DeepRL on Breakout: an Arcade Game

Sudhanshu Raj 160050041
Ankit 160050044
Maitrey Gramopadhye 160050049
Neelesh Verma 160050062

November 2019

1 Introduction

We are trying to simulate an arcade game: Breakout in which a layer of bricks lines the top third of the screen and the goal is to destroy all the bricks with the help of a ball which bounces off the sides of the wall and a horizontal moving paddle.
We use existing implementations of the breakout game for the GUI of the game and a baseline model to build upon. These implementations use raw frames from the game as states and basic rewards that accumulate over every timestep for training the agent. In this project we achieve faster training by employing domain knowledge of the game and reward shaping. The baseline models we use [1], [2] are built upon OpenAI's Atari Gym. To reduce the state space, we preprocess the raw frames to extract a feature vector to use as states. In reward calculation too, in addition to the number of bricks destroyed, we take factors like time of tour and distance of ball from paddle in consideration.
In this report, section 2 explains the problem as an MDP and also the algorithms used for training the baseline models. Section 3 explains our method and describes in detail the features we extract from the frames, the modified model architecture and the reward function we chose. In section 4 we detail our experiments and provide our results, comparing them with the baselines. We conclude by providing, as future work, some methods that could improve the results and reduce training time further.

2 Background

2.1 Posing the problem as MDP

This section describes how the game Breakout can be posed as a MDP, so that it can be trained with reinforcement learning algorithms.

2.1.1 State space

Each time-step of the game can be specified with a state. The state space will represent the game at point of time. It will include information about the location of bricks, ball and the paddle. The state space will also include the velocity of ball.
In the baseline models, the state is defined as an array of the last n frames. We have defined it as a vector, derived from the raw frames, that contains the information required by the agent to take optimal action.

2.1.2 Actions

The action space includes four actions : left, right, noop and start game.

2.1.3 Reward

The agent gets a reward every time it interacts with the environment. In the baselines for breakout, it gets a positive reward of 1 if it breaks a brick and 0 reward otherwise. This is essentially the score we aim to maximise. However, we extend the reward function to include various other factors to speedup training.

2.2 Algorithms used to train baselines

This section describes the baseline models we chose and how they were trained.

2.2.1 Atari: DDQN [1]

Size of State Tensor:

```
- 4 x 84 x 84
```

The model architecture:

```
- Conv2D (None, 32, 20, 20)
- Conv2D (None, 64, 9, 9)
- Conv2D (None, 64, 7, 7)
- Flatten (None, 3136)
- Dense (None, 512)
- Dense (None, 4)

Trainable params: 1,686,180
```

DDQN Algorithm

Deep Q-Learning includes a maximization step over estimated action values which may sometimes learn unrealistically high action values. But we are not sure whether over-estimations effects (negatively) the performances of AI agents or not. Because, if all values are uniformly higher, then relative action preferences is still preserved and we won't expect the resulting policy to be any worse. On the other hand, if these over-estimations are not uniform, they might negatively affect the quality of the resulting policy.

It stores two models consecutively at same time. Both models take state as input and predict the Q(state=input,action) value for every action. First model is the main model which is used to get(predict) the next action using greedy method .In training, it uses the second model to update the first model using Q(s,a) = reward + best of Q(next_state,next_action) equation.

Pseudo Code

We have two models, First model(M1) and second Model(M2).
In training, it randomly takes some samples from the memory and update the model as below:
For every sample(current_state,action,reward,next_state,Is_next_state_a_terminal):

```
    Predict the q2(next_state,each_action) using M2(next_state)
    Take the max_next_q_value as best of q2(next_state,each_action)
    Predict the q1(current_state,each_action) using M1(current_state)
    if Is_next_state_a_terminal is True:
        $q1(current\_state, action) = reward$
    else :
        $q1(current\_state, action) = reward + \gamma * max\_next\_q\_value$
Fit(train) M1 such that M1(current_state) = q1(current_state,each_action)

Copy the weights of M1 to M2 after some training (40000 is used in baseline model)
```

