Reinfocerment Learning Project

Juan Guevara

MAY, 2023

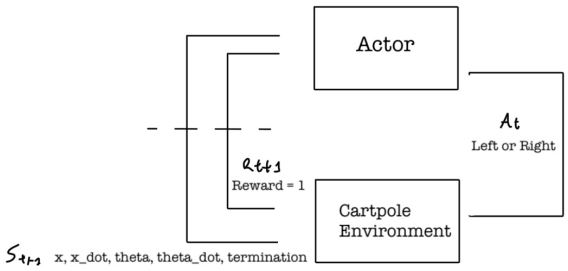
2 Cartpole Environment

The cartpole environmnet is "a schematic representation of a cart to which a rigid pole is hinged." The cart moves along the horizontal axis, applying an impulsive force, either to the left or the right, to balance the pole that moves on the vertical plane (and has a tendency to fall).

To describe the MDP of this environment, we need the state space, the action space, the transition kernel, and the reward. The physics of the system (our *transition kernel*) are described with kinematics equations that depend on 4 variable: the position of the cart on the track x, the angle of the pole with the vertical axis θ, cart velocity $\dot{x}$, and the rate of change of the angle $\dot{\theta}$. This tuple of 4 numbers will be our *state*. Upon inspection of the *state space*: $x \in (-2.4, 2.4)$; $\theta \in (-\frac{\pi}{15}, \frac{\pi}{15})$; $\dot{x} \in (-\infty, \infty)$; $\dot{\theta} \in (-\infty, \infty)$. The first two ranges are given by

```
self.theta_threshold_radians = 12 * 2 * math.pi / 360
self.x_threshold = 2.4
```

in the carpole.py code. As mentioned earlier, our cart will apply an impulsive force to the left or right. So our *action space* has cardinality 2. The *reward* is going to be 1 in every step, until the termination condition is reached (x or θ fall out of range).



3 Policy deep neural network

As we saw, our *states* (the input) are 4 variable tuples and the cardinality of our *action space* (the number of actions) is 2.

We select the action as the one outputted with the greatest probablity according to our polcy. The performance of our policy so far is terrible (the pole immediately hits the ground) because we have not trained our model yet.

```
def __init__(self):
    """Initialize model."""
    super(ActorModel, self).__init__()
    # ---> TODO: change input and output sizes depending on the environment

    #our state is describe by x, theta, x_dot, and theta_dot
    input_size = 4
    #we can go either left or right
    nb_actions = 2
```

```
    # ---> TODO: how to select an action
    # select action with the greatest probability according to policy
    action = int(torch.argmax(probabilities))
```

4 REINFORCE algorithm

1. See code and comments

```
# ---> TODO: compute discounted rewards
# new_r represents the cumulative discounted reward at each time step.
new_r = 0
# we work backwards to efficiently calculate the powers of DISCOUNT_FACTOR
for r in saved_rewards[::-1]:
    # the DISCOUNT_FACTOR's exponent increases by one for every future reward
    new_r = r + DISCOUNT_FACTOR*new_r
    # append it to the list
    discounted_rewards.append(new_r)
#restore temporal order
discounted_rewards.reverse()
```

2. See code and comments

```
 # For each time step
 actor_loss = list()
 for p, g in zip(saved_probabilities, discounted_rewards):
     # ---> TODO: compute policy loss

     # loss = discounted reward * log of the policy probability
     # we put the negative so that backpropagation peroforms gradient ASCENT
     time_step_actor_loss = -g * torch.log(p)

     # Save it
     actor_loss.append(time_step_actor_loss)

 # Sum all the time step losses
 actor_loss = torch.cat(actor_loss).sum()
```

3. See code and comments

```
#We tested 0.90 and 0.99. Both got similar results
DISCOUNT_FACTOR = 0.90

# ---> TODO: change the learning rate to solve the problem

# we tested 0.1, 0.01, and 0.001
# 0.1 did not converge
# 0.01 was unstable
# 0.001 proved the best of the three tests in terms of speed and stability
LEARNING_RATE = 0.001
```

