Lab 7: DQN I (NIPS 2013)

Reinforcement Learning with TensorFlow & OpenAI Gym
Sung Kim <hunkim+ml@gmail.com>
Code review acknowledgement

• Donghyun Kwak, J-min Cho, Keon Kim and Hyuck Kang

• Reference implementations
  - https://github.com/devsisters/DQN-tensorflow

• Feel free to report bugs/improvements
  - hunkim+ml@gmail.com
DQN

Human-level control through deep reinforcement learning, Nature
http://www.nature.com/nature/journal/v518/n7540/full/nature14236.html
Q-values for all possible actions in a given state with only a single forward pass through the network.

In which there is a separate output unit for each possible action, and only the state representation is of architecture is that a separate forward pass is required to compute the Q-value of each action, to the neural network by some previous approaches [20, 12]. The main drawback of this type action pairs to scalar estimates of their Q-value, the history and the action have been used as inputs There are several possible ways of parameterizing preprocessing to the last stage is only required because we use the GPU implementation of 2D convolutions from [11], which cropping an input dimensionality. The raw frames are preprocessed by first converting their RGB representation can be computationally demanding, so we apply a basic preprocessing step aimed at reducing the variance of the updates. Third, when learning on-policy the current parameters determine the next data sample that the parameters are trained on. For example, if the maximizing action is to move left then the training samples will be dominated by samples from the left-hand side; if the maximizing action then switches to the right then the training distribution will also switch. It is easy to see how even diverge catastrophically [25]. By using experience replay the behavior distribution is averaged over many of its previous states, smoothing out learning and avoiding oscillations or divergence in

4.1 Preprocessing and Model Architecture

importance to all transitions in the replay memory. A more sophisticated sampling strategy might recent transitions due to the finite memory size since the memory buffer does not differentiate important transitions and always overwrites with uniformly at random from DQN 2013

end for

end for

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights

for episode = 1, M do
    Initialise sequence \( s_1 = \{x_1\} \) and preprocessed sequenced \( \phi_1 = \phi(s_1) \)
    for \( t = 1, T \) do
        With probability \( \epsilon \) select a random action \( a_t \), otherwise select \( a_t = \max_a Q^*(\phi(s_t), a; \theta) \)
        Execute action \( a_t \) in emulator and observe reward \( r_t \) and image \( x_{t+1} \)
        Set \( s_{t+1} = s_t, a_t, x_{t+1} \) and preprocess \( \phi_{t+1} = \phi(s_{t+1}) \)
        Store transition \( (\phi_t, a_t, r_t, \phi_{t+1}) \) in \( D \)
        Sample random minibatch of transitions \( (\phi_j, a_j, r_j, \phi_{j+1}) \) from \( D \)
        Set \( y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases} \)
        Perform a gradient descent step on \( (y_j - Q(\phi_j, a_j; \theta))^2 \) according to equation 3

end for

end for

Algorithm 1 Deep Q-learning with Experience Replay

\[
\text{x} = \text{np.reshape(s, [1, input_size])}
\]

\[
\text{return sess.run(self._Qpred, feed_dict={self._X: x})}
\]
Algorithm 1: Deep Q-learning with Experience Replay

1. Initialize replay memory $D$ to capacity $N$
2. Initialize action-value function $Q$ with random weights
3. for episode = 1, $M$ do
   4. Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
   5. for $t = 1, T$ do
      6. With probability $\epsilon$ select a random action $a_t$ otherwise select $a_t = \text{max}_a Q^*(\phi(s_t), a; \theta)$
      7. Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
      8. Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
      9. Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $D$
     10. Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $D$
     11. Set $y_j = \begin{cases} r_j, & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \text{max}_{a'} Q(\phi_{j+1}, a'; \theta), & \text{for non-terminal } \phi_{j+1} \end{cases}$
    12. Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
8. end for
9. end for

# terminal?
if done:
  $Q[0, \text{action}] = \text{reward}$
else:
  # Obtain the $Q'$ values by feeding the new state through our network
  $Q[0, \text{action}] = \text{reward} + \text{dis} \times \text{np.max(DQN.predict(next_state))}$

if np.random.rand(1) < $\epsilon$:
    action = env.action_space.sample()
else:
    # Choose an action by greedily from the Q-network
    action = np.argmax(mainDQN.predict(state))

# Save the experience to our buffer
replay_buffer.append((state, action, reward, next_state, done))

minibatch = random.sample(replay_buffer, 10)
loss, _ = simple_replay_train(mainDQN, minibatch)

Human-level control through deep reinforcement learning, Nature
http://www.nature.com/nature/journal/v518/n7540/full/nature14236.html
DQN’s three solutions

