Daniel Slater

Building a Pong playing AI

Project repo: https://github.com/DanielSlater/PyDataLondon2016

We will talk about...

Google deepmind recently got the worlds best performance at learning a variety of atari games.

Here we are going to look how it works and re-implementing their work in PyGame with TensorFlow (you could maybe do it in Theano?)

Subjects covered

- Neural networks
- Reinforcement learning
- Q-learning
- TensorFlow
- Convolutional networks

Why pong?

- Why Pong
- Pong - Classic, simple, dynamic game
- We want to train a computer to play it, just from watching it.
- Why?

Why do we care about this?

- It’s fun
- It’s challenging
- If we can develop generalized learning algorithms they could apply to many other fields
- It will allow us to build our future robot overlords who will inherit the earth from us

Resources

Resources to go with this talk are in this repo https://github.com/DanielSlater/PyDataLondon2016

You will need:

- **Linux** Sorry…
- **Python** either 2 or 3
- **PyGame**
- **TensorFlow** And an nvidia GPU, you could also follow along the reengineer in Theano(if you do please submit)

PyGame

- http://pygame.org/
- Most popular python games framework
- 1000’s of games, all free, all open source
- All written in Python

PyGamePlayer

https://github.com/DanielSlater/PyGamePlayer

- Allows running of PyGame games with zero touches
- Handles intercepting screen buffer and key presses
- Fixes the game frame rates

Mini Pong

- 640x480 is a bit big to run a network against
- Requires resizing the screen down to a more manageable 80x80
- Mini-Pong allows you to set the screen size to as small as 40x40 and save on processing

Half Pong

- To a machine even Pong can be hard
- Half pong is an even easier version of pong.
- Just one bar, you get points just for hitting the opposite wall
- Also can be small like mini-pong
- Hopefully it will be able to train in hours not days.

Running half pong in PyGame player

- Build something that can play Half Pong by just making random moves
- RandomHalfPongPlayer

https://github.com/DanielSlater/PyDataLondon2016/blob/master/examples/1_random_half_pong_player.py

Inheriting from PyGame Player

```
from resources.PyGamePlayer.pygame_player import PyGamePlayer
from resources.PyGamePlayer.games.half_pong import run


class RandomHalfPongPlayer(PyGamePlayer):
  def __init__(self):
    super(RandomHalfPongPlayer, self).__init__(run_real_time=True)

   def start(self):
      super(RandomHalfPongPlayer, self).start()

      run(screen_width=640, screen_height=480)
```

Running half pong in PyGame player

```python
def get_keys_pressed(self, screen_array, feedback, terminal):
    action_index = random.randrange(3)
    if action_index == 0:
        return [K_DOWN]
    elif action_index == 1:
        return []
    else:
        return [K_UP]



def get_feedback(self):
    from resources.PyGamePlayer.games.half_pong import score

    # get the difference in scores between this and the last frame
    score_change = score - self._last_score
    self._last_score = score
    return float(score_change), score_change == -1
```

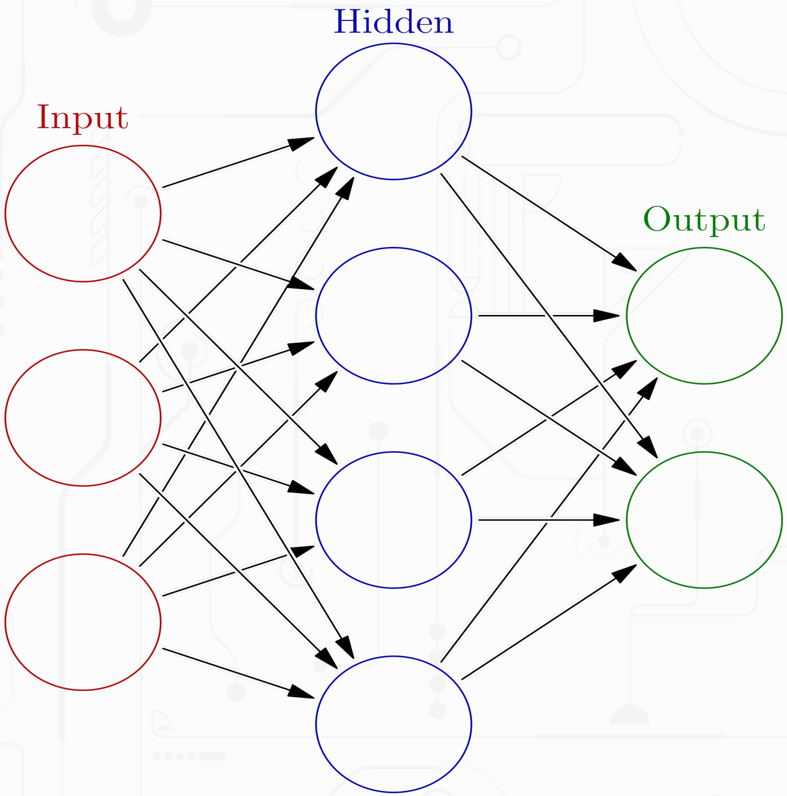
How good is RandomHalfPong?

