# Spider Research Writeup

## Environment

We are using the CarRacing-v3 environment from gymnasium which contains 96x96x3 RGB pixel observations (27,648 dimensions) and is a continuous 3d space. With steering clamped to (-1,1), gas and brake to (0,1).

All three algorithms follow the basic Markov Decision Process (MDP) framework

Definition: Tuple(S, A, P, R, gamma)

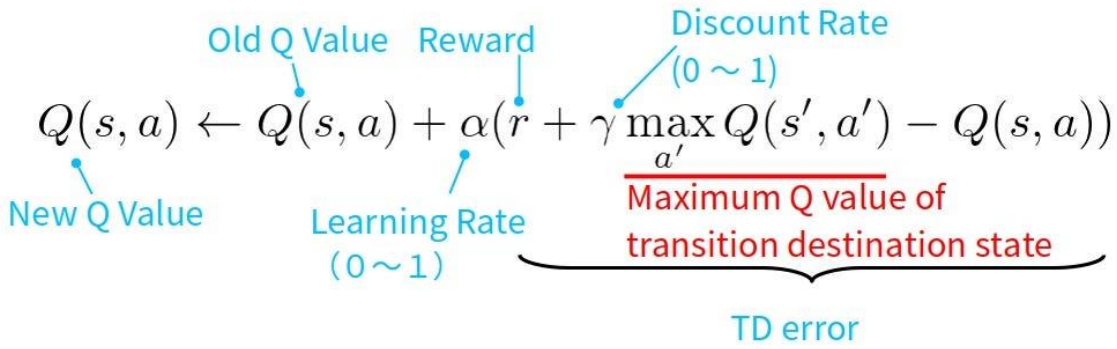
- S: State space
- A: Action space
- P (s'|s, a): Transition dynamics
- R (s, a, s'): Reward function
- Gamma belongs to 0,1: Discount Factor

## Q Learning

For the baseline model we use Q learning which is a basic model-free reinforcement learning algorithm that learns, how good it is to take an action a1 in state s1.

The measure of how good this act is stored as Q (s1, a1) which is the expected future reward

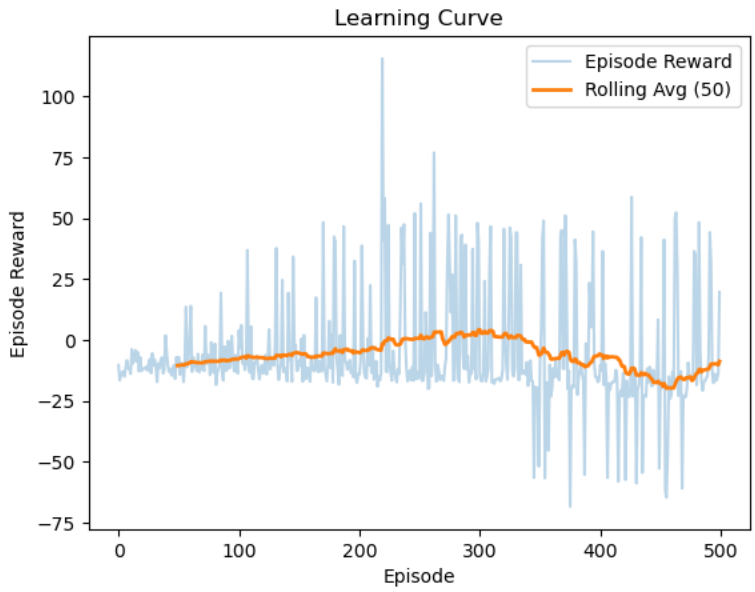
This model does not learn any policy directly. It simply tries to maximize the reward function. It is deterministic in nature which implies that after training one state will always lead to the same action which is the best known one (leading to higher reward overall)




It maintains a Q-table with entry for each (state, action) pair but for the CarRacing environment we need Q(s, a) for every possible pixel configuration, even after

grayscale 84x84: 256^(84x84) states which is ridiculously huge and since we have to discretize this continuous action space, we lose a lot of control leading to poor performance.

**Analysis**




This is the plot of the episode reward along with the rolling average with a window size of 50, the reward is a bit low than what we might expect, but that's because normal q learning is not suitable for handling this particular problem of CarRacing-v3 as this particular environment is a continuous image and doesn't quite fit in the assumption that Q learning takes with finite state space as even after discretization we have slightly different road positions, for each pixel in the road(96x96x3). This causes something called state aliasing. This mismatch of continuous and discretion-based approaches along with the how basic Q learning works by memorization leads to such low rewards

# Deep Q-Learning

The one major difference between normal Q learning and Deep Q learning is that, Q learning uses a normal Q table to map out the actions while DQN uses a CNN network to map out the states to the actions. For this particular environment of CarRacing-V3 we have 96x96 RGB images which we can resize to a suitable size and grayscale them to reduce computational complexity. To capture temporal dynamics such as velocity, direction, and acceleration, four consecutive frames are stacked together and treated as a single state representation. By processing these

stacked frames simultaneously, the DQN is able to infer motion-related information and learn how the car's behavior evolves over time within the environment.

```
DQN(
  (conv): Sequential(
    (0): Conv2d(4, 32, kernel_size=(8, 8), stride=(4, 4))
    (1): ReLU()
    (2): Conv2d(32, 64, kernel_size=(4, 4), stride=(2, 2))
    (3): ReLU()
    (4): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1))
    (5): ReLU()
  )
  (fc): Sequential(
    (0): Linear(in_features=1024, out_features=512, bias=True)
    (1): ReLU()
    (2): Linear(in_features=512, out_features=4, bias=True)
  )
)
```

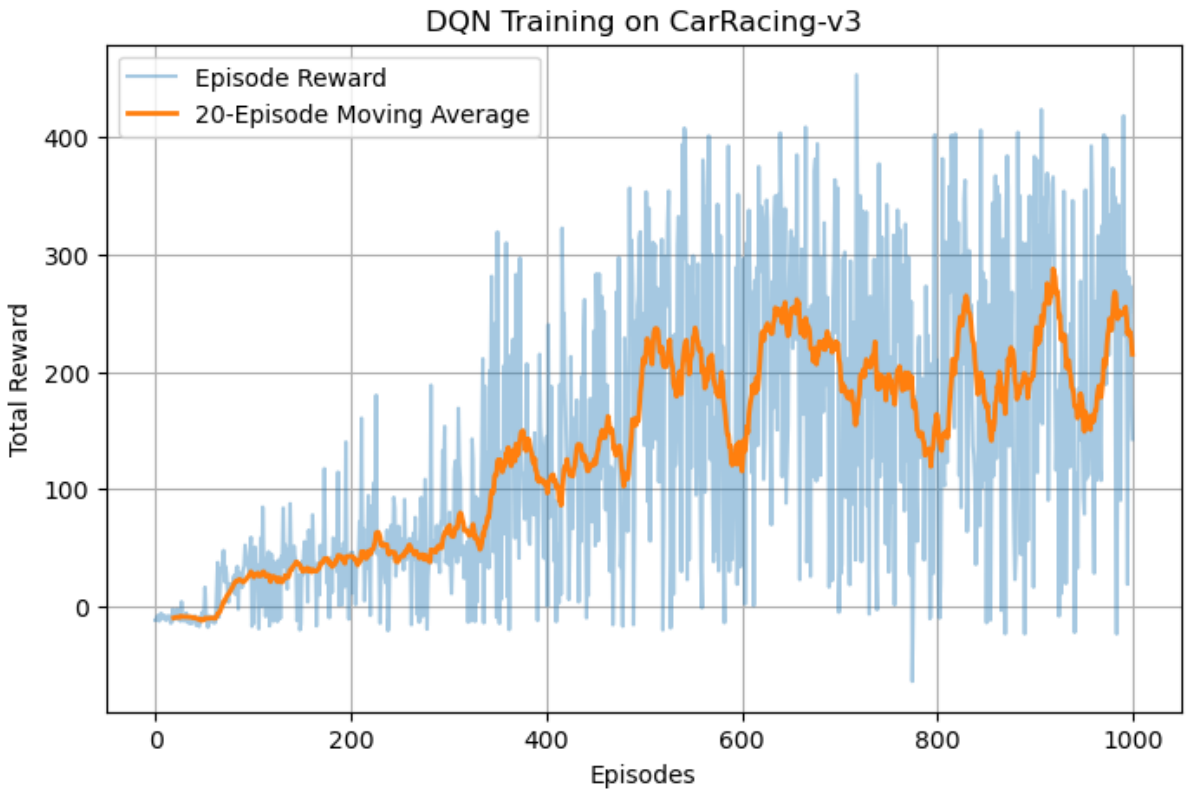
Additionally, an experience replay buffer is employed to store past transitions. During training, random minibatches are sampled from this buffer rather than learning from consecutive transitions. This randomization breaks temporal correlations in the data, improves sample efficiency, and stabilizes learning by preventing the network from overfitting to recent experiences.

