UCLA ECE C147/C247 - Neural Networks and Deep Learning Final Project

Guy Ohayon
Technion - Israel Institute of Technology
Haifa, Israel
ohayonguy@campus.technion.ac.il

Aidan Cookson
University of California, Los Angeles
CA 90024
aucookson@g.ucla.edu

Max Gong
University of California, Los Angeles
CA 90024
maxtheg@g.ucla.edu

Abstract

In this project, we explored two different reinforcement learning methods, Deep Q-Learning and REINFORCE algorithm (Policy Gradient) with a deep learning agent, to solve the OpenAI Gym environment "Cart-Pole." We optimized hyperparameters in two successive experiments for each agent to find a good configuration that solves the environment. We compared the results of the best agents and validated our intuitions.

1. Introduction

1.1. Project description

Reinforcement Learning deals with constructing an agent that learns to perform a sequence of actions on an environment, so as to attain the goal of the environment. Usually, the environment is assumed to obey the rules of a Markov Decision Process (MDP), and the agent is assumed to perform valid actions. We denote $\pi(S_t)$ to be an agent's policy, which is a function that returns a valid action given any state S_t. Let $\tau_\pi = \tau = (S_1, A_1, R_2, S_2, A_2, R_3, \ldots, S_T, A_T, R_{T+1})$ be a trajectory of some episode following the policy π (the trajectory is simply a sequence of the states observed in an episode, the actions performed by the agent after each observed state, and the rewards returned by the environment after each performed action). Notice that in our notation, each reward is considered to be at time step $t+1$ after performing an action at time step t. Also, notice that T, S_t, A_t, R_{t+1} are all random variables (in the general case). Let $G_t(\tau) = G_t = \sum_{i=t}^{T} \gamma^{i-t} R_{i+1}$ be the return of the trajectory τ from time step t onwards, where $\gamma \in (0, 1]$ is called the discount factor (the return is simply the discounted cumulative reward). The goal of a Reinforcement Learning agent is then to learn a policy $\pi^*_\theta(S_t)$ that maximizes the expected return following a state S_t:

$$\boxed{\mathbb{E}_{\pi^*_\theta}[G_t|S_t] = \mathbb{E}_{\pi^*_\theta}[\sum_{i=t}^{T} \gamma^{i-t} R_{i+1}|S_t] = \max_\theta \mathbb{E}_{\pi_\theta}[G_t|S_t]}$$

In this project, we explored the following approaches to construct an agent:

1. An agent that learns the quality function $Q_\theta(S_t, A_t) = \mathbb{E}[G_t|S_t, A_t]$ of a (state, action) pair. Then, after learning, at each state the policy of that agent would be to pick the action that maximizes $Q_\theta(s, a)$:

   $$\boxed{\pi(S_t) = \pi^*_\theta(S_t) = arg\max_{a_t} Q_\theta(S_t, a_t)}$$

   This method is called Q-Learning.

2. An agent that learns the policy directly, following a state S_t. Then, after learning, the action choice at each time step would simply be:

   $$\boxed{\pi(S_t) = \pi^*_\theta(S_t)}$$

   This method is called Policy Gradient.

In our work, we explored both of these techniques to solve a famous environment called Cart-Pole [3]:

- A pole is attached by an un-actuated joint to a cart, which moves along a frictionless track.
- The system is controlled by applying a force of +1 or -1 to the cart.
- The pole starts upright, and the goal is to prevent it from falling over.
- A reward of +1 is provided for every time step that the pole remains upright.

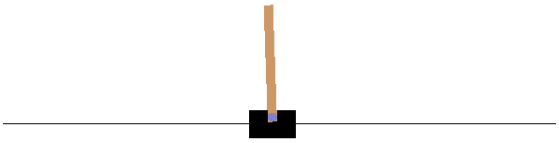

Figure 1: A visual representation of the Cart-Pole environment

- An episode ends when the pole is more than 12 degrees from vertical, or when the cart moves more than 2.4 units from the center.
- Each state is a vector that consists of 4 values:

Observation	Min value	Max value
Cart Position	-4.8	4.8
Cart Velocity	$-inf$	inf
Pole Angle	$-24°$	$24°$
Pole Velocity at Tip	$-inf$	inf

  Notice that the actual min and max values implemented in OpenAI's code are different than the values stated here: https://gym.openai.com/envs/CartPole-v1/.
- Each action is a vector that consists of 2 values:

Action	Value
Push cart to the left	0
Push cart to the right	1

According to OpenAI Gym: the environment is considered as solved if the average total reward from the last 100 episodes is at least 475.

