

INT404 - Artificial Intelligence

Project - Final Report



Submitted by:

Roll No.	Name	Registration No.
22	Arjun S	11813193
23	Prateek Rathod	11814621

Abstract

This project demonstrates the generalizability of “Deep Q-Learning” by learning control policies from visual outputs of two different environments. The model aims to model the end-to-end relation between the visual outputs (rendered 2-d visuals of a game) to the next action (control signals like move up, move right) so as to gain maximum rewards (score). The learning phase of the model involves simultaneously training the model while making predictions at each step (frame to action). In order to reduce the instability in the model introduced as a result of training at each step, a separate buffer model will be maintained whose weights will be synchronized with the original model at the end of each episode.

Related Work

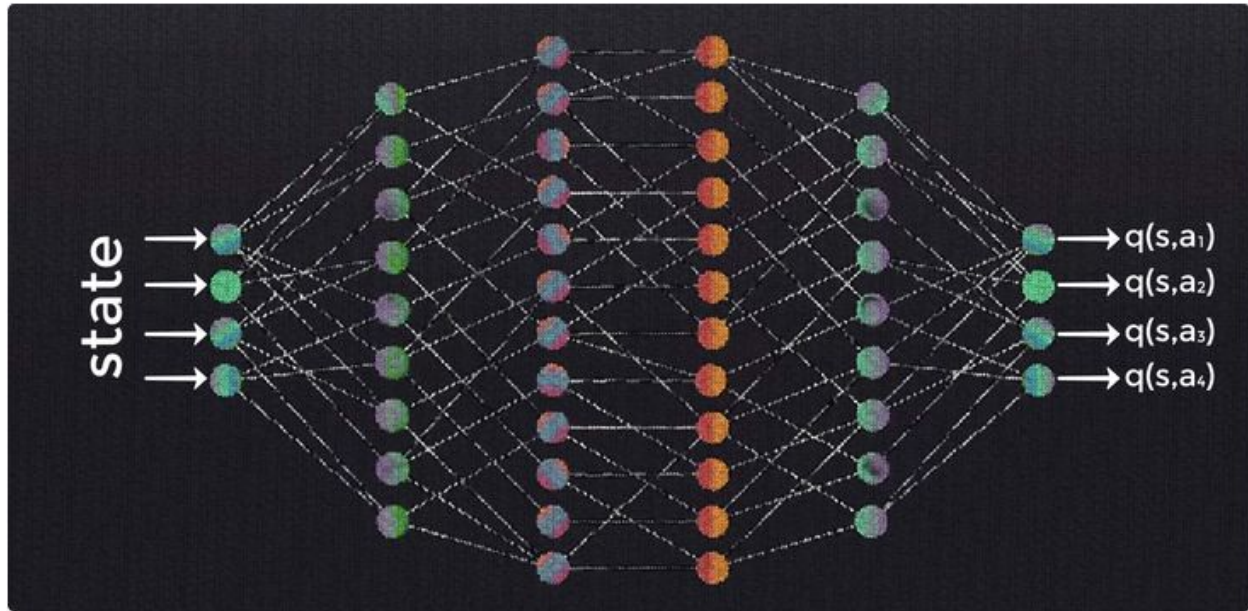
The projects listed below thoroughly analysed:

- Deep learning Flappy Bird: Deep q-learning to train a deep neural network to play the game flappy bird. [Github](#)
- Minimal DQL: A minimal Deep Q Learning implementation using Keras. [Github](#)

Implementation

This project makes use of neural networks to map the relation between the visual outputs of a simulated environment to the best action that can be taken next so as to maximize the cumulative score obtained by the agent in an episode. The purpose of a deep neural network in this project is to calculate the approximate q-value for each action of the given state. This project uses a custom-built env built specifically for this project.

The implementation consists of building a neural network that takes the current frame of the game window as input and outputs q-values for each action.



The above visualization of a network takes four states i.e. four consecutive frames as input but this project is implemented using a single state i.e. singular frame due to a simpler environment.

The loss for this neural network is calculated by comparing the output with the output of q-function (Bellman equation) which is the optimal value. The goal is to minimize the loss, for this, the weights of the neural network are updated using stochastic gradient descent and backpropagation.

A concept of replay memory is involved to execute such a process. It is nothing but a fixed-size collection (mostly deque) of tuples containing information like current state, action taken, reward received and the following state. This is required for training the network to break the correlation between continuous states. Replay memory is past experience and from this, a smaller batch of tuples is chosen at random to train the network.

This is how it works:

- Initialize replay memory capacity.
- Initialize the neural network with random weights.
- For each episode i.e. until the game finishes:
 - Initialize the starting state.
 - This state i.e. the frame of the screen window is preprocessed. The rgb image is converted into a grayscale image and then scaled down if required.
 - For each time step:
 - Select an action which can either be chosen at random or predicted by the model. The action chosen depends upon a decaying factor which decays as the model learns therefore more random actions are chosen in the beginning.
 - Execute the selected action.

- Observe the reward and the next state.
- Store experience in replay memory.
- Select a batch from the replay memory for training.
- Calculate loss between output Q-values and the Q-values for the following state to compute the loss.
- Gradient descent updates weights in the network to minimize loss.



