AA228 Final Report

Peter Schleede, Richard Hsieh, Ben Share

December 2018

1 Abstract

Reinforcement learning applied to game-playing has garnered considerable attention in the past few years. In this project, various reinforcement learning techniques are surveyed to train an agent to play the Atari racing game Enduro. Advantage Actor Critic (A2C) was implemented and proved to be ineffective, matching results published by OpenAI. Deep Q-Network was also explored which highlighted the challenges of requiring long training times, especially for a game like Enduro with sparse rewards. The most successful algorithm implemented was a shallow, quasi-linear neural network that chooses the optimal policy based on a learned scoring function. A high score of 65 was achieved by this method over a training run of 100 episodes, far better than both our baseline and the score obtained with DQN. This highlights the fact that using a relatively simple algorithm tuned with domain-specific knowledge can achieve favorable results.

2 Introduction

This project focused on implementing reinforcement learning on the Atari game Enduro. This game was simulated using the OpenAI `gym` package [1]. We wanted to try to implement several different approaches to compare them against each other, as well as against simple heuristics we came up with on our own.

Reinforcement learning methods seek to find optimal policies for Markov Decision Processes with uncertainty over the model, meaning that we do not know with certainty the results of taking certain actions from various states that we may be in. It uses an approximation that relates the current state and action to an expected future reward. The actual observed rewards, along with observed state transitions are used to train this approximation to more aptly match the data. Some methods, such as Q-learning and Sarsa store lookup tables that can be used to model the probability of state transitions, along with expected values. The addition of eligibility traces helps them assign credit to the states and actions leading up to a reward [2].

One of the drawbacks to using the above methods is that they require a lookup table of size $O(|S|x|A|)$. This is intractable for problems with a large state and/or action space. Instead, approximators are required. These approximators will likely be global because it is unlikely you will actually visit every state and take any action enough times to have a good estimate at each location. Deep learning methods such as Deep Q Network [3] and Advantage Actor Critic [4] have proven successful in recent times. They allow use of neural networks that accept high dimensional states as inputs, including convolutional neural networks that operate on images. The outputs of these networks can be viewed as a distribution over possible actions. We investigate these methods.

3 Problem

Enduro is a racing game, shown in Fig. 1, with the objective to pass as many opponent vehicles as possible without getting passed yourself. In the `gym` package, a reward of +1 is given for each car passed and -1 for each car that passes the agent. However, the net reward cannot drop below 0. There are 9 available actions.

Figure 1: Screenshot of the Enduro racing game (from [1])

There were two different versions of the game we tried. The first, `Enduro-ram-v0`, has a state consisting of the entire 128 MB of memory of the Atari for a given state of the game. Each byte can take on 256 unique values. The other, `Enduro-v0`, uses the 210x160x3 rendered game as the state, with each pixel able to take on 256 unique values. We used each for different approaches.

4 Algorithms

4.1 Base Heuristic

For this, we simply applied a single action as an input for all time. This provides a baseline that any useful algorithm should be able to beat. We find that there is no net reward for choosing actions 0 and 2-8. We find a net reward of 16 for choosing action 1 and holding to it. This reward is deterministic because the reset for the environment always goes to the same state. We consider 16 the useful baseline.

4.2 Neural Network Action Estimators

This approach takes the full visual data (x) as input, and generates scores for each action. Actions are then chosen as:

$$action = \arg\max score(a) \tag{1}$$

where the scoring function is learned by the network. We used two network architectures to generate scores.

The first is a shallow, quasi-linear network. Its internal model consists of weights W, where W.shape = x.shape.extend($num\_actions$). For a given input, its action choice is thus:

$$\arg\max x \otimes W \tag{2}$$

The second model is a more complex convolutional network. It applies convolutional filters to the (visual) input data, passes it through a dense layer, then funnels it down to $num\_actions$ values. These are then used to choose an action in the same way as above. We experimented with several features of this network: number of convolutional layers; convolution kernel size and number; activation function for both the convolutional

and dense layers; and hyperparameters including initializer and learning rate. Our final version of the network used a single convolutional layer, kernel sizes of 5 with 16 distinct filters, and ReLU activation on both convolutional and linear layers:

