Report

September 5, 2019

1 Continuous Control

In this project, we seek to solve the Unity Reacher problem. A set of double-jointed arms exist in the field and the goal is to manipulate these arms so that the end of the arm is in the goal location (where the ball is). A positive reward of 0.1 is provided for each time step when the goal is met. The definition of solved for the Udacity task is that an average game score of >30 across 20 of these agents is achieved over the course of 100 games.

[6]:
```python
from IPython.display import HTML

# Youtube
HTML('<iframe width="560" height="315" src="https://www.youtube.com/embed/
↪2N9EoF6pQyE" frameborder="0" allow="accelerometer; autoplay; encrypted-media;
↪ gyroscope; picture-in-picture" allowfullscreen></iframe>')
```

[6]:
```
<IPython.core.display.HTML object>
```

2 Approach

The approach here is to follow the Deep Deterministic Policy Gradients (DDPG) algorithm. We seek to optimize a policy that is used to identify an action within a continuous action space. In this case, the action manipulates the two joints in the arm by values ranging between -1 and 1. The value of the policy at a given time step is approximated using Deep Q Learning where the action taken from at the time step t is that identified by the policy. The optimum policy is one that maximizes this value.

3 Background

We define a *policy* π as the probability that a given action a is taken when the agent is in a particular state s. The goal is to approximate an optimal set of parameters θ such that π_θ maximizes the expected return.

$$J(\theta) = \mathbb{E}[\sum_{t=0}^{T-1} r_{t+1}|\pi_\theta]$$

where r_t is the return at timestep t and π_θ is the policy parameterized by theta. Equivalently,

$$J(\theta) = \mathbb{E}[Q(s,a)|_{s=s_t,a_t=\mu(s_t)}]$$

where $Q(s,a)$ is the expected return of selecting action a from policy μ at state s and timestep t.

The actor in an actor-critic method selects an action from a *policy* while the critic in the DDPG algorithm estimates the value of selecting a given action from a given state essentially following the DQN algorithm. We learn the actor and the critic in parallel and they inform each other.

3.1 Algorithm

for episode=1,M do
>for t=1,T do
>> Select an action a_t according to the current policy μ subject to some external noise N_t >> $a_t = \mu(s_t|\theta^\mu) + N_t$
>> Execute action a_t and observe reward r_t and next state s_{t+1}
>> Store transition (s_t,a_t,r_t,s_{t+1}) in the replay buffer R
>> Sample a mini-batch of N transitions from R (s_i,a_i,r_i,s_{t+1})
>> Calculate the expected return for each sampled transition using the target value network and target policy
>> $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
>> Update the local critic network by minimizing the mean-squared error. >> For each entry, calculate the delta between the expected return using the current reward plus the expected future return versus the expected return from the current state and action
>> $L = \frac{1}{N}\sum_i(y_i - Q(s_i, a_i|\theta^Q))^2$
>> Update the local actor policy using the sampled policy gradient
>> $\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i(\nabla_a Q(s,a|\theta^Q)|_{s=s_i,a=\mu(s_i)} \nabla_{\theta^\mu}\mu(s|\theta^\mu))|_{s_i}$
>> Update the target networks
>> $\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$
>> $\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$

3.2 Implementation

First, define the actor and critic networks in model.py structured as defined in the ddpg paper as feed-forward two hidden layer networks with 400 units and 300 units respectively. Additionally, there is a batch normalization layer as also recommended in the paper.

```
class Actor(nn.Module):
    """Actor (Policy) Model."""
    def __init__(self, state_size, action_size, seed, fc1_units=400, fc2_units=300, use_batch_n
        """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.use_batch_norm = use_batch_norm
        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."""
        if self.use_batch_norm:
            x = F.relu(self.bn1(self.fc1(state)))
        else:
            x = F.relu(self.fc1(state))
        x = F.relu(self.fc2(x))
        return F.tanh(self.fc3(x))

class Critic(nn.Module):
    """Critic (Value) Model."""

    def __init__(self, state_size, action_size, seed, fc1_units=400, fc2_units=300, use_batch_
        """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 the first hidden layer
            fc2_units (int): Number of nodes in the second hidden layer
        """
        super(Critic, 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+action_size, fc2_units)
```

```
        self.fc3 = nn.Linear(fc2_units, 1)

        self.use_batch_norm = use_batch_norm

        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, action):
        """Build a critic (value) network that maps (state, action) pairs -> Q-values."""
        if self.use_batch_norm:
            xs = F.relu(self.bn1(self.fc1(state)))
        else:
            xs = F.relu(self.fc1(state))
        x = torch.cat((xs, action), dim=1)
        x = F.relu(self.fc2(x))
        return self.fc3(x)
```

