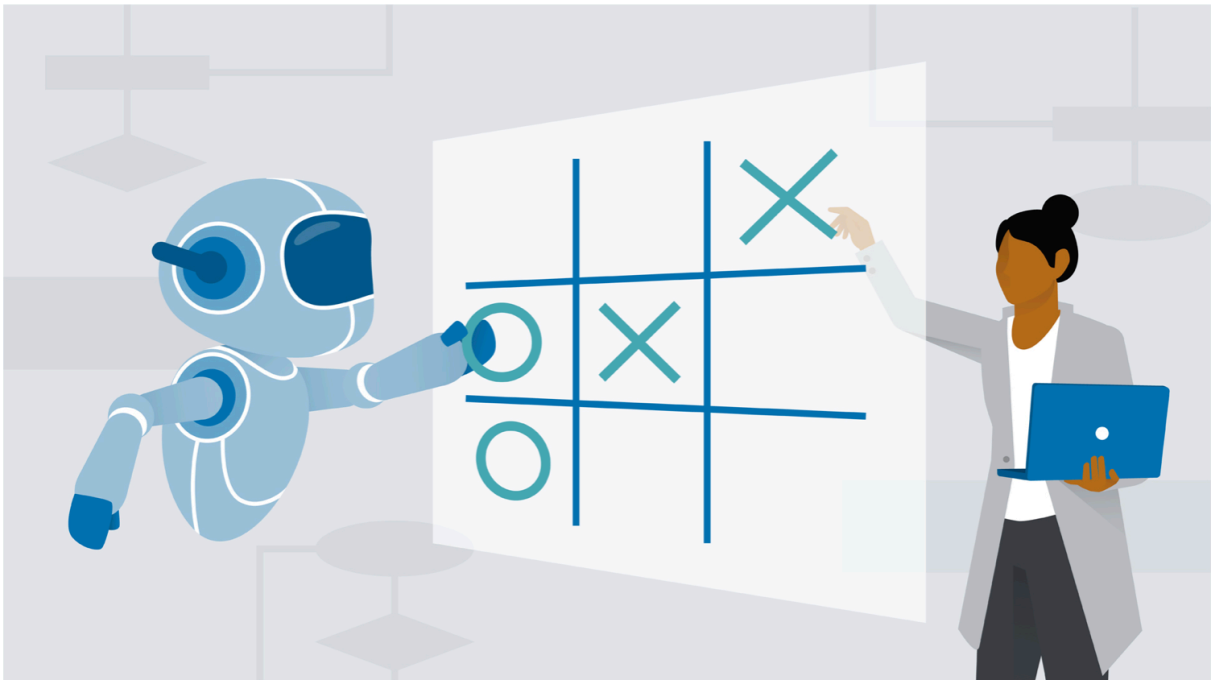
UNIVERSITY OF LIVERPOOL

IMPLEMENTING BREAKOUT AI VIA REINFORCEMENT LEARNING

REPORT



Prepared for: COMP532 - ML and BioInspired Optimisation 2022

Prepared by:

Aditya Naik
Student ID: 201574102
sganaik@liverpool.ac.uk
Rahul Kumar
Student ID: 201605507
sgrkumar@liverpool.ac.uk
Reza Fazli
Student ID: 201538160
sgrfazli@liverpool.ac.uk
James Cragg
Student ID:200365697
J.J.Cragg@liverpool.ac.uk

Introduction

The purpose of this project is to develop artificial intelligence that can learn to play Breakout through reinforcement learning and to use that AI to explore how modern advances in artificial neural networks and deep learning may change what is considered 'thinking' in a computer. We shall create a learning agent, a program, that can play in the hard-coded environment at Atari Breakout. By 'hard-coded' we mean programs that don't make use of machine learning algorithms. In other words, the AI is the result of a human programmer analyzing and developing an algorithm that is specifically designed to play Breakout. Contrary to this, our AI will employ a general learning algorithm and get better at playing Breakout over time by playing more games. This is the goal. A breakout-playing AI is not very impressive in this day and age, but an AI that learned to play Breakout? These programs, which have just recently been introduced, have far more fascinating implications.