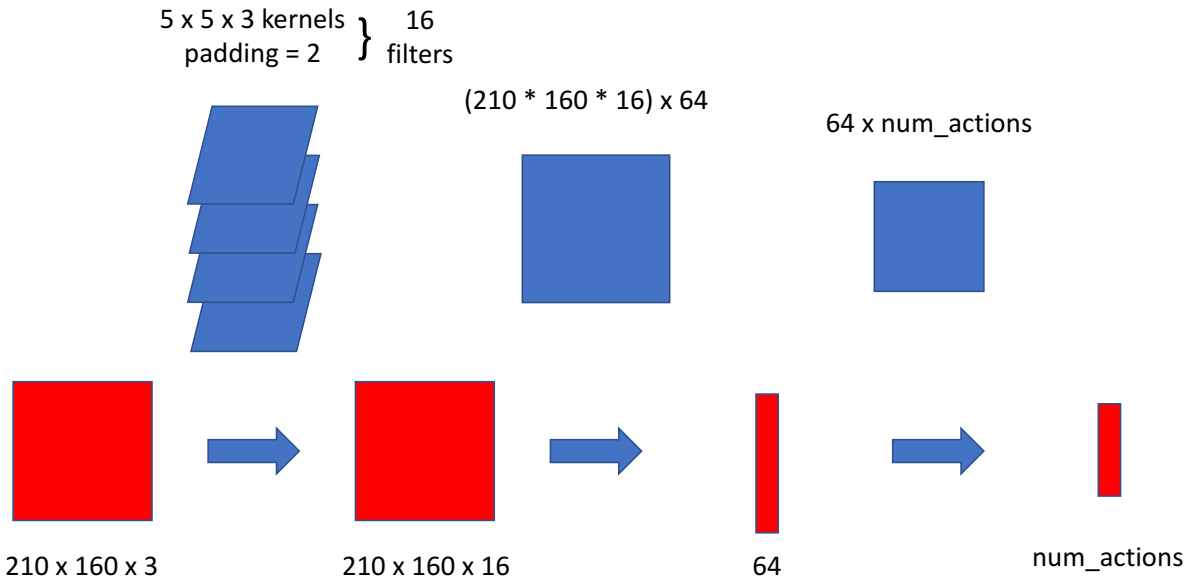

Figure 2: The convolutional architecture used

4.3 Advantage Actor Critic (A2C)

A2C is an extension of policy gradient algorithms. Policy gradient algorithms, upon receiving rewards, increase the probability of choosing all the actions that were taken in the states that led to that reward. However, this has a downside in that it can increase the likelihood of taking a poor action when that poor action was part of a chain leading to a reward. Similarly, a good action can be penalized if it was part of a chain that led to a negative reward.

A2C instead consists of two function approximators. These are the actor and critic. The actor evaluates the current state and returns a distribution over possible actions. The action taken is sampled from this distribution. The critic also evaluates states and attempts to predict how valuable it is to be in each state. In our implementation, the actor and critic shared convolutional layers of a neural network and then branched off separately into fully connected layers.

The actor is evaluated at each state. A record is kept of the states, the actions taken, and the rewards received. Every n steps, the network is trained by reflecting on the past decisions. The critic is used to predict the future value of subsequent states and actions, and of each state that was encountered. The actual rewards received are compared to the expected value of each state, and the differences between the predicted reward and actual reward are calculated, known as the advantages. In addition, a loss based on the action probabilities and rewards received is calculated. Finally, an entropy term is added to encourage exploration. These three terms are summed to form a loss function used to train the network. The loss function is given below.

$$L = L_v + L_a - \epsilon \tag{3}$$

$$L = \frac{1}{N}\sum_{i=1}^{N}(V(s)_i)^2 - \frac{1}{N}\sum_{i=1}^{N}(\log P(a_c)_i * V(s)_i) - \frac{1}{N}\sum_{i=1}^{N}(P(a_i)\log P(a_i)) \tag{4}$$

In the above equations, the L terms are the losses corresponding to the advantage and actor, respectively and ϵ is the entropy term. $V(s)$ is the advantage of a state, $P(a_c)$ is the probabilities of the action chosen. $P(a_i)$ is the predicted probability of action i. An excellent visual explanation of A2C is given at [5].

4.4 Deep Q Network - Enduro RAM

Deep Q Networks (DQN) perform global function approximation for Q-learning. Specifically, the algorithm utilizes neural networks for function approximation to automatically extract features and learn weights. This is well suited for the `Enduro-ram-v0` version of the game since the 128 MB state vector is incomprehensible to a human user. In essence, DQN performs incremental updates to the Q-values as such:

$$w \leftarrow w - \eta[Q(s,a;w) - (r + \gamma \underset{a'}{max} Q(s',a';w))]\theta(s,a) \tag{5}$$

where η is the learning rate, γ is the discount factor for future rewards, $\theta(s,a)$ are feature vectors, and w are the corresponding weights.

Since DQN is an online method, the agent builds up a set of training data by interacting with the game environment following an ϵ-greedy policy. As observations are collected at each time step, these are first stored in a replay memory queue. The agent then randomly samples a batch of data from this replay memory at each step to update the weights (as shown in Fig. 3. The neural network is then refitted following stochastic gradient descent where the weights are updated for each sample in the batch by minimizing the squared loss function in Eq. (4.4), where $r + \gamma \underset{a'}{max} Q(s',a')$ is the target and $Q(s,a)$ is the prediction.

Figure 3: Replay memory for DQN

$$loss = [(r + \gamma \underset{a'}{max} Q(s',a')) - Q(s,a)]^2 \tag{6}$$

During training the DQN algorithm also keeps track of two neural network models: one for predicting the optimal policy and one for maintaining target Q-values. Since we are refitting the policy model at every time step, the Q-values are constantly changing. To avoid refitting the policy model to a moving target, a separate target model keeps track of target Q-values and is only periodically updated by copying the weights from the policy model.