The DDPG agent acts upon these models. Extract the sections from the algorithm and illustrate the code implementation.

3.2.1 Select an action a_t according to the current policy μ subject to some external noise N_t

$a_t = \mu(s_t|\theta^\mu) + N_t$

```
def act(self, state, add_noise=True):
    """Returns actions for given state as per current policy."""
    state = torch.from_numpy(state).float().to(device)
    self.actor_local.eval()
    with torch.no_grad():
        action = self.actor_local(state).cpu().data.numpy()
    self.actor_local.train()
    if add_noise:
        action += self.noise.sample()
    return np.clip(action, -1, 1)
```

3.2.2 Noise sample N_t

So, we select an action using the local actor and add some noise. The noise defined in the algorithm is the Ornstein-Uhlenbeck Noise Process.

```
class OUNoise:
    """Ornstein-Uhlenbeck process."""

    def __init__(self, size, seed, mu=0., theta=0.15, sigma=0.2, random_fn=np.random.randn):
```

```python
        """Initialize parameters and noise process."""
        self.mu = mu * np.ones(size)
        self.theta = theta
        self.sigma = sigma
        self.seed = random.seed(seed)
        self.random_fn = random_fn
        self.reset()

    def reset(self):
        """Reset the internal state (= noise) to mean (mu)."""
        self.state = copy.copy(self.mu)

    def sample(self):
        """Update internal state and return it as a noise sample."""
        x = self.state
        dx = self.theta * (self.mu - x) + self.sigma * np.array([self.random_fn() for i in rang
        self.state = x + dx
        return self.state
```

3.2.3 Execute action a_t, observe reward r_t, and next state s_{t+1}

```python
actions = agent.act(states)
env_info = env.step(actions)[brain_name]
next_states = env_info.vector_observations
rewards = env_info.rewards
dones = env_info.local_done
```

3.2.4 Add sample to replay buffer

```python
    def step(self, states, actions, rewards, next_states, dones):
        """Save experience in replay memory, and use random sample from buffer to learn."""
        # Save experience / reward

        for state, action, reward, next_state, done in zip(states,actions,rewards, next_states
            self.memory.add(state, action, reward, next_state, done)
```

3.2.5 Calculate the expected return

$y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
μ' is the actor_target and Q' is the critic target

```python
        # ---------------------------- update critic ---------------------------- #
        # Get predicted next-state actions and Q values from target models
        actions_next = self.actor_target(next_states)
        Q_targets_next = self.critic_target(next_states, actions_next)
        # Compute Q targets for current states (y_i)
        Q_targets = rewards + (gamma * Q_targets_next * (1 - dones))
```

3.2.6 Calculate the critic loss

$L = \frac{1}{N}\sum_i(y_i - Q(s_i, a_i|\theta^Q))^2$

```
        Q_expected = self.critic_local(states, actions)
        critic_loss = F.mse_loss(Q_expected, Q_targets)

        # Minimize the loss
        self.critic_optimizer.zero_grad()
        critic_loss.backward()
        torch.nn.utils.clip_grad_norm_(self.critic_local.parameters(), 1)
        self.critic_optimizer.step()
```

