Atari DQN TFAgents

Prepared by – Aakash Maskara

Overview

In this project, I implemented Deep Q-Network (DQN) agents to play two Atari games - Seaquest and Space Invaders using the TF-Agents library. The aim was to develop a shared architecture for both games, train for a computationally feasible number of steps, evaluate the learned policy, and generate gameplay videos.

This report documents the architectural design, training process, experimentation journey, limitations, and analysis of the resulting models.

Problem Formulation

I modeled each Atari game as a Markov Decision Process (MDP) where :

- State : Stack of 4 grayscale frames from the game environment
- Action : Discrete movement/actions allowed by the environment
- Reward: Clipped to [-1, 1] to stabilize training

The agent interacts with the environment using an $\varepsilon-$greedy policy and learns $Q-$values via experience replay.

Preprocessing & Environment Setup

- Used RewardClippingWrapper to normalize rewards
- Used ActionRepeat (times = 4) to repeat actions for 4 frames to simulate faster reaction

- Environments were loaded using suite_atari.load() with default wrappers for frame stacking

Model Architecture

Both games share the same model architecture :

- Input Preprocessing: Normalized pixel values using : tf.keras.layers.Lambda(lambda obs: tf.cast(obs, tf.float32) / 255.0)
- Convolutional Layers :
  - Conv2D (32 filters, 8x8 kernel, stride 4)
  - Conv2D (32 filters, 8x8 kernel, stride 4)
  - Conv2D (64 filters, 3x3 kernel, stride 1)
- Fully Connected Layer :
  - fc_layer_params=(512,) for both Seaquest and Space Invaders.
- Q-Network: q_network.QNetwork() from TF-Agents
- Optimizer: RMSProp (both games)
- Loss: Huber Loss

Training Strategy

- Epsilon Decay : Polynomial decay from 1.0 to 0.01
- Target Update : Every 2000 steps
- Replay Buffer : 100,000 capacity
- Batch Size : 128
- Training Steps : 50,000 (due to computational limits)
- Warmup : 20,000 steps with random policy

Reason for such small training model?

Training on Atari environments is computationally expensive and time-consuming. Due to hardware constraints, I opted to train each game for 50,000 steps, followed by 50,000 more using resumeTraining.py, totaling 100,000 steps per game.

Experiments Conducted Before Reaching Final Model

I began with varied architectures and strategies. Some notable experiments :