2.2.2 A3C [2]

Size of State Tensor:

```
- 1 x 80 x 80
```

The model architecture:

```
- Conv2D (None, 32, 40, 40)
- Conv2D (None, 32, 20, 20)
- Conv2D (None, 32, 10, 10)
- Conv2D (None, 32, 5, 5)
- Flatten(None, 800)
- GRUCell (None, 256)
- (Fully connected (None, 1), Fully connected (None, 4))

Trainable params: 841,893
```

Actor-Critic Algorithm

In DQN, we use a single agent (also known as **worker**) to interact with a single environment. The way it interacts with the environment is already described in the section for DQN algorithm. Instead of deploying a single agent, A3C launches several workers asynchronously and allows them to interact with their own instance of their environment. They train their own copy of the network and share their results at the end of simulation.

Why we go for A3C ?

We came across multiple reasons for this after reading through some of the papers and some implementations of A3C. First of all, as the name says, the asynchronous part comes into play. By asynchronously launching more workers, **we are able to collect more training data**, thus making collection of data faster. Now, every deployed worker has it's own instance of the environment, thus we expect to get more diverse data which in turn makes network more robust and generate better results (We will display the results in later section).

In our case, we are launching total of 20 processes (that can be changed as well from command line arguments). The network uses **shared Adam Optimizer** so that the gradients are shared across all the 20 processes. The advantage function used is (as described in some papers) :

$$A(s_t, a_t) = r_{t+1} + \gamma V_v(s_{t+1}) - V_v(s_t) \tag{1}$$

So, the new policy gradient equation will be :

$$\Delta_\theta J(\theta) \sim \sum_{t=0}^{T-1} \Delta_\theta \log \pi_\theta(a_t|s_t) A(s_t, a_t) \tag{2}$$

On each learning step, we update both the Actor parameter and the Critic parameter. Action space and state spaces have already been described.

3 Our Method

3.1 Model Architecture

Depending on the Baseline, we changed our model for our state space.

Atari: DDQN

- Conv2D (None, 16, 42, 1)
- Flatten (None, 672)
- Fully Connected (None, 128)
- Fully Connected (None, 4)

Trainable params: 96,148

baby-a3c: Actor-Critic

- Fully connected (None, 128)
- Fully connected (None, 64)
- GRUCell (None, 64)
- (Fully connected (None, 1), Fully connected (None, 4))

Trainable params: 792,069

3.2 Reward Shaping

The score of the game is calculated by the number of bricks destroyed. We try to maximize the score which means destroying maximum number of walls in five lives. Baseline Model takes reward as 1 only when any brick is destroyed otherwise 0. This way baseline model distinguish between a favourable action and random action after training multiple times.

As the brick will not be destroyed in a single step, instead it requires a series of actions. So we tried to give some extra reward and also penalise the reward sometime based on certain conditions, so that the model will learn faster in choosing optimal action based on the current state, which will not directly result in destroying a brick in next step, instead will help in later steps such as destroying balls later or saving lives.

We mainly gave the reward at the end of every episode.

- When we lost any life , we **penalise** the reward with **absolute difference** between ball and paddle,so that whenever the ball is coming downward our model tries to move the paddle to same place where ball is going to fall.
- Now when the ball went down, we calculate the last two **absolute relative distance** between ball and peddle. If this distance **increased in the last moment**, we **penalise** the reward a bit **more**, whereas **less penalise** when the **distance decreased**. And didn't gave any reward if the distance remains same.This way the model learns to went in the ball direction when ball is coming downward. And even penalising a small reward when the relative distance decrease will help model to not just end the game for some reward in the end.
- We also give a **small reward at every step**, so that it will try to increase the steps in a run to take more rewards which will result in breaking more walls in a run.
- Whenever we **end a life**, we **penalise** the reward with a **high** amount.So the model will try to save the life every time.

