = (begin_train_time-begin_time)
print("Seconds to intialize: "+str(initialize_time))

for _ in range(num_iterations):
  iteration_start_time = time.time()

  for _ in range(collect_steps_per_iteration):
    collect_step(train_env, agent.policy, replay_buffer)

  experience, unused_info = next(iterator)
  train_loss = agent.train(experience).loss

  step = agent.train_step_counter.numpy()

  if step % log_interval == 0:
    print('step = {0}: loss = {1}'.format(step, train_loss))

  iteration_end_time = time.time()
  iteration_time = iteration_end_time-iteration_start_time
  #print("Iteration time: "+str(iteration_time))

  if step % eval_interval == 0:
    begin_eval_time=time.time()
    avg_return = compute_avg_return(eval_env, agent.policy, \
                                    num_eval_episodes)
    print('step = {0}: Average Return = {1}'.format(step, avg_return))
    returns.append(avg_return)
    end_eval_time = time.time()
    if avg_return > max_return:
      max_return = avg_return
      best_agent = agent
    eval_time = end_eval_time-begin_eval_time
    print("Evaluation time: "+str(iteration_time))

end_time = time.time()
total_time = end_time-begin_time
```

```
print("Total Time: "+str(total_time)+" seconds.")
iterations = range(0, num_iterations + 1, eval_interval)

plot_learning_curve("dqn", env_name, iterations, returns)
save_data(best_agent, "dqn", iterations, returns,
    learning_rate, save_dir="./data/")

train_env.close()
eval_env.close()
```

Categorical DQN for Atari

```
import matplotlib.pyplot as plt
import numpy as np
import time
import tensorflow as tf

from tf_agents.agents.categorical_dqn import categorical_dqn_agent
from tf_agents.environments import suite_atari
from tf_agents.environments import tf_py_environment
from tf_agents.networks import categorical_q_network, network
from tf_agents.policies import random_tf_policy
from tf_agents.replay_buffers import tf_uniform_replay_buffer
from tf_agents.trajectories import trajectory
from tf_agents.utils import common
from tf_agents.specs import tensor_spec
from tf_agents.trajectories import time_step as ts

from utilities import plot_learning_curve, save_data

begin_time = time.time()
num_iterations = 300000

initial_collect_steps = 200
collect_steps_per_iteration = 10
replay_buffer_max_length = 100000

batch_size =   32
learning_rate = 1e-3
log_interval =   100
num_eval_episodes = 10
eval_interval = 250

env_name = 'ALE/Breakout-v5'
max_episode_frames=27000

env = suite_atari.load(
    env_name,
    max_episode_steps=max_episode_frames,
    gym_env_wrappers=suite_atari.DEFAULT_ATARI_GYM_WRAPPERS_WITH_STACKING)


train_py_env = suite_atari.load(
    env_name,
    max_episode_steps=max_episode_frames,
    gym_env_wrappers=suite_atari.DEFAULT_ATARI_GYM_WRAPPERS_WITH_STACKING)

eval_py_env = suite_atari.load(
    env_name,
    max_episode_steps=max_episode_frames,
    gym_env_wrappers=suite_atari.DEFAULT_ATARI_GYM_WRAPPERS_WITH_STACKING)
```

```
train_env = tf_py_environment.TFPyEnvironment(train_py_env)
eval_env = tf_py_environment.TFPyEnvironment(eval_py_env)

class AtariCategoricalQNetwork(network.Network):
  """CategoricalQNetwork subclass that divides observations by 255."""

  def __init__(self, input_tensor_spec, action_spec, **kwargs):
    super(AtariCategoricalQNetwork, self).__init__(
        input_tensor_spec, state_spec=())
    input_tensor_spec = tf.TensorSpec(
        dtype=tf.float32, shape=input_tensor_spec.shape)
    self._categorical_q_network = \
        categorical_q_network.CategoricalQNetwork(
        input_tensor_spec, action_spec, **kwargs)

