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/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/improvement
  - https://www.facebook.com/groups/TensorFlowKR/
  - 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

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 $\mathcal{D}$
    $\boxed{\text{Sample random}}$ 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

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

Implementing Nature Paper

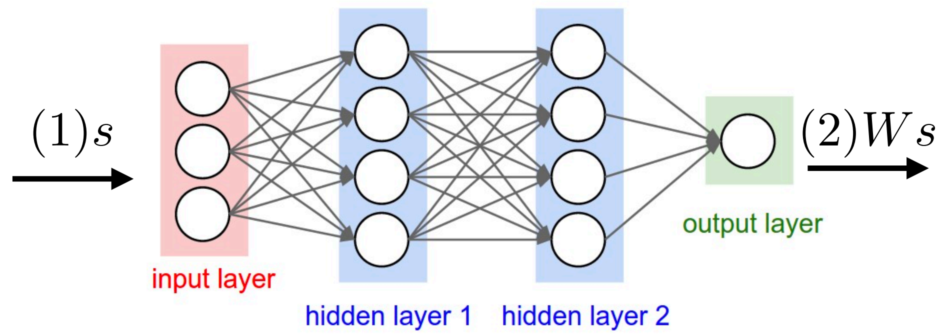
Algorithm 1: deep Q-learning with experience replay.
Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
For episode $= 1, M$ **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For $t = 1$,T **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \mathrm{argmax}_a Q(\phi(s_t), a; \theta)$
    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 $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  End For
End For

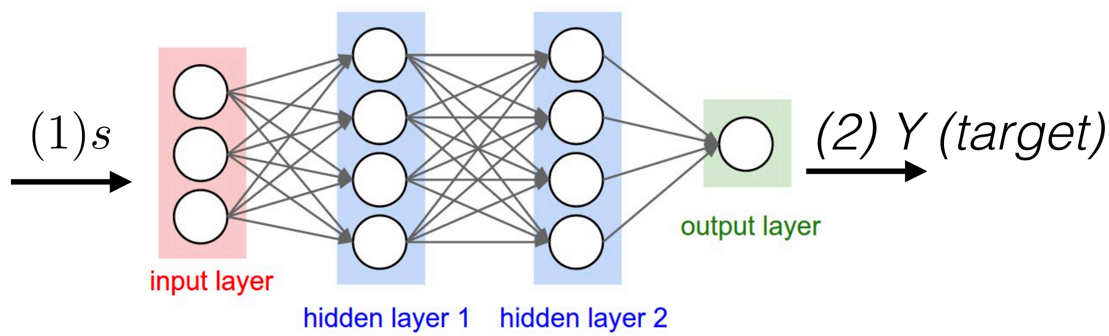
```
x = np.reshape(s, [1, input_size])
return sess.run(self._Qpred, feed_dict={self._X: x})
```

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

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[0, action] = reward + dis * np.max(DQN.predict(next_state))
```

Y

```
# get target from target DQN (Q')
Q[0, action] = reward + dis * np.max(targetDQN.predict(next_state))
```

mainDQN

Network class (same)

```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})
```

Handling two networks

dqn.py

```
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)
```

targetDQN == mainDQN

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

mainDQN

