Learning algorithm

The training algorithm employed is similar to that used in the paper *Continuous Control with Deep Reinforcement Learning* by Lillicrap, et al. This approach, known as the Deep Deterministic Policy Gradient algorithm (DDPG), borrows ideas from policy gradient, actor-critic and Deep Q-Learning Network (DQN) methods.

Policy gradient methods use models like deep neural networks to directly find the optimal policy by maximizing the expected return given a state using gradient ascent. DPG differs in that actions resulting from such models are deterministic rather than stochastic – the network just outputs an action, not the probability of taking an action. This has two benefits. First, in theory, the resulting policy gradient should require less samples to estimate as one of the elements contributing to the variance, namely the action value, is thrown out. We are just left with the variance from the state. Second, a DPG network can output continuous values, unlike policy gradient methods which just output the probability of a discrete action. A potential downfall of course is that we don't explore the action space. To remedy this, the DPG is made off-policy (i.e. we select an action, but hold off on applying back-propagation to the parameters of the network) with random noise added to the action to allow for exploration. As pointed out in the forum, the addition of noise to actions had to vary by agent and perhaps more importantly for convergence, had to lessen over time to minimize exploration closer to convergence.

Deep DPG may be viewed as an actor-critic RL method in that the DPG network acts as an "actor" network to predict an action using the current policy. The gradient estimate used to update that network uses another "critic" network which is used to estimate expected return. In theory, doing this at every time step instead of using the actual return at the end of episode allows the algorithm to potentially learn if specific steps were good or bad for the (estimated) return.

Actor-critic methods use the advantage function rather than q-values because the latter lead to a lot of instability. Instability is exactly what I what I saw in my experiments when I updated at every time step by blindly following the algorithm specified in the DDPG paper. As suggested in the Distributed Distributional DDPG (D4PG) paper, it's best to wait a number of time steps before updating. I got the best results when I waited until the end of an episode to update the networks. I also got much better results by using their suggestion of having multiple agents gather experience but draw from one update buffer and use the same critic. Finally, like almost any network used in non-trivial deep learning problems, the stability of the critic network improved when gradient clipping was applied, as suggested in the benchmark implementation.

Finally, DDPG borrows ideas from DQNs to stabilize approximating the above action and q-values using deep neural networks. In particular, DDPG uses a replay buffer to break the correlation between between time steps so networks can actually learn, and has separate target networks (one each for the actor and critic networks) to stabilize learning. The parameters of these separate target networks are updated using a soft update – i.e. a small

percentage of these are updated every time the original local networks get updated rather than at set number of time steps as in the original DQN method.

The specific version of DDPG used in this project comprises three components: a set of 20 agents to explore the environment, a single model containing the actor and critic networks, and the DDPG training algorithm itself.

The full training algorithm is described in Algorithm 1 and 1a below.

Algorithm 1 Training procedure for Reacher DDPG

Require:
The environment *env*
The *agent*

- With 20 sub-agents interacting with the environment
- Initialized with its local actor network $\mu(s|\theta^{\mu})$ and target network $\mu'(s|\theta^{\mu'})$ - initialize these to have the same weights
- Initialized with its local critic network $Q(s,a|\theta^{Q})$and target critic network $Q'(s,a|\theta^{Q'})$ - initialize these to have the same weights

update_after_episode - whether to wait until at least one sub-agent's episode is done to update agent's model
update_t – if latter is false, how often to update the weights of all networks, in timesteps
update_size – number of updates to networks' parameters per agent
The maximum number of episodes *M*
The maximum number of time steps per episode *T*
Initialize:
Experience replay buffer *R* to capacity *n*
Steps:
for episode = 1, M do
  state $s_{t=1,1..20}$ ← reset env
  Initialize random process $\mathcal{N}$ for exploration
  for t = 1, T **do**
    Calculate dampening factor df ← square root of the episode inverted
    Select action for each sub-agent i such that $a_{t,i} \leftarrow \mu(s_{t,i}|\theta^{\mu}) + df * \mathcal{N}_{t,i}$
    For all sub-agents, execute actions; observe rewards r_t, new states s_{t+1} and done flags
    For each sub-agent i, store transition ($s_{t,i}$, $a_{t,i}$, $r_{t,i}$, $s_{t+1,i}$, $done\_flag_i$) in R
    if update_after_episode = False **and** t = update_t **do**
      Update Actor-Critic Local and Target Networks (Algorithm 1a)
    end if
    break if any agent has the done flag set
  end for
  if update_after_episode = True **do**
    Update Actor-Critic Local and Target Networks (Algorithm 1a)

end if
end for

Algorithm 1a Update Actor-Critic Local and Target Networks

Require:
Local actor network $\mu(s|\theta^{\mu})$ and target network $\mu'(s|\theta^{\mu'})$ – see Algorithm 1
Local critic network $Q(s,a|\theta^{Q})$and target critic network $Q'(s,a|\theta^{Q'})$ – see Algorithm 1
update_size – see Algorithm 1
R – replay buffer, see Algorithm 1
γ - discount factor
τ - for soft update of target parameters
Steps:
n_updates ← number of agents * update_size
for j = 1, n_updates **do**
Sample a random minibatch of N transitions (s_j, a_j, r_j, s_{j+1}) from R
Set $y_j = r_j + \gamma Q'(s_{j+1}, \mu'(s_{j+1}|\theta^{\mu'})|\theta^{Q'})$
Update critic by minimizing the loss: $L = \frac{1}{N}\sum_j (y_j - Q(s_j, a_j|\theta^{Q})^2$
Update the actor policy by using the sampled gradient:

$$\nabla_{\theta^{\mu}} J \approx \frac{1}{N}\sum_j \nabla_a Q(s,a|\theta^{Q})|_{s=s_j, a=\mu(s_j)} \nabla_{\theta^{\mu}} \mu(s|\theta^{\mu})|_{s_j}$$

Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^{Q} + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^{\mu} + (1-\tau)\theta^{\mu'}$$

end for

Model

Actor: Each model (target and local network) is simply a 3-layer fully connected network. The input layer takes the state of the Reacher environment. The two hidden layers have ReLU activations. The output layer uses the tanh activation for the action value.

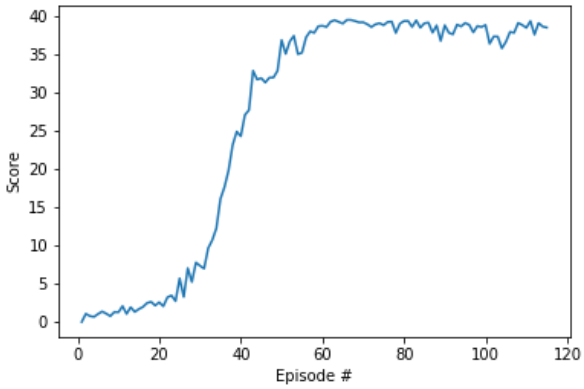
Critic: Each model (target and local network) is also a 3-layer fully connected network. The input layer takes the state of the Reacher environment and uses a ReLU activation. The second layer concatenates the output from the first layer and the action from the actor network; it too uses a ReLU activation. The output layer only does a linear transformation of the second layer's output to generate the q-values.

Hyperparameters

Name	Description	Value
n	experience replay buffer size	100000
N	SGD batch size	512
γ	discount factor	0.99
τ	for soft update of target parameters	0.001
lr_actor	actor network learning rate	0.001
lr_critic	critic network learning rate	0.001
M	maximum number of episodes	1000
T	maximum number of time steps per episode	1000
update_t	how often (in timesteps) to update networks; -1 means wait until episode end	-1
update_size	# of updates of networks per sub-agent	20
actor_fc1_units	# of units in actor's hidden layer 1	400
actor_fc2_units	# of units in actor's hidden layer 2	300
critic_fc1_units	# of units in critic's hidden layer 1	400
critic_fc2_units	# of units in critic's hidden layer 2	300
weight_decay	L2 weight decay	0
seed	seed to set random number generator for reproducible results	2

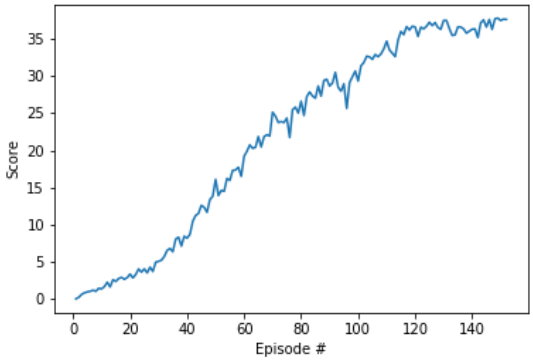
Results

Below is a plot of the average score (i.e. the sum of the rewards) per episode over a number of episodes averaged over all 20 sub-agents.



The number of episodes needed to achieve a score greater than 30 (30.32) is 115 episodes using the hyper-parameters defined above.

If the frequency of when to update the network is changed to 20 timesteps with 10 updates, as suggested in the benchmark implementation, the results are slightly worse, with the problem solved in 152 episodes.



Curiously, batch normalization did not help as much as expected. I got better results simply by waiting to update the network until I got way more samples and then updating multiple times so that the changes to parameters between updates weren't so large.

Ideas for Future Work

I only implemented some of the ideas from D4PG and not quite exactly as they did. For example, adding prioritized experience replay would likely help. And the sampling for transitions just considers one transition, and not the N-step returns.

Network stability was a serious problem for this project. I simply could not get the algorithm to work if it updated at every time step, or if only one agent was used. I needed a lot of samples to updated the critic network and thus minimize the variance of the q-values (which in turn

impacts the gradients of the actor/policy network.) As pointed out elsewhere, DDPG is only as good as the q-value estimate of the critic network.

I would also perhaps just switch to Trust Region Policy Optimization (TRPO). TRPO prevents large changes to the policy network's parameters by enforcing a KL divergence constraint on the size of the size of updates at each iteration. Another option is to use Truncated Natural Policy Gradient (TNPG) which effectively does the same thing (minimize gradient step size) using the Fisher Information Matrix.

Another problem was that training was really slow, thus impeding progress. I ended up running the code on my own GPU box and noticed that GPU utilization was at less than 10 percent most of the time. There's a lot of switching numpy arrays to tensors and back. Also, I could not readily have separate instances of even the non-vis Unity environment so I could try out different models on different GPUs (I have 4 on my own box). I thus need to investigate how to do this.