4. We decide to stop the training with the hyperparameter STOPAGE. We will stop the training when our simulated episode receives the maximum reward STOPAGE number of times consecutively.

```
#training will stop when we achieve the maximum score STOPAGE amount of times
STOPAGE = 10

# cirular queue will store the last STOPAGE rewards
final_reward_queue = deque(maxlen = STOPAGE)
```

```
        # stop training when maximum reward is achieved STOPAGE consecutive times
        final_reward_queue.append(int(episode_total_reward))
        # we check it by comparing the sum of the circular queue
        if sum(final_reward_queue) == STOPAGE*HORIZON:
            break
```

5 Markov Property

We remove the accelarations:

```
# We remove the accelerations creating a new state with only x and theta
processed_state = np.array([self.state[0], self.state[2]])
```

As expected, the policy did not converge. Our environment is described by the 4 values $x, \dot{x}, \theta, \dot{\theta}$. This means that the environment requires the 4 quantities of the previous state to accurately determine what happens in the next state. If we eliminate the accelerations, the transition between states is not possible, and the Markov property is broken. The Markov property states that the future state depends only on the current state and not on the past history. Our REINFORCE algorithm heavily relies on MDP structure, so failing to satisfy this condition renders the policy unusable. Still, it achieved decent scores (compared to our random policies) because it had some description of the environment to work with. By the way, for our neural netowrk to work, we had to modify the input space to 2.

```
iteration 3115 - last reward: 32.00
iteration 3120 - last reward: 30.00
iteration 3125 - last reward: 71.00
iteration 3130 - last reward: 28.00
iteration 3135 - last reward: 11.00
```

One possible way to restore the lost information is by "encapsulating" two states into a single state. Then, our new state will consist on the current x_t and θ_t and the the previous x_{t-1} and θ_{t-1}. Let's call this tuple $x_a, \theta_a, x_b, \theta_b$. With this, our model has complete description of the environment's physics and the markovian property is restored. To transition to the next state, the values x_{b_t} and θ_{b_t} become $x_{a_{t+1}}$ and $\theta_{a_{t+1}}$, and $x_{b_{t+1}}$ and $\theta_{b_{t+1}}$ is calculated by the environmet, since it can recover the accelaration from the state. Therefore, we only depend on the previous state to know the next one.

```
# we create a new variable to keep track of the previous state
# since we are "encapsulating" this information  in a new type
# of state, we are not breaking the markovian property, but matining it
self.prev_state = self.state
```

```
# ---> TODO: if no accelerations, determine a new working state
# we use two steps to create our new state
processed_state = np.array([self.state[0], self.state[2],
    self.prev_state[0], self.prev_state[2]])
```

6 Freedom of Choice: Actor Critic and What's under the hood

Sarsa and Q-learning are instances of action-value methods, in which we learn a function (also called a table, since its input space is $|\mathcal{S}| \times |\mathcal{A}|$) that outputs the expected reward given a state and action. Then our policy can greedily select the next move. This table if usually updated with some variation of iterative TD learning.

Policy gradient methods, like the one in this code, do not rely on state-value tables. Instead, we consider the family of policies $\pi_\theta(a|s)$, parametrized by a vector of weights θ. Here, we maximize our reward function $J(\pi_\theta) = v_{\pi_\theta} = E(\sum_{t=0}^{\infty} \mathcal{R}_{t+1})$, assuming no discount for simplicity (we are only talking about the episodic case! Discount is mandatory for continuous problems) via gradient ascent. The idea is iteratively updating the values $\theta_{t+1} = \theta_t + \alpha \nabla J(\theta_t)$.

Computing the gradient is complicated in a model-free context because future rewards depend on future states, which distirbution is, by definition, unkown. Still, we have a theroem that is able to express this gradient in a form that does not rely on the transition kernel $p(s'|s, a)$. Enters, the policy gradient theorem.

The policy gradient theorem says.

$$\nabla J(\theta) \propto \sum_s \mu(s) \sum_a Q^{\pi_\theta}(s,a) \nabla \pi_\theta(a|s) \tag{1}$$

Where $\mu(s)$ is the on-policy distribution under π_θ, which is simply the expected amount of times we visit a given state in our policy. The proof of this theorem is within this project's scope but outside of my brain's scope after final exams :(.