1. Go deep
2. Capture and replay
   • Correlations between samples
3. Separate networks
   • Non-stationary targets

Tutorial: Deep Reinforcement Learning, David Silver, Google DeepMind
1. Go deep (class)
2. Replay memory

```python
# store the previous observations in replay memory
replay_buffer = deque()

# Save the experience to our buffer
replay_buffer.append((state, action, reward, next_state, done))
if len(replay_buffer) > REPLAY_MEMORY:
    replay_buffer.popleft()

if episode % 10 == 1:  # train every 10 episodes
    # Get a random batch of experiences.
    for _ in range(50):
        # Minibatch works better
        minibatch = random.sample(replay_buffer, 10)
        loss, _ = simple_replay_train(mainDQN, minibatch)
```

https://github.com/awjuliani/DeepRL-Agents

\[
s_1, a_1, r_2, s_2
\]
\[
s_2, a_2, r_3, s_3
\]
\[
s_3, a_3, r_4, s_4
\]
\[
... 
\]
\[
s_t, a_t, r_{t+1}, s_{t+1}
\]
2. Train from replay memory

```python
def simple_replay_train(DQN, train_batch):
    x_stack = np.empty((0, DQN.input_size)).reshape(0, DQN.input_size)
    y_stack = np.empty((0, DQN.output_size)).reshape(0, DQN.output_size)

    # Get stored information from the buffer
    for state, action, reward, next_state, done in train_batch:
        Q = DQN.predict(state)

        # terminal?
        if done:
            Q[0, action] = reward
        else:
            # Obtain the Q' values by feeding the new state through our network
            Q[0, action] = reward + dis * np.max(DQN.predict(next_state))

        y_stack = np.vstack([y_stack, Q])
        x_stack = np.vstack([x_stack, state])

        # Train our network using target and predicted Q values on each episode
    return DQN.update(x_stack, y_stack)
```

Sample random minibatch of transitions \((\phi_j, a_j, r_j, \phi_{j+1})\) from \(D\)
Set \(y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\
 r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}\)
Perform a gradient descent step on \((y_j - Q(\phi_j, a_j; \theta))^2\) according to

https://github.com/awjuliani/DeepRL-Agents
np.vstack

In [1]: ```
import numpy as np
```  

In [2]: ```
a = np.arange(5)
b = np.arange(5,10)
c = np.arange(10,15)
print(a)
print(b)
```

```
[0 1 2 3 4]
[5 6 7 8 9]
```  

In [3]: ```
x = np.vstack([a,b])
print(x)
```

```
[[0 1 2 3 4]
 [5 6 7 8 9]]
```  

In [4]: ```
x = np.vstack([x,c])
print(x)
```

```
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
[10 11 12 13 14]]
```
2. Train from replay memory

```python
def simple_replay_train(DQN, train_batch):
x_stack = np.empty(0).reshape(0, DQN.input_size)
y_stack = np.empty(0).reshape(0, DQN.output_size)

# Get stored information from the buffer
for state, action, reward, next_state, done in train_batch:
    Q = DQN.predict(state)

    # terminal?
    if done:
        Q[0, action] = reward
    else:
        # Obtain the Q' values by feeding the new state through our network
        Q[0, action] = reward + dis * np.max(DQN.predict(next_state))

    y_stack = np.vstack([y_stack, Q])
x_stack = np.vstack([x_stack, state])

# Train our network using target and predicted Q values on each episode
return DQN.update(x_stack, y_stack)
```

Sample random minibatch of transitions \((\phi_j, a_j, r_j, \phi_{j+1})\) from \(\mathcal{D}\)

Set \(y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}\)

Perform a gradient descent step on \((y_j - Q(\phi_j, a_j; \theta))^2\) according to

https://github.com/awjuliani/DeepRL-Agents
Recap

1. Net - Build - init

2. Env

    \[ a = \frac{?}{?} \]

    \[ s = \text{env-step}(a) \]

    buff

    \[ \text{random-sample} \]
Code1: setup

```python
import numpy as np
import tensorflow as tf
import random
import dqn
from collections import deque

import gym
env = gym.make('CartPole-v0')

# Constants defining our neural network
input_size = env.observation_space.shape[0]
output_size = env.action_space.n

dis = 0.9
REPLAY_MEMORY = 50000
```

https://github.com/awjuliani/DeepRL-Agents
class DQN:
    
    def __init__(self, session, input_size, output_size, name="main"): 
        self.session = session 
        self.input_size = input_size 
        self.output_size = output_size 
        self.net_name = name 
        self._build_network()

    def _build_network(self, h_size=10, l_rate=1e-1):
        with tf.variable_scope(self.net_name):
            self.X = tf.placeholder(
                tf.float32, [None, self.input_size], name="input_x")