5 Results

5.1 DQN

The results in Table 1 were achieved using DQN with exponential decay in ϵ (exploration rate):

5.2 A2C

After getting A2C running, we trained it for a while and found that we never received any net reward. We initially decided that must be because there was not enough time to train. However, as we researched more, we found a baseline from OpenAI [6], that showed that, in 10 million episodes, they never received a net reward from A2C. We concluded this was thus a less promising method to pursue.

Layers	Nodes per Hidden Layer	ϵ_{max}	ϵ_{min}	Episodes	Training Time	Average Score
3	256	1.0	0.87	10	0.83 hr	1.50
3	256	1.0	0.1	100	8.3 hr	0.34

Table 1: DQN results training over 10 and 100 episodes

5.3 Neural Network Estimators

The simple network we used ended up achieving the overall highest performance of any of the methods we tried. Its aggregate scores for individual runs during 100 episodes of training are shown in Fig. 4:

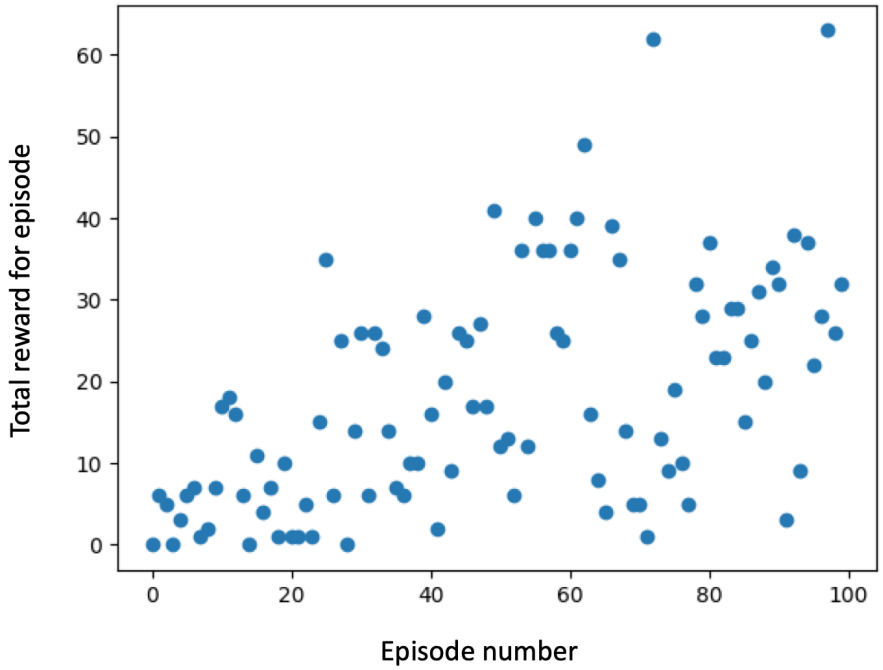

Figure 4: Performance of neural network model during training

Its peak score over this period was 65–better than any of our other methods, and a considerable improvement on the baseline of 16. This suggests the model was able to genuinely capture some of the strategy that goes into improving on the obvious tactics. The data has a strong upwards trend as well, suggesting that additional training might continue to improve the model's performance. One thing to note is that the model experiences a very high degree of variability, as the updates to its weights resulted in significant differences in performance from episode to episode.

Meanwhile, the convolutional network performed more poorly, achieving a max score of 20 and generally hovering around the baseline range. There are several likely explanations for this. First and foremost, the additional complexity of the network made it much slower to train, and we weren't able to tune it as much as the linear model. More training time (and better management of GPU resources) would almost certainly improve its performance. And second, due to the complexity and black box nature of deep models, it was harder to insert domain-specific knowledge into the algorithm's training process–a factor that, as discussed below, appears to have been extremely significant.

6 Conclusions

Overall, we were able to implement a number of complex algorithms to tackle the problem of Enduro, and achieved reasonable results. With the shallow neural network predictor, in particular, we achieved considerable improvements over the baseline methods. Nevertheless, our work highlighted several interesting challenges of Enduro for traditional AI models.

One of the primary challenges is that the rewards for this environment are very sparse. The agent only receives a positive reward of +1 when it passes a car. For the majority of state-action combinations, the reward is 0. As such, even for a training run using DQN over 100 episodes for 8.3 hours (which equates to roughly 200,000 iterations of refitting the model) the average score is a sub-par 0.34. Online methods like DQN can take a long time to train as they require the agent to perform significant exploration in order to propagate non-zero Q-values through the entire state-action space. This is highlighted by Table 1 where a higher average score is achieved in the 10 episodes than in 100 episodes. In the 10-episode run, ϵ is high which allows the agent to explore the state-action space more. On the other hand, with the 100-episode run as ϵ decays the agent begins exploiting a sub-optimal model that hasn't seen the majority of the state-action space. This leads the agent down dead-ends as it gets stuck in local optima. One improvement for improving our DQN algorithm would be to optimize it for using GPUs and increasing the training time. Another improvement would be to utilize eligibility traces to help propagate Q-values through the state-action space faster.