Observe that the right hand side of the equation above is the sum over states weighted by how often the states occur under the target the policy π_θ. So we can interpret it as the expectation under π_θ of the value funciton $\sum_a Q^{\pi_\theta}(s,a)\nabla\pi_\theta(a|s)$

$$\sum_s \mu(s) \sum_a Q^{\pi_\theta}(s,a) \nabla \pi_\theta(a|s) = E_{\pi_\theta}[\sum_a Q^{\pi_\theta}(S_t,a) \nabla \pi_\theta(a|S_t)] \tag{2}$$

And we are a few tricks away from the final REINFORCE algorithm. Multiplying and dividing by $\pi_\theta(a|S_t)$

$$\nabla J(\theta) \propto E_{\pi_\theta}[\sum_a \pi_\theta(a|S_t) Q^{\pi_\theta}(S_t,a) \frac{\nabla \pi_\theta(a|S_t)}{\pi_\theta(a|S_t)}]$$
$$\nabla J(\theta) \propto E_{\pi}[\sum_a Q^{\pi_\theta}(S_t,a) \nabla \ln \pi_\theta(a|S_t,\theta)]$$

For the last step, we define $G_t = \sum_{k=t+1}^{T} R_k$. We observe that G_t is just a montecarlo estimator for the state value function. $E_\pi(G_t|S_t,A_t) = Q^\pi(S_t,A_t)$. So, we finalize the gradient policy by interpreting the sum over a as an expectection over A_t

$$\begin{aligned}\nabla J(\theta) &\propto E_{\pi_\theta}[\sum_a Q^{\pi_\theta}(S_t,a) \ln \pi_\theta(a|S_t)] \\ &= E_{\pi_\theta}[Q^{\pi_\theta}(S_t,A_t) \ln \pi_\theta(A_t|S_t)] \\ &= E_{\pi_\theta}[G_t \ln \pi_\theta(A_t|S_t)]\end{aligned}$$

One last realization. The family of weights we are updating can be the weights of a neural network. So, the ascent performed in the Monte-Carlo Policy Gradient algorithm (attached below, from Sutton's book) is just a backpropagation!

REINFORCE: Monte-Carlo Policy-Gradient Control (episodic) for π_*

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
Algorithm parameter: step size $\alpha > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ (e.g., to $\mathbf{0}$)

Loop forever (for each episode):
  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot,\boldsymbol{\theta})$
  Loop for each step of the episode $t = 0, 1, \ldots, T-1$:
    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$ $\quad (G_t)$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \gamma^t G \nabla \ln \pi(A_t|S_t,\boldsymbol{\theta})$

In the alogirithm above (and in our implementation), the discount factor is taken into account. The ideas are the same, though.

The problem with this algorithm is that, despite its nice theoretical convergence properties, it has high variance. We deal with this with actor-critic models.

An Actor-Critic model is a type of reinforcement learning algorithm that combines the benefits of both policy-based methods (our policy gradient) and value-based methods (we will see which later!). The actor

and the critic interact and learn from each other iteratively. The critic provides feedback to the actor to guide its policy updates, while the actor explores the environment and collects experiences for the critic to learn better value estimates.

The updating rule of the policy is given by Sutton and Barto

$$\begin{aligned}\theta_t &= \alpha(G_{t:t+1} - \hat{v}(S_t, w))\nabla \ln(\pi(A_t|S_t)) \\ &= \alpha(R_t - \gamma\hat{v}(S_{t+1}, w) - \hat{v}(S_t, w))\nabla \ln(\pi(A_t|S_t)) \\ &= \alpha\delta_t \nabla \ln(\pi(A_t|S_t))\end{aligned}$$

Where $\hat{v}(S_t, w)$ is the state value function estimated by our critic, parameterized by the vector of weights w (aha! Another neural network).