Not very good.

Score is around -0.03

Lets try using neural networks!

What is a neural network?



- Inspired by the brain
- Sets of nodes are arranged in layers
- Able to approximate complex functions

MLPHalfPong

```
import cv2
import numpy as np
import tensorflow as tf
from common.half_pong_player import HalfPongPlayer

class MLPHalfPongPlayer(HalfPongPlayer):
   def __init__(self):
       super(MLPHalfPongPlayer, self).__init__(run_real_time=False,
                                                  force_game_fps=6)
       self._input_layer, self._output_layer = self._create_network()
       init = tf.initialize_all_variables()
       self._session = tf.Session()
       self._session.run(init)

 def _create_network(self):
       input_layer = tf.placeholder("float", [self.SCREEN_WIDTH, self.
SCREEN_HEIGHT])
       feed_forward_weights_1 = tf.Variable(tf.truncated_normal([self.
SCREEN_WIDTH, self.SCREEN_HEIGHT], stddev=0.01))
       feed_forward_bias_1 = tf.Variable(tf.constant(0.01, shape=[256]))

       feed_forward_weights_2 = tf.Variable(tf.truncated_normal([256,
self.ACTIONS_COUNT], stddev=0.01))
       feed_forward_bias_2 = tf.Variable(tf.constant(0.01, shape=[self.
ACTIONS_COUNT]))

       hidden_layer = tf.nn.relu(
           tf.matmul(input_layer, feed_forward_weights_1) +
feed_forward_bias_1)

       output_layer = tf.matmul(hidden_layer, feed_forward_weights_2)
+ feed_forward_bias_2

       return input_layer, output_layer
```

MLPHalfPong

```
def _create_network(self):
    input_layer = tf.placeholder("float", [self.SCREEN_WIDTH, self.SCREEN_HEIGHT])

    feed_forward_weights_1 = tf.Variable(tf.truncated_normal([self.SCREEN_WIDTH, self.SCREEN_HEIGHT], stddev=0.01))
    feed_forward_bias_1 = tf.Variable(tf.constant(0.01, shape=[256]))

    feed_forward_weights_2 = tf.Variable(tf.truncated_normal([256, self.ACTIONS_COUNT], stddev=0.01))
    feed_forward_bias_2 = tf.Variable(tf.constant(0.01, shape=[self.ACTIONS_COUNT]))

    hidden_layer = tf.nn.relu(tf.matmul(input_layer, feed_forward_weights_1) + feed_forward_bias_1)

    output_layer = tf.matmul(hidden_layer, feed_forward_weights_2) + feed_forward_bias_2

    return input_layer, output_layer
```

Neural network controlling actions

```
def get_keys_pressed(self, screen_array, feedback, terminal):
    # images will be black or white
    _, binary_image = cv2.threshold(cv2.cvtColor(screen_array, cv2.COLOR_BGR2GRAY), 1, 255,
                                    cv2.THRESH_BINARY)


    output = self._session.run(self._input_layer, feed_dict={self._output_layer: binary_image})
    action = np.argmax(output)


    return self.action_index_to_key(action)
```

How does it do?