            # First layer of weights
            W1 = tf.get_variable("W1", shape=[self.input_size, h_size],
                                 initializer=tf.contrib.layers.xavier_initializer())
            layer1 = tf.nn.tanh(tf.matmul(self.X, W1))

            # Second layer of weights
            W2 = tf.get_variable("W2", shape=[h_size, self.output_size],
                                 initializer=tf.contrib.layers.xavier_initializer())

            # Q prediction
            self.Qpred = tf.matmul(layer1, W2)

            # We need to define the parts of the network needed for learning a
            # policy
            self.Y = tf.placeholder(
                shape=[None, self.output_size], dtype=tf.float32)

            # Loss function
            self._loss = tf.reduce_mean(tf.square(self.Y - self.Qpred))
            # Learning
            self._train = tf.train.AdamOptimizer(lr_rate).minimize(self._loss)

    def predict(self, state):
        x = np.reshape(state, [1, self.input_size])
        return self.session.run(self.Qpred, feed_dict={self.X: x})

    def update(self, x_stack, y_stack):
        return self.session.run([self._loss, self._train], feed_dict={
            self.X: x_stack, self.Y: y_stack})

https://github.com/awjuliani/DeepRL-Agents
```python
def simple_replay_train(DQN, train_batch):
    x_stack = np.empty(0).reshape(0, DQN.input_size)
    y_stack = np.empty(0).reshape(0, DQN.output_size)

    # Get stored information from the buffer
    for state, action, reward, next_state, done in train_batch:
        Q = DQN.predict(state)

        # terminal?
        if done:
            Q[0, action] = reward
        else:
            # Obtain the Q' values by feeding the new state through our network
            Q[0, action] = reward + dis * np.max(DQN.predict(next_state))

        y_stack = np.vstack([y_stack, Q])
        x_stack = np.vstack([x_stack, state])

    # Train our network using target and predicted Q values on each episode
    return DQN.update(x_stack, y_stack)
```

https://github.com/awjuliani/DeepRL-Agents
Code 4: bot play

```python
def bot_play(mainDQN):
    # See our trained network in action
    s = env.reset()
    reward_sum = 0
    while True:
        env.render()
        a = np.argmax(mainDQN.predict(s))
        s, reward, done, _ = env.step(a)
        reward_sum += reward
        if done:
            print("Total score: {}".format(reward_sum))
            break
```

https://github.com/awjuliani/DeepRL-Agents
```python
def main():
    max_episodes = 5000
    replay_buffer = deque()

    with tf.Session() as sess:
        mainDQN = dqn.DQN(sess, input_size, output_size)
        tf.global_variables_initializer().run()

        for episode in range(max_episodes):
            e = 1. / ((episode / 10) + 1)
            done = False
            step_count = 0

            state = env.reset()

            while not done:
                if np.random.rand(1) < e:
                    action = env.action_space.sample()
                else:
                    # Choose an action by greedily from the Q-network.
                    action = np.argmax(mainDQN.predict(state))

                # Get new state and reward from environment
                next_state, reward, done, _ = env.step(action)

                if done:
                    reward = -100

                # Save the experience to our buffer
                replay_buffer.append((state, action, reward, next_state, done))
                if len(replay_buffer) > REPLAY_MEMORY:
                    replay_buffer.popleft()

                state = next_state
                step_count += 1
                if step_count > 10000:
                    break

            if episode % 10 == 0:
                # Train every 10 episodes
                minibatch = random.sample(replay_buffer, 10)
                loss, _ = simple_replay_train(mainDQN, minibatch)

            print("Episode: {} steps: {}".format(episode, step_count))
            if step_count > 10000:
                pass
                break

        print("Episode: {} steps: {}".format(episode, step_count))
        if step_count > 10000:
            pass
            break
```

https://github.com/awjuliani/DeepRL-Agents
How to read results

| Episode: 510  | 5 steps: 105 |
| Episode: 511  | 6 steps: 113 |
| Episode: 512  | 7 steps: 46  |
| Episode: 513  | 8 steps: 55  |
| Episode: 514  | 9 steps: 62  |
| Episode: 515  | 0 steps: 49  |
| Episode: 516  | 1 steps: 40  |
| Episode: 517  | 2 steps: 39  |
| Episode: 518  | 3 steps: 126 |
| Episode: 519  | 4 steps: 600 |
| Episode: 520  | 5 steps: 228 |
| Episode: 521  | 6 steps: 223 |
| Episode: 522  | 7 steps: 128 |
| Episode: 523  | 8 steps: 143 |
| Episode: 524  | 9 steps: 424 |
| Episode: 525  | 0 steps: 62  |
| Episode: 526  | 1 steps: 267 |
| Episode: 527  | 299 steps: 96 |
| Episode: 528  | 300 steps: 51 |
| Episode: 529  | 422 steps: 1689 |
| Episode: 530  | 423 steps: 334 |
# How to read results

