**Solving Excitebike With Reinforcement Learning**

Supervisor: Michael Ryoo Author: Alexander Hu

**Abstract**

Excitebike is a game well suited for the purposes of reinforcement learning, yet little work has been done to explore how to utilize well known methods to create an agent capable of playing the classic retro game. We prepare the environment necessary to test the environment, build a baseline model as well as two DQN implementations, and analyze the results. Ultimately, we will show how proper setup can allow even basic reinforcement learning concepts to learn how to play Excitebike.

**Introduction**

Excitebike is an arcade and retro console game first made in 1984 by Nintendo. The game consists of a motorbike racer trying to go from the start to the finish of a straight line track with four lanes as quickly as possible. Along the way are multiple obstacles like ramps of various shapes and sizes as well as mud pits which the racer must navigate through.

Although this environment is well suited for basic reinforcement learning due to the inability to reach a gameover state beyond completing a race, few papers have been written in regards to this topic. Thus, this paper will explore how to tackle the Excitebike environment using behavioral cloning as well as Deep-Q networks.

The desired goal is as follows: Finish the first level of Excitebike fast enough to make the top 3 podium - in other words, finish before the time of 01:24:00 (one minute and twenty-four seconds).

**Environment Setup**

To provide an environment for the algorithms to work on, a custom environment was created using the Gym-Retro framework. The relevant custom integration files are provided at the end of the paper - please refer to relevant resources for proper setup. For simply loading up pre-trained weights, refer to the README file.

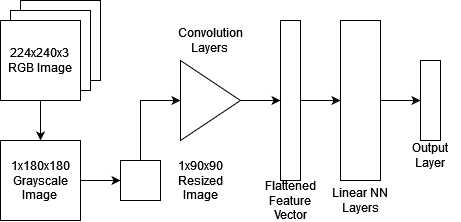
The environment rewards 5000 points for finishing the race and imposes a 1 point penalty for every action that does not finish the race. This approximates finishing the race in the alloted time, although it turns out that the total reward could be a relatively small negative value and still achieve the desired goal.

**Image Preprocessing/Neural Net Structure**

With the Excitebike environment set up, the original observations received from the environment come as an RGB 3-channel image with height 224 and width 240.

Parts of the top, right, and left portions of this image are cropped out as the bike racer will never end up there and obstacles past the racer cannot be interacted with. In addition, the time to beat and actual time is removed as they are extraneous - the time to beat is represented by the environment’s reward function and the time spent racing is calculated by the number of steps in any given full trajectory.

A small part of the right side of the screen is also cropped out to bring the dimensions to 180x180. The slight loss of information should not make it significantly harder for a convolutional network to differ between obstacles.



*Figure 1: Basic Structure of Neural Network Design Across Algorithms*

After acquiring an 180x180 reformatted image, the image is converted to grayscale, reducing the channels from 3 to 1. The image is finally scaled down to 90x90 and reshaped for proper use as the inputs to pytorch neural networks.

These images are first fed into convolutional neural network layers that transform the image array into a single dimensional feature vector. That feature vector is then fed into linear neural network layers to provide the necessary calculations for the desired outputs at the output layers. The output layer represents the score assigned to each possible action the algorithm may take to interact with the Excitebike environment.

Figure 1 provides a visual of the overall design across all algorithms.

**Algorithms**

One design choice stands out across all three algorithms that severely impacts the way they all approach the Excitebike problem - the choice to eliminate the possibility of performing the “do-nothing” action.

While Excitebike only has one gameover state, aka finishing the race, it is possible for an agent to never finish by always taking the option to not press a button to move forward. To avoid the complications from this, steps are taken in each algorithm to remove that possibility.

All algorithms utilize the same convolutional layer scheme to process the 1 x 90 x 90 grayscale image into a feature vector. The convolutional network parameters were not well understood and therefore will be glossed over for readers to consider as a possible avenue of exploration.

The following three algorithms are setup and utilized with the environment as follows:

Baseline Model

For the baseline model, we seek to train it with supervised learning, providing it with data from a human expert. In this specific instance, the problem is to identify the correct action given some image input. This could be structured as a classifier problem using a convolutional neural network that takes the environment observation as input and returns projected values of each possible action it could take.