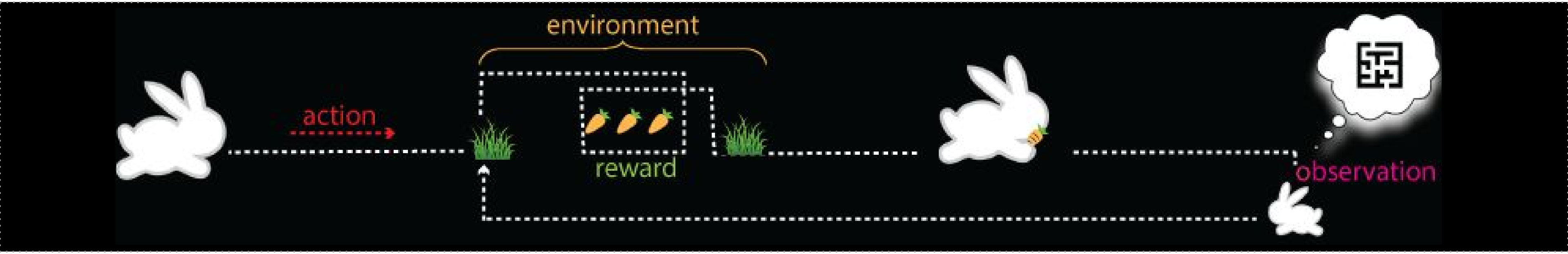
MLPHalfPong

- Awful...

- We need to train it
- But what is the loss function for the game pong?

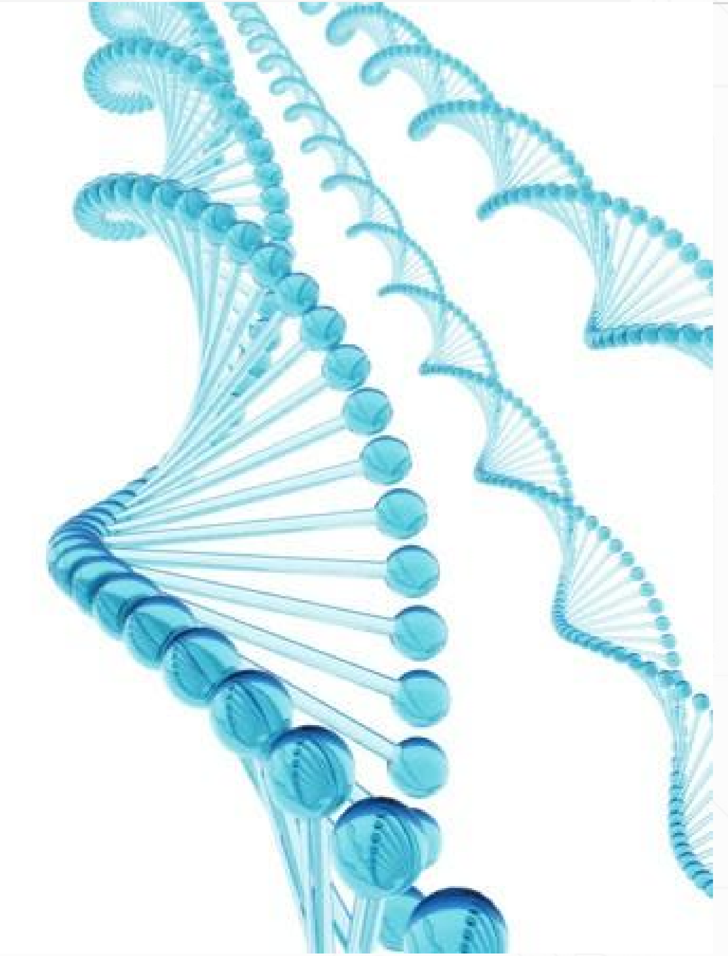
Reinforcement learning

- Agents are run within an environment.
- As they take actions they receive feedback
- They aim to maximize good feedback and minimize bad feedback
- Computer games are a great way to train reinforcement learning agents. we know we can learn games from just a sequence of images, so computer agents should be able to do the same thing (given enough computational power, time and the right algorithms).



Approaches to reinforcement learning

- Genetic algorithms are very popular/successful
- But very random and unprincipled
- Doesn’t feel like how humans learn
- What else could we try?



