Lab 7: DQN 2 (Nature 2015)

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/improvement
  - hunkim+ml@gmail.com
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
Human-level control through deep reinforcement learning, Nature
http://www.nature.com/nature/journal/v518/n7540/full/nature14236.html
Algorithm 1: deep Q-learning with experience replay.
Initialize replay memory D to capacity N
Initialize action-value function Q with random weights \( \theta \)
Initialize target action-value function \( \hat{Q} \) with weights \( \theta^- = \theta \)
For episode = 1, M do
  Initialize sequence \( s_1 = \{x_1\} \) and preprocessed sequence \( \phi_1 = \phi(s_1) \)
  For \( t = 1, T \) do
    With probability \( \epsilon \) select a random action \( a_t \)
    otherwise select \( a_t = \arg \max_a Q(\phi(s_t), a; \theta) \)
    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 \( y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max \ Q(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases} \)
    Perform a gradient descent step on \( \mathcal{L}(\theta) = \left(y_j - Q(\phi_j, a_j; \theta)\right)^2 \) with respect to the network parameters \( \theta \)
    Every \( C \) steps reset \( \hat{Q} = Q \)
  End For
End For

Implementing Nature Paper

Human-level control through deep reinforcement learning, Nature
http://www.nature.com/nature/journal/v518/n7540/full/nature14236.html
Solution 3: separate target network

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

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

# Obtain the \(Q'\) values by feeding the new state through our network
\[
Q[\theta, \text{action}] = \text{reward} + \text{dis} \times \text{np.max(DQN.predict(next_state))}
\]

# get target from target DQN (\(Q'\))
\[
Q[\theta, \text{action}] = \text{reward} + \text{dis} \times \text{np.max(targetDQN.predict(next_state))}
\]
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("{}\_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(  
                learning_rate=l_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})
Handling two networks

```python
with tf.Session() as sess:
    mainDQN = dqn.DQN(sess, input_size, output_size, name="main")
    targetDQN = dqn.DQN(sess, input_size, output_size, name="target")
    tf.global_variables_initializer().run()

    # initial copy q_net -> target_net
    copy_ops = get_copy_var_ops(dest_scope_name="target",
                                src_scope_name="main")
    sess.run(copy_ops)

    # Every C steps reset \( \hat{Q} = Q \)

    if i % 10 == 0:  # train every 10 episode
        # copy q_net -> target_net
        sess.run(copy_ops)
```
Solution 3: copy network

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

Set \( y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases} \)

Perform a gradient descent step on \( (y_j - Q(\phi_j, a_j; \theta))^2 \) with respect to the network parameters \( \theta \)

Every \( C \) steps reset \( \hat{Q} = Q \)
Copy network (trainable variables)

Every $C$ steps reset $\hat{Q} = Q$

```python
def get_copy_var_ops(*, dest_scope_name="target", src_scope_name="main"):
    # Copy variables src_scope to dest_scope
    op_holder = []
    src_vars = tf.get_collection(
        tf.GraphKeys.TRAINABLE_VARIABLES, scope=src_scope_name)
    dest_vars = tf.get_collection(
        tf.GraphKeys.TRAINABLE_VARIABLES, scope=dest_scope_name)
    for src_var, dest_var in zip(src_vars, dest_vars):
        op_holder.append(dest_var.assign(src_var.value()))
    return op_holder

# initial copy q_net -> target_net
copy_ops = get_copy_var_ops(dest_scope_name="target",
                             src_scope_name="main")
with tf.Session() as sess:
    sess.run(copy_ops)
```
Recap

1. $\text{Net} \times 2$

2. $\text{target} = \text{main\ Net}$

3. $\alpha \Rightarrow S; t, \ldots$

\[ D \]

$\text{train}$

$\text{f} = \text{target}$

$\text{main\ loss}$
```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(
            learning_rate=l_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
def replay_train(mainDQN, targetDQN, train_batch):
    x_stack = np.empty(0).reshape(0, input_size)
    y_stack = np.empty(0).reshape(0, output_size)

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

        # terminal?
        if done:
            Q[0, action] = reward
        else:
            # get target from target DQN (Q')
            Q[0, action] = reward + dis * np.max(targetDQN.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 mainDQN.update(x_stack, y_stack)

https://github.com/awjuliani/DeepRL-Agents
def get_copy_var_ops(*, dest_scope_name="target", src_scope_name="main"):  
    # Copy variables src_scope to dest_scope
    op_holder = []

    src_vars = tf.get_collection(
        tf.GraphKeys.TRAINABLE_VARIABLES, scope=src_scope_name)
    dest_vars = tf.get_collection(
        tf.GraphKeys.TRAINABLE_VARIABLES, scope=dest_scope_name)

    for src_var, dest_var in zip(src_vars, dest_vars):
        op_holder.append(dest_var.assign(src_var.value()))

    return op_holder

https://github.com/awjuliani/DeepRL-Agents
Code 6: bot play (same)

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

    with tf.Session() as sess:
        mainDON = dqn.DQN(sess, input_size, output_size, name="main")
        targetDON = dqn.DQN(sess, input_size, output_size, name="target")
        tf.global_variables_initializer().run()

        # Initial copy q_net -> target_net
        copy_ops = get_copy_var_ops(dest_scope_name="target",
                                     src_scope_name="main")
        sess.run(copy_ops)

        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(mainDON.predict(state))
                # Get new state and reward from environment
                next_state, reward, done, _ = env.step(action)
                if done: # Penalty
                    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: 
                    # Good enough. Let's move on
                    break

            print("Episode: {} steps: {}".format(episode, step_count))
            if step_count > 10000:
                pass
                # Break
            if episode % 10 == 0:
                # Train every 10 episodes
                minibatch = random.sample(replay_buffer, 10)
                loss, _ = replay_train(mainDON, targetDON, minibatch)
                # Loss: loss
                sess.run(copy_ops)
                bot_play(mainDON)

    if __name__ == "__main__":
        main()
```

https://github.com/awjuliani/DeepRL-Agents
DQN works reasonably well

CartPole-v0 defines "solving" as getting average reward of 195.0 over 100 consecutive trials.
DQN works reasonably well

CartPole-v0 defines "solving" as getting average reward of 195.0 over 100 consecutive trials.
DQN works reasonably well

(1) Fun
(2) Still
(3) shaky 1
(4) shaky 2

DQN on Cart Pole
state = next_state
step_count += 1
if step_count > 10000:  # Good enough. Let's move on
    break
step_history.append(step_count)
print("Episode: {} steps: {}".format(episode, step_count))

CartPole-v0 defines "solving" as getting average reward of 195.0
over 100 consecutive trials.
Exercise 1

• Hyper parameter tuning
  - Learning rate
  - Sample size
  - Decay factor

• Network structure
  - add bias
  - test tanh, sigmoid, relu, etc.
  - improve TF network to reduce sess.run() calls

• Reward redesign
  - 1,1,1,1,1,.. -100
  - 1, 0.9, 0.99, ..., 0
Exercise 2

• Simple block based car race?
  - DQN 2013?

• Rewrite it using DQN 2015 algorithm?
Exercise 3

• DQN implementations
  - https://github.com/songrotek/DQN-Atari-Tensorflow
  - https://github.com/devsisters/DQN-tensorflow

• Other games

• RMA approach
  - Run
  - Modify
  - Adapt (to your new game/problems)
Next
Policy Gradient
(better than DQN?)