This points to a broader challenge with this type of task: the need to customize generic learning algorithms to fit the specifics of the problem. For instance, MDP algorithms learned in class formulate the problem in terms of states, actions, and immediate rewards dependent on state and action. While this approach is sufficient to describe the problem, straightforward application of these algorithms is fundamentally unable to generalize to this problem. For instance–since the agent receives rewards only for passing other cars, the only action that directly elicits positive rewards is the forward action. Naive early implementations of our algorithms thus reduced to the baseline agent. On the other hand, without a semi-functioning policy, the agent will simply never pass other cars, and thus never receive rewards–making it unable to learn.

One strategy we found effective in solving the direct reward problem was to collect rewards over a defined timeframe, then propagate changes backwards given the aggregate reward over the period. This allows non-forward actions–i.e. moving sideways to avoid a car that we'll subsequently pass–to receive rewards for the progress made afterwards. Similarly, we were able to break the no-progress loop by initializing agents with a preference for the forward action–essentially bootstrapping their learning process. In a sense, this is akin to reframing the problem: instead of simply learning which action to take, the agent should learn when to *deviate* from its default action of forward movement, and which action to take. This method was especially effective with the linear network model, which was able to quickly incorporate this knowledge into its own action choices; future work should include experimenting with alternate ways to incorporate this formulation into other models. Given the relative simplicity of this model and its high resulting performance, this underlines the importance of domain-specific knowledge and fine-tuning algorithms to the problem domain. Our code can be obtained from https://github.com/rhsieh91/aa228_project.

7 Group Contributions

Peter - A2C code and description, Introduction, Problem
Richard - DQN code and description, Abstract, Conclusion
Ben - NN Estimators code and description, Results

Appendix A A2C Implementation in Python

A.1 train.py

```
from model import ActorCritic
import gym
import a2c_config as cfg
import torch
import torch.optim as optim
from torchvision import transforms
import numpy as np
import time
import os

def save_checkpoint(state, path='results', filename='checkpoint.pth.tar'):
    if not os.path.isdir(path):
        os.makedirs(path)
    path += ('/' + filename)
    torch.save(state, path)

'''
Function to get a subwindow of the state
Inputs:
    s:          state, numpy array
    tl:         list, top left corner [y,x]
    br:         list, bottom right corner [y,x]
Output:
    s_sub:      state, as a subwindow
'''
def GetSubWindow(s, tl, br):
    s_sub = s[tl[0]:br[0], tl[1]:br[1]]
    return s_sub

'''
Function to convert a list of numpy images to a torch Tensor that
can be input to the network.
Inputs:
    s:                  either a list of images as ndarrays or a
                        single ndarray
Outputs:
    torch_tensor:       torch FloatTensor of s
'''
def StateToTorch(s):
    # if it is a list, convert it to a numpy array
    s = np.array(s)

    # if it is only one image, unsqueeze the first dimension
    if len(s.shape) == 3:
        s = np.expand_dims(s, axis=0)

    # move the channel axis to second
    s = np.moveaxis(s, [0,1,2,3], [0,2,3,1])

    # convert to Tensor
```

```
    torch_tensor = torch.from_numpy(s).float() / 255

    return torch_tensor

def GetTotalLoss(states, actions, rewards, dones, net, cuda):

    # Calculating the ground truth "labels" as described above
    state_values_true = net.CalcActualStateValues(states,
                                                  rewards,
                                                  dones,
                                                  StateToTorch,
                                                  cuda)

    s = StateToTorch(states)
    if cuda:
        s = s.cuda()
    action_probs, state_values_est = net.EvaluateActions(s)
    action_log_probs = action_probs.log()

    a = torch.stack(actions).view(-1,1)
    chosen_action_log_probs = action_log_probs.gather(1, a)

    # This is also the TD error
    advantages = state_values_true - state_values_est

    entropy = (action_probs * action_log_probs).sum(1).mean()
    action_gain = (chosen_action_log_probs * advantages).mean()
    value_loss = advantages.pow(2).mean()
    total_loss = value_loss - action_gain - 0.0001*entropy

    return total_loss

# create our actual environment
env = gym.make('Enduro-v0')

# define the network
n_actions = env.action_space.n
net  = ActorCritic(cfg.input_size, cfg.filter_sizes, n_actions, cfg.gamma)

# move onto GPU is we can and want to
if cfg.use_cuda and cfg.cuda_option:
    print('Using GPU')
    net = net.cuda()

optimizer = optim.Adam(net.parameters(), lr=cfg.lr)

s = GetSubWindow(env.reset(), cfg.tl, cfg.br)
finished_games = 0
prev_finished = 0
iter = 0
reward_counter = 0
best_reward = 0
per_game_rewards = []
```

```
start_time = time.time()
while finished_games < cfg.total_games:
    states, actions, rewards, dones = [], [], [], []

    for i in range(cfg.n_steps):
        s_tensor = StateToTorch(s)
        if cfg.use_cuda and cfg.cuda_option:
            s_tensor = s_tensor.cuda()

        action_probs = net.GetActionProbs(s_tensor)
        a = action_probs.multinomial(1).data[0][0]
        sp, r, done, _ = env.step(a)

        states.append(s)
        actions.append(a)
        rewards.append(r)
        dones.append(done)

        reward_counter += r

        if r != 0:
            print('Recieved a reward of %d' %r)

        if done:
            s = GetSubWindow(env.reset(), cfg.tl, cfg.br)
            finished_games += 1
            per_game_rewards.append(reward_counter)
            print('Reward for game %d: %d' % (finished_games, reward_counter))

            if reward_counter > best_reward:
                print('Saving Model')
                save_checkpoint({'net_state_dict': net.state_dict(),
                                 'reward': reward_counter},
                                path='/model', filename='checkpoint_model.pth.tar')
                best_reward = reward_counter

            reward_counter = 0
            iter = 0
        else:
            s = GetSubWindow(sp, cfg.tl, cfg.br)

    # reflect on last few states and actions
    if cfg.use_cuda and cfg.cuda_option:
        total_loss = GetTotalLoss(states, actions, rewards, dones, net, 1)
    else:
        total_loss = GetTotalLoss(states, actions, rewards, dones, net, 0)
    optimizer.zero_grad()
    total_loss.backward()
    # nn.utils.clip_grad_norm(model.parameters(), 0.5)
    optimizer.step()
    iter += 1
    if iter % 100 == 0:
        print('Now on iteration %d of game %d' % (iter, finished_games+1))
```

```
    if finished_games != prev_finished:
        prev_finished = finished_games
        print('Now finished %d game(s)' % finished_games)

end_time = time.time()

print('Per Game Rewards:')
print(per_game_rewards)

print('Total time: %f' % (end_time-start_time))
print('Time per game: %f' % ((end_time-start_time)/cfg.total_games))
```