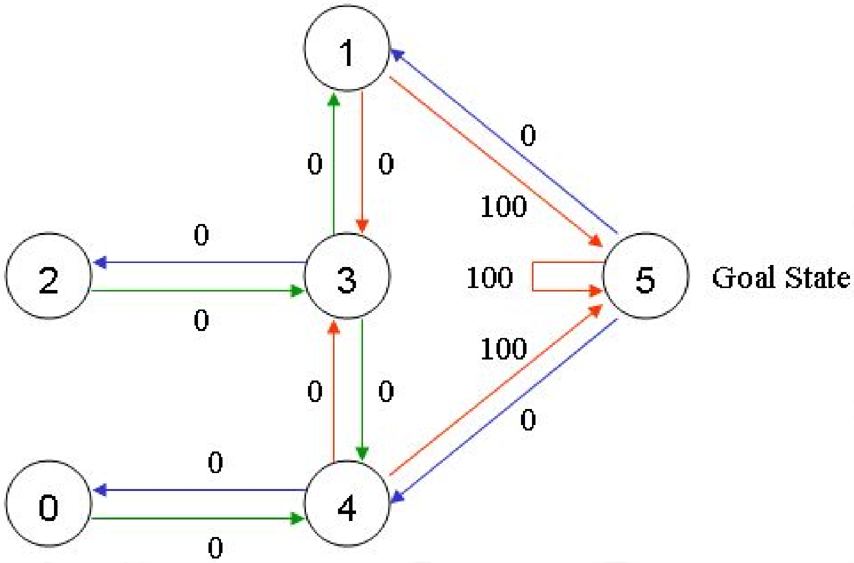
Q-Learning

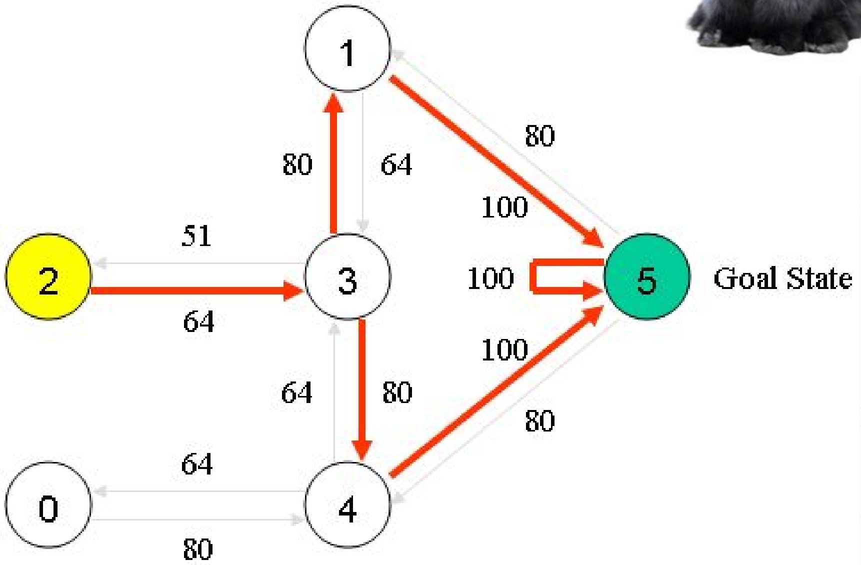
- Given a state and an a set of possible actions determine the best action to take to maximize reward
- Any action will put us into a new state that itself has a set of possible actions
- Our best action now depends on what our best action will be in the next state, and so on
- For example...

Q-Learning Maze example

Bunny must navigate a maze

Reward = 100 in state 5 (a carrot)
Discount factor = 0.8





Images stolen from http://mnemstudio.org/path-finding-q-learning-tutorial.htm

After optimization….

Q Learning

$$Q^*(s,a) = \mathbb{E}_{s' \sim \mathcal{E}}\left[r + \gamma \max_{a'} Q^*(s',a') \Big| s,a\right]$$

$$Q_{t+1}(s_t,a_t) = \underbrace{Q_t(s_t,a_t)}_{\text{old value}} + \underbrace{\alpha_t(s_t,a_t)}_{\text{learning rate}} \cdot \left(\overbrace{\underbrace{R_{t+1}}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \underbrace{\max_a Q_t(s_{t+1},a)}_{\text{estimate of optimal future value}}}^{\text{learned value}} - \underbrace{Q_t(s_t,a_t)}_{\text{old value}} \right)$$

- Q-function is the concept of the perfect action state function
- We will use a neural network to approximate this Q-function

World's simplest game

States with rewards

states = [0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]

- Agent exists in one state and can move forward or backward (with wrap around).
- Tries to get to the maximum reward
- We want to determine the maximum reward we could get in each state. The best action is to move to the state with the best reward

