Report of Deep Reinforcement learning Project

REINFORCE and PPO Implementation
on the CartoPole-V0 environnement

Authors:
Asma Ben Nasr
Mohamed Znouda

Academic year :2021/2022

Table of contents:

1. The environment:

   CartPole-v0: CartPole, also known as inverted pendulum, is a game in which you try to balance the pole as long as possible. It is assumed that at the tip of the pole, there is an object which makes it unstable and very likely to fall over.
   So, CartPole-v0 is a reinforcement learning concept on cartpole. It consists of a pole where it 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 pendulum starts upright, and the goal is to prevent it from falling over. A reward of +1 is provided for every timestep that the pole remains upright. The episode ends when the pole is more than 15 degrees from vertical, or the cart moves more than 2.4 units from the center.

2. Implementation:
   a. REINFORCE and hyperparameters:
      REINFORCE is a Monte Carlo variant of policy gradients. With our packages imported we are going to set up a class called Policy that will contain our neural network. It's going to have two hidden layers with a ReLU activation function and w softmax output defined in the forward

function. We will also give it a method called act to sample action from the distribution and get its log probability.

```
class Policy(nn.Module):
    def __init__(self, s_size=4, h_size=16, a_size=2):
        super(Policy, self).__init__()
        self.fc1 = nn.Linear(s_size, h_size)
        self.fc2 = nn.Linear(h_size, a_size)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return F.softmax(x, dim=1)

    def act(self, state):
        state = T.from_numpy(state).float().unsqueeze(0).to(device)
        probs = self.forward(state).cpu()
        m = Categorical(probs)
        action = m.sample()
        return action.item(), m.log_prob(action)
```

After setting up our optimizer to Adam optimizer with **a learning rate of 1e-2**, we implemented the REINFORCE algorithm in a function called reinforce.

In each episode, for each step we get the action and its log probability via the function act we defined, we save the log probability, apply the action to the environment and collect the reward.

All the rewards are used to calculate the discounted reward function i.e.

$G_t = \sum_{t=1}^{T} \gamma^t R_t$.

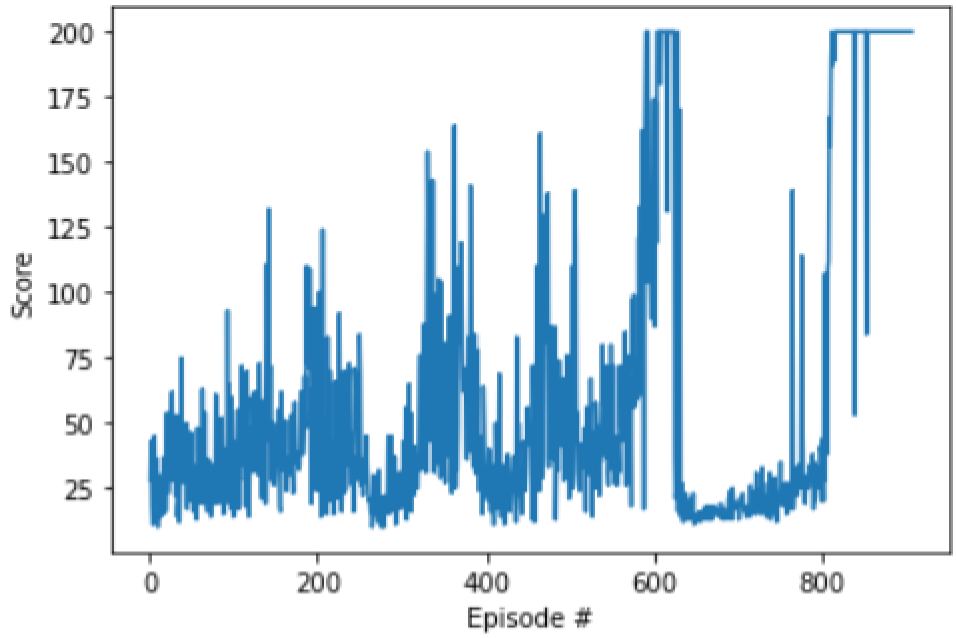
Then the policy loss is calculated as : $-\sum_{t}^{T} ln(G_t \pi(a_t|s_t, \theta))$

```python
policy = Policy().to(device)
optimizer = optim.Adam(policy.parameters(), lr=1e-2)

def reinforce(n_episodes=1000, max_t=1000, gamma=0.99, print_every=100):
    scores_deque = deque(maxlen=100)
    scores = []
    for i_episode in range(1, n_episodes+1):
        saved_log_probs = []
        rewards = []
        state = env.reset()
        for t in range(max_t):
            action, log_prob = policy.act(state)
            saved_log_probs.append(log_prob)
            state, reward, done, _ = env.step(action)
            rewards.append(reward)
            if done:
                break
        scores_deque.append(sum(rewards))
        scores.append(sum(rewards))

        discounts = [gamma**i for i in range(len(rewards)+1)]
        R = sum([a*b for a,b in zip(discounts, rewards)]) #discounted reward function

        policy_loss = []
        for log_prob in saved_log_probs:
            policy_loss.append(-log_prob * R)
        policy_loss = T.cat(policy_loss).sum() #calculate policy loss for all values in saved log probs

        optimizer.zero_grad()
        policy_loss.backward()
        optimizer.step()

        if i_episode % print_every == 0:
            print('Episode {}\tAverage Score: {:.2f}'.format(i_episode, np.mean(scores_deque)))
        if np.mean(scores_deque)>=195.0:
            print('Environment solved in {:d} episodes!\tAverage Score: {:.2f}'.format(i_episode-100, np.mean(scores_deque)))
            break

    return scores
```