3.3 Feature Extraction

The state space of Atari games are way too large, so, learning with such huge state space takes a lot of time and space. Like, in baseline model in a3c the input was the whole frame and used a RNN layer for extracting temporal features like velocity. In atari DDQN baseline model, the input features was four consecutive frames. So, instead of using such methods we processed each frame and extracted useful features from it to represent it as a feature vector.
The feature we used are:

- **Brick Presence**: The bricks were presented in a small subsection of the frame. So, we extracted that subsection from the frame and encoded it with one-hot encoding with 0 for brick absent and 1 for brick present. One hot encoding helped us to remove redundant color information which was not a necessary data.
- **Paddle Position**: The feature vector also contain the position of the paddle in that frame. Only x-position of paddle was important as the y-position is always fixed so, only x-position is included in the feature vector
- **Ball position**: The relative ball position of ball from the paddle was extracted and included in the feature vector. Instead of the absolute ball position, relative ball position is used as it makes more sense than the former.
- **Ball velocity**: The ball velocity is calculated from the ball position of previous feature vector and the present feature vector. It includes the temporal information of the frames and hence instead of feeding previous n frames to the Neural Network, extracting this feature from the frame made the network compact.

Shape of the feature vector was different for both (ddqn and a3c) the models. For a3c, the flattened matrix is used as the input but in ddqn, 2d matrix is used as input. Shape of the matrix is same as the extracted brick shape along with added columns for paddle position and ball position and velocity.

4 Experiments and Results

We trained the Breakout game on the baseline model and various variants of modified model.

4.1 DDQN

The model details of the baseline model and our modified model is given above. The input for the baseline model is a matrix of four sequential frames whereas in our modified model it is the feature vector. We tried two variants of our modified model: one with flattened feature vector and second was 2D matrix of the feature vector. The reward was shaped for both the variants as explained above. Due to lack of spacial information of bricks in the flattened variant of feature_vector model, its performance was not as good as the one with the 2D matrix.
The results of the 2D-feature_vector model and baseline model are shown in Figure 1 and 2.

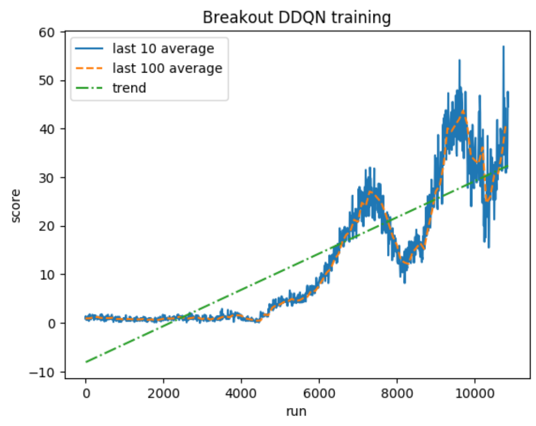

Figure 1: Score vs Runs for Baseline Model

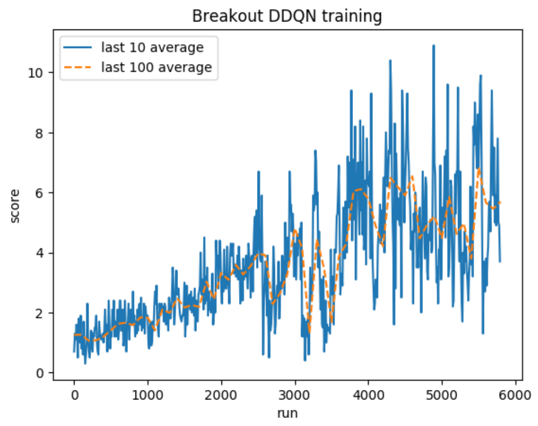

Figure 2: Score vs Runs for Modified Model with 2D feature Matrix

Here we could see that the performance of the modified model with feature vector and reward shaping is much better than the baseline model. The modified model always performed better than the baseline model at the corresponding runs. The score in the modified model touches an average of 10 as soon as 4000-4500 runs but the baseline model doesn't touch the score of 10 till 6000 runs.

Figure 3 and 4 shows the comparison of our modified model when input space was 1D feature vector than when it was 2D feature matrix. It can be seen that the 2D matrix gave better results than the former one.
Figure 5 shows the average step length for last 10 and last 100 runs and it can be seen clearly that the step length gradually increased as the run increased indicating that our agent was performing better and better and played for longer number of steps as the runs increased.