```
if i % 10 == 0:  # train every 10 episode

    # copy q_net -> target_net
    sess.run(copy_ops)
```

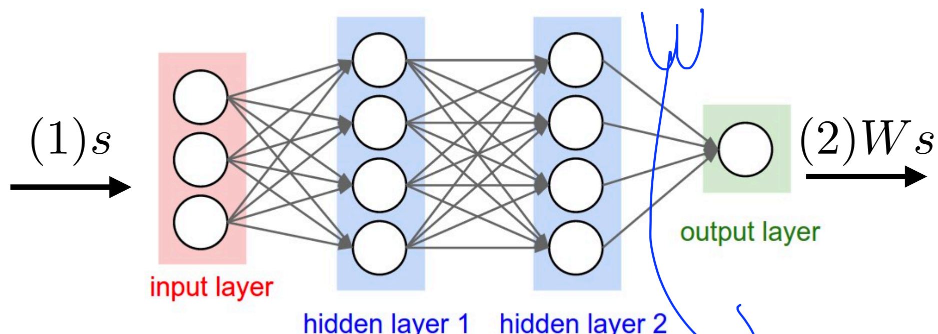
Solution 3: copy network

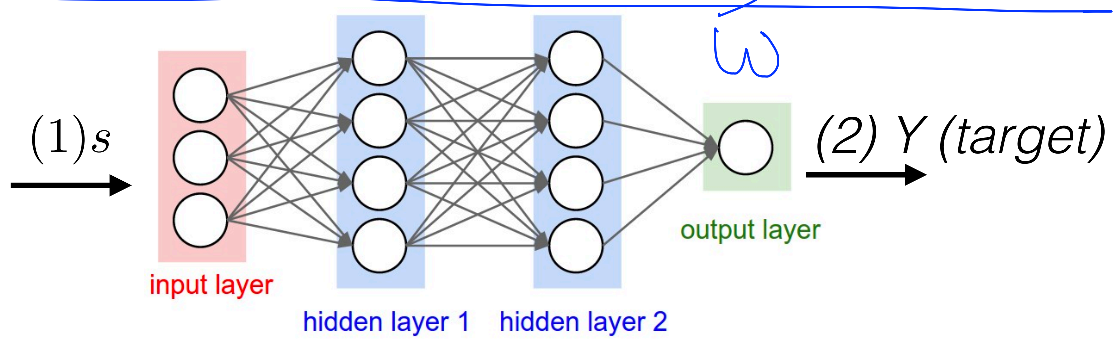
Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D

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 $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters θ

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
```

a = b

tensor

dest_var = src_var

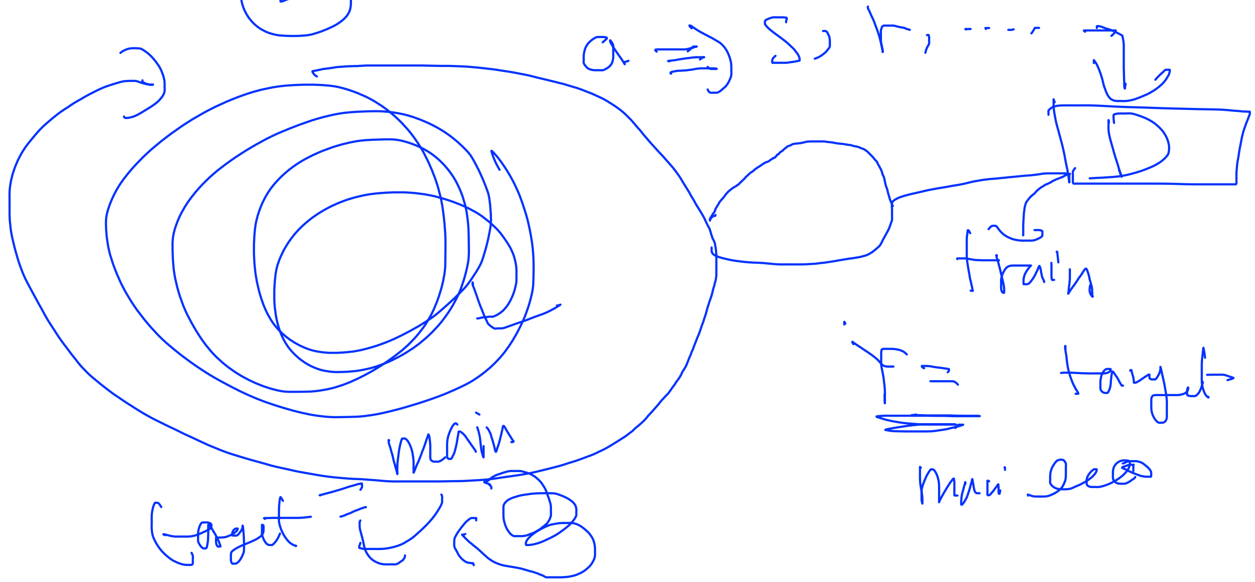
```python
# initial copy q_net -> target_net
copy_ops = get_copy_var_ops(dest_scope_name="target",
                            src_scope_name="main")
sess.run(copy_ops)
```