A.2 model.py

```
import torch
from torch import nn
import torch.nn.functional as F
import numpy as np

'''
Function to create a 3x3 convolutional layer with a given stride
Inputs:
    input:              int, number of input filters
    output:             int, number of output filters
    stride:             int, stride
Outputs:
    layer
'''
def Conv3x3(input, output, stride):
    if stride == 1:
        return nn.Conv2d(input, output, kernel_size=3, stride=stride, padding=0, bias=False)
    elif stride == 2:
        return nn.Conv2d(input, output, kernel_size=3, stride=stride, padding=1, bias=False)

'''
Function to flatten the output of convolutional layers
Input:
    x:                  output of conv layers
Output:
    x_flat:             flattened version for linear operations
'''
def Flatten(x):
    return x.view(x.size(0), -1)

'''
Class to create the model. Based on code at:
https://github.com/rgilman33/simple-A2C/blob/master/3_A2C-nstep-TUTORIAL.ipynb.
'''
class ActorCritic(nn.Module):
    def __init__(self, input_size, filter_sizes, n_actions, gamma):
        super(ActorCritic, self).__init__()
        self.gamma = gamma

        self.conv1 = Conv3x3(filter_sizes[0], filter_sizes[1], 2)
```

```
        self.conv2 = Conv3x3(filter_sizes[1], filter_sizes[2], 2)
        self.conv3 = Conv3x3(filter_sizes[2], filter_sizes[3], 2)

        output_edge = np.array(input_size) / 8
        output_size = int(np.prod(output_edge) * filter_sizes[3])
        self.actor = nn.Linear(output_size, n_actions)
        self.critic = nn.Linear(output_size, 1)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.relu(self.conv2(x))
        x = F.relu(self.conv3(x))
        return x

    def GetActionProbs(self, x):
        x = Flatten(self(x))
        return F.softmax(self.actor(x), dim=1)

    def GetStateValue(self, x):
        x = self(x)
        return self.critic(Flatten(x))

    def EvaluateActions(self, x):
        x = self(x)
        return F.softmax(self.actor(Flatten(x)), dim=1), self.critic(Flatten(x))

    def CalcActualStateValues(self, states, rewards, dones, transform, cuda):
        R = []
        rewards.reverse()

         # If we happen to end the set on a terminal state, set next return to zero
        if dones[-1] == True:
            next_return = torch.Tensor([0]).data[0]
            if cuda:
                next_return = next_return.cuda()

        # If not terminal state, bootstrap v(s) using our critic
        else:
            s = transform(states[-1])
            if cuda:
                s = s.cuda()
            next_return = self.GetStateValue(s).data[0][0]

        # Backup from last state to calculate "true" returns for each state in the set
        R.append(next_return)
        dones.reverse()
        for r in range(1, len(rewards)):
            # discount rewards
            if not dones[r]:
                this_return = rewards[r] + next_return * self.gamma
            else:
                this_return = torch.Tensor([0]).data[0]
                if cuda:
                    this_return = this_return.cuda()
```

```
            R.append(this_return)
            next_return = this_return

        R.reverse()
        state_values_true = torch.stack(R).unsqueeze(1)

        return state_values_true

def test():
    input_size = [152, 152]
    filter_sizes = [3, 64, 128, 256]
    n_actions = 9
    net = ActorCritic(input_size, filter_sizes, n_actions)
    print(net)
    x = torch.randn(1,filter_sizes[0],input_size[0], input_size[1])
    y = net(x)
    print(y.size())
    y = net.get_action_probs(x)
    print(y.size())
    print(y)
    print(y.sum())
    return net

if __name__ == '__main__':
    net = test()
```

A.3 deep_rl_agent.py

```
import gym
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.models import Model
import numpy as np
import random
from matplotlib import pyplot as plt
import sys
import argparse
import copy

env = gym.make('Enduro-v0')

class RandomAgent:
        def __init__(self, num_actions):
                self.num_actions = num_actions

        def act(self):
                return random.randrange(self.num_actions)

        def train(self):
                pass

class BoringAgent:
        def __init__(self, action):
                self.action = action
```

```
        def act(self):
                return self.action

        def train(self):
                pass

class LinearAgent:
        def __init__(self, num_actions=9, input_shape = [210, 160, 3]):
                self.weights = np.random.rand(input_shape[0], input_shape[1], input_shape[2], num_actio
                self.num_actions = num_actions

        def act(self, observation):
                action_vals = np.sum((np.expand_dims(observation, axis=3) * \
                        self.weights).reshape(-1, self.num_actions), axis=0)
                return np.argmax(action_vals)

        def act_rand(self, observation, p):
                rand = random.random()
                if rand > 0.2 * (1 - p):
                        rand = random.random()
                        if p > rand:
                                return self.act(observation)
                        else:
                                return 1
                return random.randrange(self.num_actions)

        def train(self, num_episodes=20.0, window=30, alpha=1e-3):
                reward_list = []
                step_list = []
                best_weights = np.zeros(self.weights.shape)
                best_reward = 0
                for i in range(int(num_episodes)):
                        observation = env.reset()
                        p = i / num_episodes
                        done = False
                        total_reward = 0
                        steps = 0
                        rewards = []
                        actions = []
                        while not done:
                                a = self.act_rand(observation, p)

                                new_observation, r, done, _ = env.step(a)
                                actions.append(a)
                                cur_alpha = alpha / (steps + 1)
                                if steps >= window:
                                        rewards[steps % window] = r
                                        self.weights[:, :, :, actions[(steps + 1) % window]] += cur_alp
                                else:
                                        rewards.append(r)
                                if steps == window - 1:
                                        self.weights[:, :, :, actions[(steps + 1) % window]] += cur_alp
                                observation = new_observation
```

```
                                total_reward += r
                                steps += 1
                        reward_list.append(total_reward)
                        if total_reward > best_reward:
                                best_reward = total_reward
                                best_weights = self.weights
                        step_list.append(steps)
                        print "Episode %d:\n\tp = %.2f\n\tsteps = %d\n\treward = %d" %(i, p, steps, tota
                plt.scatter(range(int(num_episodes)), reward_list)
                plt.savefig("linear_rewards")
                np.save("linear_weights", best_weights)
                self.weights = best_weights

        def loadFromSaved(self, fn):
                self.weights = np.load(fn)

class ConvolutionalAgent:
        def __init__(self, num_actions=9, input_shape = [210, 160, 3], nums_of_filters = [16], learning_
                initializer = tf.keras.initializers.random_normal()
                self.input_shape = input_shape
                self.num_actions = num_actions
                self.learning_rate = learning_rate

                obs_input = keras.Input(shape=input_shape)

                layers = []
                conv_shape = copy.deepcopy(input_shape)
                for num_filters in nums_of_filters:
                        layers.append(
                        tf.layers.Conv2D(input_shape=conv_shape, filters=num_filters, kernel_size=5,\
                                                strides=1, padding="same", activation=tf.nn.relu
                        conv_shape[2] = num_filters
                layers.extend([
                        tf.layers.Flatten(input_shape=input_shape),
                        tf.layers.Dense(64, kernel_initializer=initializer, activation=tf.nn.relu),
                        tf.layers.Dense(num_actions, kernel_initializer=initializer)
                ])
                m = keras.Sequential(layers)
                out = m(obs_input)

                self.model = Model(inputs=obs_input, outputs=out)

        def act(self, observation, sess):
                x = tf.placeholder(tf.float32, self.input_shape)
                newShape = [-1]
                newShape.extend(self.input_shape)
                batch = tf.to_float(tf.reshape(x, newShape))
                action_vals = action_vals = tf.reshape(self.model(batch), [-1])
                action = tf.argmax(action_vals)
                a = sess.run(action, feed_dict={x: observation})
                return a

        def act_rand(self, p):
                rand = random.random()
```

```
        if rand > 0.2 * (1 - p):
                rand = random.random()
                if p > rand:
                        -1
                else:
                        return 1
        return random.randrange(self.num_actions)

def train(self, num_episodes=100, window=30):
        x = tf.placeholder(tf.float32, self.input_shape)
        newShape = [-1]
        newShape.extend(self.input_shape)
        batch = tf.reshape(x, newShape)

        action_vals = tf.reshape(self.model(batch), [-1])
        action = tf.argmax(action_vals)
        accum_reward = tf.placeholder(tf.float32)

        trainer = tf.train.GradientDescentOptimizer(self.learning_rate)
        score = -action_vals[action] * accum_reward
        update_step = trainer.minimize(score)

        init = tf.global_variables_initializer()

        reward_list = []
        step_list = []
        with tf.Session() as sess:
                sess.run(init)
                for i in range(int(num_episodes)):
                        if (i % 1e2) == 0:
                                print "Episode %d" %i
                        p = i / num_episodes
                        observation = env.reset()
                        done = False
                        total_reward = 0
                        steps = 0
                        rewards = []
                        actions = []
                        while not done:
                                if steps % 10 == 0:
                                        print steps, "-", total_reward
                                a = self.act_rand(p)
                                if a < 0:
                                        a = sess.run(action, feed_dict={x: observation})
                                next_observation, r, done, info = env.step(a)

                                total_reward += r
                                actions.append(a)

                                if steps >= window:
                                        rewards[steps % window] = r
                                        _ = sess.run(update_step, feed_dict={x: observation, ac
                                else:
                                        rewards.append(r)
```

```
                                        if steps == window - 1:
                                                _ = sess.run(update_step, feed_dict={x: observation, ac

                                        observation = next_observation
                                        steps += 1
                                reward_list.append(total_reward)
                                step_list.append(steps)
                        self.model.save("conv_weights.h5")

                def loadFromSaved(self, fn):
                        self.model = keras.models.load_model(fn)


if __name__ == "__main__":
        parser = argparse.ArgumentParser()
        parser.add_argument("--agent")
        parser.add_argument("--use_saved_weights", action="store_true")
        parser.add_argument("--animate", action="store_true")
        args = parser.parse_args()

        if args.agent:
                if "conv" in args.agent:
                        agent = ConvolutionalAgent()
                else:
                        agent = LinearAgent()
        else:
                agent = LinearAgent()

        use_saved_weights = args.use_saved_weights
        animate = args.animate


        observation = env.reset()
        if use_saved_weights:
                if args.agent and "conv" in args.agent:
                        agent.loadFromSaved("conv_weights.h5")
                else:
                        agent.loadFromSaved("linear_weights.npy")
                print "Weights loaded"
        else:
                agent.train()
                print "\n\nDone training"
                observation = env.reset()

        if animate:
                env.render()
        done = False
        reward = 0
        sess = tf.Session()
        init = tf.global_variables_initializer()
        sess.run(init)
        while not done:
                if args.agent and "conv" in args.agent:
                        action = agent.act_rand(0.5)
```

```
                        if action < 0:
                                action = agent.act(observation, sess)
                else:
                        action = agent.act(observation)
                print action
                observation, r, done, info = env.step(action)
                reward += r
                if animate:
                        env.render()
        print reward
```