3.2.7 Update the local actor policy using the sampled policy gradient

$\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i(\nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu}\mu(s|\theta^\mu))|_{s_i}$
pytorch's autograd function means we really just need to calculate the mean Q value using the actions from the local actor. We invert the result so that the Adam optimizer will search for the minimum.

```
        # ---------------------------- update actor ---------------------------- #
        # Compute actor loss
        actions_pred = self.actor_local(states)
        actor_loss = -self.critic_local(states, actions_pred).mean()
        # Minimize the loss
        self.actor_optimizer.zero_grad()
        actor_loss.backward()
        torch.nn.utils.clip_grad_norm_(self.actor_local.parameters(), 1)
        self.actor_optimizer.step()
```

3.2.8 Update the target networks

$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$
$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$
The target network update is performed in the soft_update method

```
        # ----------------------- update target networks ----------------------- #
        self.soft_update(self.critic_local, self.critic_target, TAU)
        self.soft_update(self.actor_local, self.actor_target, TAU)

    def soft_update(self, local_model, target_model, tau):
        """Soft update model parameters.
        _target = *_local + (1 - )*_target

        Params
        ======
            local_model: PyTorch model (weights will be copied from)
            target_model: PyTorch model (weights will be copied to)
            tau (float): interpolation parameter
```

```
"""
for target_param, local_param in zip(target_model.parameters(), local_model.parameters
    target_param.data.copy_(tau*local_param.data + (1.0-tau)*target_param.data)
```

4 Hyperparameter Selection

We run scenarios with and without batch normalization, with uniform versus normally distributed noise, with multiple batch sizes, and multiple buffer sizes. Below we show the combined results varying only one of the parameters at a time to illustrate parameters most likely to be successful.

[10]:
```
import seaborn as sns
import matplotlib.pyplot as plt
import pickle
import pandas as pd
import numpy as np
sns.set()
```

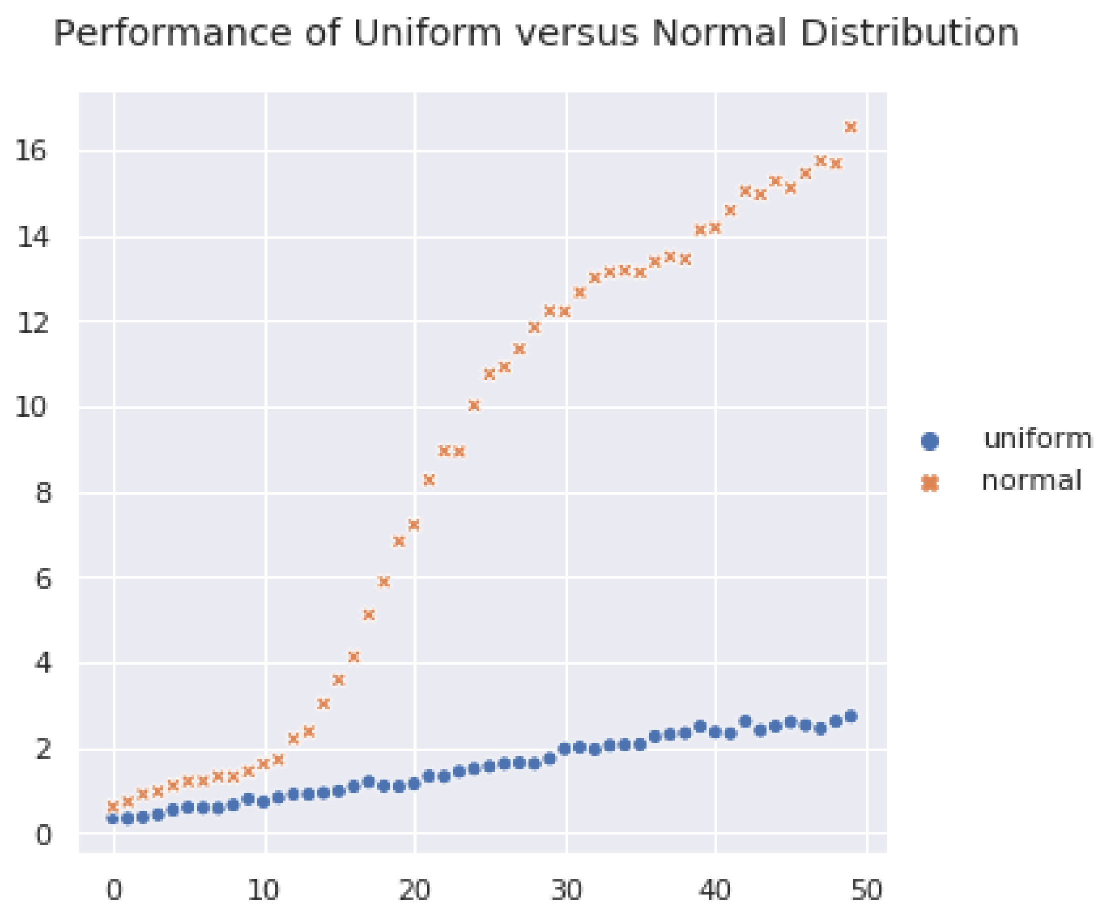
[11]:
```
all_scores = []
for batch_size in [64,128,256]:
    for buffer_size in [100000,1000000]:
        for use_batch_norm in ["False", "True"]:
            for random_fn in ["uniform", "normal"]:
                fname =␣
 ↪'time_and_scores_{batch_size}_{buffer_size}_{use_batch_norm}_{random_fn}.
 ↪pkl'.format(
                    batch_size=batch_size, buffer_size=buffer_size,␣
 ↪use_batch_norm=use_batch_norm, random_fn=random_fn)
                time_and_scores = pickle.load(open(fname, 'rb'))
                epoch = 0
                for s in time_and_scores[1]:
                    all_scores.append((fname, batch_size, buffer_size,␣
 ↪use_batch_norm, random_fn, epoch, s))
                    epoch += 1
df=pd.DataFrame(data=all_scores)
df.columns=['fname', 'batch_size', 'buffer_size', 'use_batch_norm',␣
 ↪'random_fn', 'epoch', 'score']
```

