Reinforcement Learning Minichallenge Report

Introduction

This report aims to provide an in-depth analysis of DQN (Deep Q-Learning) based methods, their motivations, and their goals, The methods will be evaluated on the SpaceInvader game, made available by gym.

The initial idea behind DQN is to combine deep learning with reinforcement learning. Traditional Q-Learning struggles with high dimensional state spaces, a problem deep learning can help overcome. DQN uses a deep neural network as a function approximator to estimate the Q-values of a given state-action pair, allowing it to handle environments with large state spaces effectively due to its neural network architecture.

As a baseline, randomized actions are used. The code can be seen in random_baseline.py. This approach involves selecting actions randomly from the

available action space without any learning or strategy. Looking at its results can show what reinforcement learning brings to the table in this specific setting.

The primary goal of DQN is to maximize the cumulative reward over time. However, focusing solely on the rewards as a performance metric can be misleading. This is because the reward signal in reinforcement learning is often noisy and delayed, making it difficult to attribute changes in the reward to specific actions or policies.

The following metrics can be analyzed:

- Episode Frames and Episode Steps:
  - These metrics show the number of frames in an episode as well as the number of steps the agent takes. These are inherently correlated. In the gym's SpaceInvader environment, there is a parameter called frame-skipping which sets the number of frames passing by for each action. In the DQN trained, the amount of episode steps correlated to episode frames divided by 3. Looking at the episode frames we can see whether the agent is surviving longer.
- FPS:
  - Looking at the script performance using FPS (frames per second), we can see how difficult/easy it is to simulate with different learning methods.
- Epsilon:
  - This is the exploration rate in the epsilon-greedy policy. Tracking this can show how the balance between exploration and exploitation changes over time.
- Gradient Norm, Loss, Q-values:
  - Large changes in the gradient norm, loss, or Q-values might indicate instability in the learning process. The loss shows the value of the loss function for any given neural network. The Q-values show similar insights into the agent's learning and decision-making process.
- Videos:
  - Videos provide a qualitative assessment of the agent's behavior and strategy. How does the agent act in one episode?
- Speed of convergence:
  - How quickly does the algorithm learn?

Randomized Actions

This method samples a random action for the agent. The resulting training looks like this:

```
step=380000 | ALE/SpaceInvaders-v5/episode_frames=2083.85 |
ALE/SpaceInvaders-v5/episode_reward=152.16 |
ALE/SpaceInvaders-v5/episode_steps=695.95 | fps=7048.02

step=390000 | ALE/SpaceInvaders-v5/episode_frames=2123.32 |
ALE/SpaceInvaders-v5/episode_reward=155.15 |
ALE/SpaceInvaders-v5/episode_steps=709.10 | fps=7160.28

step=400000 | ALE/SpaceInvaders-v5/episode_frames=2060.82 |
ALE/SpaceInvaders-v5/episode_reward=155.36 |
ALE/SpaceInvaders-v5/episode_steps=688.27 | fps=7244.53

step=410000 | ALE/SpaceInvaders-v5/episode_frames=2130.31 |
ALE/SpaceInvaders-v5/episode_reward=154.06 |
ALE/SpaceInvaders-v5/episode_steps=711.46 | fps=7120.17

step=420000 | ALE/SpaceInvaders-v5/episode_frames=2059.49 |
ALE/SpaceInvaders-v5/episode_reward=149.57 |
ALE/SpaceInvaders-v5/episode_steps=687.84 | fps=7325.31
```

We can see that the agent doesn't learn and the metrics obviously look stable. The episode_reward is basically an average reward when every action is random. The FPS is very pretty high on my local system since no neural network is trained during the steps. If we look at the game recorded:

Randomized Action Video of SpaceInvaders

The agent makes randomized actions and dies very quickly due to it. As we can see, the score stays consistently very low across multiple steps.

DQN

In the DQN script, first, a couple of parameters are defined as flags for training. Then the game environment is defined. The environment is resized to 84X84 pixels. The deep Q network is defined using the DQN class that uses a PyTorch sequential module to build the architecture of a simple convolutional neural network. This takes the input (frames of SpaceInvader) and convolves it into the best next action. The train function is used to train the DQN model. Then it starts a loop that runs for a total number of steps specified by the parameter "FLAGS.total_steps". The epsilon value depending on the current step is calculated using a linear scheduler: LinearSchedule. It starts high

and decreases over time. The epsilon value is used to take a random action with its probability or use the model's current policy. The action based on the deep neural network is chosen by an argmax function. Taking the steps in the gym environment results in new observations, rewards, done flags (whether the episode has ended), and additional information. This is done for multiple environments in parallel. Using a quick check whether an environment has ended, the ended environments metrics are pushed to a logger. All of the observed observations, next observations, actions, rewards, and done states are pushed to the replay buffer that is used later on to train the neural network. All the environments are managed by the "gym.vector.SyncVectorEnv" class which automatically resets finished environments to the start state. This means, the environments keep going in parallel and no environment needs to wait until the others finish.