The Artificial Neural Network

Deep learning models heavily rely on artificial neural networks, which is a subset of machine learning. Artificial neural networks are used to solve composite jobs like training computers on how to obtain a high-level understanding of digital images and video games. The main idea is that the learning model learns the rules and transcribes the next move or come up with the winning strategies in the game so that the neural network can achieve high score or rewards.

The idea is that our current state is the tensor of a pixel value from the 4 most recent frames. We will be basing our model on the greedy epsilon strategy where we apply a random action, or our current state placed within a convolutional neural network (CNN) to gain action. From our action, we will receive a reward and enter a new state and the process of moving from state to state will be stored in memory for training. This data will be used to train the agent using the HUBER loss function.

Libraries used

The TF-Agents library is a Reinforcement learning library based on TensorFlow developed at google and open-sourced in 2018. It provides many environments and wrappers for OpenAI gym environments. It is fast, scalable and also allows us to implement Reinforcement Learning algorithms like DQN, DDQN, REINFORCE etc and efficient replay buffers and metrics. Tensorflow and Keras were also used to build the neural network We also used matplotlib to show the game playing out at the end of training and NumPy for mathematical calculations.

The Game

The game environment is simulated via the TF-Agents python package and is one of many Atari games included in it. This fact isn't too important. Breakout is a relatively simple game with little variation between versions. The aim of the game is simple, you have to destroy all the blocks using the ball. The difficult part is lining yourself up so you don't miss it.

The Algorithm:

After each life is lost the game requires you to press fire to restart playing. The agent may initially learn that pressing FIRE means quickly losing the game. To avoid this problem we create and use a subclass of the `AtariPreprocessing wrapper class called `GameWithAutoFire` which presses FIRE (automatically) at the start of the game and after each life lost.

Training architecture

The training is split into two sections which consist of the data collections and training parts.

Data collection:

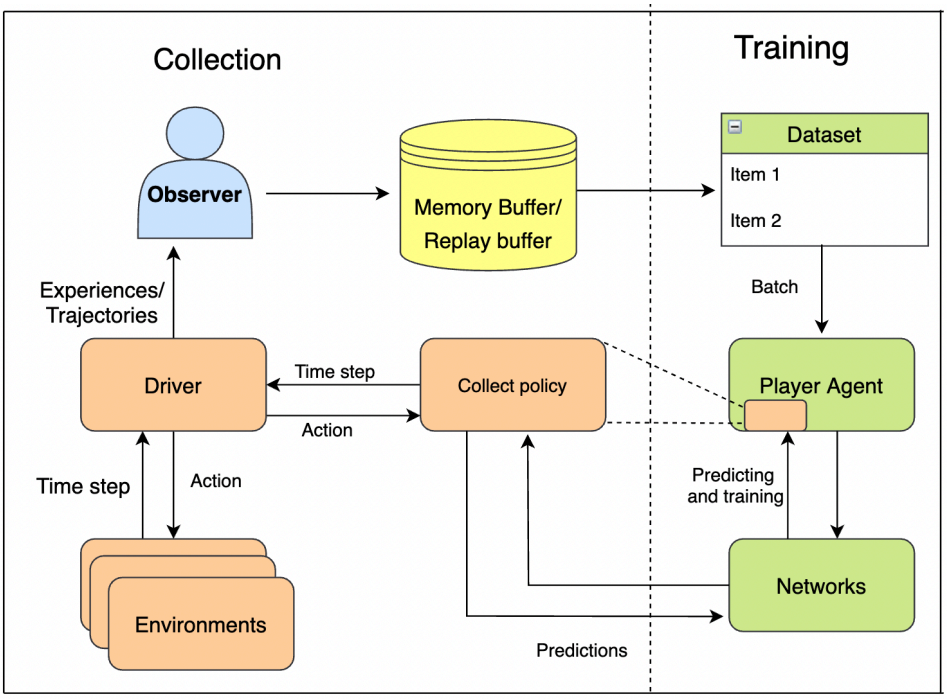
There is a driver object that explores the environment using a collect policy and collects experiences which are handed over to the observer who stores these experiences into a memory buffer.