The data was provided by a human expert playing twenty games manually and storing the moves in data files. The data files are converted into a data set with each entry representing one observation-action pair. These pairs are parsed through to generate the set of unique moves made by the human expert.

For the baseline model, the major concern is identifying the initial starting observation with the passive “do-nothing” action. This is particularly prevalent due to the technical limitations forcing the human expert to always have a late start at the beginning of each run.

This results in a majority of the observations matching the initial state to correspond to doing nothing. If such a classifier is then used to choose actions to run a race itself, the agent would never inch past the starting position. Thus, the algorithm eliminates all observation-action pairs with the passive action to prevent such a scenario. In the end, 15 other unique actions were identified.

From there, the data is split into a training and testing set in a 70/30 split. The classifier neural network is trained on the training set over two epochs before being evaluated on the testing set. Finally, this neural network is then utilized to play Excitebike and chooses actions every step based on given observation inputs.

The neural network for the baseline model has notably more nodes compared to the DQN models, utilizing 3136 x 6000, 6000 x 3000, 3000 x 1024, and 1024 x 15 layers. Each layer besides the final utilizes a ReLU activation function. The relative complexity of this network in particular is done to improve the accuracy of the classifier.

Basic DQN Model

The basic Deep-Q Network model utilizes the idea of taking an epsilon greedy action at every step, then training the neural network over a 128 sized batch of SARS data gathered from the replay buffer containing 20000 entries of the most recent steps taken by the neural network.

Both this basic model and the following DQN variant model handles the passive problem by limiting the action space to the following 8 actions: A, A+Right, A+Up, A+Down, B, B+Right, B+Up, B+Down. This allows the racer to always move forward, to change lanes while moving forward, and to angle the bike down when it is in the air to properly land back onto the track. While it limits the strategies the network could develop, it comes with the upset of massively reducing the amount of training time.

Both DQN models utilize a similar linear layer of 3136 x 1024, 1024 x 512, and 512 x 8 layers. All but the last utilize a ReLU activation function.

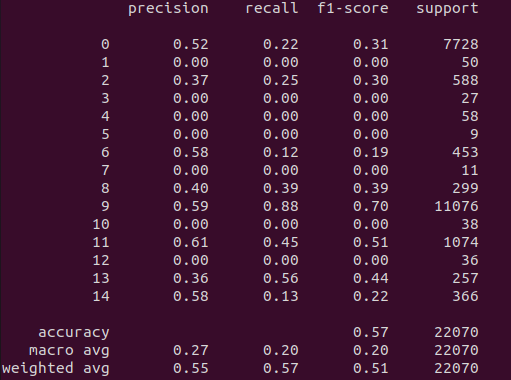
DQN Model with Best Trajectory Stored

The main differing point of this variation from the basic DQN model is the utilization of a separate buffer consisting of the trajectory that produced the best total reward as the neural network trains. This trajectory is used periodically by chance to pull batches from to train the model instead of the normal replay buffer.

The reasoning is that it is far easier to fill the replay data with junk trajectories since bad trajectories in this problem will always take a significantly larger amount of steps compared to preferred trajectories. Therefore, storing the best possible trajectory and using it as training may reinforce the model’s ability to pursue more optimal strategies. This is specifically accomplished by giving the algorithm a 1 in 6 chance to pull the batch from the best trajectory instead of from the normal replay buffer.

**Results**

Baseline Model



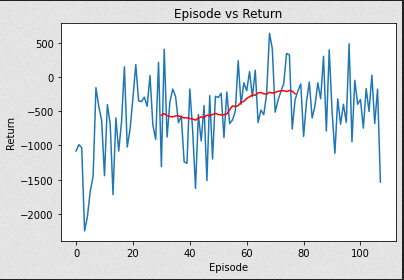
*Figure 2: Classification Report of Baseline Algorithm*

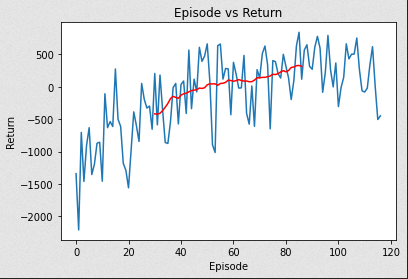
Figure 2 shows the classification report evaluating the testing set after training on the training set. Overall, these results suggest that the largest bucket was predicted most often, even to the detriment of other classification. However, some less common moves were still identified with surprising probability.