Training the neural networks:

After each step taken in the parallel environments, a batch is sampled from the replay_buffer. This replay_buffer is full of (state, action, reward, next_state, done) pairs that were added during individual steps. How many steps are initially added before training the neural network is defined using the flag "warmup_steps". We can also define the maximum amount of values stored in the replay buffer which defaults to 100'000.

The batch of the replay buffer is used to train the DQN. It uses the DQN and a target network that is simply a copy of the DQN. The Q-network is responsible for learning and updating the Q-values based on the agent's interactions with the environment. It is updated at every step, and its goal is to approximate the optimal Q-function, which gives the maximum expected future rewards for each action in each state. On the other hand, the target network is a separate network that has the same architecture but its parameters are updated less frequently. The purpose is to provide stable Q-value targets when calculating the loss for updating the Q-network. If we used the constantly updating Q-network to generate these targets, it could lead to harmful correlations and instability during training. By keeping the target network fixed for a number of steps (given by the flag "target_network_update_freq"), we reduce the risk of the targets changing too rapidly, which could cause the Q-network to "chase a moving target" which results in unstable or divergent learning.

The target network is used on the next observation instead of the current one. It estimates the future rewards that the agent can expect to receive. The idea is that the

agent has taken an action in the current state and observed the immediate reward as well as the next state. Now, to update its Q-value estimate for the current state-action pair, it needs to estimate the total reward it can get from this point onwards. This total reward is the sum of the immediate reward and the discounted maximum expected reward at the next state, which is estimated by the target network. The agent is essentially trying to look one step into the future. This approach touches on the principle of optimality by Bellman, which states that the optimal policy in the current state must lead to the optimal policy in subsequent states.

The target network is also used as a guiding force for the Q-network since it doesn't have a ground truth to be trained on per batch. Also, the temporal difference (TD) and the Huber Loss are used to calculate the loss for the Q-network.

Compute the estimate of best Q-values starting from the next states:

$$Q_{\text{target}}(s', a') = \max_{a'} Q_{\text{target}}(s', a')$$

Mask Q-values where the episode has ended at the current step:

$$Q_{\text{target}}(s', a') = (1 - \text{done}) \cdot Q_{\text{target}}(s', a')$$

Compute TD target:

$$\text{TD}_{\text{target}} = r + \gamma \cdot Q_{\text{target}}(s', a')$$

Compute estimated Q-values of actions taken in the current step:

$$Q(s, a) = Q_{\text{network}}(s, a)$$

Compute Huber Loss:

$$L(y, f(x)) = \begin{cases} 0.5 \cdot (y - f(x))^2 & \text{for } |y - f(x)| \leq \delta \\ \delta \cdot (|y - f(x)| - 0.5 \cdot \delta) & \text{otherwise} \end{cases}$$

Optimize the model, and update its parameters:

$$\nabla_\theta L = \nabla_\theta [0.5 \cdot (\text{TD}_{\text{target}} - Q(s, a))^2]$$

$$\theta = \theta - \alpha \cdot \nabla_\theta L$$

DQN Results

Due to time constraints, the model was still learning when the training was stopped. Still, the model went through almost 1 million steps. Also, the logs of the previous steps were lost due to a terminal error. The training logs can be seen here: Train Logs

After a lot of steps, the model generally performs better on the task and its episode reward has improved. When instantiating this model, the reward is similar to the one from the randomized model. At the 960'000th step, the reward has gone up to 274.42 compared to around 150-155 in the randomized model. This is also shown in the episode episode_frames / episode_steps which are generally higher than the ones in the randomized model. Meaning, that the model is alive for a longer time. The FPS is also pretty stable across all steps. Statements on grad_norm, loss, and q_values will be given when more data is shown for future models. Looking at the most recent video of the agent: DQN Video of SpaceInvaders Last State, we can see the model being proactive in shooting the aliens. Sometimes, it seems to be stuck in one place alternating between left move, right move, and shooting. This means that the aliens have it easy to shoot it down. There doesn't seem to be much strategy in the agent's brain, it's just shooting a lot. It's still quite hard to see what it has learned and how it's better than the randomized version. It seems to be aiming a bit better than in the video of the initial state of the model: DQN Video of SpaceInvaders First State

The model's training time was very long and it doesn't seem to learn very quickly. In the next chapters, some common techniques will be added to DQN that have been shown to improve it.