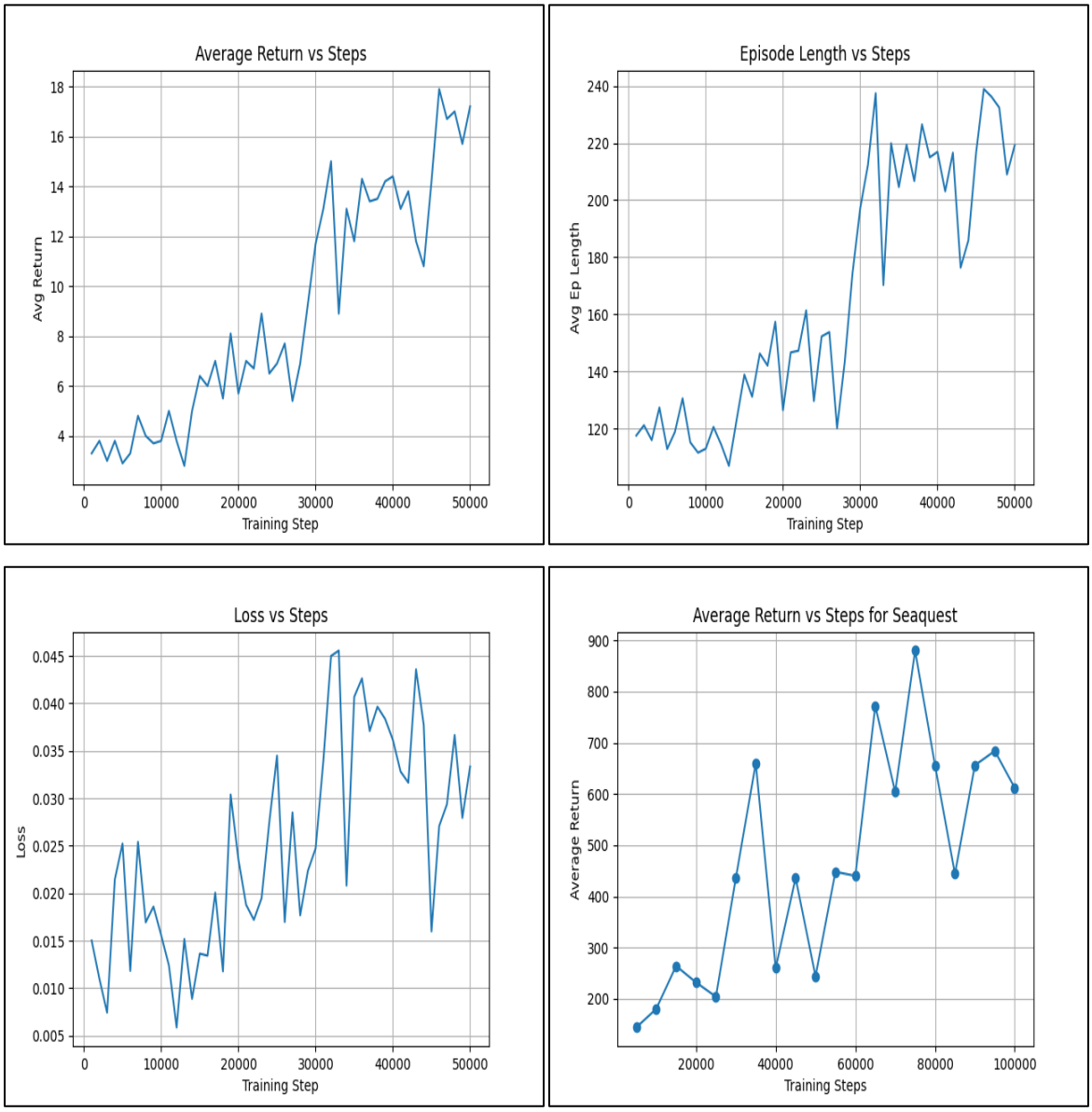
- Used lower fc_layer_params=(128, 64) for Seaquest initially led to incompatible checkpoint resume
- Initially tried different optimizers for each game, but unified to RMSprop for consistency and better stability
- Tested smaller replay buffer (50,000) but convergence was unstable
- Adjusted frame skip, reward scaling, and learning rate decay schedules and found current settings stable

All these led to the final architecture and hyperparameters, shared across both games.

Results & Plots Analysis

Plots were generated using buildAndTrainAgent_Seaquest.py, buildAndTrainAgent_SpaceInvaders.py & createTrainingCurvesAndVideos.py. Below are the key results :