[12]:
```
def display_conditional_performance(df, conditions, labels, title=""):
    my_df_scores = {}
    for i in range(len(labels)):
        cond_field, cond_value = conditions[i]
        label = labels[i]
        my_df = df[df[cond_field] == cond_value]
        my_df_scores[label] = []
        for epoch in range(50):
            my_df_scores[label].append(np.mean(my_df.score[my_df.epoch ==␣
 ↪epoch]))
```

```
    g=sns.relplot(data=pd.DataFrame(my_df_scores))
    g.fig.suptitle(title)
    g.fig.subplots_adjust(top=.9)
```

Below we illustrate the performance difference between a uniform and normal distribution over 50 episodes. It is clear that the normal distribution learns much faster.

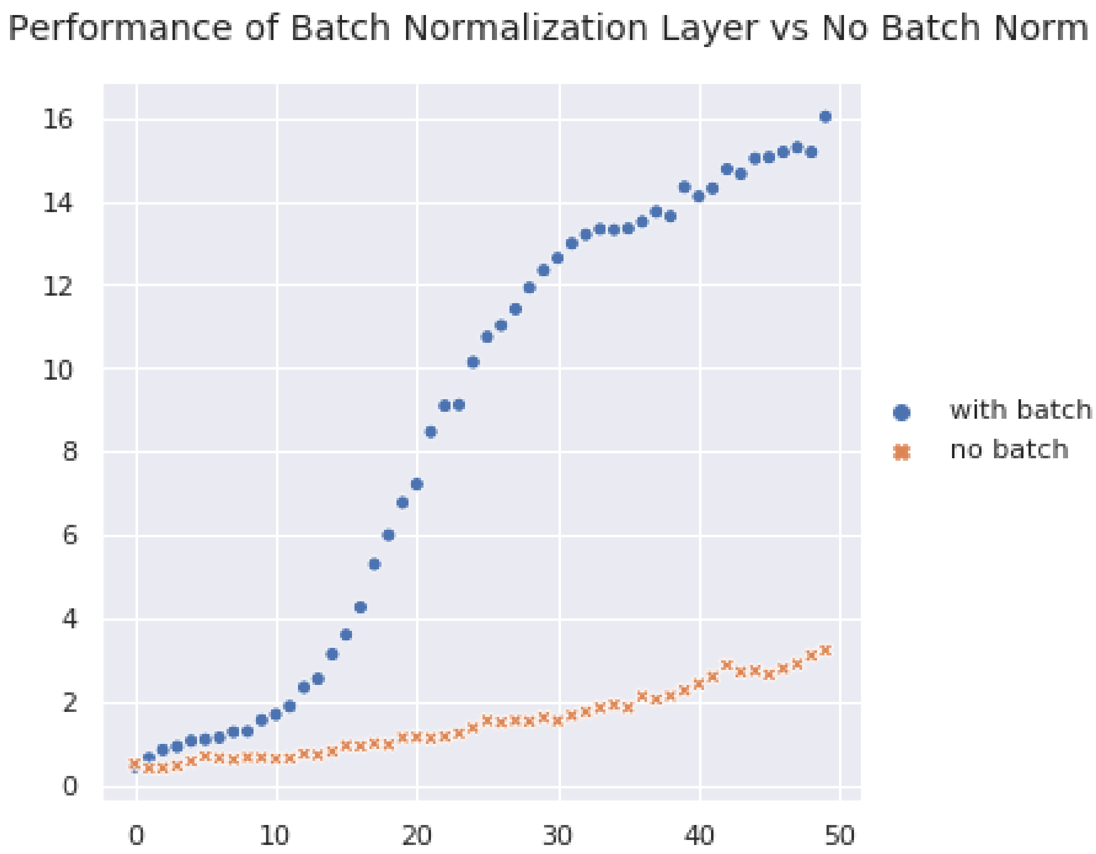
[13]:
```
display_conditional_performance(df,