Double DQN

In standard DQN, the same network (Q-network) is used to both select and evaluate an action. This can lead to an over-optimistic estimate of the Q-value, as the network might consistently overestimate the Q-value of certain actions due to naturally occurring noise. The main idea behind Double DQN is to reduce this overestimation bias that can occur. It arises due to the max operator used in the target network's "next q values" estimation, which selects the maximum value from a noisy estimate of several action values.

This video nicely shows an example of how noise can be handled and then resumes with the explanation for double DQN.

In Double DQN, the issue is handled by using the Q-network to select the best action of the next observation to then calculate the next q values using the target network and the indices of the best actions chosen by the Q-network. This makes the "next q-values" calculation use both networks at the same time reducing noise and the overestimation bias.

The only lines changed for Double DQN compared to simple DQN are the following:

The line

```
# compute estimate of best q values starting from next states
next_q_value, _ = target_network(batch['next_obs']).max(dim=1)
```

Is changed to:

```
# compute estimate of best actions starting from next states using the
online network
_, best_actions = q_network(batch['next_obs']).max(dim=1)

# compute Q-values of the best actions using the target network
next_q_value = target_network(batch['next_obs']).gather(1,
best_actions.unsqueeze(dim=1)).squeeze(dim=1)
```

Double DQN Results

Looking at the Train Logs, the reward after the same amount of steps as in the simple DQN example wasn't much higher but due to large fluctuations, some of the episodes reached over 300 points as soon as the 860'000th step which is remarkable and not seen in the simple DQN results. We can see the epsilon decreasing making the model exploit more down the line instead of exploring. The loss seems to be going up over time which is weird since it's trying to minimize the Huber loss. An increasing loss might indicate lots of things. The increase in the loss shows that the model is not finished with learning. It is also not the best metric for performance since there are a lot of different states to be looked at during training that might make it hard during training to correctly predict the sum of rewards by the Q-network. The Q-value is increasing steadily which is a good sign and shows the model's performance in predicting higher future rewards for its actions.

Another interesting metric is the FPS count which has decreased quite a lot using Double DQN. This is probably due to the fact that to calculate next_q_values, both networks are used instead of only the target network.

Looking at the saved video from the last step: Double DQN Video of SpaceInvaders Last State, the agent seems to actually follow the crowd of aliens when moving left to right. The reward is not that great but it seems to understand that moving with the aliens increases the probability that its lasers destroy the aliens. Its movement has therefore improved and it doesn't get stuck in the same place anymore. It still fails to actively dodge lasers and doesn't strategize using the orange blocks as shields which humans would intuitively do.

Dueling Double DQN

The Dueling Double DQN extends the Q-network architecture. It splits the Q-values calculated using the Q-network CNN architecture into two different parts: The value function V(s) and the advantage function A(s.a). The dueling DQN algorithm splits the neural network into two different heads, one of them to estimate the state-value function for state s (V(s)), the other one to estimate the advantage function for each action "a" (A(s, a)). In the end, it combines both parts into a single output, which will estimate the Q-values. Essentially, the idea of the advantage function (A(s, a) = Q(s, a) - V(s)) is to measure how much better or worse an action "a" is compared to the average action in state "s" under a certain policy. This split can be helpful because sometimes it is unnecessary to know the exact value of each action. Using just the V(s) can be enough.

The problem with that is that backpropagation is harder since the model doesn't know how to calculate V and A in Q = V+A, due to multiple valid equations. The following statement is true for any c: Q = (V+c) + (A-c). The model cannot uniquely determine the values of V and A during training. To address this issue, the dueling DQN architecture forces the advantage function estimator to have zero advantage at the chosen action. This is done by subtracting the mean (batch-wise) of the advantages of all possible actions from the estimated advantage, resulting in the equation Q = V + (A-mean(A)). This ensures that the advantages sum to zero and provides a unique solution to V and A that the model can learn. This strategy can be seen in the forward function of the model:

```
def forward(self, obs):
    if len(obs.shape) == 3:
        obs = torch.unsqueeze(obs, dim=1)  # add channel dim
    obs = obs * (1. / 255.)
    x = self.feature(obs)
    advantage = self.advantage(x)
    value     = self.value(x)
    return value + advantage - advantage.mean()
```

We could also use the max value of the advantages, but the paper proposes the mean.

Dueling Double DQN Results

The Train Logs show an improved reward and time alive in comparison to Double DQN, especially in the last couple of steps. It reaches 300 points much earlier but still fails to learn for a long time. Dueling Double DQN has actually a higher loss than Double DQN making the reward and loss negatively correlated for some weird reason. The FPS is only a tiny amount smaller than in Double DQN which is surprising since the model uses another calculation in the neural networks. The gradient norm is higher than in Double DQN which might indicate that the gradient steps are bigger which could be from the additional complexity of the architecture.