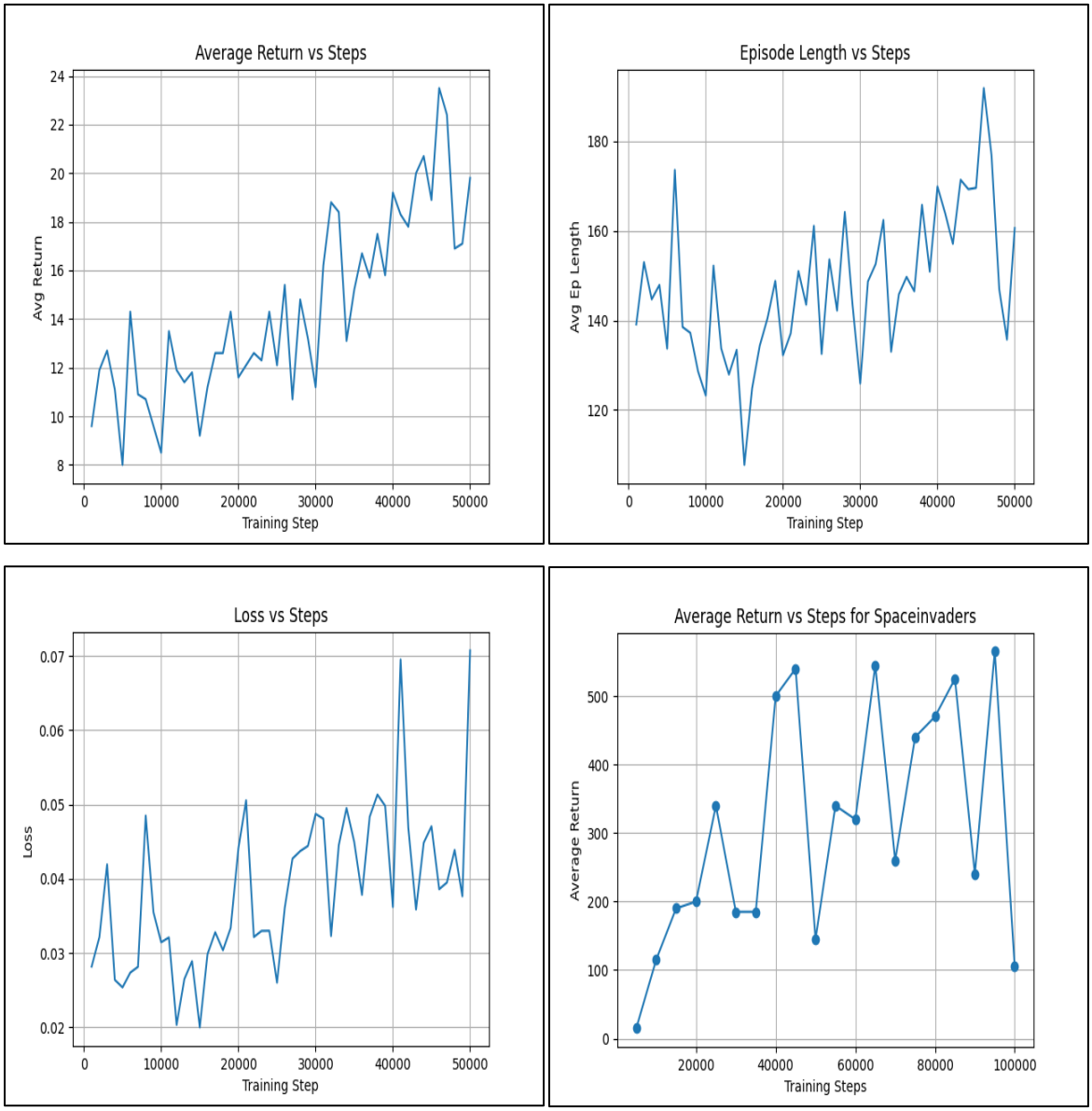
Seaquest



- Loss vs Steps : loss fluctuated mildly but trended downward
- Average Return vs Steps : started low but steadily improved

- Episode Length vs Steps : shows gradual increase, indicating agent survival improved
- Training Curve : shows stable learning trajectory