                                                                ␣
 ↪[['random_fn','uniform'],['random_fn','normal']],
                                 ['uniform','normal'],
                                 "Performance of Uniform versus Normal␣
 ↪Distribution")
```



Next we look at the impact of the batch normalization layer in the actor and critic networks. The performance has a simliar profile to the normal vs uniform plot

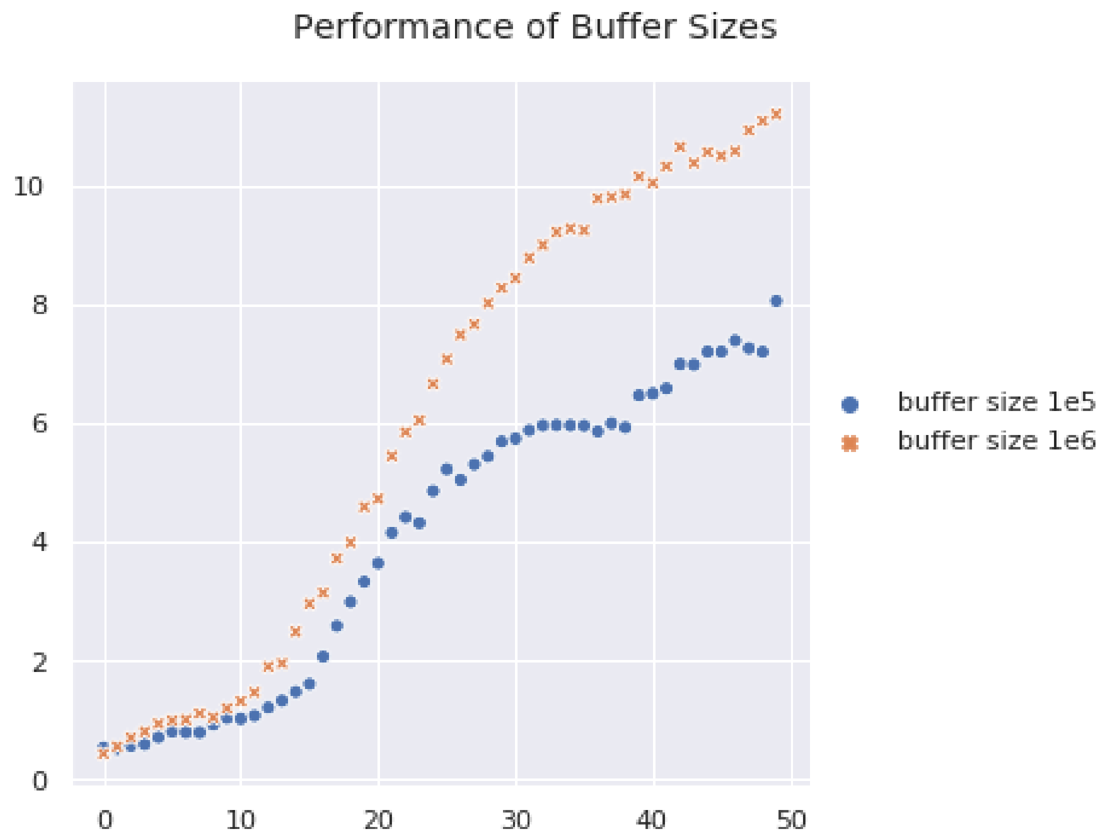
[14]:
```
display_conditional_performance(df,
                                 [['use_batch_norm','True'], ['use_batch_norm',␣
 ↪'False']],
                                 ['with batch','no batch'],
```

```
                                    "Performance of Batch Normalization Layer vs No␣
↪Batch Norm")
```