The function is run for **1000 episodes** with **gamma =0.99**. The environment is solved in 806 episodes with an average score of 195.36.

```
Episode 100     Average Score: 31.53
Episode 200     Average Score: 46.22
Episode 300     Average Score: 34.97
Episode 400     Average Score: 58.02
Episode 500     Average Score: 44.04
Episode 600     Average Score: 64.04
Episode 700     Average Score: 60.87
Episode 800     Average Score: 25.74
Episode 900     Average Score: 187.45
Environment solved in 806 episodes!     Average Score: 195.36
```

The result for plotting the score by number of episodes is the following:



b. PPO and hyperparameters:

PPO is a policy gradient method.

To implement it we started by setting up a ppo.py file where we define all the necessary functions for our ppo algorithm to run properly in the train.ipynb notebook.

ppo.py:

With our packages imported we set up a class called PPOMemory that handles saving states, log probabilities , values returned by the critic, actions, rewards and dones. In this class we implemented functions to generate batches, store memory and clear memory.

```python
class PPOMemory:
    def __init__(self,batch_size):
        self.states=[] #states
        self.probs=[] # log probabilities
        self.vals=[] #values calculated by critic
        self.actions=[] #actions
        self.rewards=[] #rewards
        self.dones=[] #terminal flags
        self.batch_size= batch_size
    def generate_batches(self): #generate random batches of size batch_size
        n_states=len(self.states) #number of states
        batch_start=np.arange(0,n_states,self.batch_size)
        indices = np.arange(n_states,dtype=np.int64)
        np.random.shuffle(indices) #for randomness
        batches=[indices[i:i+self.batch_size]for i in batch_start]
        return(np.array(self.states),np.array(self.actions),np.array(self.probs),np.array(self.vals),np.array(self.rewards),np.array(self.dones)

    def store_memory(self,state,action,probs,vals,reward,done):
        self.states.append(state) #add  state
        self.actions.append(action) #add action
        self.probs.append(probs) #add log probability
        self.vals.append(vals) #add value
        self.rewards.append(reward) #add reward
        self.dones.append(done) #add done

    def clear_memory(self): #clear the memory at the end of every episode
        self.states=[]
        self.probs=[]
        self.vals=[]
        self.actions=[]
        self.rewards=[]
        self.dones=[]
```

Then, we set up a class ActorNetwork to define the architecture of the actor. Our actor network has two hidden layers with two ReLU activation functions ,a softmax output and an Adam optimizer with learning rate alpha. The forward function will forward the state in the actor network to calculate a series of probabilities that we use to draw from a distribution to get the actions and the log probabilities. save_checkpoint and load_checkpoint functions are for saving and loading checkpoints when training the actor network.

```python
class ActorNetwork(nn.Module):
    def __init__(self,n_actions,input_dims,alpha,fc1_dims=256,fc2_dims=356,chkpt_dir='./'):
        super(ActorNetwork,self).__init__()
        self.checkpoint_file = os.path.join(chkpt_dir,'actor_ppo')
        self.actor = nn.Sequential(
            nn.Linear(*input_dims,fc1_dims),
            nn.ReLU(),
            nn.Linear(fc1_dims,fc2_dims),
            nn.ReLU(),
            nn.Linear(fc2_dims,n_actions),
            nn.Softmax(dim=-1)
        )
        self.optimizer = optim.Adam(self.parameters(),lr=alpha)#optimizer
        self.device = T.device('cuda:0' if T.cuda.is_available() else 'cpu')#device gpu or cpu
        self.to(self.device)#send the network to the device
    def forward(self,state):
        dist=self.actor(state)
        dist=Categorical(dist) #calculates a series of probabilities that we are using to draw from a distribution to get our actions,
        #and we can use it to get log probabilities
        return(dist)
    def save_checkpoint(self):
        T.save(self.state_dict(),self.checkpoint_file)
    def load_checkpoint(self):
        self.load_state_dict(T.load(self.checkpoint_file))
```