Space Invaders

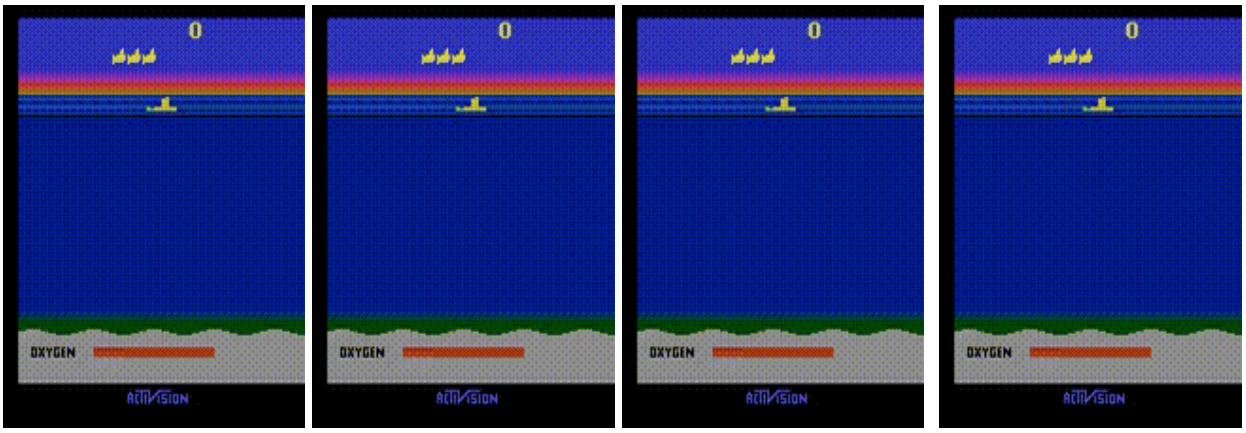


- Loss vs Steps : stable oscillation, learning not collapsed
- Average Return vs Steps : lower than Seaquest but increasing trend
- Episode Length vs Steps : increasing, shows learning happening
- Training Curve : confirms model is learning strategy slowly but stably

Policy Deployment & Videos

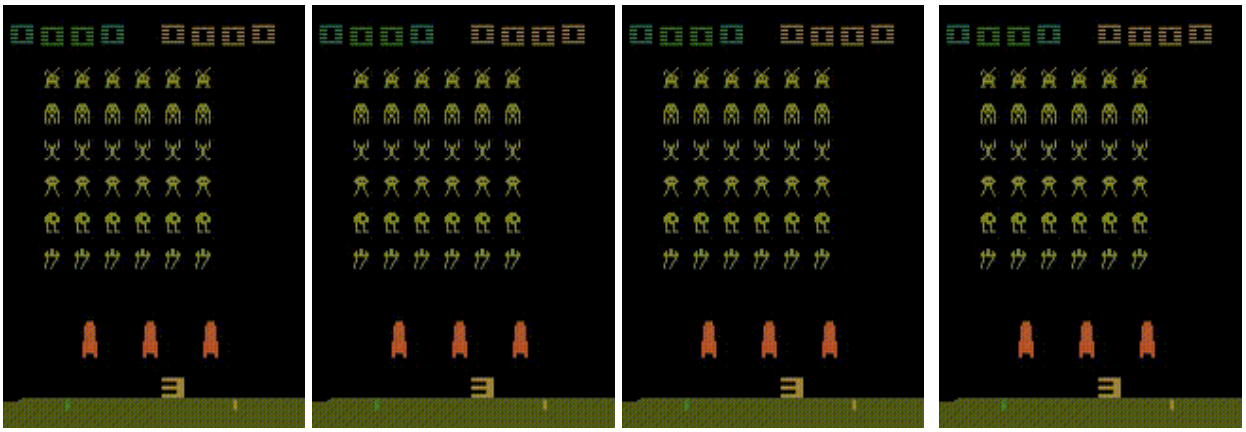
Using deployGamePlayer.py, gameplay videos were generated by running saved policies :

Seaquest



DeployGamePlayer **Best** **Intermediate** **Poor**

Space Invaders



DeployGamePlayer **Best** **Intermediate** **Poor**

Package Versioning

To ensure reproducibility, here is a list of the package versions :

AutoROM 0.4.2

autorom-accept-rom-license 0.6.1

gym 0.23.0

gym-notices 0.1.0

imageio 2.37.0

imageio-ffmpeg 0.6.0

importlib-resources 6.5.2

ipykernel 6.30.1

ipython 8.37.0

keras 2.11.0

matplotlib 3.10.5

matplotlib-inline 0.1.7

numpy 1.23.5

opencv-python 4.6.0.66

pillow 11.3.0

pip 25.1

pygame 2.1.0

python 3.10.18

tensorboard 2.11.2

tensorboard-data-server 0.6.1

tensorboard-plugin-wit 1.8.1

tensorflow 2.11.0

tensorflow-estimator 2.11.0

tensorflow-intel 2.11.0

tensorflow-io-gcs-filesystem 0.31.0

tensorflow-probability 0.17.0

tf-agents 0.14.0

typing-extensions 4.14.1

Conclusion

In this project I have successfully implemented a shared DQN agent for Seaquest and Space Invaders. Despite training for only 100k steps (vs 1M+ in literatures), the agent demonstrates learning. With more training, the average return and stability could significantly improve.