  @property
  def num_atoms(self):
    return self._categorical_q_network.num_atoms

  def call(self, observation, step_type=None, network_state=()):
    state = tf.cast(observation, tf.float32)
    state = state / 255
    return self._categorical_q_network(
        state, step_type=step_type, network_state=network_state)


fc_layer_params = (512,)
conv_layer_params=((32, (8, 8), 4), (64, (4, 4), 2), (64, (3, 3), 1))

q_net = AtariCategoricalQNetwork(
            train_env.observation_spec(),
            train_env.action_spec(),
            conv_layer_params=conv_layer_params,
            fc_layer_params=fc_layer_params)

optimizer = tf.compat.v1.train.RMSPropOptimizer(
    learning_rate=learning_rate,
    decay=0.95,
    momentum=0.0,
    epsilon=0.00001,
    centered=True)

train_step_counter = tf.Variable(0)

observation_spec = tensor_spec.from_spec(train_env.observation_spec())
time_step_spec = ts.time_step_spec(observation_spec)

action_spec = tensor_spec.from_spec(train_env.action_spec())
target_update_period=2000

agent = categorical_dqn_agent.CategoricalDqnAgent(
    time_step_spec ,
    action_spec ,
    categorical_q_network=q_net ,
    optimizer=optimizer ,
    n_step_update=1.0,
    target_update_tau=1.0,
    target_update_period=target_update_period ,
    gamma=0.99,
    reward_scale_factor=1.0,
    gradient_clipping=None,
    debug_summaries=False ,
    summarize_grads_and_vars=False)

agent.initialize()
best_agent = agent

def compute_avg_return(environment, policy, num_episodes=10, max_steps=1000):
  total_return = 0.0
  for _ in range(num_episodes):
    step_counter=0
    time_step = environment.reset()
    episode_return = 0.0

    while not time_step.is_last() and step_counter < max_steps:
      action_step = policy.action(time_step)
      time_step = environment.step(action_step.action)
      episode_return += time_step.reward
      step_counter+=1
    total_return += episode_return

  avg_return = total_return / num_episodes
  return avg_return.numpy()[0]


replay_buffer = tf_uniform_replay_buffer.TFUniformReplayBuffer(
    data_spec=agent.collect_data_spec ,
    batch_size=train_env.batch_size ,
    max_length=replay_buffer_max_length)

dataset = replay_buffer.as_dataset(
    num_parallel_calls=3,
    sample_batch_size=batch_size ,
    num_steps=2).prefetch(3)

random_policy = random_tf_policy.RandomTFPolicy(train_env.time_step_spec(),
                                                train_env.action_spec())

def collect_step(environment, policy, buffer):
  time_step = environment.current_time_step()
  action_step = policy.action(time_step)
  next_time_step = environment.step(action_step.action)
  traj = trajectory.from_transition(time_step, action_step, next_time_step)
  buffer.add_batch(traj)

def collect_data(env, policy, buffer, steps):
  for _ in range(steps):
    collect_step(env, policy, buffer)

collect_data(train_env, random_policy, replay_buffer, \
             steps=initial_collect_steps)



iterator = iter(dataset)

agent.train = common.function(agent.train)
agent.train_step_counter.assign(0)
avg_return = compute_avg_return(eval_env, agent.policy, num_eval_episodes)
returns = [avg_return]
max_return = returns[0]

begin_train_time = time.time()
initialize_time = (begin_train_time-begin_time)
print("Seconds to intialize: "+str(initialize_time))

for _ in range(num_iterations):
  iteration_start_time = time.time()

  for _ in range(collect_steps_per_iteration):
    collect_step(train_env, agent.policy, replay_buffer)

  experience, unused_info = next(iterator)
  train_loss = agent.train(experience).loss

  step = agent.train_step_counter.numpy()

  if step % log_interval == 0:
    print('step = {0}: loss = {1}'.format(step, train_loss))