In our implementation, we will improve $\hat{v}(S_t, w)$ with the semi-gradient TD(0) algorithm. We will not spend too much time on this, but it is enough to observe that it is continual and online, on top of the fact that it relies on gradient ascent, just like our new version of the policy-gradient algorithm. The gradient displayed in the pseudocode of TD(0) algorithm is the result of chainruling the minimum square error $\nabla(v(S_t, w) - \hat{v}(S_t, w))^2 \propto (v(S_t, w) - \hat{v}(S_t, w))\nabla\hat{v}(S_t, w)$, and then substituting $v(S_t, w)$ (the real state value given our policy) by a prototipical semigradient estimator estimator, such as $R + \gamma\hat{v}(s', w)$.

Semi-gradient TD(0) for estimating $\hat{v} \approx v_\pi$

Input: the policy π to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S}^+ \times \mathbb{R}^d \to \mathbb{R}$ such that $\hat{v}(\text{terminal},\cdot) = 0$
Algorithm parameter: step size $\alpha > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose $A \sim \pi(\cdot|S)$
    Take action A, observe R, S'
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R + \gamma\hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})\big]\nabla\hat{v}(S,\mathbf{w})$
    $S \leftarrow S'$
  until S is terminal

One-step Actor–Critic (episodic), for estimating $\pi_{\boldsymbol{\theta}} \approx \pi_*$

Input: a differentiable policy parameterization $\pi(a|s, \boldsymbol{\theta})$
Input: a differentiable state-value function parameterization $\hat{v}(s,\mathbf{w})$
Parameters: step sizes $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\mathbf{w}} > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^d$ (e.g., to $\mathbf{0}$)
Loop forever (for each episode):
  Initialize S (first state of episode)
  $I \leftarrow 1$
  Loop while S is not terminal (for each time step):
    $A \sim \pi(\cdot|S, \boldsymbol{\theta})$
    Take action A, observe S', R
    $\delta \leftarrow R + \gamma\hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})$ (if S' is terminal, then $\hat{v}(S',\mathbf{w}) \doteq 0$)
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}}\delta\nabla\hat{v}(S,\mathbf{w})$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} I\delta\nabla \ln\pi(A|S, \boldsymbol{\theta})$
    $I \leftarrow \gamma I$
    $S \leftarrow S'$

During our training of the actor-critic model, we had some really interesting behavior. Our policy worked quite nicely for many episodes of training, and suddenly the scores dropped to never recover.

Reading about it, we have a possible explanation. In algorithms such as TD(0), when our state value is overfitted, it no longer has the capacity to recover from mistakes, and the total amount of errors compounds quadratically. I tried multiple learning rates and discount factor combinations, as well as cliping the norm of the gradient with.

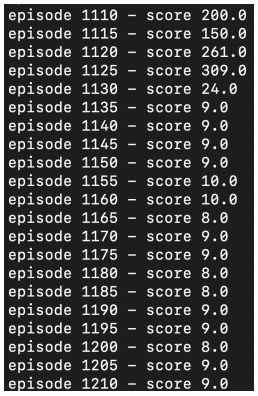


Figure 1: Catastrophic forgetting?

```
torch.nn.utils.clip_grad_norm_(critic.parameters(), 0.5)
```

To no avail. The final solution is given chaging the loss as below

```
# First try, as stated in Sutton and Barto's algorithm. Unsuccessful
advantage = reward + DISCOUNT_FACTOR * new_state_val.item() - state_val.item()
val_loss = - advantage * state_val

# Second try, multiplying loop discount (DISCOUNT_FACTOR^time_step). Unsuccessful
advantage = reward + DISCOUNT_FACTOR * new_state_val.item() - state_val.item()
val_loss = - advantage * state_val * I

# Third try, letting the gradient step apply the chainrule for us
# and multiplying by the loop discount. Successful
val_loss = F.mse_loss(reward + DISCOUNT_FACTOR * new_state_val, state_val)
val_loss *= I
```

Interestingly, adding the discount factor helped in the convergence, but it is not part of Sutton's origianl algorithm. Anyways, finally our model coverges with good enough stability. That concludes our exploration of the actor critic algorithm!