We use two networks, the policy network and the target network. The policy network interacts with the environment, given a state s it outputs the q values for all possible actions. With probability epsilon the agent does exploration(random) otherwise it does exploitation (Selects action with highest Q value) epsilon is decayed over time and while the policy network is updated with every step, the target network which we strive towards is updated every fixed number of steps. This is for stability as we need a fixed target to move towards.

DQN is also based on the Bellman optimality equation, which defines how Q-values should relate to future rewards

$$Q(s) = r + \text{gamma} * \max(Q[s1])$$

During training, the network does not know the true Q-values. Instead, it moves its predictions toward a target value computed using this equation.




DQN faces similar problems in being a something designed for discrete action spaces and so saturates at the 500-episode interval.

# Proximal Policy Optimization:

PPO is a policy updating model, it doesn't directly affect the model. It fixes an old policy and it will not change while collecting data. The agent basically acts normally, explores and makes mistakes and it stores states, actions, rewards and how confident the policy was about those actions. After this the PPO judges whether the action was better or worse than expected this is basically the "Advantage".

$$A(s,a)=Q(s,a)-V(s)$$

Positive advantage is a good move while negative advantage is a bad move. So, we create a new policy basically that decides whether it should make a particular action more likely or less and it's basically a minor change of the old policy, we don't allow it to change too much by clipping it, this is how it differentiates from other on policy algorithms. We keep repeating this and update in slow intervals to figure out the best policy.

**Clipping mechanism:**

if A>0 (good action):

want to increase probability

but clip ratio at 1 + epsilon to prevent over optimization

if A < 0 (bad action):

want to decrease probability

but clip ratio at 1 – epsilon to prevent over penalization

---

For any differentiable policy and for any policy objective function, the policy gradient is:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(a_t|s_t) R(\tau)]$$

The term ∇logπ(a|s) tells us how to nudge the policy to make an action more or less likely

PPO is a combination of 2 networks: the actor and the critic which share convolutional features

**Actor(policy network):**

```
Shared Conv Features (3136) → FC(512)
  ↓                          ↓
μ = FC(512→3)              log_σ = FC(512→3)
  ↓                          ↓
Action ~ Normal(μ, exp(log_σ))  # Gaussian policy
```

**Critic (Value network):**

```
Shared Conv Features (3136) → FC(512)
  ↓
V(s) = FC(512→1)  # State value estimate
```

The actor basically maximizes reward while the critic reduces the variance in gradiance estimates. Since true Q(s,a) is expensive we use

A(s,a)=Q(s,a)−V(s) ≈R+γV(s′)−V(s). This approximation gives us something called the TD error.