When running the game using the trained neural network, the model produces a deterministic result of -243 reward. It turns out this reward value is just enough to meet the goal and beat the time of 00:01:24.

Basic DQN Model vs DQN Model with Best Trajectory Stored

Figure X shows the rewards found at every episode for each model. Notably, the variant DQN model boasts a higher reward average as the algorithm runs compared to the default version.





*Figure 3: Reward per episode for default DQN (top) and DQN with best trajectory (bottom)*

However, the default DQN provides the far superior final result, which consists of running both algorithms using an epsilon value of zero and forcing greedy actions every time. The default algorithm scored a total reward of 620 while the variant ultimately produced a reward of -24. Both rewards are better than the classification model - in particular, the default DQN model produced a reward that is comparable to the human expert’s first few (discarded) runs.

**Video and Results Analysis**

This section refers to videos demonstrating the capabilities of the algorithms. The videos are provided in the google drive folder.

One notable issue with the baseline model has to do with the data aggregation - some of the possible moves had far too few data points for the algorithm to properly identify them in classification.

However, even for moves that only numbered in triple digits had a decent shot at classification. This manifests in the corresponding video as a relatively consistent ability to stick landings after launching from ramps.

Another issue is that the main classification buckets were between pressing only A and only B. In fact, in the recording those two are the default options for most of the run, with no ability to change lanes. It also was unable to properly identify when to press B well enough to prevent the bike from overheating, which was a common symptom from the classifier results.

For the basic DQN model, it showed a remarkable ability to avoid mud pits in particular. It also managed to learn how to stick landings decently well, although not as smoothly as the classification model. This leads to some interesting tactics that the human expert did not consider, like utilizing an “improper” landing to jump over one ramp. The default model also learned to properly manage the bike’s heat and alternates between A and B at a safe level.

The variant DQN model flubs many jumps and has a tendency to run itself into the grass at the bottom of the track. However, it also does seem to manage to move properly to avoid puddles and avoid overheating. One theory for the strange behavior may be the result of overfitting from the best trajectory recorded.

This overfitting may be the root cause of how the model achieved higher training rewards on average compared to the default model. While this was initially theorized to be a sign that the variant model was superior, such behavior actually was a sign of one of the everlasting reinforcement learning problems: catastrophic interference.

Without the reminder of more suboptimal runs possible that the default algorithm was more prone to encountering during training, the variant algorithm probably began to overfit far too heavily around the narrow scenarios defined by the best trajectory and was unable to handle observation states outside of that set properly. In some training attempts of the variant algorithm, heavy catastrophic interference was observed at the tail end of the runtime.

Indeed, this problem is potentially damaging for both algorithms - it is suggested to save the weight parameters of the model that show the best performance. This was accomplished by taking the highest average of the past 5 training runs and setting the weights at that time as the weights to save.

However, while the final results of the default algorithm utilized this “best weight” network, the variante network’s “best weight” implementation was catastrophically bad, showing a result of about -2500. Instead, the weights recorded at the end of the algorithm’s runtime were used. This reinforces the concept that higher training reward is not indicative of final performance.

**Conclusion**

All three models ultimately achieved the goal originally stated at the start of the paper - one of which managed to produce results that were comparable to a human’s. While some concessions were made in the action space of the environment to ease training, this should still provide some proof of concept of how even basic reinforcement learning methods could be used for playing Excitebike.

**Drive Folder**

<https://drive.google.com/drive/folders/1PKJlglPw4vxSFnJ-ZcPVO2S_ptpnZrdK?usp=sharing>

**References**

guiguilegui, AI Learning Excitebike with XGBoost

<https://guiguilegui.wordpress.com/2018/09/07/ai-learning-excitebike-using-xgboost/>

Gym Retro Documentation

<https://retro.readthedocs.io/en/latest/index.html>

Pytorch Tutorial - Training a Classifier

<https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html>

Berretta, Asperti, Experience Replay in Sparse Rewards Problems using Deep Reinforcement Techniques

Collinsworth, CSE525 HW1 Spring 2022