  iteration_end_time = time.time()
  iteration_time = iteration_end_time-iteration_start_time
  #print("Iteration time: "+str(iteration_time))

  if step % eval_interval == 0:
    begin_eval_time=time.time()
    avg_return = compute_avg_return(eval_env, agent.policy, \
                              num_eval_episodes)
    print('step = {0}: Average Return = {1}'.format(step, avg_return))
    returns.append(avg_return)
    end_eval_time = time.time()
    if avg_return > max_return:
      max_return = avg_return
      best_agent = agent
    eval_time = end_eval_time-begin_eval_time
    print("Evaluation time: "+str(iteration_time))

end_time = time.time()
total_time = end_time-begin_time
print("Total Time: "+str(total_time)+" seconds.")
iterations = range(0, num_iterations + 1, eval_interval)

plot_learning_curve("categorical_dqn", env_name, iterations, returns)
save_data(best_agent, "categorical_dqn",
    iterations, returns, learning_rate, save_dir="./data/")

train_env.close()
eval_env.close()
```

Cart Pole

```
import utilities
import dqn_utilities

###############################
### Simulation Parameters ###
###############################

# Agent type
# agent_type = "cartpole_dqn"
agent_type = "cartpole_categorical_dqn"

# Environment Name
# env_name = 'ALE/Breakout-v5'
env_name = 'CartPole-v0'

num_iterations = 20000 # Number of training iterations
fc_layer_params = (100,) # One hidden layer, 100 nodes
learning_rate = 5e-4 # Learning rate
batch_size = 64 # Replay buffer batch size

###################
### Environment ###
###################

train_env = dqn_utilities.get_env(env_name)
eval_env = dqn_utilities.get_env(env_name)

#############
### Agent ###
#############

tf_agent = dqn_utilities.get_agent(agent_type,
    train_env, learning_rate, fc_layer_params)

###################
### Train Agent ###
###################

tf_agent, steps, avg_rewards, exec_times = dqn_utilities.train_agent(tf_agent,
    train_env, eval_env, num_iterations, batch_size)

##########################
### Plot and Save Data ###
##########################

utilities.save_data(tf_agent, agent_type,
    steps, avg_rewards, exec_times, learning_rate)

final_avg_return = dqn_utilities.compute_avg_return(eval_env,
    tf_agent.policy, 100)
```

```
print('{0} Final Avg Return: {1}'.format(agent_type, final_avg_return))

utilities.plot_learning_curve(agent_type, env_name, steps, avg_rewards)

dqn_utilities.sim_episode(train_env, tf_agent.policy, True)

######################
### Script Cleanup ###
######################

train_env.close()
eval_env.close()
```

Cart Pole Utilities

```
import tensorflow as tf
from tf_agents.environments import suite_gym
from tf_agents.environments import tf_py_environment
from tf_agents.utils import common
from tf_agents.agents.dqn import dqn_agent
from tf_agents.agents.categorical_dqn import categorical_dqn_agent
from tf_agents.networks import q_network
from tf_agents.networks import categorical_q_network
from tf_agents.replay_buffers import tf_uniform_replay_buffer
from tf_agents.drivers import dynamic_step_driver

import matplotlib.pyplot as plt
import time

def compute_avg_return(environment, policy, num_episodes=10, plt_flag=False):

  total_return = 0.0

  if plt_flag:
    myfig = plt.figure()

  for _ in range(num_episodes):

      episode_return = sim_episode(environment, policy, plt_flag)

      total_return += episode_return

  avg_return = total_return / num_episodes
  return avg_return.numpy()[0]

def sim_step(prev_time_step, environment, policy):

    action_step = policy.action(prev_time_step)
    time_step = environment.step(action_step.action)

    return time_step, environment

def sim_episode(env, policy, plt_flag=False):

    # Initialization
    episode_return = 0.0
    trajectory = []

    # Reset Environment
    prev_time_step = env.reset()
    # prev_time_step = env.step(1)

    # Plot
    if plt_flag:
        myaxes = plt.imshow(env.render()[0])
        plt.show(block=False)

    # Loop through episode
    while not prev_time_step.is_last():

        # Simulate one step
        time_step, env = sim_step(prev_time_step, env, policy)

        # Record step results
        episode_return += time_step.reward
        # trajectory.append(traj)

        # Increment step
        prev_time_step = time_step

        # Plot
        if plt_flag:
            myaxes.set_data(env.render()[0])
            plt.draw()
            plt.pause(0.01)

    if plt_flag:
        plt.close()

    return episode_return

def get_env(env_name):

    py_env = suite_gym.load(env_name)
    env = tf_py_environment.TFPyEnvironment(py_env)

    return env

def get_agent(agent_type, env, learning_rate, fc_layer_params):

    optimizer = tf.keras.optimizers.Adam(learning_rate=learning_rate)
    train_step_counter = tf.Variable(0)

    if agent_type == "cartpole_dqn":

        # DQN Neural Network
        q_net = q_network.QNetwork(
            env.observation_spec(),
            env.action_spec(),
            fc_layer_params=fc_layer_params
        )

        # DQN Agent
        tf_agent = dqn_agent.DqnAgent(
            env.time_step_spec(),
            env.action_spec(),
            q_network=q_net,
            optimizer=optimizer,
            td_errors_loss_fn=common.element_wise_squared_loss,
            train_step_counter=train_step_counter
        )
    elif agent_type == "cartpole_categorical_dqn":

        # Categorical DQN Neural Network
        cat_q_net = categorical_q_network.CategoricalQNetwork(
            env.observation_spec(),
            env.action_spec(),
            fc_layer_params=fc_layer_params
        )

        # Categorical DQN Agent
        tf_agent = categorical_dqn_agent.CategoricalDqnAgent(
            env.time_step_spec(),
            env.action_spec(),
            categorical_q_network=cat_q_net,
            optimizer=optimizer,
            td_errors_loss_fn=common.element_wise_squared_loss,
            train_step_counter=train_step_counter
        )

    tf_agent.initialize()

    return tf_agent

def train_agent(agent, train_env, eval_env, num_iterations, batch_size,
collect_steps_per_iteration=20, replay_buffer_capacity=1000,
log_interval=50, eval_interval=50, num_eval_episodes=10):

    replay_buffer = tf_uniform_replay_buffer.TFUniformReplayBuffer(
        agent.collect_data_spec,
        batch_size=train_env.batch_size,
        max_length=replay_buffer_capacity
    )

    replay_observer = [replay_buffer.add_batch]

    dataset = replay_buffer.as_dataset(
        sample_batch_size=batch_size,
        num_steps=2,
        num_parallel_calls=3,
        single_deterministic_pass=False
    ).prefetch(3)

    iterator = iter(dataset)

    driver = dynamic_step_driver.DynamicStepDriver(
        train_env,
        agent.collect_policy,
        observers=replay_observer,
        num_steps=collect_steps_per_iteration
    )

    agent.train = common.function(agent.train)
    agent.train_step_counter.assign(0)

    initial_time_step = train_env.reset()

    plot_one = True

    steps = []
    avg_rewards = []
    exec_times = []

    for i in range(num_iterations):

        t_start = time.time()

        driver.run(initial_time_step)

        experience, _ = next(iterator)
        train_loss = agent.train(experience=experience)

        step = agent.train_step_counter.numpy()

        if step % log_interval == 0:
            print('Step␣=␣{0}:␣loss␣=␣{1}'.format(step, train_loss.loss))

        if step % eval_interval == 0:
            avg_return = compute_avg_return(eval_env, agent.policy,
                num_eval_episodes)
            steps.append(step)
            avg_rewards.append(avg_return)
            exec_times.append(time.time() - t_start)
            print('step␣=␣{0}:␣Average␣Return␣=␣{1}'.format(step, avg_return))

