CSCI 166 - Frogger DQN Report and Analysis

Introduction & Setup

Why Frogger?

We went with Frogger from the choices as our game for DQN mainly because we thought it was interesting and because it is a popular well known game (it was also cool). Still Frogger has a many interesting aspects of it that make it a good environment to study DQN

The first is sparse rewards the agent only gets positive feedback when making forward progress towards the goal with a large reward for successfully reaching the other side fully. The next aspect is the timing requirements the agent must time the movements to avoid getting hit by cars, trucks, and the moving logs and turtles. Lastly is the multiple failure modes like hitting vehicles, falling into water, or running out of time which will terminate the episode.

Overview of our Environment

Property	Description
Observation Space	84×84×4 frames
Action Space	5 discrete actions (NONE, UP, RIGHT, LEFT, DOWN)
Reward Structure	Sparse

This environment/game thus makes a good test game for DQN because it needs both exploration to find good paths and also precise controls based on timing in order to make those paths without losing. The sparse rewards also show us the algorithms use to give rewards over long sequences of actions in the environment.

Architecture of the Model

Baseline DQN

The baseline DQN implementation follows some of the given starter code template.

The network has the following architecture

- Input of 84×84×4 stacked grayscale frames
- Conv Layer 1 with 32 filters, 8×8 kernel, stride 4, ReLU
- Conv Layer 2 with 64 filters, 4×4 kernel, stride 2, ReLU
- Conv Layer 3 with 64 filters, 3×3 kernel, stride 1, ReLU
- Fully Connected 512 units, ReLU
- Output Layer 5 units (one per action)

Some notes about Baseline, First if our experience replay buffer that holds transitions (s, a, r, done, s') and samples random batches to break some correlations that may not be good. Then a target network which is a separate, slow updated copy of the Q network to make the training targets more stable, and there is also the greedy policy that balances exploration (random actions) and exploitation (greedy actions).

Double DQN Extension

The problem with baseline DQN is that it usually overestimates the Q values because in the same network we select and evaluate the next best action. Double DQN fixes this by decoupling the action selection from the action evaluation

Standard DQN Target:

```
y = r + γ * max_a' Q_target(s', a')
```

Double DQN Target:

```
a* = argmax_a' Q_online(s', a')     #
y = r + γ * Q_target(s', a*)        # Separate evaluation and best action
```

It is a small code change (literally one line) but makes a huge difference by reducing overestimation bias .

Experiments

We setup 6 experiments to see the way that different algorithm choices and hyperparameter settings make. All experiments were run for 500,000 training steps. It took around 7 hours total using the A100 GPU from a Google Colab environment allowing us to organize ourselves and results.

Experiment Config

Experiment	Algorithm	Learning Rate	Epsilon Decay	Target Sync	Replay Size	Batch Size
Baseline DQN	DQN	1e-4	150k frames	1000 frames	10,000	32
Double DQN	DDQN	1e-4	150k frames	1000 frames	10,000	32
DDQN Lower LR	DDQN	5e-5	150k frames	1000 frames	10,000	32
DDQN Longer Exploration	DDQN	1e-4	300k frames	1000 frames	10,000	32
DDQN Slower Target Sync	DDQN	1e-4	150k frames	2500 frames	10,000	32
DDQN Bigger Replay	DDQN	1e-4	150k frames	1000 frames	30,000	32

Tuning

Our experimental approach was straight forward. For the baseline establishment we started with a standard DQN to establish a performance floor. Then we made an algorithm change and switched to Double DQN (our required variant) to measure the improvements from baseline, next was the hyperparameter exploration and experiments in where we tested four different hyperparameter variations to understand their individual impacts on learning.