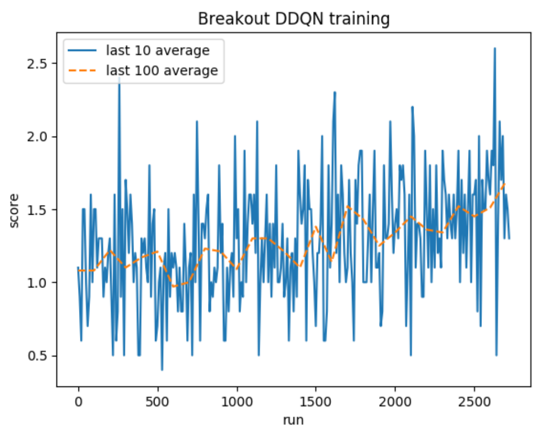

Figure 3: Modified Model with 1D feature vector

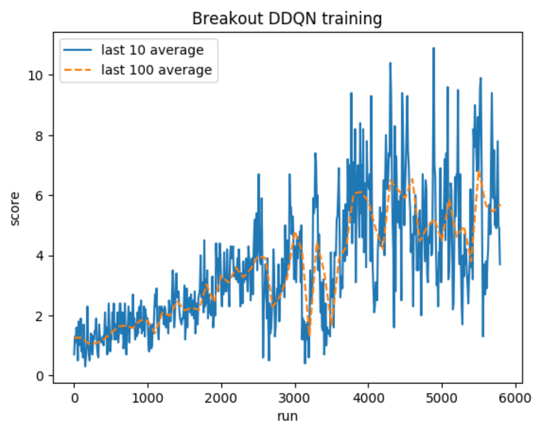

Figure 4: Modified Model with 2D feature Matrix

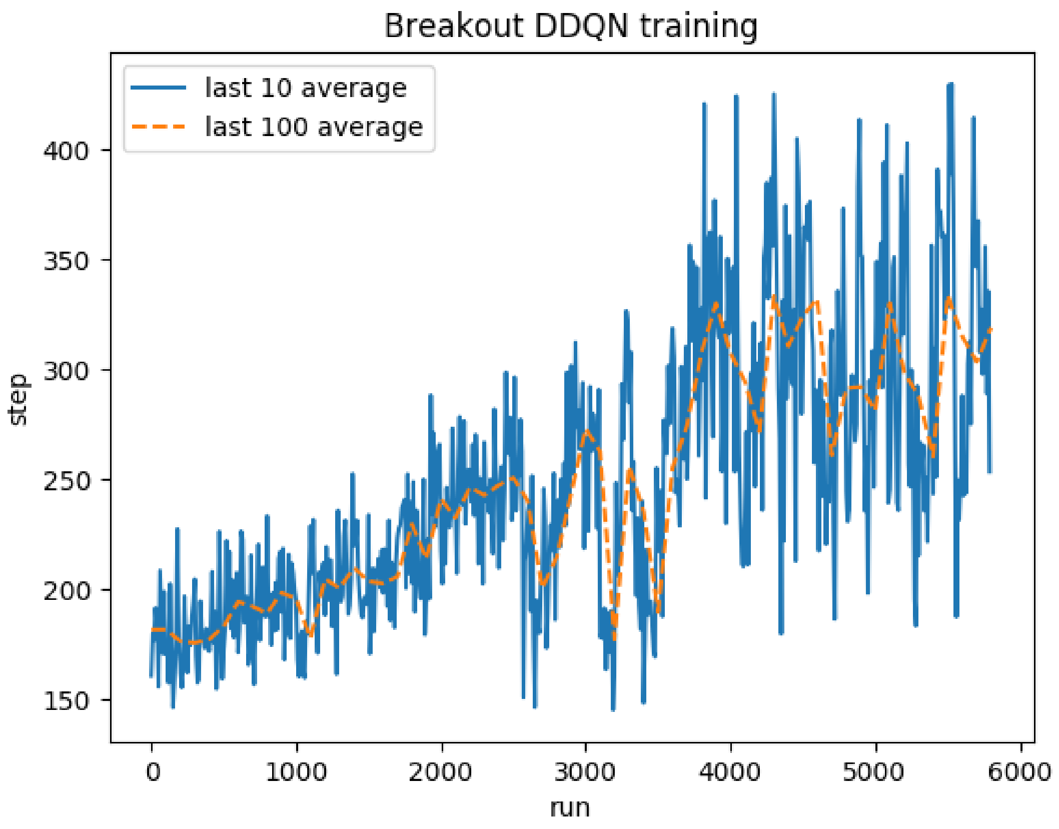

Figure 5: Episode length vs Episode number

4.2 A3C

The input of baseline model was one frame but in the modified model with reward shaping and feature extraction, it was a 1D vector of the feature vector extracted.
The reward was also shaped for the modified model as mentioned above.
The results of score vs run for these models are shown in Figure 6.