            # if avg_return > 100 and plot_one:
            #     sim_episode(train_env, tf_agent.policy, True)
            #     plot_one = False

    return agent, steps, avg_rewards, exec_times
```

REINFORCE for Atari Breakout

```
#atari_REINFORCE.py
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import base64
import imageio
import IPython
import matplotlib.pyplot as plt
import numpy as np
import PIL.Image
import pyvirtualdisplay
import reverb

import tensorflow as tf

from tf_agents.agents.reinforce import reinforce_agent
from tf_agents.drivers import py_driver
from tf_agents.environments import suite_gym
from tf_agents.environments import tf_py_environment
from tf_agents.environments import suite_gym, suite_atari
from tf_agents.networks import actor_distribution_network
from tf_agents.policies import py_tf_eager_policy
from tf_agents.replay_buffers import reverb_replay_buffer
from tf_agents.replay_buffers import reverb_utils
#from tf_agents.replay_buffers import tf_uniform_replay_buffer
from tf_agents.specs import tensor_spec
from tf_agents.trajectories import trajectory
from tf_agents.utils import common

from tf_agents.environments import batched_py_environment
from tf_agents.eval import metric_utils
from tf_agents.metrics import tf_metrics
from tf_agents.networks import q_network, network
from tf_agents.policies import random_tf_policy

from tf_agents.trajectories import time_step as ts

from utilities import save_data

import time

# Set up virtual display for rendering OpenAI gym environments.
#display = pyvirtualdisplay.Display(visible=0, size=(1400, 900)).start()

##########################################
### Hyperparameters

num_iterations = 8000
#num_iterations = 200
```

```
#num_iterations = 10
collect_episodes_per_iteration = 50
#replay_buffer_capacity = 40000
replay_buffer_capacity = 35000

#fc_layer_params = (100,)
fc_layer_params = (512,)
conv_layer_params=((32, (8, 8), 4), (64, (4, 4), 2), (64, (3, 3), 1))

#learning_rate = 1e-3
learning_rate = 2.5e-3
#learning_rate = 0.05
log_interval = 10
num_eval_episodes = 10
eval_interval = 5


#########################################
### Environment

# create two environments with one for training and the other for evaluation

# AtariPreprocessing runs 4 frames at a time, max-pooling over the last 2
# frames. We need to account for this when computing things like update
# intervals.

#env_name = 'BreakoutDeterministic-v4'
#ATARI_FRAME_SKIP = 4
#
#max_episode_frames=108000  # ALE frames
#
#env = suite_atari.load(
#     env_name,
#     max_episode_steps=max_episode_frames / ATARI_FRAME_SKIP,
#     gym_env_wrappers=suite_atari.DEFAULT_ATARI_GYM_WRAPPERS_WITH_STACKING)
#
#
#train_py_env = suite_atari.load(
#     env_name,
#     max_episode_steps=max_episode_frames / ATARI_FRAME_SKIP,
#     gym_env_wrappers=suite_atari.DEFAULT_ATARI_GYM_WRAPPERS_WITH_STACKING)
#
#eval_py_env = suite_atari.load(
#     env_name,
#     max_episode_steps=max_episode_frames / ATARI_FRAME_SKIP,
#     gym_env_wrappers=suite_atari.DEFAULT_ATARI_GYM_WRAPPERS_WITH_STACKING)
#
#train_env = tf_py_environment.TFPyEnvironment(train_py_env)
#eval_env = tf_py_environment.TFPyEnvironment(eval_py_env)
#

env_name = 'ALE/Breakout-v5'
```

```
max_episode_frames=27000

env = suite_atari.load(
    env_name,
    max_episode_steps=max_episode_frames,
    gym_env_wrappers=suite_atari.DEFAULT_ATARI_GYM_WRAPPERS_WITH_STACKING)


train_py_env = suite_atari.load(
    env_name,
    max_episode_steps=max_episode_frames,
    gym_env_wrappers=suite_atari.DEFAULT_ATARI_GYM_WRAPPERS_WITH_STACKING)

eval_py_env = suite_atari.load(
    env_name,
    max_episode_steps=max_episode_frames,
    gym_env_wrappers=suite_atari.DEFAULT_ATARI_GYM_WRAPPERS_WITH_STACKING)

train_env = tf_py_environment.TFPyEnvironment(train_py_env)
eval_env = tf_py_environment.TFPyEnvironment(eval_py_env)