In TensorFlow

https://github.com/DanielSlater/PyDataLondon2016/blob/master/examples/4_tensorflow_q_learning.py

```
session = tf.Session()

state = tf.placeholder("float", [None, NUM_STATES])
targets = tf.placeholder("float", [None, NUM_ACTIONS])

hidden_weights = tf.Variable(tf.constant(0., shape=[NUM_STATES,
NUM_ACTIONS]))

output = tf.matmul(state, hidden_weights)

loss = tf.reduce_mean(tf.square(output - targets))
train_operation = tf.train.AdamOptimizer(0.1).minimize(loss)

session.run(tf.initialize_all_variables())
```

In TensorFlow

```
for i in range(50):
   state_batch = []
   rewards_batch = []

   # create a batch of states
   for state_index in range(NUM_STATES):
      state_batch.append(hot_one_state(state_index))
      minus_action_index = (state_index - 1) % NUM_STATES
      plus_action_index = (state_index + 1) % NUM_STATES

      minus_action_state_reward = session.run(output, feed_dict={state: [hot_one_state(minus_action_index)]})
      plus_action_state_reward = session.run(output, feed_dict={state: [hot_one_state(plus_action_index)]})

      # these action rewards are the results of the Q function for this state and the actions minus or plus
      action_rewards = [states[minus_action_index] + FUTURE_REWARD_DISCOUNT * np.max(minus_action_state_reward),
                        states[plus_action_index] + FUTURE_REWARD_DISCOUNT * np.max(plus_action_state_reward)]
      rewards_batch.append(action_rewards)

   session.run(train_operation, feed_dict={
      state: state_batch,
      targets: rewards_batch})
```

Applying Q-Learning to Pong

- What the states and actions?
- Actions are the key presses.
- The state is the screen.
- Normal screen is 640x480 pixels = 307200 data points per state = 2^307200 different states
- Pong is a dynamic game a single static shot is not enough our state needs to comprise change.
- Make it store the last 4 frames.
- State is now 2^1228800 = too f***ing big number.
- Neural networks can reduce this state space.

MLP Q Learning Half Pong Player

https://github.com/DanielSlater/PyDataLondon2016/blob/master/examples/5_mlp_q_learning_half_pong_player.py

As well as Q-learning we will need:

Experience Replay

- We don’t just want to learn off the current state
- Real entities also learn from their memories
- We will collect states (experience)
- Then sample from them and learn off that (Replay)

Store observations in memory(record experience)

```
_, binary_image = cv2.threshold(cv2.cvtColor(screen_array, cv2.COLOR_BGR2GRAY), 1, 255,
                                 cv2.THRESH_BINARY)

binary_image = np.reshape(binary_image, (80 * 80,))

# first frame must be handled differently

if self._last_state is None:

    self._last_state = binary_image

    random_action = random.randrange(self.ACTIONS_COUNT)

    self._last_action = np.zeros([self.ACTIONS_COUNT])

    self._last_action[random_action] = 1.

    return self.action_index_to_key(random_action)

binary_image = np.append(self._last_state[self.SCREEN_WIDTH * self.SCREEN_HEIGHT:], binary_image, axis=0)

self._observations.append((self._last_state, self._last_action, reward, binary_image, terminal))
```

Training (Replay)

```
# sample a mini_batch to train on
mini_batch = random.sample(self._observations, self.MINI_BATCH_SIZE)

# get the batch variables
previous_states = [d[self.OBS_LAST_STATE_INDEX] for d in mini_batch]
actions = [d[self.OBS_ACTION_INDEX] for d in mini_batch]
rewards = [d[self.OBS_REWARD_INDEX] for d in mini_batch]
current_states = [d[self.OBS_CURRENT_STATE_INDEX] for d in mini_batch]
agents_expected_reward = []

# this gives us the agents expected reward for each action we might
agents_reward_per_action = self._session.run(self._output_layer, feed_dict={self._input_layer: current_states})

for i in range(len(mini_batch)):
   if mini_batch[i][self.OBS_TERMINAL_INDEX]:
      # this was a terminal frame so there is no future reward...
      agents_expected_reward.append(rewards[i])
   else:
      agents_expected_reward.append(
         rewards[i] + self.FUTURE_REWARD_DISCOUNT * np.max(agents_reward_per_action[i]))
```

Training (Replay)

```
# learn that these actions in these states lead to this reward

self._session.run(self._train_operation, feed_dict={
        self._input_layer: previous_states,
        self._actions: actions,
        self._target: agents_expected_reward})
```

Explore the space

- At first our Q-function is really bad
- Start with random movements and gradually phase in learned movements

```
def _choose_next_action(self, binary_image):
    if random.random() <= self._probability_of_random_action:
        return random.randrange(self.ACTIONS_COUNT)
    else:
        # let the net choose our action
        output = self._session.run(self._output_layer, feed_dict={self._input_layer: binary_image})
        return np.argmax(output)
```

How does it do?