**<u>Generalized Advantage Estimation (GAE):</u>**

PPO uses GAE to balance bias and variance

TD error: $\delta t = Rt + \gamma V(st+1) - V(st)$

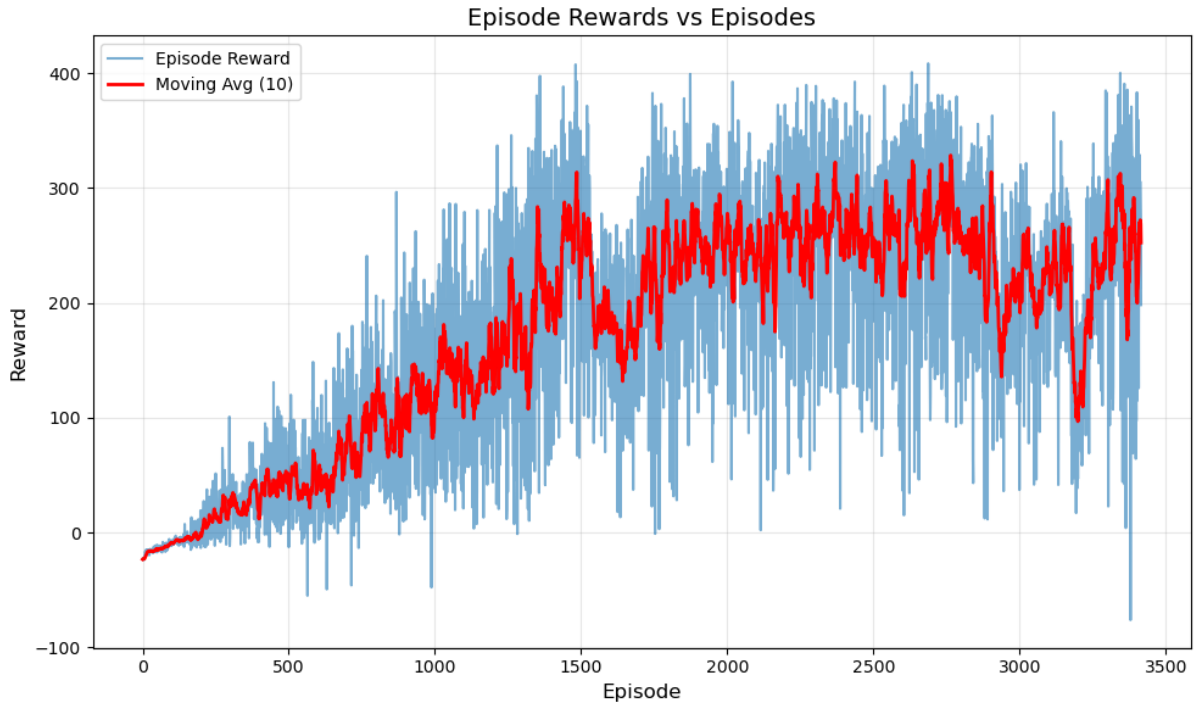
If $\delta t > 0$ critic underestimated

If $\delta t < 0$ critic overestimated

GAE Advantage: $At = k = \sum(\gamma\lambda)^k * \delta t + k$ here we add up the future TD errors but discount them twice, once by lambda (trust in the critic) and once by gamma( future uncertainty)

If lambda = 0 -> TD = 0 and it uses only one step look ahead with low variance and high bias.

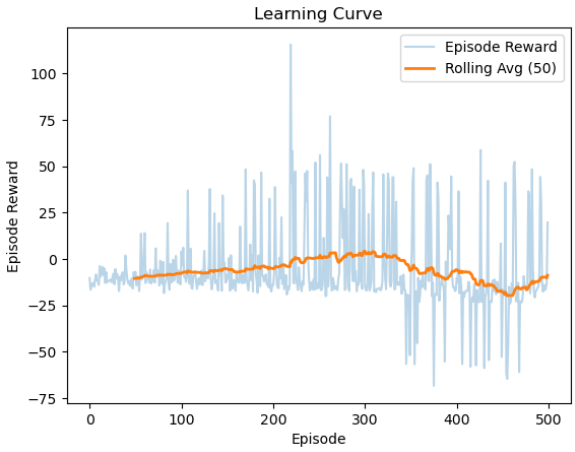
If lambda = 1 -> it uses full return with low bias and high variance, typical we use something between 0 and 1 so that it filters noise but keeps most long term info.




Unlike Q-Learning/DQN which learn value functions, PPO directly learns the policy pi(a|s, theta) and also provides native support for continuous spaces leading to a reward curve with more potential, if we increase the number of episodes, the model might get even more reward and become almost perfect at solving the environment.