<table>
<thead>
<tr>
<th>Episode</th>
<th>Steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>510</td>
<td>25</td>
</tr>
<tr>
<td>511</td>
<td>44</td>
</tr>
<tr>
<td>512</td>
<td>34</td>
</tr>
<tr>
<td>513</td>
<td>18</td>
</tr>
<tr>
<td>514</td>
<td>16</td>
</tr>
<tr>
<td>515</td>
<td>29</td>
</tr>
<tr>
<td>516</td>
<td>32</td>
</tr>
<tr>
<td>517</td>
<td>45</td>
</tr>
<tr>
<td>518</td>
<td>20</td>
</tr>
<tr>
<td>519</td>
<td>47</td>
</tr>
<tr>
<td>520</td>
<td>19</td>
</tr>
<tr>
<td>521</td>
<td>566</td>
</tr>
<tr>
<td>522</td>
<td>595</td>
</tr>
<tr>
<td>523</td>
<td>735</td>
</tr>
<tr>
<td>524</td>
<td>413</td>
</tr>
<tr>
<td>525</td>
<td>653</td>
</tr>
<tr>
<td>526</td>
<td>667</td>
</tr>
<tr>
<td>527</td>
<td>740</td>
</tr>
<tr>
<td>528</td>
<td>620</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Episode</th>
<th>Steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>283</td>
<td>55</td>
</tr>
<tr>
<td>284</td>
<td>83</td>
</tr>
<tr>
<td>285</td>
<td>57</td>
</tr>
<tr>
<td>286</td>
<td>91</td>
</tr>
<tr>
<td>287</td>
<td>53</td>
</tr>
<tr>
<td>288</td>
<td>83</td>
</tr>
<tr>
<td>289</td>
<td>87</td>
</tr>
<tr>
<td>290</td>
<td>88</td>
</tr>
<tr>
<td>291</td>
<td>106</td>
</tr>
<tr>
<td>292</td>
<td>184</td>
</tr>
<tr>
<td>293</td>
<td>118</td>
</tr>
<tr>
<td>294</td>
<td>46</td>
</tr>
<tr>
<td>295</td>
<td>168</td>
</tr>
<tr>
<td>296</td>
<td>45</td>
</tr>
<tr>
<td>297</td>
<td>62</td>
</tr>
<tr>
<td>298</td>
<td>75</td>
</tr>
<tr>
<td>299</td>
<td>96</td>
</tr>
<tr>
<td>300</td>
<td>51</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Episode</th>
<th>Steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>405</td>
<td>105</td>
</tr>
<tr>
<td>406</td>
<td>113</td>
</tr>
<tr>
<td>407</td>
<td>46</td>
</tr>
<tr>
<td>408</td>
<td>55</td>
</tr>
<tr>
<td>409</td>
<td>62</td>
</tr>
<tr>
<td>410</td>
<td>49</td>
</tr>
<tr>
<td>411</td>
<td>40</td>
</tr>
<tr>
<td>412</td>
<td>39</td>
</tr>
<tr>
<td>413</td>
<td>126</td>
</tr>
<tr>
<td>414</td>
<td>600</td>
</tr>
<tr>
<td>415</td>
<td>228</td>
</tr>
<tr>
<td>416</td>
<td>223</td>
</tr>
<tr>
<td>417</td>
<td>128</td>
</tr>
<tr>
<td>418</td>
<td>143</td>
</tr>
<tr>
<td>419</td>
<td>424</td>
</tr>
<tr>
<td>420</td>
<td>62</td>
</tr>
<tr>
<td>421</td>
<td>267</td>
</tr>
<tr>
<td>422</td>
<td>1689</td>
</tr>
<tr>
<td>423</td>
<td>334</td>
</tr>
</tbody>
</table>
Next

Lab: DQN (Nature 2015)

\[
\min_{\theta} \sum_{t=0}^{T} \left[ \hat{Q}(s_t, a_t|\theta) - (r_t + \gamma \max_{a'} \hat{Q}(s_{t+1}, a'|\bar{\theta})) \right]^2
\]