Here, it is clearly visible that the average score was always better for the modified model than the baseline model.

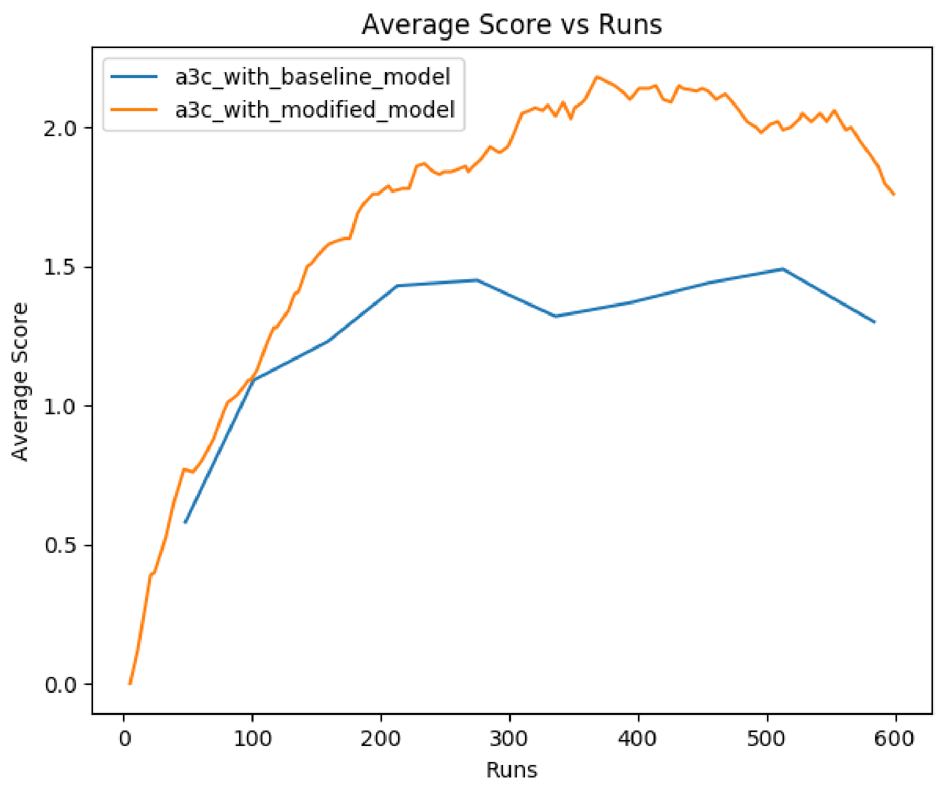

Figure 6: Score vs Runs for A3C implementation for both models

5 Conclusion and Future Works

After performing the above experiments and analyzing the results, we concluded that the feature extraction and reward shaping makes the training process faster and gives better results.Reward shaping should be done properly because there were some instances in our experiments where the model didn't seem to converge for some rewards awarded for some actions. Feature extraction worked really well in our experiments and since, it decreases the input space dramatically, the training process become extremely fast. Not only our model gave better results but also became fast due to dramatic decrease in input space. Due to time constraints, there were some experiments we planned to do but couldn't perform it. As we saw that in DDQN-atari 2D feature vetor with convolutional layer performed better that the flattened feature vector but we weren't able to do this for the a3c-atari due to time constraint. Also the baseline model used 4 consecutive frames as input and thus, this also contains some temporal information such as acceleration of the ball which we didn't included in our feature vector. This could be one more enhancement we could have done on our feature vector.

6 References

1. https://github.com/gsurma/atari

2. https://github.com/greydanus/baby-a3c