##########################################
### Agent

# first need an actor network that can learn to predict
# the action given an obsrvation from the environment

class AtariREINFORCENetwork(network.Network):
    """REINFORCENetwork subclass that divides observations by 255."""

    def __init__(self, input_tensor_spec, action_spec, **kwargs):
        super(AtariREINFORCENetwork, self).__init__(
            input_tensor_spec, state_spec=())
        input_tensor_spec = tf.TensorSpec(
            dtype=tf.float32, shape=input_tensor_spec.shape)
        self._actor_distribution_network = \
            actor_distribution_network.ActorDistributionNetwork(
            input_tensor_spec, action_spec, **kwargs)

#     @property
    def call(self, observation, step_type=None, network_state=()):
        state = tf.cast(observation, tf.float32)
        # We divide the grayscale pixel values by 255 here rather than
        # storing normalized values beause uint8s are 4x cheaper to
        # store than float32s.
        # TODO(b/129805821): handle the division by 255 for
        # train_eval_atari.py in
        # a preprocessing layer instead.
        state = state / 255
        return self._actor_distribution_network(
            state, step_type=step_type, network_state=network_state)
```

```
actor_net = AtariREINFORCENetwork(train_env.observation_spec(),
    train_env.action_spec(),conv_layer_params=conv_layer_params,
    fc_layer_params=fc_layer_params)
#actor_net = AtariREINFORCENetwork(train_env.observation_spec(),
    train_env.action_spec(),fc_layer_params=fc_layer_params)

# also need an optimizer to train the network

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

# train_step_counter variable keeps track of how many times the network
# is updated

train_step_counter = tf.Variable(0)

tf_agent = reinforce_agent.ReinforceAgent(
        train_env.time_step_spec(),
        train_env.action_spec(),
        actor_network=actor_net,
        optimizer=optimizer,
        normalize_returns=True,
        train_step_counter=train_step_counter)

tf_agent.initialize()


#########################################
### Policies

# policies represent the standard notion of policies in RL.
# given a time_step produce an action or a distribution
# over actions.o

# Agents have two policies
# - main policy that is used for evaluation/deployment
# - policy that is used for data collection

eval_policy = tf_agent.policy
collect_policy = tf_agent.collect_policy


#########################################
### Metrics and Evaluation

# return is sum of rewards obtained while running
# a policy in an environment for an episode. Usually
# average this over a few episodes.


#def compute_avg_return(environment, policy, num_episodes=10, maxSteps=1000):
def compute_avg_return(environment, policy, num_episodes=10, maxSteps=500):
    total_return = 0.0

    for episodeIdx in range(num_episodes):

        #print(episodeIdx)
        time_step = environment.reset()
        episode_return = 0.0

        steps = 0
        while time_step.is_last() == False and steps < maxSteps:
            #print(steps)
            action_step = policy.action(time_step)
            #print(action_step.action)
            time_step = environment.step(action_step.action)
            episode_return += time_step.reward
            steps += 1
        total_return += episode_return

    avg_return = total_return / num_episodes

    return avg_return.numpy()[0]