A.4 a2c_config.py

```
import torch

# define what portion of each state we will use (reduce computation in network)
tl = [0, 8]
br = [152, 160]
input_size = [br[0]-tl[0], br[1]-tl[1]]

use_cuda = torch.cuda.is_available()
cuda_option = 1
filter_sizes = [3, 64, 128, 256]
gamma = 0.95

lr = 3e-3

# gameplay configurations
total_games = 50
n_steps = 20
```

Appendix B DQN Implementation in Python

```
# This code is largely based on the Deep Q-Learning template published at:
# https://github.com/keon/deep-q-learning.git

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

EPISODES = 50
EPISODE_LEN = 10000

class DQNAgent:
    def __init__(self, state_size, action_size):
        self.state_size = state_size
        self.action_size = action_size
        self.memory = deque(maxlen=10000) # maximum number of samples stored in dataset

        self.gamma = 0.95 # discount rate
        self.epsilon = 1.0 # exploration rate
        self.epsilon_min = 0.1 # minimum exploration rate
        self.epsilon_decay = 0.99995 # exploration decay rate
        self.learning_rate = 0.001

        # Model for learning Q values and to extract optimal policy
        self.policy_model = self._build_model_3L()

        # Model for tracking target Q-values during policy_model updates. Target model is updated
        # less frequently to prevent having a moving target everytime we train the policy_model.
        self.target_model = self._build_model_3L()


    def _build_model_3L(self):
        """3-layer neural network"""
        model = Sequential()
        model.add(Dense(units=256, input_dim=self.state_size, activation='relu')) # input layer
        model.add(Dense(units=256, activation='relu'))
        model.add(Dense(units=self.action_size, activation='linear')) # output layer
        model.compile(loss='mse', optimizer=Adam(lr=self.learning_rate)) # mean squared loss
        return model


    def remember(self, state, action, reward, next_state, done):
        """Store s,a,r,s' by appending to self.memory."""
        self.memory.append((state, action, reward, next_state, done))


    def act(self, state):
```

```
    """Choose action randomly (explore) or by model prediction (exploit)."""
    # Decay exploration rate
    if self.epsilon > self.epsilon_min:
        self.epsilon *= self.epsilon_decay

    if np.random.rand() <= self.epsilon: # explore with i self.epsilon
        # return random.randrange(self.action_size)
        return random.randrange(1,4)

    act_values = self.policy_model.predict(state)
    return np.argmax(act_values[0])  # returns maximizing action


def replay(self, batch_size):
    """Train the neural net on the episodes in self.memory.
       Only N samples defined by batch_size are sampled from self.memory for training.
    """
    if len(self.memory) < batch_size:
        return

    minibatch = random.sample(self.memory, batch_size)
    for state, action, reward, next_state, done in minibatch:
        target = self.target_model.predict(state)

        if done:
            target[0][action] = reward
        else:
            target[0][action] = (reward + self.gamma * \
                                 np.amax(self.target_model.predict(next_state)[0]))

        self.policy_model.fit(state, target, epochs=1, verbose=0)


def target_update(self):
    """Update target model by copying over the weights from the policy model.
       We keep track of two different models to help
    """
    weights = self.policy_model.get_weights()
    target_weights = self.target_model.get_weights()

    for i in range(len(target_weights)):
        target_weights[i] = weights[i]

    self.target_model.set_weights(target_weights)


def load(self, name):
    self.policy_model.load_weights(name)


def save(self, name):
    self.policy_model.save_weights(name)
```

```
if __name__ == '__main__':
    env = gym.make('Enduro-ram-v0')
    state_size = env.observation_space.shape[0]
    action_size = env.action_space.n

    agent = DQNAgent(state_size, action_size)
    done = False
    batch_size = 32
    scores = [] # store scores for each episode

    for episode in range(EPISODES):
        state = env.reset()
        state = np.reshape(state, [1, state_size])
        score = 0

        start_time = time.time()
        for step in range(EPISODE_LEN):
            # env.render()
            action = agent.act(state) # DQN agent chooses next action
            next_state, reward, done, _ = env.step(action) # observe reward and new state

            score += reward # keep track of game score

            next_state = np.reshape(next_state, [1, state_size])
            agent.remember(state, action, reward, next_state, done) # add s,a,r,s' to self.memory

            # Train model
            agent.replay(batch_size)

            state = next_state # advance the state

            if step % 200 == 0:
                print('step = {}, exploration rate: {:.2}, score ={}'
                      .format(step, agent.epsilon, score))

            # If we reach the end of the game (i.e. game over, did not pass 200 cars in a day)
            if done:
                end_time = time.time()
                print('episode: {}/{}, score: {}, exploration rate: {:.2}, time elapse = {}'
                      .format(episode, EPISODES, score, agent.epsilon, end_time-start_time))
                scores.append(score)
                break

        end_time = time.time()
        scores.append(score)
        print ('episode = {}, score = {}, exploration rate: {:.2}, time elapsed = {}'.
               format(episode, score, agent.epsilon, end_time-start_time))

        # Update target_model after every episode
        agent.target_update()

        # Print average score every 10 episodes during training
        if episode % 10 == 0:
            print('AVERAGE SCORE = {}'.format(np.mean(np.asarray(scores))))
```

```
# Print final average score after all episodes are complete
if scores:
    print('FINAL AVERAGE SCORE = {}'.format(np.mean(np.asarray(scores))))

# Save weights after training is complete
agent.save(os.path.join(os.getcwd(), 'enduro_dqn_3L.h5'))
```

References

[1] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[2] Mykel J Kochenderfer. *Decision making under uncertainty: theory and application*. MIT press, 2015.

[3] Volodymyr Mnih, K Kavukcuoglu, D Silver, A Rusu, J Veness, M G Bellemare, A Graves, M Riedmiller, and A Fidjeland. Human-level control through deep reinforcement learning. *Nature,518*, pages 529–533, 2015.

[4] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937, 2016.

[5] Intuitive rl: Introduction to advantage-actor-critic (a2c. ://hackernoon.com/intuitive-rl-intro-to-advantage-actor-critic-a2c-4ff545978752.

[6] Openai atari baseline. http://htmlpreview.github.io/?https://github.com/openai/baselines/blob/master/benchmarks_atari10M.htm. Accessed: 2018-12-07.