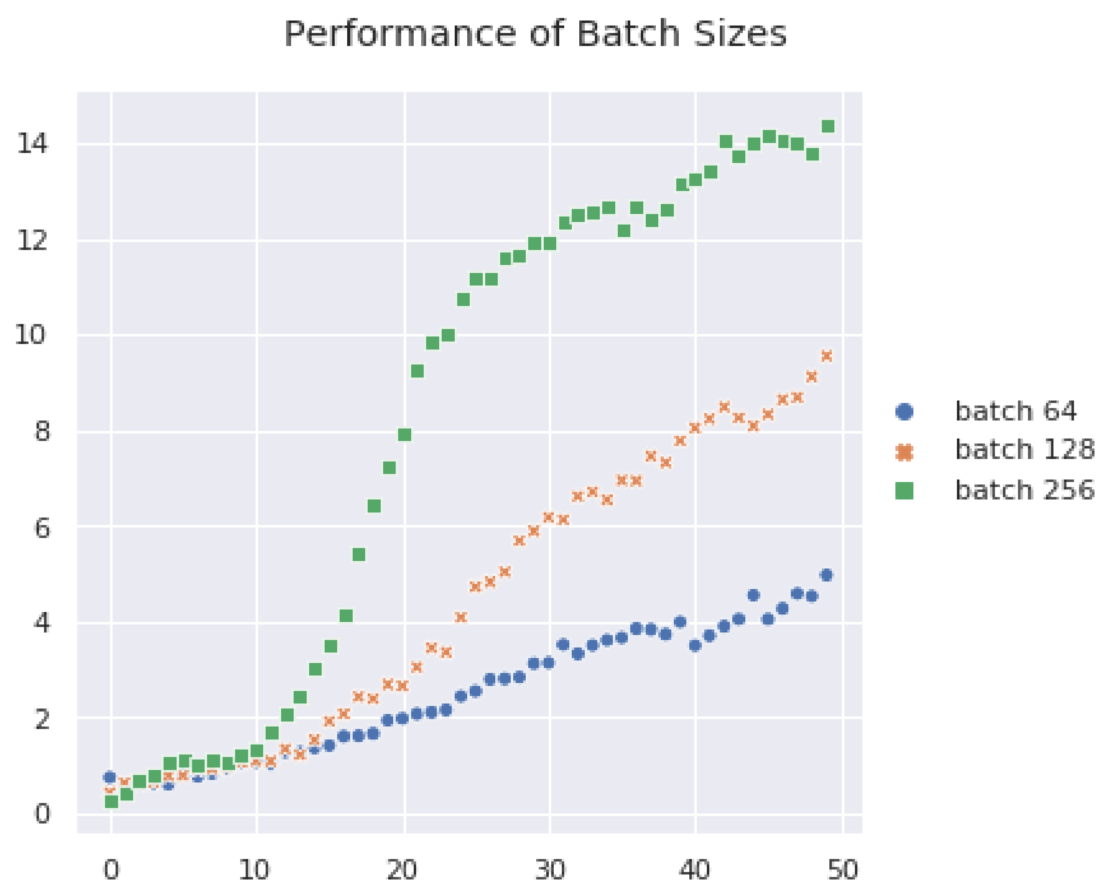
The buffer size does not show as dramatic a difference in the performance though the larger buffer does seem to improve performance.