When looking at the agent playing in Dueling Double DQN Video of SpaceInvaders Last State, the agent seems to be only slightly better in shooting and tracing the aliens than in the Double DQN example. Not much difference can be seen but it seems to be living longer. Still, no other advanced strategy can be seen.

Prioritized Experience Replay

Sampling data from the RingBuffer might not be optimal. Some experiences are inherently more frequent and don't give the model a lot of new information to learn. We can optimize the draw of samples using a prioritized replay buffer. This buffer simply uses prioritized entries in its memory to sample them more often. The priorities of those entries are easily transformed into probabilities. Entries just recently sampled get an updated priority score. This new priority depends on the td_error which just means the amount of the current q-value predicted by the Q-network is off from the TD target. If it is off by a lot, the entry gets a higher priority to be sampled again since the Q-network hasn't gotten that right yet. This strategizes learning of the Q-network.

Prioritized Experience Replay Results

This method showed extremely high q-values after the 120000th step which is not normal. Also, the gradient norm is very high. The agent doesn't learn quite as much as with the other models. There must be some kind of bug in the code for that to happen or something is missing. The agent doesn't survive long in general. Interestingly enough, the video of the last step shows something interesting: Prioritized Experience Replay Video of SpaceInvaders Last State. The agent actually doesn't die before the video ends abruptly. It is actually pretty good at dodging the alien's lasers. This might be an outlier that was captured by the video but it might show an underlying problem with the model's metrics if the video is accurate. I don't know exactly how the videos of the agent are made and whether there is something weird going on. The reward is not high in the video because it's not destroying aliens fast enough but the survival time is good which is weirdly not shown in episode_frames + episode_steps in the training logs: Train Logs. Since the model was only trained for 2 hours, the epsilon is still very high, so the model is still in the exploration phase. Therefore it's difficult to make assumptions about the model.

Dueling Double DQN with more complex CNN

The CNN of the DQN is very simple. Maybe using a different architecture can improve training time. The architecture added was one additional convolutional layer, batch norms for each layer, and one skip connection at the end for the last convolutional layer. The value and advantage streams also got a small dropout layer. The result was underwhelming since the model learned so slowly that it wasn't really noticeable. Due to time constraints, only 500'000 steps were done, but it was only at around 180 rewards at that time. The other metrics also don't seem stable as shown in Train Logs. The grad_norm metric is also higher than in previous models indicating that the gradient is very high due to the complexity of the model. It also starts very high. The FPS has decreased by a very small amount.

Another experiment was done using a decreased epsilon starting value. It decreased to 0.4 from 1.0. The idea was to see what the reward looks like when the exploitation is higher. The CNN is pretty small compared to CNNs used for image classification. Maybe it doesn't need to explore that much. The results show that the reward metric is

very unstable across different steps. Due to time constraints, the model couldn't fully be trained but up to the 250000th step, it looks slightly worse than the basic Dueling Double DQN approach which it is based on.

Conclusion

Using common approaches to improve DQN, some slight training improvements could be made that weren't tested on very high step counts. With "only" 1 million steps, a significant difference could still be made visible. This shows that those common approaches work. The Prioritized Experience Replay algorithm was implemented but might have an error since its Q-values are extremely high. All in all, all models perform better than the randomized approach and show machine learning. In general, parameter fine-tuning, training longer, and some more tweaking of the code could potentially lead to even faster training convergence.

Further Ideas

The images could be further processed and used to adjust the reward signal for the model. By using object detection one could detect the aliens, their lasers, the orange blocks, and all of their coordinates. For example, the agent could be rewarded by shooting down aliens, avoiding bullets, being behind an orange block, and being penalized for destroying orange blocks. This is essentially reward shaping which can also lead to unintended behaviors. Image processing would be hard since right now the images are in greyscale and pretty small.

Another idea would be to learn a model of the environment with a model-based reinforcement learning algorithm. Additionally, exploring other reinforcement learning algorithms, such as policy gradient methods or actor-critic methods, could provide interesting comparisons and potential improvements.

In general, a lot of finetuning can be made by adjusting the model architecture and even changing the way actions are handled ("softmax" instead of "hardmax"), and adding noise can help improve the training process.

Reflection

Unfortunately, I didn't have much time to evaluate each model. Nevertheless, I learned some methods that can be used to improve DQN learning. These methods were mysterious at the start but after a bit of reading, they didn't seem that bad and this encourages me to keep on learning about reinforcement learning algorithms. While ideas are abstract in the beginning, there is always a way to look at them intuitively.