1.2. Project settings

In our project, we solved the Cart-Pole environment by using two different kinds of algorithms:

1. **Deep Q-Learning**[2]: A Q-Learning algorithm in which the agent consists of a neural network that receives an input vector S_t and returns an output vector $(Q_\theta(S_t, a_t = 0), Q_\theta(S_t, a_t = 1))$.
2. **REINFORCE Algorithm**[3, 1]: A Policy Gradient algorithm in which the agent consists of a neural network that receives an input vector S_t and returns an output vector $(\pi_\theta(S_t) = 0, \pi_\theta(S_t) = 1)$, where $\pi_\theta(S_t) = k$ is the probability that the policy would choose the action $a_t = k$ at a given state S_t.

To examine each of these agents, we experimented with different hyperparameter configurations (all of which used 2 hidden layers). With each hyperparameter configuration, we attempted to solve the environment 5 times and averaged the number of episodes it took to solve the environment. This was our performance metric for each configuration (the lower this metric is, the better the performance). If an agent fails to solve the environment within 5000 episodes in any of the 5 trials, we considered this agent as failure - it's not able to solve the environment. We optimized the performance of each agent with two experiments, the first to test a broad range of hyperparameter values, and the second to fine-tune each parameter.:

1. In the first experiment, we tested:
   - Different sizes (number of neurons) of the 2 hidden layers (both hidden layers are of the same size): 1, 10, 20, 80, or 256.
   - Different activation functions between the hidden layers: ReLU, LeakyReLU, tanh, and identity activation.
   - Using batch normalization or not.
   - Different weight initialization: constant initialization of either 0.1 or 30 to all weights, uniform distribution from $U[-0.1, 0.1]$ or $U[-30, 30]$ to all weights, Xavier, and PyTorch Default initialization (which we'll call Default from now on).

   In each hyperparameter setting, we used:
   - $\gamma = 1$, since future rewards are crucial in this environment.
   - Adam optimizer, with learning rate $\epsilon = 0.01$.
   - $L2$ weights normalization, with $\lambda = 0.005$.
   - Maximum number of episodes = 5000.
   - Random seed = 123.

   We analyzed the performance of all tested hyperparameter configurations, determined the best settings, and hypothesized reasons behind the results.

2. In the second experiment, we fine-tuned the best hyperparameter configurations found in experiment 1 (both for DQN and REINFORCE, separately), and further improved the agents' performance. We achieved this by varying $\gamma, \lambda, \epsilon$ and the size of the hidden layers. Then the average performance was compared to previous configurations (the fine-tuned hyperparameters tested will be reported in section 2, after stating the results of experiment 1).

2. Results

2.1. REINFORCE Results

In experiment 1, we discovered that hidden size of 10, $tanh$ activation, no batch normalization, and constant 0.1 initialization resulted in the best performing agent, which solved the environment in 178 episodes on average. We then proceeded to experiment 2 by testing the following hyperparameters (by fine tuning the best hyperparameters that we found before):

- $\gamma \in [0.92, 0.96, 1]$.
- $\epsilon \in [0.5, 0.05, 0.005, 0.0005, 0.01]$.
- $hidden\_sizes \in [8, 10, 12, 14, 16]$.

- $\lambda \in [0.5, 0.05, 0.005, 0.0005]$ (L2 regularization hyperparameter).
- $tanh$ activation.
- No batch normalization.
- Constant 0.1 initialization.

From experiment 2, we discovered that $\gamma = 1$, $\epsilon = 0.01$, $hidden\_size = 8$, and $\lambda = 0.005$ resulted in the best performing agent, which solved the environment in 236 episodes on average.

2.2. Deep Q-Learning Results

In experiment 1, we discovered that hidden size of 256, $tanh$ activation, no batch normalization, and Default initialization resulted in the best performing agent, which solved the environment in 392.8 episodes on average. We then proceeded to experiment 2 by testing the following hyperparameters:

- $\gamma \in [0.92, 0.96, 1]$.
- $\epsilon \in [0.5, 0.05, 0.005, 0.0005, 0.01]$.
- $hidden\_sizes \in [200, 250, 300]$.
- $\lambda \in [0.5, 0.05, 0.005, 0.0005]$ (L2 regularization hyperparameter).
- $tanh$ activation.
- No batch normalization.
- Default initialization.

From experiment 2, we discovered that $\gamma = 1$, $\epsilon = 0.005$, $hidden\_size = 200$, and $\lambda = 0.0005$ resulted in the best performing agent, which solved the environment in 383.8 episodes on average.

3. Discussion

In our project, we experimented with different kinds of neural networks to explore the REINFORCE and DQN agents. We discovered that:

- $\gamma = 1$, $\epsilon = 0.01$, $hidden\_size = 8$, $\lambda = 0.005$, $tanh$ activation, no batch normalization, and constant 0.1 initialization resulted in the best performing REINFORCE agent. $\gamma = 1, \epsilon = 0.005, hidden\_size = 200$, $\lambda = 0.0005$, $tanh$ activation, no batch normalization, and Default initialization resulted in the best performing DQN agent.
- In both agents, using $\gamma = 1$ made both REINFORCE and DQN agents to consistently perform better. This makes sense since γ measures how much the agent should care about future rewards when it picks the next action ($\gamma = 1$ means future rewards are equally important to immediate rewards). In the Cart-Pole environment, an agent should strive to balance the pole long-term, since the objective is to be able to prevet the pole from dropping for as long as possible. Thus, $\gamma = 1$ was expected to result in better performing agents.
- We observed that the REINFORCE agent required a significantly smaller hidden size to behave optimally (REINFORCE required a hidden size of 8, and DQN required a hidden size of 200). Firstly, this gives REINFORCE a significant advantage in terms of computational complexity. Secondly, this result was expected since in the Cart-Pole environment it's much easier to predict the best next action directly rather than predicting the quality (Q) of the next action. For example, if the pole is leaning to the right, the best action is clearly to move right so as to try balancing the pole to the left, but it's a lot harder to approximate the Q for each action (how much reward would the agent get if it chose to move right or left). REINFORCE agent approximates the policy directly, and thus requires a less complicated model to be able to perform optimally in our environment.
- The REINFORCE agent consistently performed better than DQN. This might be because we didn't test enough hyperparameters for DQN, but we think that this also follows because of the simplicity of determining the best next action and the difficulty of determining what's the Q for each action, as explained above.
- We observed that the REINFORCE agent solved the environment in many more hyperparameter configurations, while DQN was highly sensitive to hyperparameters. This makes sense in general, since DQN requires good choices of exploration-exploitation scheduling, replay buffer size, and batch size, all of which depend on the environment and the other hyperparameters. This doesn't mean that DQN performs worse than REINFORCE, but rather that DQN requires more attention in hyperparameter selection, and thus claiming that REINFORCE strictly performs better than DQN on this environment is not proven and is only true empirically. But, since in our environment it's very simple to determine the best action at any state, we think that given the optimal hyperparameters of REINFORCE and DQN, REINFORCE agent would still perform better and require much less computation.
- We observed that increasing the hidden layer size for the REINFORCE agent resulted in worse performance than using a smaller hidden size (we were even able to solve the environment with hidden size of 1!). This result is expected, since increasing the hidden size more than necessary makes the agent's decision function over-determined, which increases the noise in actions and decreases the performance of the agent. If the agent requires a hidden layers size of 8 to perform optimally, adding more neurons would mean that many neurons ought to learn the same thing, but increases the noise that's coming out to the decision layer.
- Empirically, tanh activation seemed to lead to better performance.

References

[1] R. S. S. et al. Policy gradient methods for reinforcement learning with function approximation, 2000.

[2] V. M. et al. Playing atari with deep reinforcement learning, 2013. DeepMind Technologies.

[3] R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning, 1992.