[15]:
```
display_conditional_performance(df,
                                    ␣
↪[['buffer_size',100000],['buffer_size',1000000]],
                                     ['buffer size 1e5','buffer size 1e6'],
                                    "Performance of Buffer Sizes")
```



Finally, we look at the impact of three different mini-batch sizes of 64 versus 128 versus 256 samples. The larger mini-batch does perform better but all batch sizes seem to learn.

[16]:
```
display_conditional_performance(df,
                                [['batch_size',64],['batch_size',128],
                                 ['batch_size',256]],
                                ['batch 64','batch 128','batch 256'],
                               "Performance of Batch Sizes")
```



5 Results

For the official scoring we used batch normalization, normal distribution, batch size of 128, and buffer size of 1e6.

[18]:
```
time_and_scores = pickle.
  ↪load(open('time_and_scores_128_1000000_True_normal_official.pkl', 'rb'))
scores = time_and_scores[1]
d = {'raw scores':scores, 'smoothed x100 scores':[]}
d['smoothed x100 scores'] = [np.mean(scores[max(i-100,0):i+1]) for i in␣
  ↪range(len(scores))]
pdf = pd.DataFrame(d)
ax=sns.relplot(data=pdf,kind='line')
```

[19]:
```
print("Smoothed Score exceeded 30 at {}".format(pdf.index[pdf['smoothed x100␣
↪scores'] > 30][0]))
```

```
Smoothed Score exceeded 30 at 142
```

The average score over 100 samples exceeds the threshold of 30 in episode 142. The plot of the raw averages shows that the variance isn't too large.

[21]:
```
df[df.use_batch_norm=="False"]
```

[21]:
```
                                    fname  batch_size  buffer_size  \
0     time_and_scores_64_100000_False_uniform.pkl          64       100000
1     time_and_scores_64_100000_False_uniform.pkl          64       100000
2     time_and_scores_64_100000_False_uniform.pkl          64       100000
3     time_and_scores_64_100000_False_uniform.pkl          64       100000
4     time_and_scores_64_100000_False_uniform.pkl          64       100000
...                                           ...         ...          ...
1095  time_and_scores_256_1000000_False_normal.pkl        256      1000000
1096  time_and_scores_256_1000000_False_normal.pkl        256      1000000
1097  time_and_scores_256_1000000_False_normal.pkl        256      1000000
1098  time_and_scores_256_1000000_False_normal.pkl        256      1000000
1099  time_and_scores_256_1000000_False_normal.pkl        256      1000000

     use_batch_norm random_fn  epoch     score
0             False   uniform      0  0.806000
```

```
1            False   uniform     1   0.614000
2            False   uniform     2   0.631000
3            False   uniform     3   0.616000
4            False   uniform     4   0.366000
...            ...       ...   ...        ...
1095         False    normal    45  22.795499
1096         False    normal    46  24.315499
1097         False    normal    47  26.295999
1098         False    normal    48  27.423999
1099         False    normal    49  28.884499

[600 rows x 7 columns]
```

[ ]:

```

```