Results

Learning Curves Comparison Chart

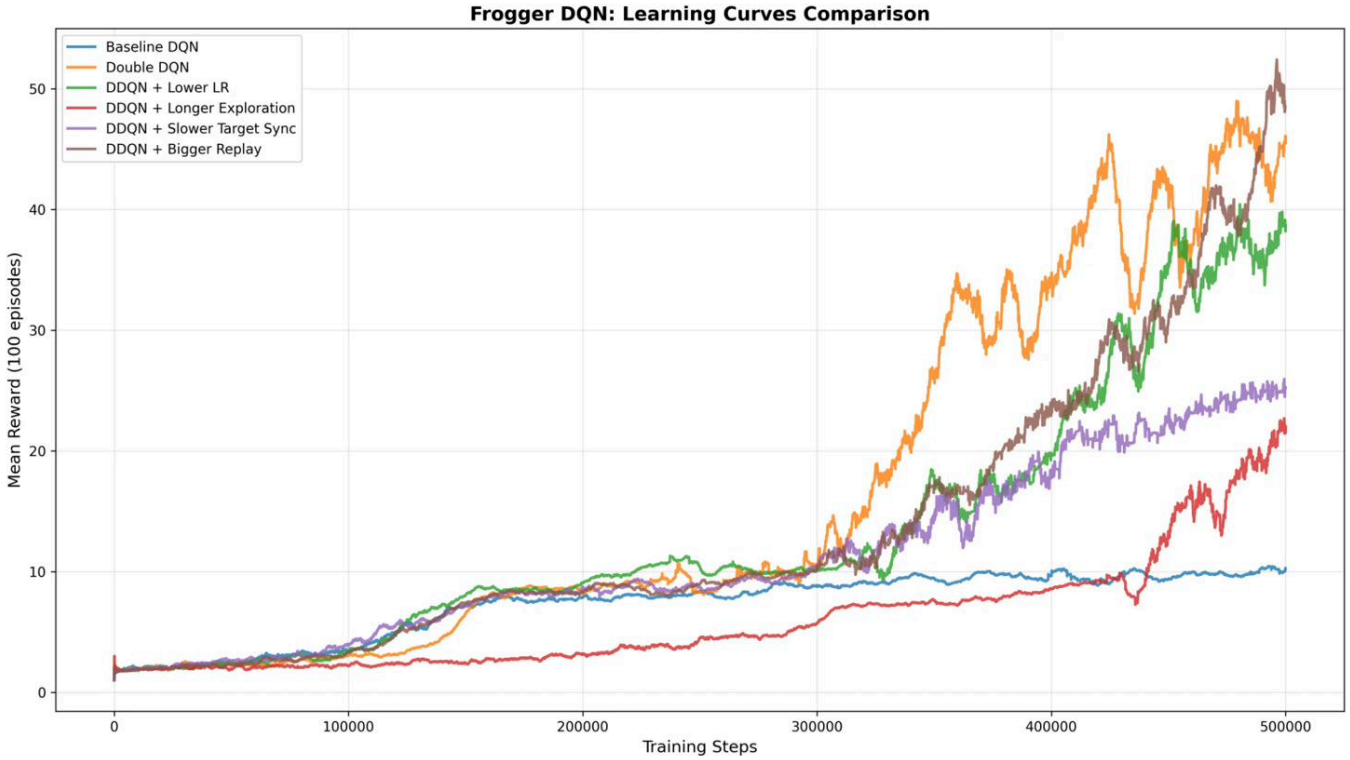

Figure 1: Mean reward (100-episode average) across all six experiments over 500,000 training steps.

Summary of Expirements

Experiment	Final Mean Reward	Peak Reward	Total Episodes
Baseline DQN	10.23	53	7,059
Double DQN	45.52	216	7,671
DDQN Lower LR	38.69	176	7,796
DDQN Longer Exploration	21.96	109	10,002
DDQN Slower Target Sync	25.24	88	8,855
DDQN Bigger Replay	48.52	157	7,944

Findings

Double DQN results is a very large improvement over the baseline, the standard Double DQN achieved a peak final mean reward of 45.52 compared to the baseline DQN mean of 10.23 which is a 345% improvement. This tells us that overestimation is a very big bottleneck in vanilla DQN.

The two best-performing configurations were Double DQN and Bigger Replay Buffer with a 45.52, and 48.52 mean reward. The bigger 30,000 buffer gave more diverse training experiences which resulted in much better results had we not had that buffer. Especially after

500,000 steps as seen in Figure 1.Also Standard DDQN which came in second with the 45.52 mean reward the default hyperparameters gave good performance, showing the power that some different tweaks can make to an entire agents performance.

The worst performer that was not baseline was the DDQN with the Longer Exploration config with a 21.96 mean reward. Extending the epsilon decay from 150,000 to 300,000 frames meant the agent spent way too long exploring randomly which prevented it from learning efficiently within the 500k step budget.

Our learning curve observations graph show us that baseline DQN (blue line) converges around a mean reward of 10 never achieving meaningful improvement even with 500k steps. DDQN with variants show way larger learning curves especially after 300k steps. The bigger replay buffer variant (brown) shows us the best performance for later game runs, which beats out the baseline DDQN at the 500k mark. The Lower learning rate (green) was next in line after both DDQN and DDQN with replay, and showed the third best results showing us how changing learning rate by making it smaller is more beneficial in Frogger, If we combined both the reduced LR and a larger replay we might've gotten the very best results, isolating the experiments as we did allows us to see what what changes were most effective like we did in our case.

As for some visual observations from the recorded videos, In early training the agent moves randomly, typically dying or just staying still and moving left and right within the spawn point. Towards late training the agent demonstrates correct moves, getter correct timed movements through the traffic lanes and getting to the lake consistently.

Reflection

We chose Frogger because it has interesting mechanics for reinforcement learning (plus its nostalgic) in that there are sparse rewards that only arrive with correct moves in sequence, and where precise timing actually matters, there are also the multiple failure modes that the agent must learn to avoid, like cars and the platforming in the lake. The progress from staying still in early runs to good runs later on was clearly visible in our recorded videos. It was very satisfying to see how our models can learn to play a game through Reinforcement Learning.

As for algorithms DDQN was the single biggest input that changed the outcome. By decoupling action selection from action evaluation we reduced the Q value overestimation resulting in more stable learning. The baseline DQN failed to learn Frogger in any way, while the DDQN variants consistently got better with more and more steps.

The main challenges we encountered were first sparse rewards, that made early learning difficult. The agent often died a lot before receiving meaningful positive feedback making it hard

to learn the actions that were beneficial. This is visible in the slow initial progress across all experiments in our graph refer to *Figure 1*.

Another problem was the exploration vs. exploitation balance which was important. Our experiment with longer exploration (300k frame epsilon decay) shows us that too much exploration actively hurts learning because the agent wastes training time taking random actions instead of actively refining good behaviors.

One last problem was how to give correct credits over long action sequences. Successful Frogger runs require many correct decisions in sequence, but the agent only receives reward at the end. The larger replay buffer helped by providing more diverse experiences to learn from in later training steps, while DDQNs more accurate Q value estimations led to more confident decisions.

Future Improvements

Based on our results there are many improvements to be made such as reward shaping by adding small rewards for forward progress. There is also Prioritized Experience Replay where sampling important transitions like successful moves, more frequently could improve learning. Another possible improvement is using N step returns because combining rewards over multiple steps would improve credit assignment for the long sequences needed in Frogger. Of course we may have benefited from combining experiments like reduce LR, and a Bigger Replay.