Training section:

In the training section, the dataset object samples a few experiences from the memory buffer and uses those memories in batches to train an agent whose policy is defined by a deep Q-Network which also determines the experience collection policy described in the previous section.

Multiple environments are used to take advantage of all CPU cores while the GPUs are busy training the agent which also helps in obtaining less correlated experiences reducing overfitting.

Deep Q-Network



To implement the DQN we use the tf_agents.networks package. Our network contains a preprocessing layer which normalises the input after which comes three convolutional layers consisting of32, 8x8 filters with a stride of 4; 64, 4x4 filters with a stride of 2; 64, 3x3 filters with a stride of 1 and a fully connected Dense layer with 550 hidden units. Each of these units uses a relu activation function by default and the final output layer has 4 outputs with a linear activation function which outputs the q-values for each of the four actions ('NOOP', 'FIRE', 'LEFT', 'RIGHT').

Agent

The tf_agents.agents package is used to initialise the agent. We first set the steps to 0 indicating the start of training. Then we specify the value of the epsilon which will control the exploration of the agent. We use a Polynomial decay schedule taken from the Keras library to reduce the value of epsilon as the agent trains. The value is initially set to 1 and gradually decays every 250000 steps.

We then pass in the DQN initialised previously as a parameter as well as time step and agent action specifications, the optimiser which is RMSProp which uses a learning rate of 2.5e-4, the loss function (Huber loss function), the discount factor (gamma) and the Keras scheduler that returns the epsilon value.

We then initialize the agent with the name of the player agent.

Memory Buffer (replay buffer) and the Observer

The tf_agents.replay_buffers package is used to implement this. We will use the TFUniformReplayBuffer class in the package. This provides a uniform sampling of experiences with a good performance.

data_spec: This gives the specification of the data to the buffer so that the agent knows what the collected data will look like.

barch_size: The number of experiences that will be accumulated at each step of training. Since our agent will be executing one action per step the batch size will be one

max_len: the maximum size of the replay buffer. We have created a large buffer with a size of 100000. In case of an out of memory issue, this should be reduced as it will utilise a lot of RAM.

We then create the observer which is an object that takes the experiences or trajectories and strokes them in the replay buffer.

Training Metrics

While training the agent we observed that the loss was not a very good measure of how well an agent was learning as there would be some times when the agent would perform badly even though the loss was very low so instead we use the average rewards per episode to decide how well an agent is playing. To get the metrics we use the tf_agents.metrics.tf_metrics sub-package which will help us log the number of episodes, steps taken in the environment, average return and average episode length. All these metrics will be displayed every 1000 iterations.

Collect Driver

This is the object that explores the environment, gathers experiences and sends them to the observer who interns saves these to the memory or replay buffer. The driver sends the time step to the collection policy (this policy could be either defined by the online model or a target model depending on which model computes the values of the next state) which then, given the time step returns an action step object consisting of the action to be performed. It then transfers this action to the environment and returns the next time step. Ultimately the driver created an experience (trajectory object) and sends it to the observer.

There are two main types of drivers DynamicStepDriver and DynamicEpisodeDriver. The former collects experiences from a predefined number of steps while the latter stores experience from a given number of episodes.

Since we want to collect experiences for four steps in each training iteration we will use the DynamicStepDriver and pass in the environment, the collect policy, the observers and the number of steps to collect data for in each iteration which is 4 in our case.