# Experimental Analysis

**Q learning:**

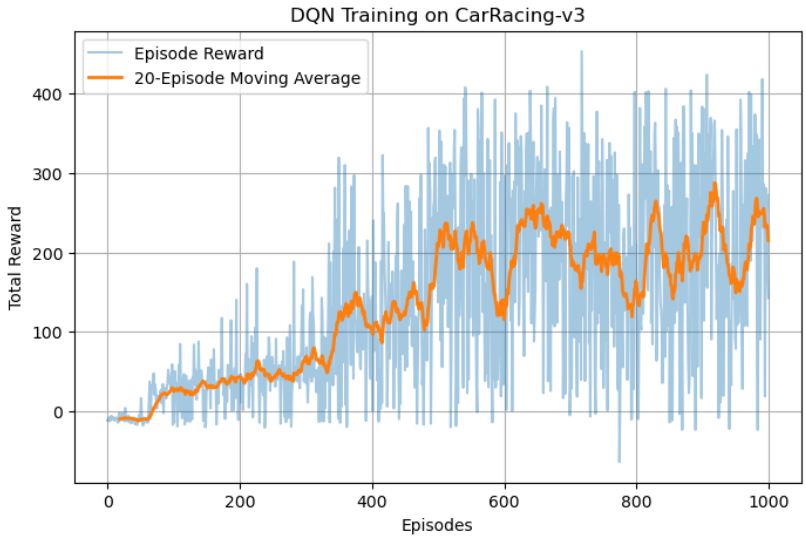



No of episodes = 500

Best Episode Reward: 4.22

Behaviour was random and kept going off track

**DQN:**




No of Episodes = 800

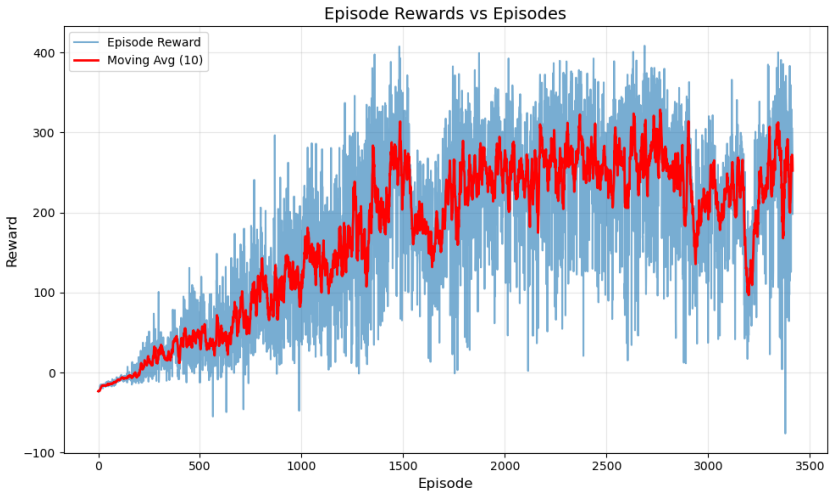
Mean Reward = 284.336

Std Reward = 111.743

Immensely better than normal Q learning but rewards did kinda saturate around the 500$^{th}$ episode and didn't really improve.

**PPO:**

| *Aspect* | *Off-Policy (DQN)* | *On-Policy (PPO)* |
|---|---|---|
| *Data Reuse* | *High (replay buffer)* | *Low (discard after use)* |
| *Sample Efficiency* | *Higher (theoretical)* | *Lower (theoretical)* |
| *Stability* | *Moderate (needs target net)* | *High (clipping)* |
| *Implementation* | *Complex (buffer, target)* | *Simpler* |
| *Continuous Actions* | *Requires modification* | *Native support* |




No of Updates = 500

No of episodes = 3410

Mean Reward = 201.317

Std Reward = 117.862

Although the mean and std reward are lesser than DQN, it would have gradually outperformed DQN if given a few more updates as PPO does usually take a few hundred more updates to gradually get more reward. Near the 490 mark it was getting around 400 rewards. Given sufficient time and system resources PPO might get massive reward.

# Theoretical Analysis

### Off policy (Q learning, DQN) vs On policy (PPO)

Despite lower theoretical sample efficiency, PPO converges faster on CarRacing due to having native continuous action support, stable learning(clipping) and better exploration

**Bias vs Variance**

**Bias**:

Error from approximating complex functions, this is usually because the model is too simple or makes strong assumptions which are systematically wrong

Eg: Linear model for a non linear problem

**Variance:**

Error from sensitivity to training data i.e the model is too flexible, small data changes causes big output changes, It has very noisy leaerning

Eg: monte carlo returns

Total error = Bias^2 + Variance + Irreductible error

Reducing bias usually increases variance and reducing variance usually increases bias

| Method | Bias | Variance |
|---|---|---|
| Monte Carlo | Low | High |
| TD(0) | High | Low |
| GAE(λ) | Tunable | Tunable |

In the case of GAE:

| λ | Use | Bias | Variance |
|---|---|---|---|
| 0 | Critic only | High | Low |
| 1 | Rewards only | Low | High |
| 0.95 | Mix | Balanced | Balanced |

So since TD(0) error has high biasing, Q learning and DQN also in turn have high bias and low variance as the target itself is a biased approximation (Q approximation)

| Algorithm | Bias | Variance | Best For |
|---|---|---|---|
| **Q-Learning (TD(0))** | High | Low | Short episodes, tabular |
| **DQN (TD(0))** | High | Low | Discrete actions, short horizon |
| **PPO (GAE, λ=0.95)** | Medium | Medium | **Long horizon, continuous control** |
| **PPO (λ=1, MC)** | Low | High | Simple problems, low noise |

## Stability Mechanisms

### DQN:

Problem: Correlation in sequential data

Consecutive frames are highly correlated so we create the experience replay system from which we sample random batches in no particular order to break correlation

Problem: Non stationary target

The target basically changes with every update so we make a target network that's a frozen copy of the q network and is updated only every n-steps

### PPO:

Problem: Drastic policy Updates

A small parameter change can cause a large policy change so we clip our surrogate objective.

To maintain stability in PPO we also do value function clipping, gradient clipping to prevent exploding gradients in deep networks, advantage normalization to maintain consistent gradient scales and entropy regularization to prevent premature convergence and maintain exploration