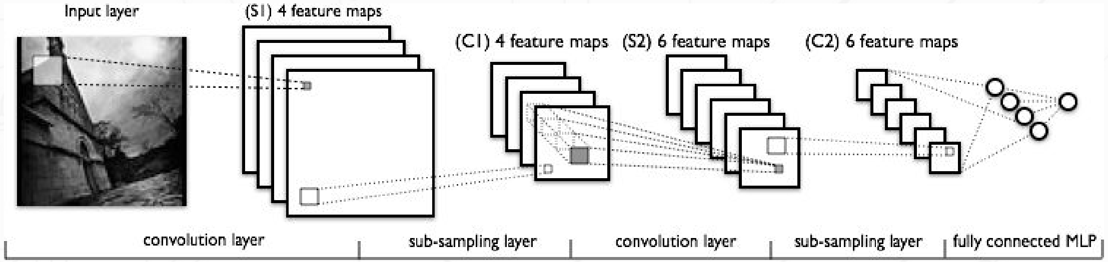
How does it do?

- After training for x was y: 0.0
- Still really bad
- Why, There is no linear/shallow mapping from screen pixels to the action

- Convolutional/Deep networks might do better?

Convolutional networks

Convolutional net:



- Use a deep convolutional architecture to turn a the huge screen image into a much smaller representation of the state of the game.

- Key insight: pixels next to each other are much more likely to be related...

Conv Net Half Pong Player

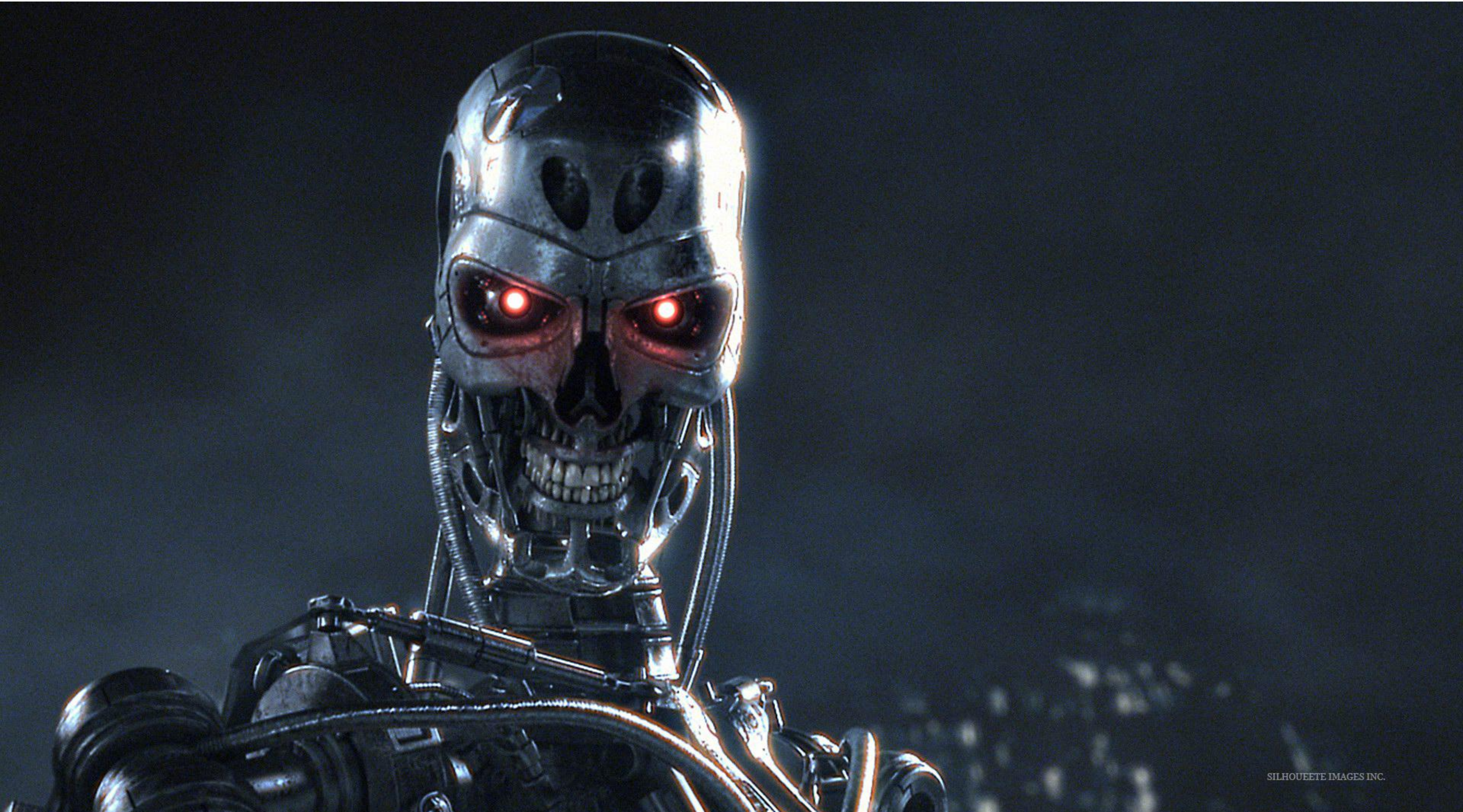
https://github.com/DanielSlater/PyDataLondon2016/blob/master/examples/6_conv_net_half_pong_player.py