Dataset

The experiences from the memory buffer are sampled and stored in the dataset object to be used to train the agent

Training the agent

To improve the training speed we converted the functions to TensorFlow functions to take advantage of the GPUs and parallel processors. The initial state of the collection policy is fed into the function. Since the policy is stateless it returns an empty tuple. We then iterate over the dataset and execute the training. At each iteration, we call the drivers run() method to run the collect policy and collect experience for 4 steps, broadcasting the collected experiences from the dataset and sending it forward to the agent's train() method which returns the loss of training. The metrics are all logged and the agent is trained for 100000 iterations.

Final experiments and evaluation

We ran two experiments

Experiment 1 :

Here we trained the agent using a simple Deep Q learning algorithm where the same DQN model is used to make Q-value predictions and set its targets. This was done by setting the parameter target_q_network to None while initialising the PlayerAgent.It was observed that while the training progressed there were many instances where the Average return

metric was quite high (~20.0) after a few further iterations it would randomly drop down to a low value (~8 or 9) and training would again continue and this would repeat a few times. We realized that this was because of an event called 'catastrophic forgetting' which results in the agent forgetting what it learned in one part of the environment while learning something new. The reason why this happened is probably that we used the same DQN model to train and set its targets.

Experiment 2:

In this experiment, we use two DQN models to make predictions and set targets separately. The online model to make predictions is DQNetworkModel while the TargetModel is used to set the targets. The two models are exact copies of each other. The only difference is that the target model is updated every 2000 iterations, unlike the DQNetworkModel which is updated every iteration. This is implemented by initialising both the models and passing them in the agent initialiser with parameters q_network = DQNetworkModel, target_q_network = TargetModel, target_update_period = 2000.
Making these changes greatly improved performance and reduced the effect of catastrophic forgetting as well as training instabilities and resulted in a more uniform rise in the average return metric.

The final results after training for 100000 iterations are displayed below:

```
99000 loss:0.00224INFO:absl:
        NumberOfEpisodes = 1275
        EnvironmentSteps = 396004
        AverageReturn = 22.600000381469727
        AverageEpisodeLength = 602.2000122070312
```

Conclusion

From this project, we were able to demonstrate the power of deep learning in the field of Reinforcement Learning by training an agent to play the Atari Breakout game using only the pixel frames as input to the Deep Q-network model. Upon completion of the training for 100000 iterations, the agent was successfully able to play the game on its own as was shown in the animation at the end of training. The agent would have performed better with a higher score if trained for many more iterations but due to constraints on time and computational resources we were forced to limit the training to only 100000 iterations.

Challenges of the project

Understanding the theory and implementing code takes a lot of time. Likewise, running the code required enormous computational power, which was the biggest challenge for the group.

Future Works:

We used Deep Q Learning for our RL algorithm. Q Learning is often used as an introduction to deep RL, but it is not the most effective algorithm. The Advanced Actor-Critic algorithm (A2C) may be an option in the future. Perhaps it would be more effective. As its lightweight framework and its stabilizing effect on the parallel actor-learners to successfully train the neural networks. The asynchronous variant trains for half the time on a single multi-core CPU instead of a GPU.

In future, deep learning reinforcement models may not be that data-hungry and can be implemented for other uses. It could save computational time and may not require extensive CPUs and GPUs. It may also be used to solve simple tasks and problems.

References

[1]Mnih, Volodymyr, et al. Asynchronous Methods for Deep Reinforcement Learning, Google DeepMind, 2016
[2]Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D. and Riedmiller, M. (2013). Playing Atari with Deep Reinforcement Learning. [online] arXiv.org. Available at: https://arxiv.org/abs/1312.5602.
[3]Aurélien Géron (2019). Hands-on machine learning with Scikit-Learn and TensorFlow concepts, tools, and techniques to build intelligent systems. O'Reilly Media, Inc.
[4] Sutton, Richard S., and Andrew G. Barto. Reinforcement learning: An introduction. MIT press, 2018.
[5] Gagniuc, Paul A. Markov chains: from theory to implementation and experimentation. John Wiley & Sons, 2017.