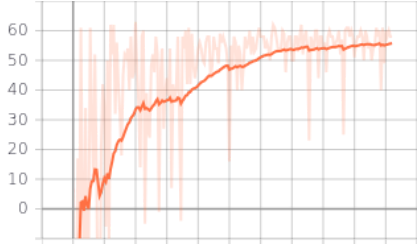


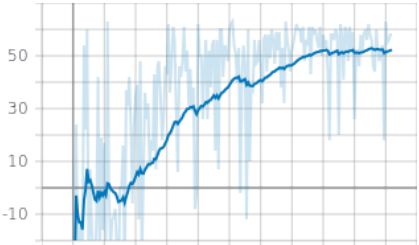
To parallel train multiple agents with different models each with different sets of hyper-parameters, this project follows a master-worker architecture, wherein multiple “remote workers” train the model, and stream performance logs back to the master through a http endpoint exposed by the “master” machine. The “master” stores the received logs as scalar metrics into TensorBoard, hence enabling a seamless monitoring experience.

Status:		
Attribute	colab-worker-2	colab-worker-cnn-2
env	rover_lander_2	rover_lander_2
max_episodes	5000	5000
current_episode	0	0
connected_at	2020-04-03 12:38:54.392266	2020-04-03 12:38:58.187328
updated_at	2020-04-03 13:53:09.738833	2020-04-03 13:50:57.223795
last_ep_score	56	58
avg_ep_score	42.75396825396825	47.930348258706466
num_step	42	
acc	0.703125	0.734375
loss	0.45802921056747437	0.15102538466453552
mse	0.45802921056747437	0.15102538466453552
episodes	200	

Results

The agent was trained from scratch using two variants of neural networks - a Fully Connected Neural Network (FCNN) and a Convolutional Neural Network (CNN). Below are the observations from each model being used to train each environment. The performance metrics were logged using TensorBoard and can be inspected from the links given in the table below.


Env: rover_lander_1

Model	Performance	Episode 20	Episode 200	Train Score / episode
FCNN	Open TensorBoard			
CNN	Open TensorBoard			

This table contains GIF, and may not be animated on your editing tool. Please check the [project repository](#) to view this table.

Env: rover_lander_2

Model	Performance	Episode 20	Episode 200	Train Score / episode
FCNN	Open TensorBoard			

CNN	Open TensorBoard			
-----	---------------------	---	--	---

This table contains GIF, and may not be animated on your editing tool. Please check the [project repository](#) to view this table.

It was observed from the logs and metrics recorded that agents operating in different environments followed a similar trend in learning speed and performance regardless of the environment as long as they shared comparable levels of complexity.

The metric “last_ep_score” that on training the agent for 200 episodes, the “average score / episode” plateaued, and did not continue to increase anymore. It was also observed that having a reward function that not only rewards the agent for expected behavior but penalises for undesirable behaviour showed much better performance. This was mainly because, an agent with only positive reward tends to take the same action when in a state that has no possible state to take next that has a positive reward.

Instructions to try out this project

First, on your local machine run:

```
python train_master.py
```

Note: Use a port-forwarding tool like [ngrok](#) to expose the endpoint created

To monitor logs streamed from remote workers on your local machine, run:

```
tensorboard --logdir logs
```

Now, on each remote workstation run:

```
python train_worker.py \
    --env <ENV_NAME> \
    --master-endpoint <MASTER_ENDPOINT> \
    --worker-name <WORKER_NAME>
```

To train using the CNN based model run:

```
python train_worker_cnn.py \  
    --env <ENV_NAME> \  
    --master-endpoint <MASTER_ENDPOINT> \  
    --worker-name <WORKER_NAME>
```

Or, run remote worker from Google Colab -

https://colab.research.google.com/github/ArjunInventor/Deep-Q-Learning-Agent/blob/master/train_worker.ipynb

Testing agent

```
python play.py --model <MODEL_PATH> --env <ENV_NAME>
```

When using a CNN based model, run:

```
python play_cnn.py --model <MODEL_PATH> --env <ENV_NAME>
```

Use --save-gif to save the gameplay as a gif

Important Libraries Used

- Tensorflow: To define transformation steps, train model and make inferences at each step
- TensorBoard: To log scalar metrics streamed from remote workers and visualize together
- pyGame: To build the custom environments, that can compute states without having to render or need video drivers, hence making it possible to train faster on cloud
- Flask: To set up an endpoint for facilitating a master-slave federated-learning process, where multiple remote workers train with different sets of hyper-parameters and report back results at a set interval

Team responsibilities

- **Arjun S (11813193)**
 - i. Implement “environment” and “agent” wrapper class with a gym-like interface
 - ii. Implement scripts and flask application to follow master-worker architecture, to parallelly train, multiple models, with separate remote workers each with different sets of hyper-parameters
 - iii. Implement Q-function and replay-memory for the agent to randomly sample previous transitions to learn from and estimate maximum possible reward
 - iv. Hyper-parameter tuning, making use of this project’s unique remote worker support to parallelly train multiple instances of models with different sets to hyper-parameters
- **Prateek Rathod (11814621)**
 - i. Implement rover_lander environments and enable state computation without video drivers
 - ii. Implement reward functions for each environment that facilitates effective learning
 - iii. Implement deep-learning-based models using Keras and TensorFlow for the agent to map visual feed to acceptable moves in the action space
 - iv. Modify TensorBoard’s default behavior to optimize for continuous training at each step

References

- [Playing Atari with Deep Reinforcement Learning - arXiv:1312.5602](#)
- [Deep Recurrent Q-Learning for Partially Observable MDPs - arXiv:1507.06527](#)
- [Flask Documentation](#)
- [Pygame physics simulation](#)