Recap

① Net × 2

② target = mainNet

③ Env



Code1: setup (same)

```
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 (same)

```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: replay train (targetDQN added)

```
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

Code 5: network (variable) copy

```
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

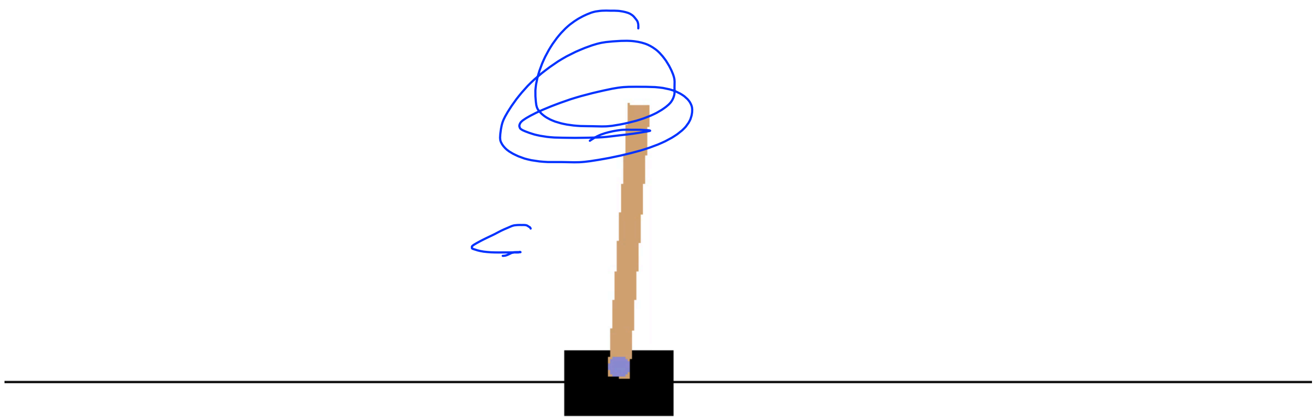
Code 7: main (network copy part added)

```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, 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)

        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:  # 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
```

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

            if episode % 10 == 1:  # train every 10 episode
                # Get a random batch of experiences.
                for _ in range(50):
                    minibatch = random.sample(replay_buffer, 10)
                    loss, _ = replay_train(mainDQN, targetDQN, minibatch)

                print("Loss: ", loss)
                # copy q_net -> target_net
                sess.run(copy_ops)

        bot_play(mainDQN)

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.

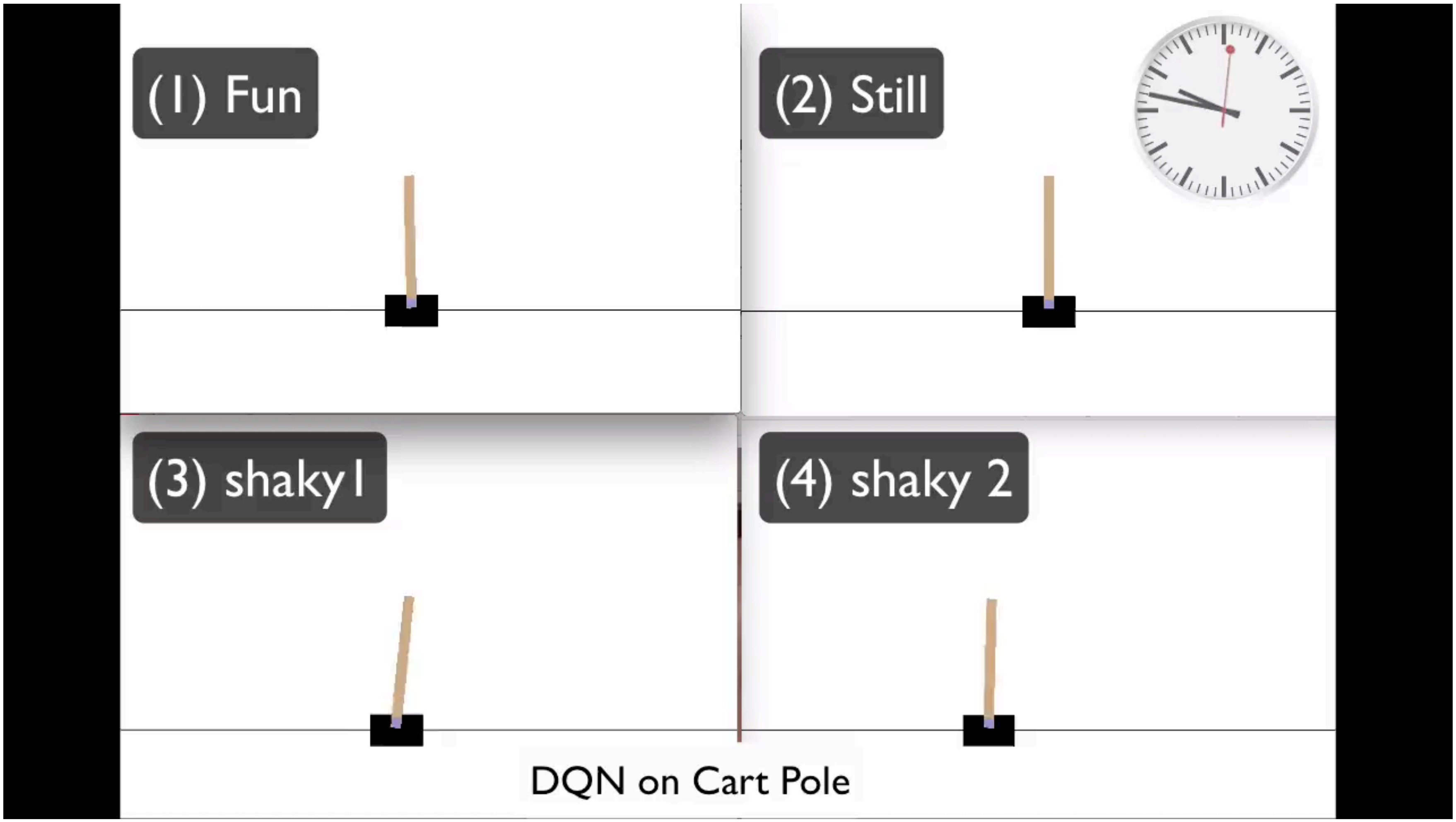
DQN works reasonably well

```
        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
```

```
Average reward for episode 4997: 10001.0. Loss: -0.0013621606631204486
Solved in 4998 episodes!
Average reward for episode 4998: 10001.0. Loss: -0.0013621606631204486
Solved in 4999 episodes!
Average reward for episode 4999: 10001.0. Loss: -0.0013621606631204486
Traceback (most recent call last):
  File "/Users/hunkim/deeplearning/qlearning/08_1_1_policy_learning_cartpole-decay.py", line 140, in <module>
    env.render()
  File "/Users/hunkim/gitWorkspace/gym/gym/core.py", line 175, in render
    return self._render(mode=mode, close=close)
  File "/Users/hunkim/gitWorkspace/gym/gym/envs/classic_control/cartpole.py", line 120, in _render
    from gym.envs.classic_control import rendering
  File "/Users/hunkim/gitWorkspace/gym/gym/envs/classic_control/rendering.py", line 23, in <module>
    from pyglet.gl import *
  File "/Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6/site-packages/pyglet/gl/__init__.py", line 236, in <module>
    import pyglet.window
  File "/Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6/site-packages/pyglet/window/__init__.py", line 1816, in <module>
    gl._create_shadow_window()
  File "/Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6/site-packages/pyglet/gl/__init__.py", line 205, in _create_shadow_window
    _shadow_window = Window(width=1, height=1, visible=False)
  File "/Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6/site-packages/pyglet/window/__init__.py", line 496, in __init__
    screen = display.get_default_screen()
  File "/Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6/site-packages/pyglet/canvas/base.py", line 74, in get_default_screen
    return self.get_screens()[0]
IndexError: list index out of range

Process finished with exit c
```

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

DQN works reasonably well



DQN
2013 VS 2015

```
Episode: 4142  steps: 194
Episode: 4143  steps: 163
Episode: 4144  steps: 229
Episode: 4145  steps: 91
Episode: 4146  steps: 140
Episode: 4147  steps: 129
Episode: 4148  steps: 139
Episode: 4149  steps: 135
Episode: 4150  steps: 279
Episode: 4151  steps: 163
Loss:  3.01422
Episode: 4152  steps: 17
Episode: 4153  steps: 11
Episode: 4154  steps: 17
Episode: 4155  steps: 11
Episode: 4156  steps: 21
Episode: 4157  steps: 13
Episode: 4158  steps: 18
Episode: 4159  steps: 35
Episode: 4160  steps: 23
Episode: 4161  steps: 12
Loss:  0.876567
Episode: 4162  steps: 10001
Episode: 4163  steps: 10001
Episode: 4164  steps: 10001
Episode: 4165  steps: 10001
Episode: 4166  steps: 10001
Episode: 4167  steps: 10001
Episode: 4168  steps: 10001
Episode: 4169  steps: 10001
Episode: 4170  steps: 10001
Episode: 4171  steps: 10001
Loss:  2.94949
Episode: 4172  steps: 10001
Episode: 4173  steps: 10001
Episode: 4174  steps: 10001
Episode: 4175  steps: 10001
Episode: 4176  steps: 10001
Episode: 4177  steps: 10001
Episode: 4178  steps: 10001
Episode: 4179  steps: 10001
Episode: 4180  steps: 10001
Episode: 4181  steps: 10001
Loss:  2.70244
Episode: 4182  steps: 10001
Episode: 4183  steps: 10001
Episode: 4184  steps: 10001
Episode: 4185  steps: 10001
Episode: 4186  steps: 10001
Episode: 4187  steps: 10001
Episode: 4188  steps: 10001
Episode: 4189  steps: 10001
Episode: 4190  steps: 10001
```

```python
        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.

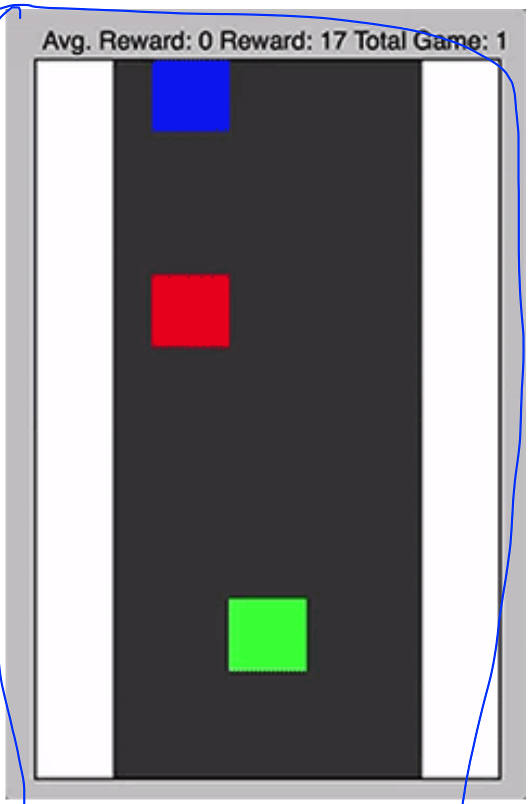
```
Loss:  0.97495
Episode: 322  steps: 309
Episode: 323  steps: 5392
Episode: 324  steps: 1180
Episode: 325  steps: 2176
Episode: 326  steps: 336
Episode: 327  steps: 2047
Episode: 328  steps: 341
Episode: 329  steps: 243
Episode: 330  steps: 919
Episode: 331  steps: 275
Loss:  1.42303
Episode: 332  steps: 10001
Episode: 333  steps: 10001
Episode: 334  steps: 10001
Episode: 335  steps: 10001
Episode: 336  steps: 10001
Episode: 337  steps: 10001
Episode: 338  steps: 9807
Episode: 339  steps: 10001
Episode: 340  steps: 10001
Episode: 341  steps: 10001
Loss:  1.05019
Episode: 342  steps: 10001
Episode: 343  steps: 10001
Episode: 344  steps: 10001
Episode: 345  steps: 10001
Episode: 346  steps: 10001
Episode: 347  steps: 10001
Episode: 348  steps: 10001
Episode: 349  steps: 10001
Episode: 350  steps: 10001
Episode: 351  steps: 10001
Loss:  0.711296
Episode: 352  steps: 10001
Episode: 353  steps: 10001
Episode: 354  steps: 10001
Episode: 355  steps: 10001
Episode: 356  steps: 10001
Episode: 357  steps: 10001
Episode: 358  steps: 10001
Episode: 359  steps: 10001
Episode: 360  steps: 10001
Episode: 361  steps: 10001
Loss:  1.11969
Episode: 362  steps: 10001
Episode: 363  steps: 10001
Episode: 364  steps: 10001
Episode: 365  steps: 10001
Episode: 366  steps: 10001
Episode: 367  steps: 10001
Episode: 368  steps: 10001
Episode: 369  steps: 10001
```

Exercise 1

- Hyper parameter tuning
  - Learning rate
  - Sample size
  - Decay factor
- Network structure → FC / RNN / CNN
  - 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

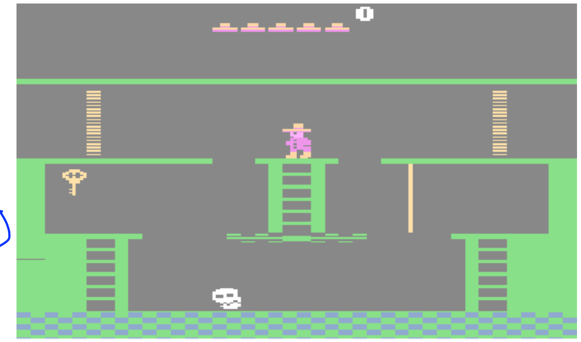
Exercise 2

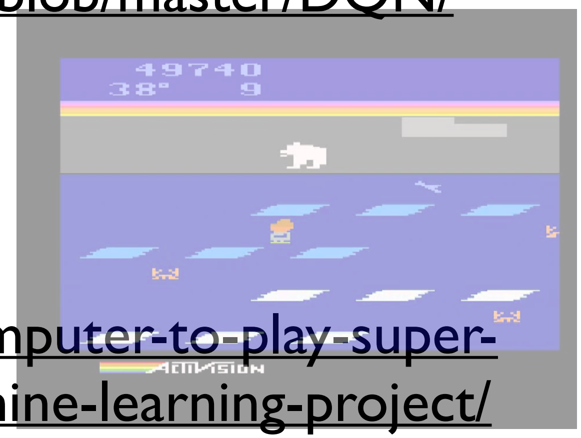
- Simple block based car race?
  - https://github.com/golbin/TensorFlow-Tutorials/tree/master/07%20-%20DQN
  - DQN 2013?
- Rewrite it using DQN 2015 algorithm?

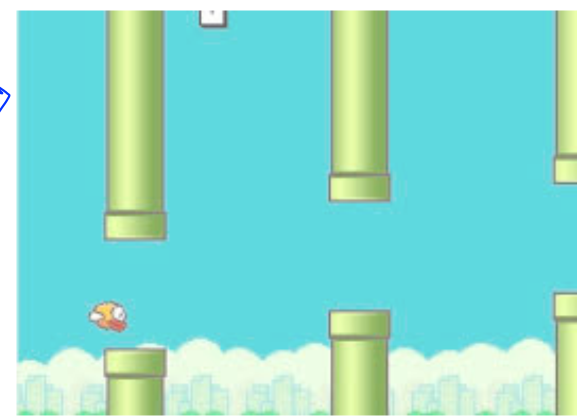


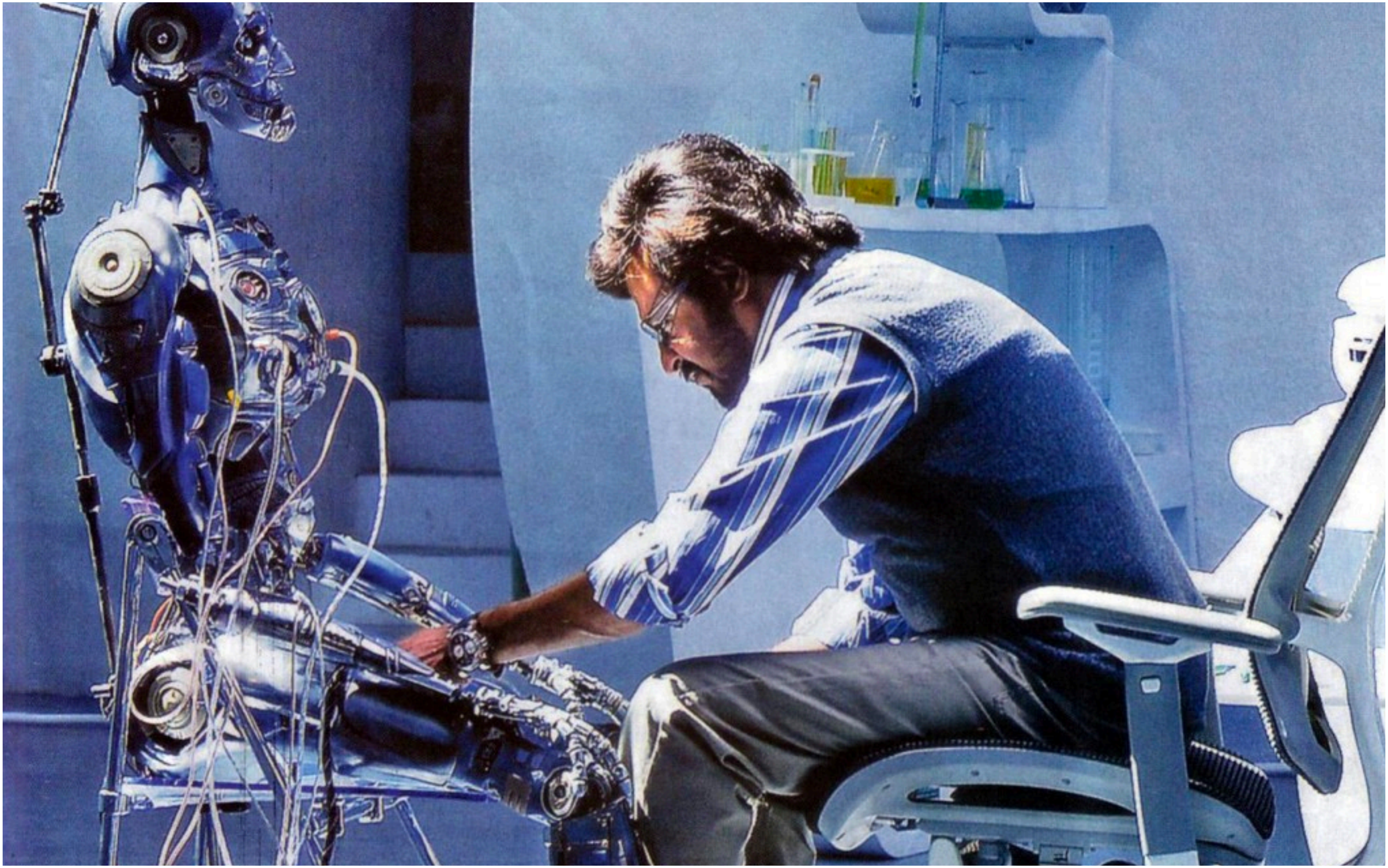
Exercise 3

- DQN implementations
  - https://github.com/songrotek/DQN-Atari-Tensorflow
  - https://github.com/dennybritz/reinforcement-learning/blob/master/DQN/dqn.py
  - https://github.com/devsisters/DQN-tensorflow
- Other games
  - http://www.ehrenbrav.com/2016/08/teaching-your-computer-to-play-super-mario-bros-a-fork-of-the-google-deepmind-atari-machine-learning-project/
- RMA approach
  - Run
  - Modify
  - Adapt (to your new game/problems)









Next

Policy Gradient

(better than DQN?)