Create convolutional network

```
input_layer = tf.placeholder("float", [None, self.SCREEN_WIDTH,self.SCREEN_HEIGHT, self.STATE_FRAMES])

hidden_convolutional_layer_1 = tf.nn.relu(tf.nn.conv2d(input_layer, convolution_weights_1, strides=[1, 4, 4, 1], padding="SAME") + convolution_bias_1)

hidden_max_pooling_layer_1 = tf.nn.max_pool(hidden_convolutional_layer_1, ksize=[1, 2, 2, 1],  strides=[1, 2, 2, 1], padding="SAME")

hidden_convolutional_layer_2 = tf.nn.relu(
   tf.nn.conv2d(hidden_max_pooling_layer_1, convolution_weights_2, strides=[1, 2, 2, 1],
                padding="SAME") + convolution_bias_2)

hidden_max_pooling_layer_2 = tf.nn.max_pool(hidden_convolutional_layer_2, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding="SAME")

hidden_convolutional_layer_3_flat = tf.reshape(hidden_max_pooling_layer_2, [-1, 256])

final_hidden_activations = tf.nn.relu(
   tf.matmul(hidden_convolutional_layer_3_flat, feed_forward_weights_1) + feed_forward_bias_1)

output_layer = tf.matmul(final_hidden_activations, feed_forward_weights_2) + feed_forward_bias_2
```

How does it do?

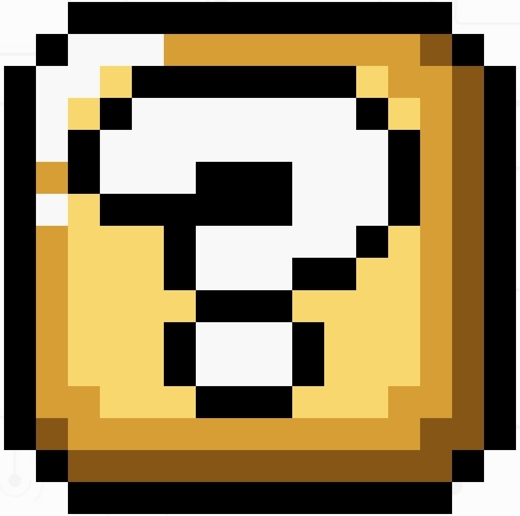

Now it’s amazing!


SILHOUEETE IMAGES INC.

Well pretty good

- Score is: +0.3
- Much better than random
- Appears to be playing the game
- The same architecture can work on all kinds of other games:
    - Breakout
    - Q*bert
    - Seaquest
    - Space invaders

Thank you! Hope you enjoyed this!



contact me @:

http://www.danielslater.net/