########################################
### Replay Buffer

table_name = 'uniform_tabel'
replay_buffer_signature = tensor_spec.from_spec(
        tf_agent.collect_data_spec)

replay_buffer_signature = tensor_spec.add_outer_dim(
        replay_buffer_signature)

table = reverb.Table(
        table_name,
        max_size=replay_buffer_capacity,
        sampler=reverb.selectors.Uniform(),
        remover=reverb.selectors.Fifo(),
        rate_limiter=reverb.rate_limiters.MinSize(1),
        signature=replay_buffer_signature)

reverb_server = reverb.Server([table])

replay_buffer = reverb_replay_buffer.ReverbReplayBuffer(
        tf_agent.collect_data_spec,
        table_name=table_name,
        sequence_length=None,
        local_server=reverb_server)

rb_observer = reverb_utils.ReverbAddEpisodeObserver(
        replay_buffer.py_client,
        table_name,
        replay_buffer_capacity
        )


#########################################
### Data Collection

# REINFORCE learns from whole episodes.
# need a function to collect an episode using data collection policy
# and save data as trajectories in the replay buffer

def collect_episode(environment, policy, num_episodes, maxStepsPerEpisode):
    driver = py_driver.PyDriver(
            environment,
#             py_tf_eager_policy.PyTFEagerPolicy(
#                 policy, use_tf_function=True),
            policy,
            [rb_observer],
            max_steps=maxStepsPerEpisode,
            max_episodes=num_episodes)
    initial_time_step = environment.reset()
    driver.run(initial_time_step)


#########################################
### Training the agent

#try:
#     %%time
#except:
#     pass
#
# Optimize by wrapping some of the code in a graph using TF function
tf_agent.train = common.function(tf_agent.train)

# Reset the train step
tf_agent.train_step_counter.assign(0)

# Evaluate the agent's policy once before training
avg_return = compute_avg_return(eval_env, tf_agent.policy, num_eval_episodes)
returns = [avg_return]

for iterationIdx in range(num_iterations):

    print("Iteration:␣",iterationIdx)

    #start_time = time.time()
    # Collect a few episodes using collect_policy and
    # save to the replay buffer
    maxStepsPerEpisodeVal = 1000
    #if iterationIdx < 10:
    #     maxStepsPerEpisodeVal = 1000
    #else:
    #       maxStepsPerEpisodeVal = 10


    print("collecting episodes")
    collect_episode(
            train_py_env, tf_agent.collect_policy,
            collect_episodes_per_iteration, maxStepsPerEpisodeVal)


    #print(time.time()-start_time)

    # Use data from the buffer and update the agent's network
    #iterator = iter(replay_buffer.as_dataset(sample_batch_size=1))
    print("preparing training data")

    iterator = iter(replay_buffer.as_dataset(sample_batch_size=1))
    trajectories, _ = next(iterator)




    print("training agent")
    #start_time = time.time()
    train_loss = tf_agent.train(experience=trajectories)
    #print(time.time()-start_time)

    replay_buffer.clear()

    step = tf_agent.train_step_counter.numpy()

    if step % log_interval == 0:
        print('step = {0}: loss = {1}'.format(step, train_loss.loss))

    if step % eval_interval == 0:
        print("computing avg return")
#         avg_return = compute_avg_return(eval_env, tf_agent.collect_policy,
            num_eval_episodes)
        avg_return = compute_avg_return(eval_env, tf_agent.policy,
            num_eval_episodes)

        print('step = {0}: Average Return = {1}'.format(step, avg_return))
        returns.append(avg_return)


##########################################
### Visualization

iterations = range(0, num_iterations + 1, eval_interval)
exec_times = 0

save_data(tf_agent, "REINFORCE", iterations, returns,
    exec_times, learning_rate,
    save_dir="./data/REINFORCE_8000iterations_50episodesPerIteration_")

steps = range(0, (num_iterations+1)*collect_episodes_per_iteration,
    eval_interval*collect_episodes_per_iteration)
plt.plot(steps, returns)
plt.ylabel('Average Return')
plt.xlabel('episodes')
plt.ylim(top=50)
plt.savefig("learningCurve_atari_REINFORCE.png")
```