Then, we set up a class CriticNetwork to define the architecture of the critic. Our critic network has one hidden layer with two ReLU activation functions, a linear output layer and an Adam optimizer with learning rate alpha. The forward function passes the state into the critic

```python
class CriticNetwork(nn.Module):
    def __init__(self,input_dims,alpha,fc1_dims=256,fc2_dims=256,chkpt_dir='./'):

        super(CriticNetwork,self).__init__()
        self.checkpoint_file=os.path.join(chkpt_dir,'critic_ppo')
        self.critic = nn.Sequential(
            nn.Linear(*input_dims,fc1_dims),
            nn.ReLU(),
            nn.Linear(fc1_dims,fc2_dims),
            nn.ReLU(),
            nn.Linear(fc2_dims,1)
        )
        self.optimizer=optim.Adam(self.parameters(),lr=alpha) #optimizer
        self.device = T.device('cuda:0' if T.cuda.is_available() else 'cpu') #device gpu or cpu
        self.to(self.device) #send the network to the device

    def forward(self,state):
        value=self.critic(state) #value by the critic
        return(value)
    def save_checkpoint(self):
        T.save(self.state_dict(),self.checkpoint_file)
    def load_checkpoint(self):
        self.load_state_dict(T.load(self.checkpoint_file))
```

network and return the critic value. save_checkpoint and load_checkpoint functions are for saving and loading checkpoints when training the critic network.

Finally, we set up the Agent class where we initialize the hyperparameters : gamma,policy clip, the number of epochs, gae lambda as well as the actor network , the critic network and the PPO memory.
We define a remember function to handle the interface between the agent and its memory as well as a save_models function and a load_models function. Also, we define a function choose_action where we sample an action from the distribution given by the actor. The function returns the action, its probability and the critic value.
Finally, we implement our learn function. For each epoch, we collect a set of trajectories then we calculate the generalized advantage estimate (GAE). As a matter of a fact, with GAE we create a mix between monte carlo and td updates, monte carlo methods have low bias and update with true rewards but high variance. TD methods have a high bias because we are using estimates to update another estimate but have low variance, a mix between them combines the best in both. After that, we update the policy by maximizing the PPO clip objective. Then we calculate the actor and critic loss and define the total loss as actor_loss+0.5* critic_loss.
At the end of all epochs, we clear the memory.

train.ipynb:

We define the **horizon N=20** (the number of steps before doing an update) , **batch_size=5**,**n_epochs = 4**, **alpha=0.0003**.

```
def learn(self):

    for _ in range(self.n_epochs):

        #collect a set of trajectories
        state_arr,action_arr,old_probs_arr,vals_arr,reward_arr,done_arr,batches=self.memory.generate_batches()
        values=vals_arr
        advantage=np.zeros(len(reward_arr),dtype=np.float32)

        for t in range(len(reward_arr)-1):

            discount=1
            a_t=0
            for k in range(t,len(reward_arr)-1):
                #compute generalized advantage estimate (GAE)
                a_t+=discount*(reward_arr[k]+self.gamma*values[k+1]*(1-int(done_arr[k]))-values[k]) #1-dones as a multiplicative value because the value of the terminal state
                discount *=self.gamma*self.gae_lambda
                #calculate the advantage
                advantage[t]=a_t
            advantage = T.tensor(advantage).to(self.actor.device)
            values = T.tensor(values).to(self.actor.device)
            for batch in batches:
                states=T.tensor(state_arr[batch],dtype=T.float).to(self.actor.device)
                old_probs=T.tensor(old_probs_arr[batch]).to(self.actor.device)
                actions=T.tensor(action_arr[batch]).to(self.actor.device)

                dist = self.actor(states)
                critic_value=self.critic(states)
                critic_value=T.squeeze(critic_value)
                new_probs=dist.log_prob(actions)
                prob_ratio=(new_probs-old_probs).exp() #exp(new_probs-old_probs) = exp(new_probls)/exp(old_probs) probs being : log(x)
                weighted_probs=advantage[batch]*prob_ratio
                weighted_clipped_probs=T.clamp(prob_ratio,1-self.policy_clip,1+self.policy_clip)*advantage[batch]
                actor_loss=-T.min(weighted_probs,weighted_clipped_probs).mean()
                returns=advantage[batch] + values[batch]
                critic_loss=(returns-critic_value)**2
                critic_loss = critic_loss.mean()
                total_loss=actor_loss + 0.5*critic_loss
                self.actor.optimizer.zero_grad()
                self.critic.optimizer.zero_grad()
                total_loss.backward()
                self.actor.optimizer.step()
                self.critic.optimizer.step()
    self.memory.clear_memory() #clear memory at the end of all epochs
```

Then we define our agent and train it on the environment for 1000 episodes. We only save the model corresponding to an improved average score.

The results given by the PPO are the following: