Lab 7: DQN 1 (NIPS 2013)

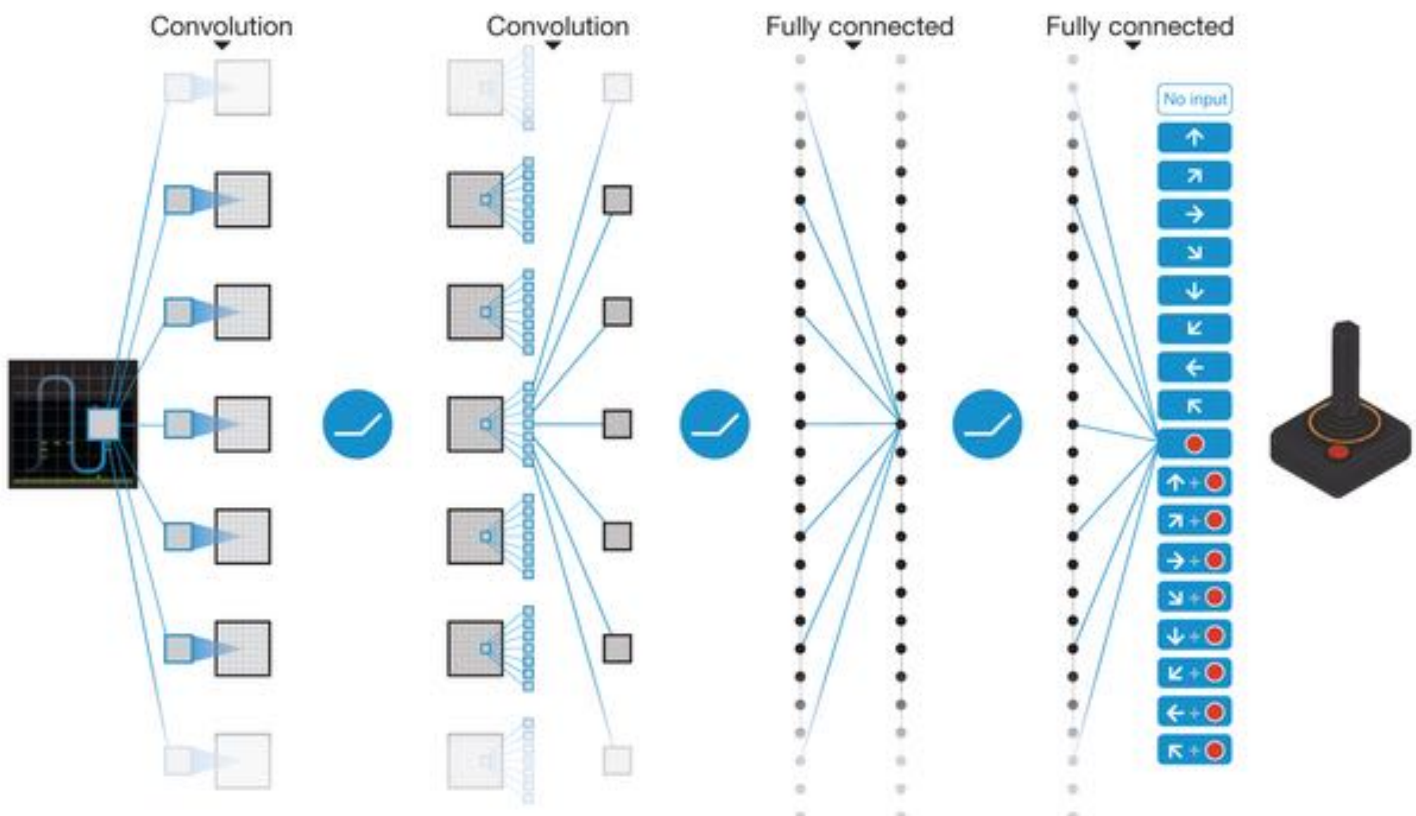
Reinforcement Learning with TensorFlow&OpenAI Gym

Sung Kim <hunkim+ml@gmail.com>

Code review acknowledgement

- Donghyun Kwak, J-min Cho, Keon Kim and Hyuck Kang
- Reference implementations
  - https://github.com/awjuliani/DeepRL-Agents
  - https://github.com/devsisters/DQN-tensorflow
  - https://github.com/dennybritz/reinforcement-learning/blob/master/DQN/dqn.py
- Feel free to report bugs/improvements
  - https://www.facebook.com/groups/TensorFlowKR/
  - hunkim+ml@gmail.com

DQN



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

DQN 2013

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ 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 ϵ 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 $\mathcal{D}$
        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 equation 3
    end for
end for

```
x = np.reshape(s, [1, input_size])
return sess.run(self._Qpred, feed_dict={self._X: x})
```

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 $\mathcal{D}$ 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 ϵ 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 $\mathcal{D}$
    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 equation 3
  end for
end for

```
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))
```

```
# 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)
```

```
# 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))
```

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)

```python
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})
```

```python
X = tf.placeholder(tf.float32, [None, input_size], name="input

# First layer of weights
W1 = tf.get_variable("W1", shape=[input_size, output_size],
                     initializer=tf.contrib.layers.xavier_initi
Qpred = tf.matmul(X, W1)
```

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

2. Replay memory

s_1, a_1, r_2, s_2
s_2, a_2, r_3, s_3
s_3, a_3, r_4, s_4
$\ldots$
$s_t, a_t, r_{t+1}, s_{t+1}$

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

1

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

2

```
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

2. Train from replay memory

```
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 t

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

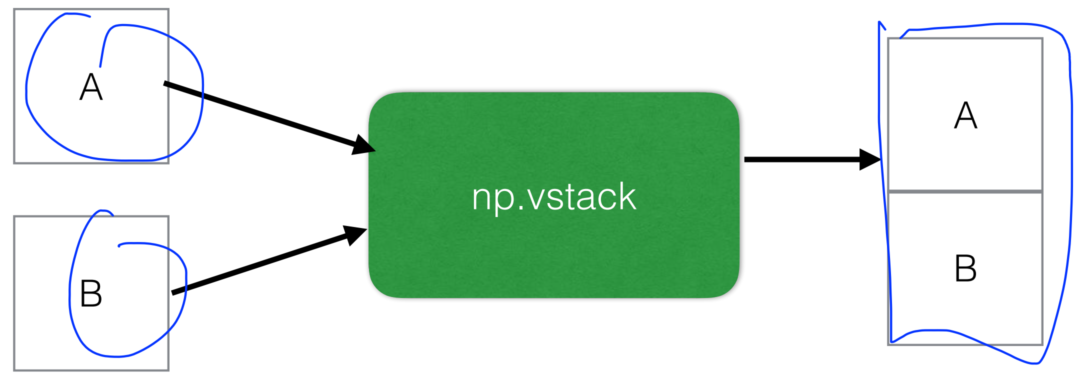
np.vstack

```
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)
```

Q_next_state

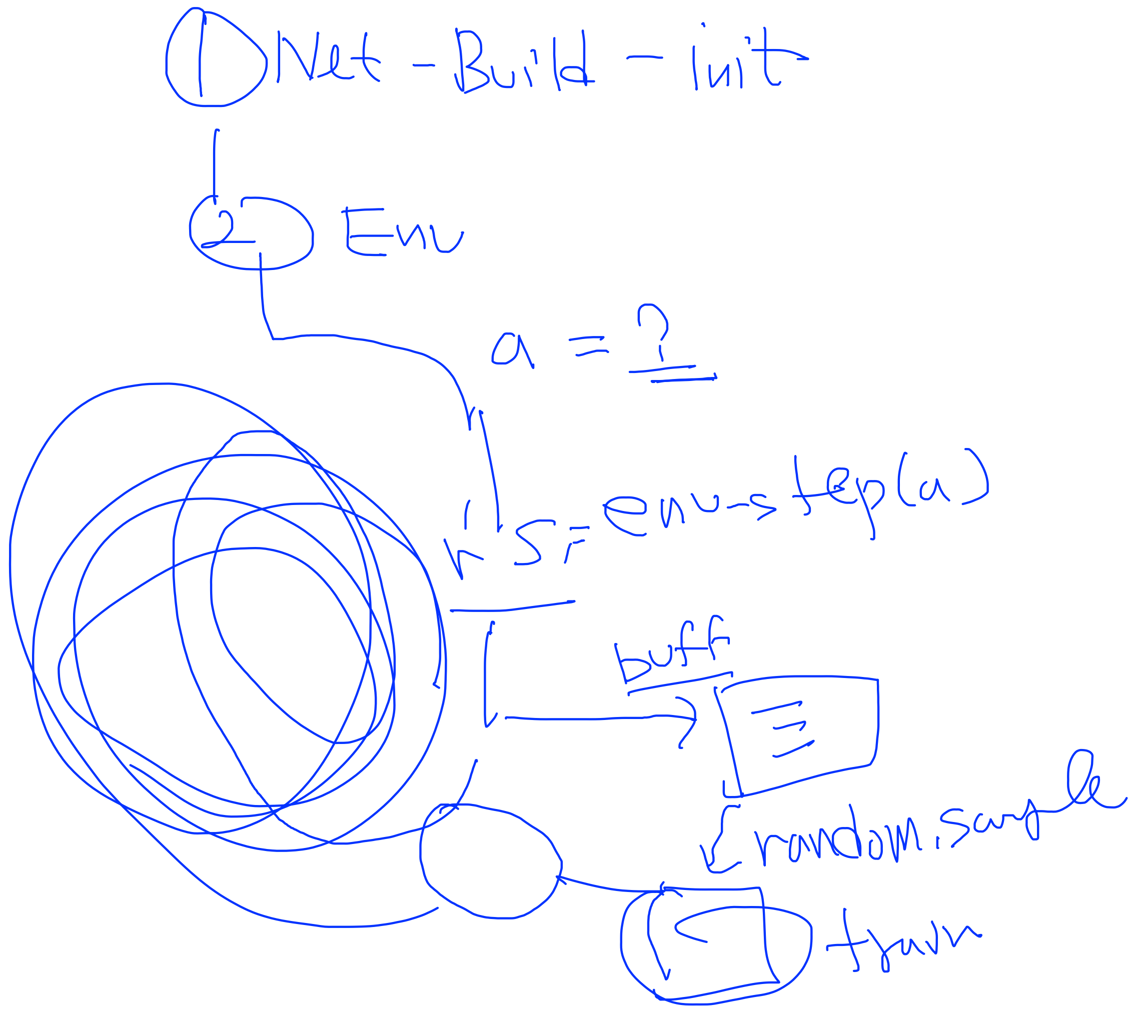
Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$

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 t

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

Recap



Code1: setup

```
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

Code 2: Network

```python
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

Code 3: Train from Replay Buffer

```
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

```
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

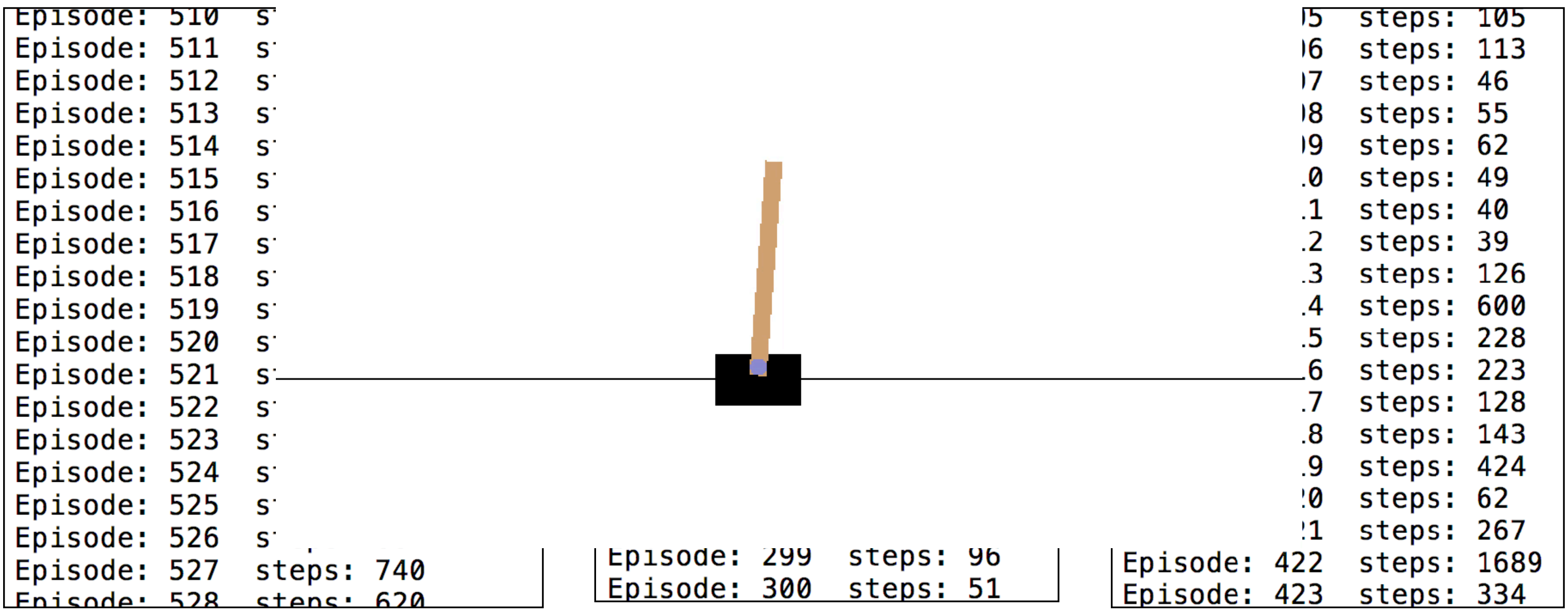
Code 6: main

```python
def main():
    max_episodes = 5000

    # store the previous observations in replay memory
    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-networ
                    action = np.argmax(mainDQN.predict(state))

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

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

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

            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)
                print("Loss: ", loss)

        bot_play(mainDQN)

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

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

How to read results



How to read results

```
Episode: 510   steps: 25
Episode: 511   steps: 44
Episode: 512   steps: 34
Episode: 513   steps: 18
Episode: 514   steps: 16
Episode: 515   steps: 29
Episode: 516   steps: 32
Episode: 517   steps: 45
Episode: 518   steps: 20
Episode: 519   steps: 47
Episode: 520   steps: 19
Episode: 521   steps: 566
Episode: 522   steps: 595
Episode: 523   steps: 735
Episode: 524   steps: 413
Episode: 525   steps: 653
Episode: 526   steps: 667
Episode: 527   steps: 740
Episode: 528   steps: 620
```

```
Episode: 283   steps: 55
Episode: 284   steps: 83
Episode: 285   steps: 57
Episode: 286   steps: 91
Episode: 287   steps: 53
Episode: 288   steps: 83
Episode: 289   steps: 87
Episode: 290   steps: 88
Episode: 291   steps: 106
Episode: 292   steps: 184
Episode: 293   steps: 118
Episode: 294   steps: 46
Episode: 295   steps: 168
Episode: 296   steps: 45
Episode: 297   steps: 62
Episode: 298   steps: 75
Episode: 299   steps: 96
Episode: 300   steps: 51
```

```
Episode: 405   steps: 105
Episode: 406   steps: 113
Episode: 407   steps: 46
Episode: 408   steps: 55
Episode: 409   steps: 62
Episode: 410   steps: 49
Episode: 411   steps: 40
Episode: 412   steps: 39
Episode: 413   steps: 126
Episode: 414   steps: 600
Episode: 415   steps: 228
Episode: 416   steps: 223
Episode: 417   steps: 128
Episode: 418   steps: 143
Episode: 419   steps: 424
Episode: 420   steps: 62
Episode: 421   steps: 267
Episode: 422   steps: 1689
Episode: 423   steps: 334
```

Next

Lab: DQN (Nature 2015)

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