Report

December 30, 2019

1 Report

1.0.1 Deep Deterministic Policy Gradient (DDPG)

As a first implementation I used the Deep Determistic policy gradient algorithm to solve the Reacher enviroment. This algorithm is a a model-free, off-policy actor-critic algorithm that uses a deterministic policy for selecting actions. This algorithm is purposed for continous action spaces and combines elements of DQN in that it uses for example a replay buffer and actor/critic networks but also uses ideas from David Silver et al's Deterministic policy gradient paper.

DDPG is a policy gradient algorithm that uses a stochastic behavior policy for good exploration but estimates a deterministic target policy, which is much easier to learn. Since DDPG is off-policy and uses a deterministic target policy, this allows for the use of the Deterministic Policy Gradient theorem (which will be derived shortly).

DDPG uses two neural networks, one for the actor and one for the critic.

These networks compute action predictions for the current state and generate a temporal-difference (TD) error signal each time step.

The input to the actor network is the state (at time t) and it outputs a vector of floats reresenting the real valued actions the agent is to take in the enviroment. The critic's input is also the state (at time t) as well the action taken by the determinisitic policy (i.e. (s_t,actor_policy(s_t)) and it returns the expected rewards expected following the current policy onwards from that state(i.e. the value function) .

The critic network is updated using the gradient from the mean squared bellman error(i.e. just the squared loss between a) the sum of the reward at time t and an estimate of the future rewards using a frozen copy of the critic network parameters ; b) the current critic network evaluated at at the current state s_t using the current sampled action a_t).

The deterministic policy gradient theorem proved by Silver et al provides the update rule for the weights of the actor network. This update is essentially the gradient of the critic network holding the action taken as a constant multiplied by the gradient of the actor network with respect to the parameters of the actor network (which reminicient of the chain rule).

Function Approximator architecture choices

I used a vanilla feed forward architecture for both the actor and the critic network. I found that using batch-norm helped stablize the learning process and so included in the architecture. There 3 layers with a tanh output to scale the continous action between -1 and 1 for the actor. The critic had no scaling of the outputs.

Below is an example of the actor network

```
In [ ]: class Actor(nn.Module):
            """Actor (Policy) Model."""
```

```
def __init__(self, state_size, action_size, seed, fc1_units=400, fc2_units=300):
    """Initialize parameters and build model.
    Params
    ======
        state_size (int): Dimension of each state
        action_size (int): Dimension of each action
        seed (int): Random seed
        fc1_units (int): Number of nodes in first hidden layer
        fc2_units (int): Number of nodes in second hidden layer
    """
    super(Actor, self).__init__()
    self.seed = torch.manual_seed(seed)
    self.fc1 = nn.Linear(state_size, fc1_units)
    self.bn1 = nn.BatchNorm1d(fc1_units)
    self.fc2 = nn.Linear(fc1_units, fc2_units)
    self.fc3 = nn.Linear(fc2_units, action_size)
    self.reset_parameters()
def reset_parameters(self):
    self.fc1.weight.data.uniform_(*hidden_init(self.fc1))
    self.fc2.weight.data.uniform_(*hidden_init(self.fc2))
    self.fc3.weight.data.uniform_(-3e-3, 3e-3)
def forward(self, state):
    """Build an actor (policy) network that maps states -> actions."""
    x = F.relu(self.bn1(self.fc1(state)))
    x = F.relu(self.fc2(x))
    return F.tanh(self.fc3(x))
```

1.0.2 Exploration vs Exploitation

One challenge with using the a deterministic policy is that ... well its determinstic. This means that exploration techniques used in DQN such as the epsilon greedy selection cannot be used as there in no well defined probability distribution from which to choose action over.

One way suggested by the DDPG to get around this was to use the output of the determinstic policy as the mean parameter (mu) for the Ornstein-Uhlenbeck process (a stochastic process very closely related to Gaussian processes with the special property that it is mean reverting i.e. will eventually converge to mean of the parameter of the process which happens to be the deterministic action given by the current policy) The OU process also is correlated in time mean that the noise added to actions will tend not to counter action the given action (this was hypothesized by the paper). Open AI and other researchers however stated they they found that gaussian noise seemed to do the job. I also found this to be true and in fact my agent trained faster with Gaussian noise added as opposed to OU noise. I only decreased the varaince of the Gaussian noise the further on in trained the agent progressed.

1.0.3 Updating Scheme

Reading the bench mark section of the Udacity I decided to update my agented every 20th divisible step progressed in an epsiode and decided to perform 10 gradient updates with new randomly

sampled minibatches .

1.0.4 Number of Agents

I couldnt seem to get enough reward signal to get my agent of the ground and learning when using a single agent even after large number of episodes of training. I opt'ed for using the 20 agent enviroment instead because it provided more reward signal (hence non-zero gradients in the backward pass for training the function approximators)

1.0.5 Future Work and Improvements

As can be seen I also tried using A2C with Actor critic methods. Although DDPG was muuuuch more sample inefficient than the A2C algorithm, A2C was more cpu friendly and ran much faster(which may be beneficial if the time-to-train/sample efficiency trade is acceptable). Another problem with the A2C as currently implemented was how high variance the agent was(the agent was also unstable and suceptable to degragation in performance after having had slved the enviroment -- however I suspect that this is a me problem I could decreased exploration or implemented early stopping).

Future improvements I could include is generalized advantage estimation(GAE) with A2C. I could also try different RL algorithms which attempt to maximise a stochastic policy directly such TRPO or PPO(the approaches are also compatible with GAE).

There have been further improvements on DDPG that I could also try. For example :

D4PG Distributed Distributional DDPG (D4PG) applies a set of improvements on DDPG to make it run in the distributional fashion.These improvements include (all the below comes from https://lilianweng.github.io/lil-log/2018/04/08/policy-gradient-algorithms.html)

(1) Distributional Critic: The critic estimates the expected Q value as a random variable ~ a distribution Zw parameterized by w and therefore Qw(s,a)=Zw(x,a).

(2) N-step returns: When calculating the TD error, D4PG computes N-step TD target rather than one-step to incorporate rewards in more future steps.

(3) Multiple Distributed Parallel Actors: D4PG utilizes K independent actors, gathering experience in parallel and feeding data into the same replay buffer.

(4) Prioritized Experience Replay (PER)

TD3 The Q-learning algorithm is commonly known to suffer from the overestimation of the value function. This overestimation prograte through the parameters hurting the learning process

Twin Delayed Deep Deterministic (short for TD3; Fujimoto et al., 2018) proposes a couple of fixes to help with the overstimation of Q values (1) Clipped Double Q-learning: In Double Q-Learning, the action selection and Q-value estimation are made by two networks separately. The minimum of the estimated Q values is used in order to try discourage the overestimation of Q values

(2) Delayed update of Target and Policy Networks: In the actor-critic model, policy and value updates are deeply coupled(critic is used to evaluate the gradients of the actor). This leads to high variance

To reduce the variance, TD3 updates the policy at a lower frequency than the Q-function. The policy network stays the same until the value error is small enough after several updates. The idea is similar to how the periodically-updated target network stay as a stable objective in DQN.

(3) Target Policy Smoothing: Given a concern with deterministic policies that they can overfit to narrow peaks in the value function, TD3 introduced a smoothing regularization strategy on the value function: adding a small amount of clipped random noises to the selected action and averaging over mini-batches.

In [ ]: