Learning about (Deep) Reinforcement Learning

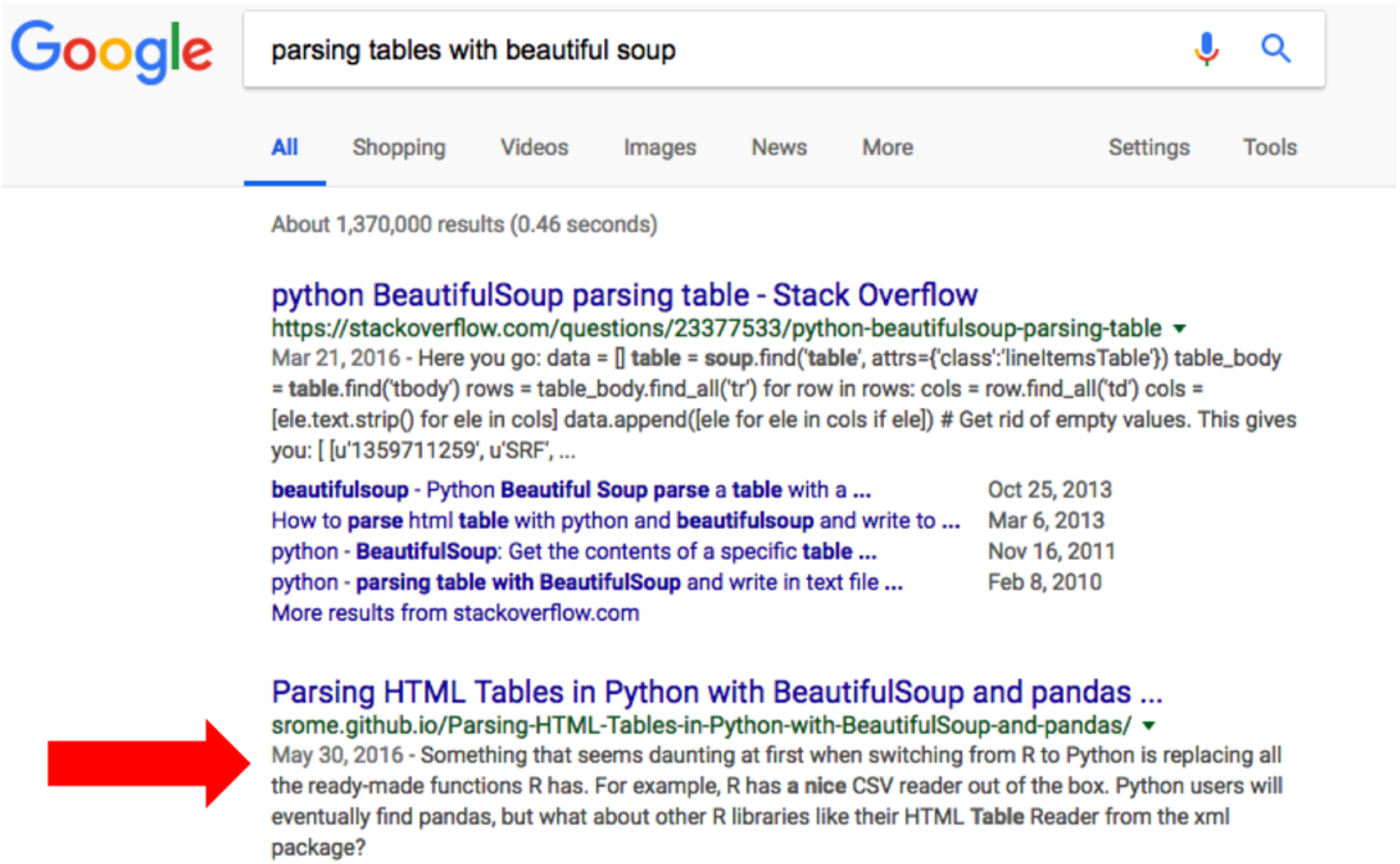
Scott Rome

romescott@gmail.com

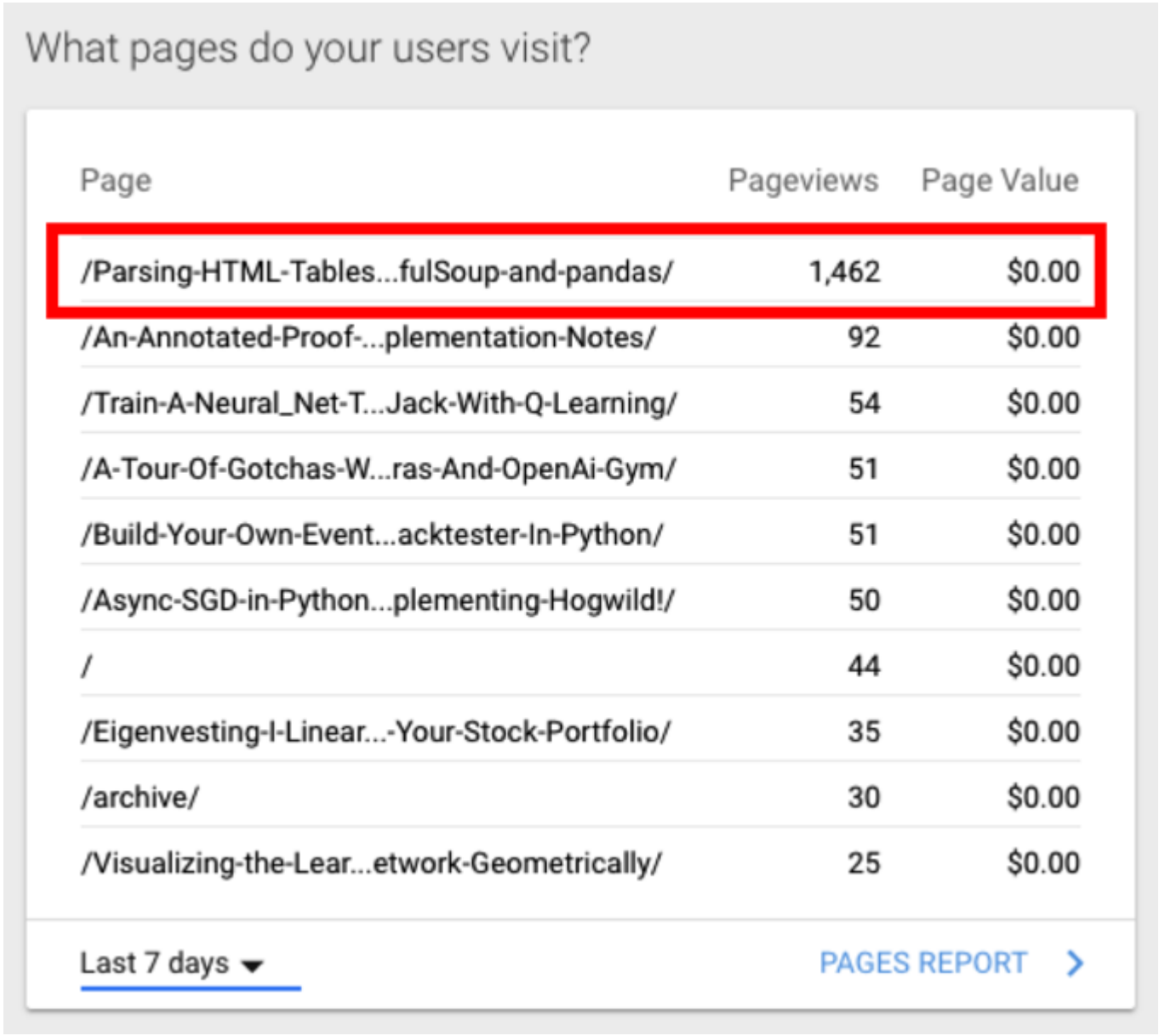
About Me

- 2 Cats
- Currently the Lead Data Scientist at an ad tech firm
  - Previously a Data Scientist in healthcare
- Ph.D. in Mathematics
- Run a ML blog: http://srome.github.io

You May Also Know Me From...



Sadly, I'm A One Hit Wonder



PSA: Volunteers Wanted!

techgirlz™

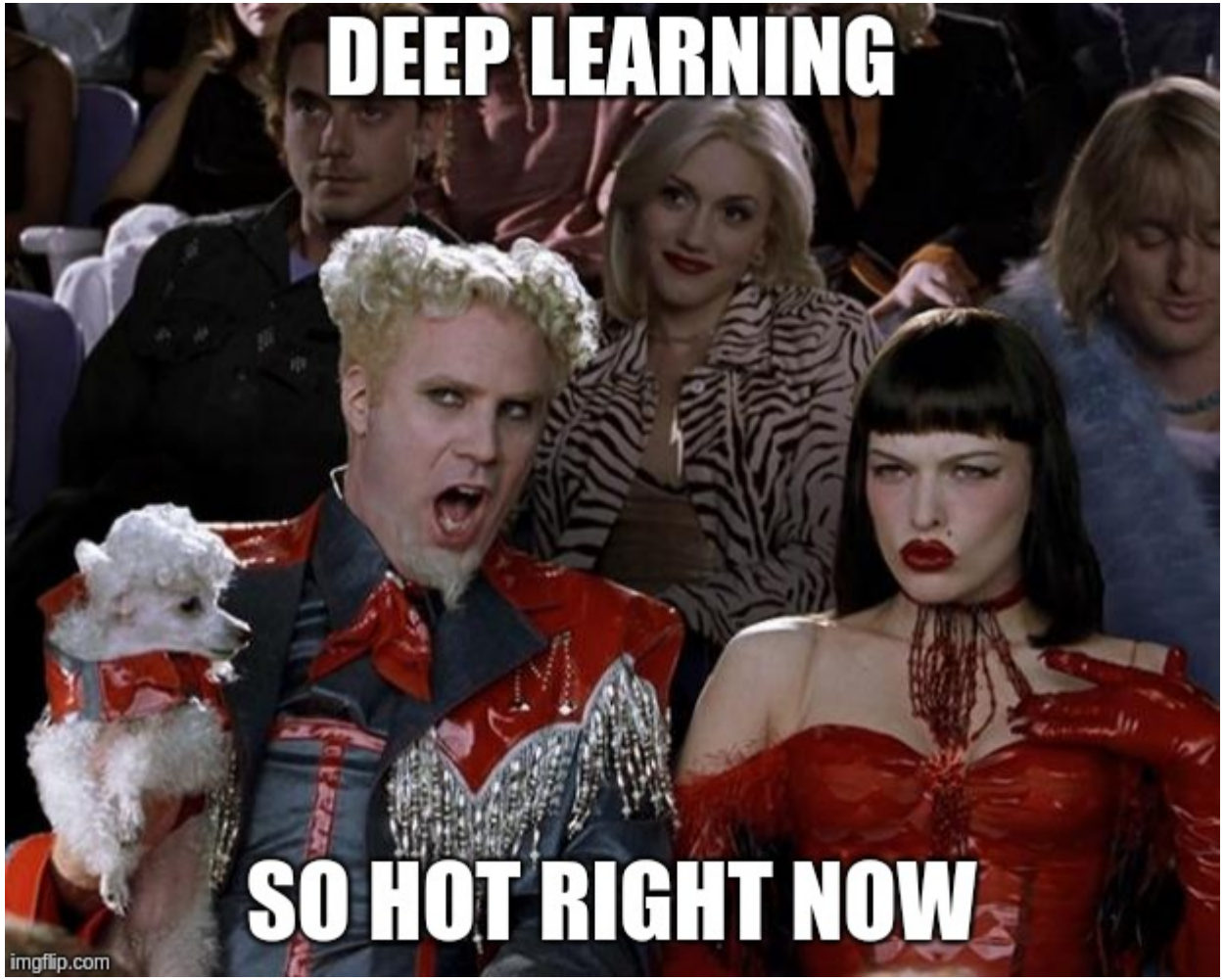
You should leave with...

- Some ideas about what (deep) reinforcement learning is
- An understanding of the basic training process for reinforcement learning
- Having seen more than enough code snippets from https://github.com/srome/ExPyDQN
- Not having heard all the annoying gotchas when implementing a reinforcement learning framework (see the post "A Tour of Gotchas When Implementing Deep Q Networks with Keras and OpenAi Gym" from my blog for that).

Reinforcement Learning: A Brief History

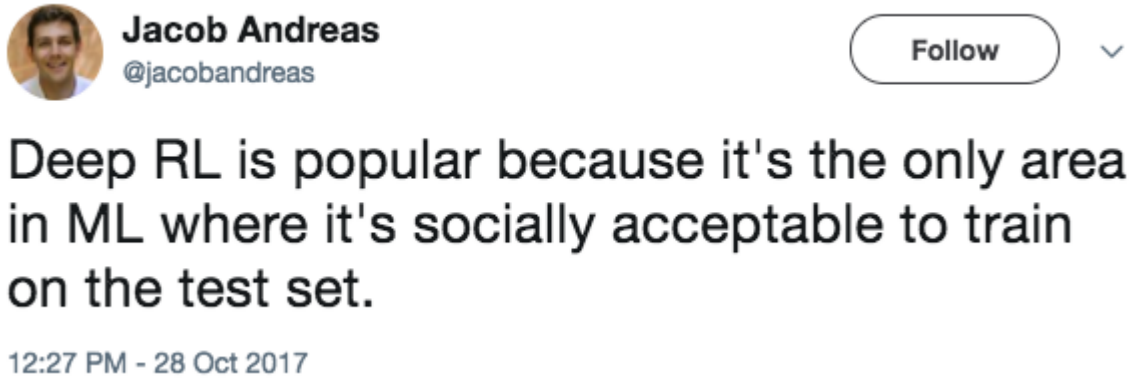
- (1950s) Roots in Optimal Control / dynamic programming, developed by Bellman at the RAND Corporation
  - Called "dynamic programming" because the Secretary of Defense "actually had a pathological fear and hatred of the word, research"
- (1977) Formal study of Temporal Difference Learning began
- (1989) Q-Learning was published
- (1992) TD-Gammon developed
- (2013) Deep Q Learning published (Neural Networks used to play Atari)
- (2016) AlphaGo beats Lee Sidol
- (2017) AlphaGo Zero beats Alpha Go

Notice, the activity starting around the early 2010's... that is in part due to...


DEEP LEARNING
SO HOT RIGHT NOW
imgflip.com

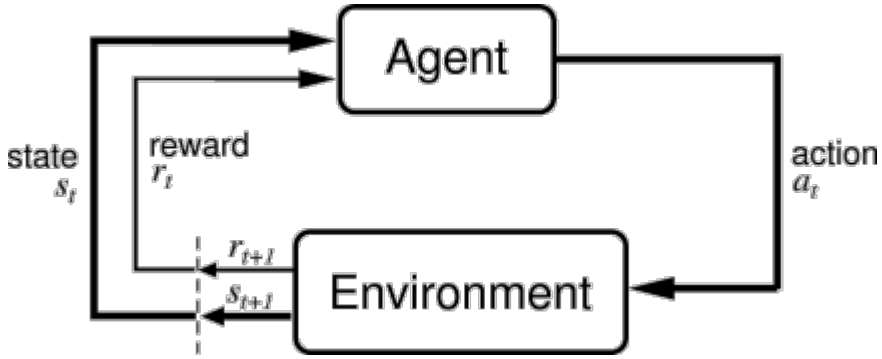
Goal of Reinforcement Learning

Training an agent (e.g., a model) to interact with an environment



Training Algorithm at a High Level

From (7),



In [ ]:

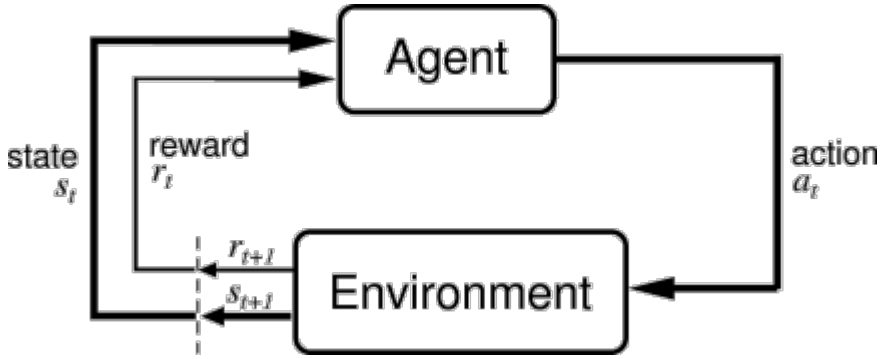
```
env  # emulator environment
state # initial state from environment
agent # the thing we want to train

while game_still_going:
    action = agent.get_action(state) # get action from the agent
    next_state, reward, game_still_going = env.step(action) # pass action to emulat
or, get next state
    agent.update(state, next_state, action, reward) # update the agent
```

Differences vs. "Typical" Machine Learning Problems

- Training data is usually generated during training
- The agent (model) is not independent from the data it is trained on:
  - The agent affects the generation of new training data.
  - The target variable at each update step (as we will see) depends on a version of the agent.
- It's possible to miss part of your possible training set due to data collection decisions

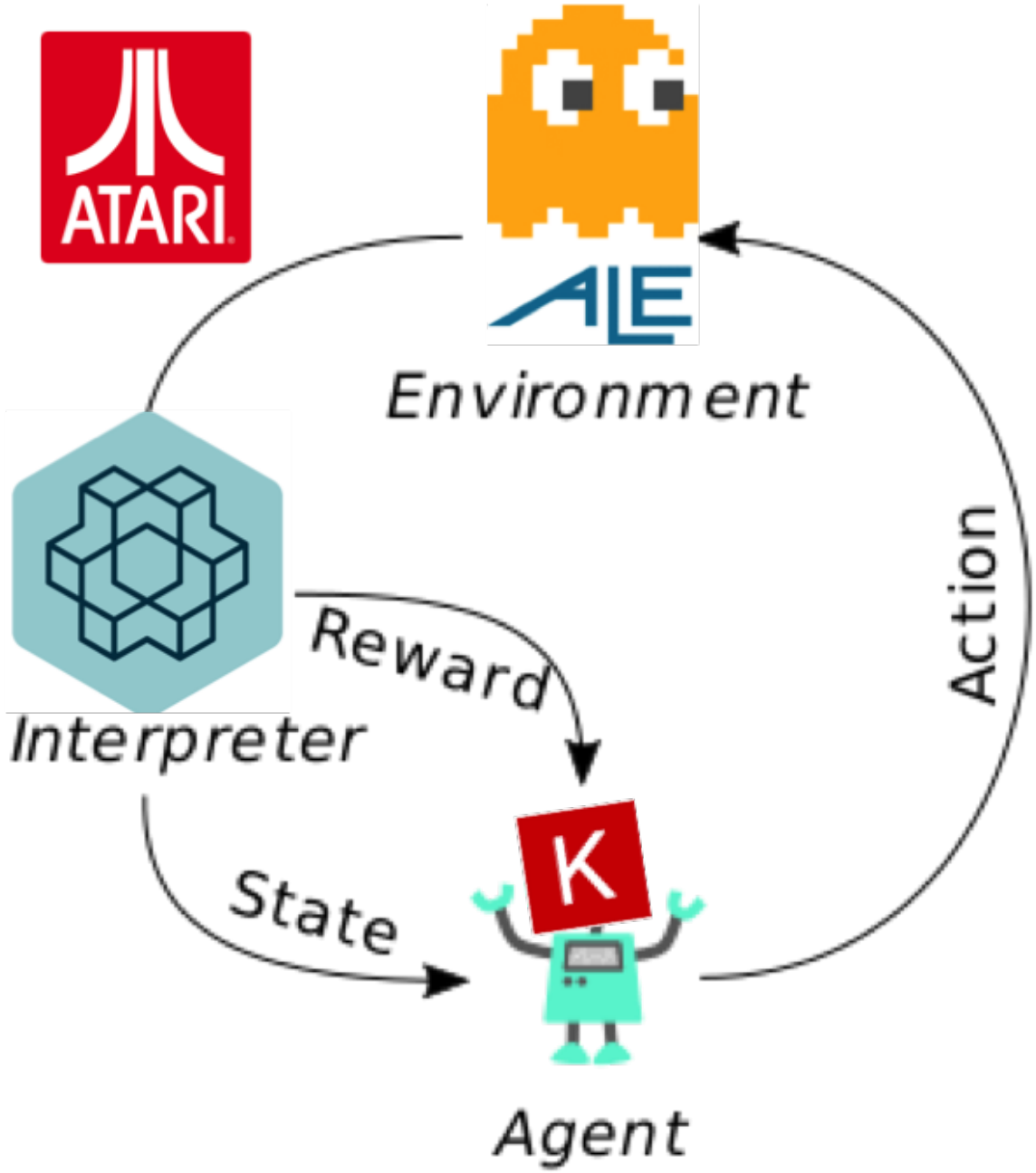
In [ ]:
```
while game_still_going:
    action = agent.get_action(state)
    next_state, reward, game_still_going = env.step(action)
    agent.update(state, next_state, action, reward)
```

Big Questions



- What libraries do we use?
- What (who?) is the agent?
- How do we select a_t?
- What is s_t?
- What is our target y_t?
    - What is the minibatch?

Python Libraries



* Disclaimer: Not a graphic artist.

The Agent

From (3),

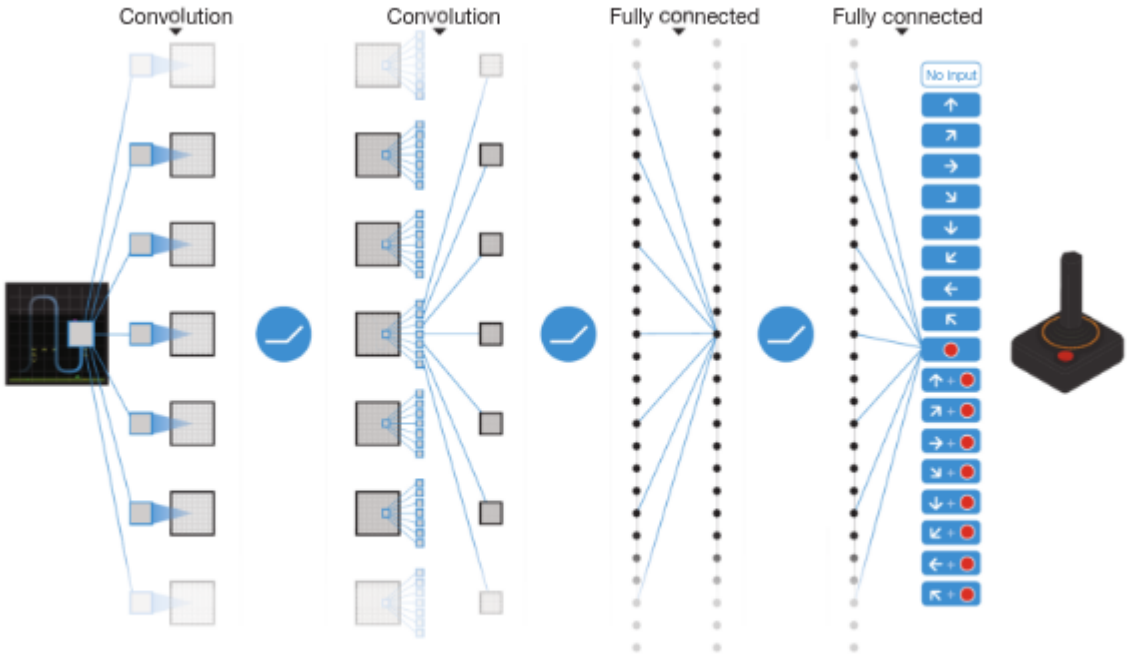
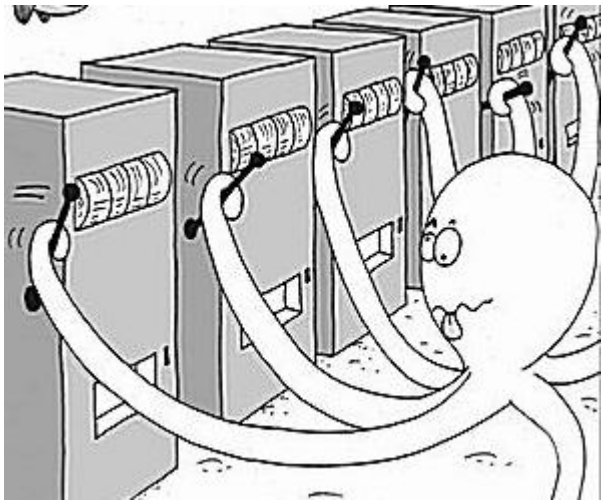


Figure 1 | Schematic illustration of the convolutional neural network. The details of the architecture are explained in the Methods. The input to the neural network consists of an $84 \times 84 \times 4$ image produced by the preprocessing map ϕ, followed by three convolutional layers (note: snaking blue line symbolizes sliding of each filter across input image) and two fully connected layers with a single output for each valid action. Each hidden layer is followed by a rectifier nonlinearity (that is, $\max(0,x)$).

How Do We Select a_t: Exploration / Exploitation

The "Multi-Armed Bandit" Problem



"Originally considered by Allied scientists in World War II, it proved so intractable that, according to Peter Whittle, the problem was proposed to be dropped over Germany so that German scientists could also waste their time on it." - Wikipedia

Approximate Solution to MAB: ϵ - Greedy Strategy

- For each state, select a random action (level) with probability ϵ.
- Otherwise, allow the agent to choose the action.
- ϵ annealing is used to allow the agent to select more actions over time.

In [ ]:

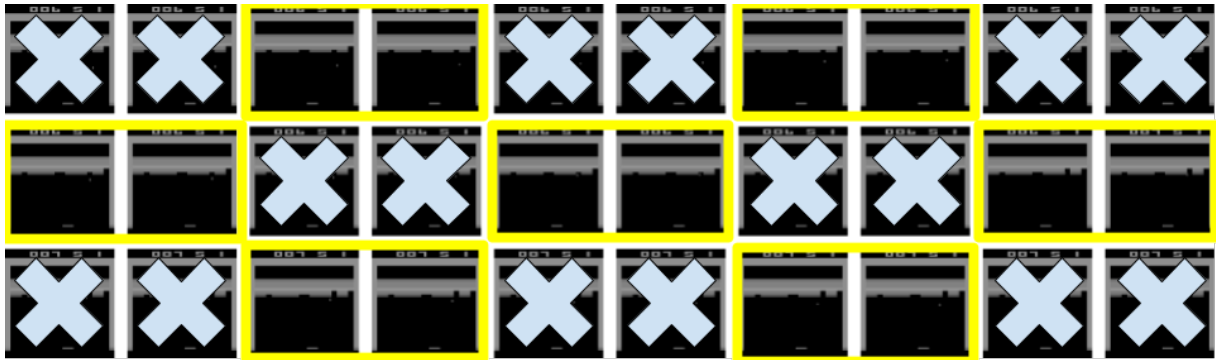
```
def get_action(self, image=None, epsilon_override=None):
    epsilon = self.get_epsilon(epsilon_override) # Annealing code is hidden in here
    if np.random.uniform(0, 1) < 1 - epsilon:
        rewards = self._get_q_values(image=image)
        action = self._actions[np.argmax(rewards)]
    else:
        action = np.random.choice(self._actions)

    return action
```

What is s_t?

The emulator returns an image after each action. To form s_t, we must

- Drop image to grayscale
- Frame skip
  - Every n-th frame is considered for the definition of the current state
  - n is sometimes referred to as ϕ length
- Frames are stacked as an input
  - For a (84,84) grayscale image and frame skip of 4: an instance state has the dimension (4,84,84,1)
- Consecutive max
  - Take pixel-wise max of $(n-1)$-th and n-th image from the emulator as the n-th frame.

From (5),

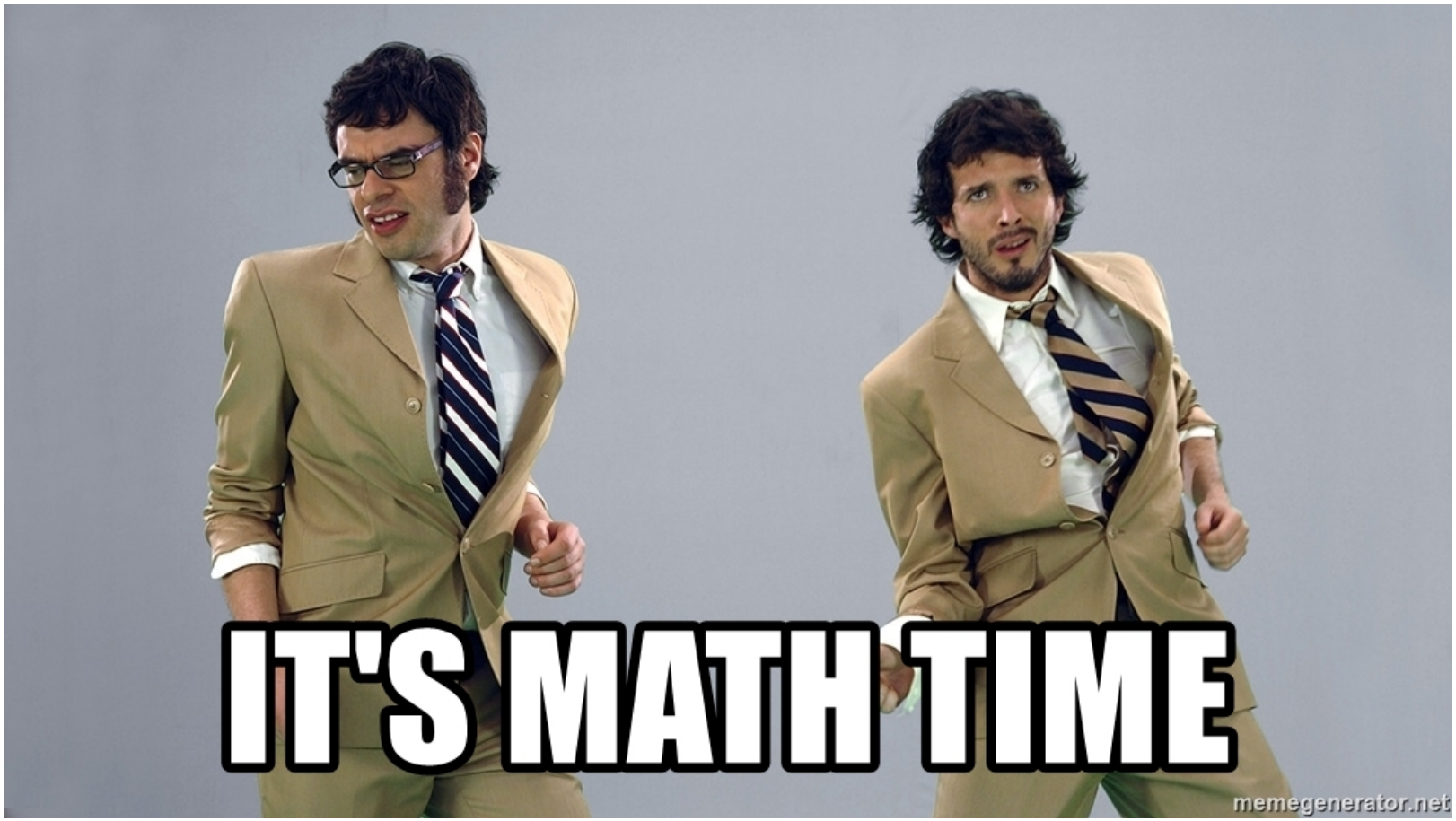


Frame Skip / Consecutive Max Code during Emulator Step

In [6]:

```
def step(self, action):
    """ This relies on the fact that the underlying environment creates new images
for each frame.
        By default, opengym uses atari_py's getScreenRGB2 which creates a new array
for each frame."""

    total_reward = 0

    obs = None
    for k in range(self.frame_skip):
        last_obs = obs

        # Make sure you are using a "NoFrameskip-v4" ROM
        obs, reward, is_terminal, info = self._env.step(action)
        total_reward += reward

        if is_terminal:
            # End episode if is terminal
            if k == 0 and last_obs is None:
                last_obs = obs
            break

    if self.consecutive_max and self.frame_skip > 1:
        obs = np.maximum(last_obs, obs)

    # Store observation, greyscale and cropping are applied in here
    self.store_observation(obs)

    return self._get_current_state(), total_reward, is_terminal
```

As Promised in the Abstract...



What is y_t?

Let $r' \in \mathbb{R}$ be a reward from performing action a at state s. The action-value function Q provides a mapping from $(s, a) \rightarrow r'$.

- Using Q, you can define a policy for the agent.
- Greedy Policy:
  - At each time step, the agent selects the action associated with the highest reward.

More Definitions

Assume $r \in \mathbb{R}$ is the resulting reward from the emulator for performing a at state s, and let $\gamma \in \mathbb{R}$ be a discount.

We define the optimal policy as the actions chosen via Q^*, which satisfies the *Bellman Equation*:

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

i.e., the reward plus the discounted future rewards for following the policy.

y_t defined

We define our neural network with weights θ as $Q(s, a, \theta)$. For each example (s_t, s_{t+1}, a_t, r_t), we select weights θ_t and define the training target to be:

$$y_t(\theta_t) := r_t + \gamma \max_{a'} Q(s_{t+1}, a', \theta_t)$$

- Notice, our target depends on θ_t -- it's not fixed like traditional machine learning problems!
- Some out-of-scope math proves that this y_t allows us to learn Q^*
  - (Technically, we don't optimize a single loss function, but a sequence of loss functions)

Wait? y_t depends on θ? What θ?

- New targets are generated every minibatch using a recent, fixed set of weights $\hat{\theta}_t$.
- $\hat{\theta}_t$ is typically updated with recent weights every ~10000 steps (for example).
  - i.e. $\hat{\theta}_t = \hat{\theta}_{t_i}$ for some fixed older set of weights.
- This technique is referred to as the *fixed target network*.

Other notes on y_t

- Notationally, it is easier to define the neural network as $Q(s, a, \theta)$.
- Practically, it is defined as $Q(s, \theta)$ and $y_t \in \mathbb{R}^{|A|}$, i.e. y_t has a reward entry for every action.

Calculating the $\max_{a'} Q(s_{t+1}, a', \theta_i)$

In [4]:

```
def _build_training_variables(self, future_states, actions, is_terminal):
    future_rewards = self._get_model(fixed=True).predict(future_states) # Get the f
ixed target network!
    max_rewards = np.max(future_rewards, axis=1).reshape((len(future_rewards), 1))

    # Code for an optimization around training
    # which updates reward for ONLY the completed action
    # training_mask masks rewards for other actions so
    # we only update weights according to the action we chose
    training_mask = np.zeros(future_rewards.shape)
    action_indexes = np.array([[np.where(self._actions == action)[0][0]] for action
in actions])
    is_terminal = is_terminal * 1.  # convert to float
    for index0, index1 in zip(range(self._minibatch), action_indexes):
        training_mask[index0, index1] = 1.

    return max_rewards, training_mask, is_terminal
```

What is the minibatch?

- Time correlation of examples causes divergence during training
- The solution is "Replay Memory"
  - Store m examples in memory D and sample minibatches from D for training!

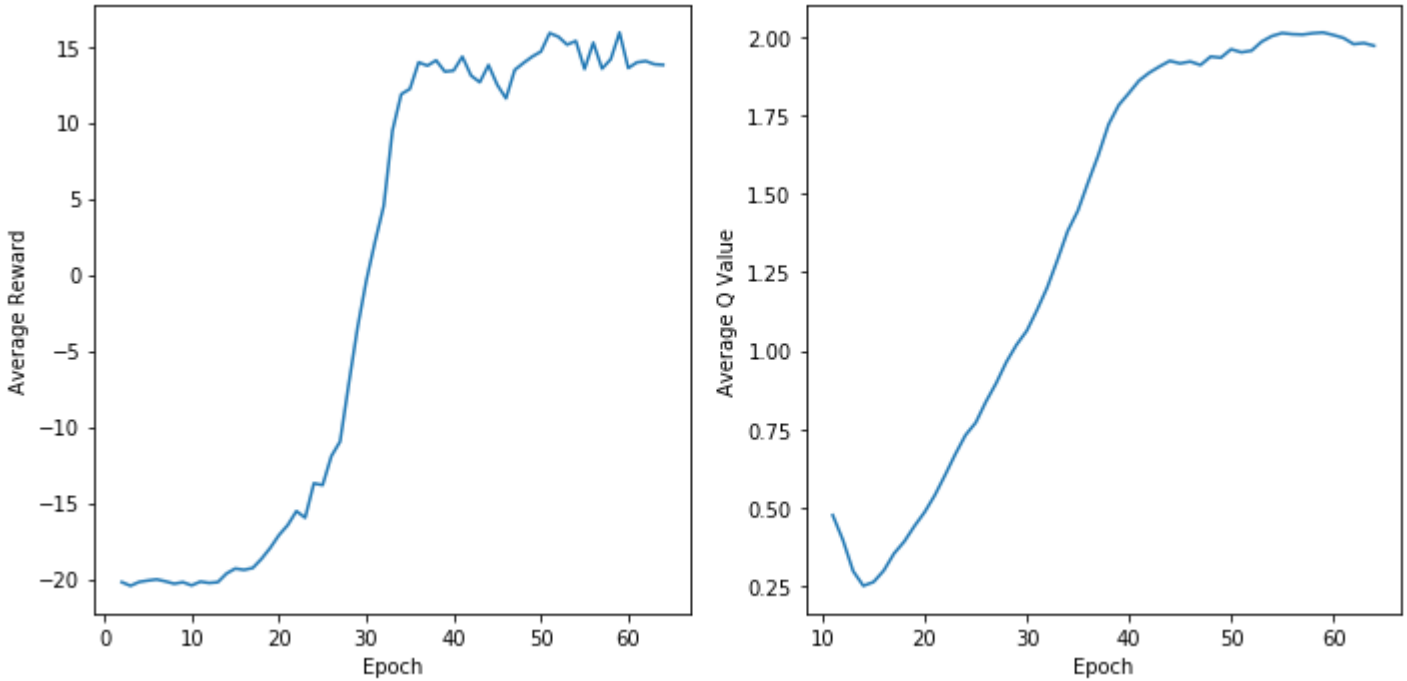
In [ ]:
```
def sample(self):
    # Code to sample from the replay memory

    ind = np.random.choice(self._size, size=self._mini_batch_size)

    # Avoiding a copy action as much as possible
    self._mini_batch_state[:] = self._memory_state[ind,:,:,:]
    self._mini_batch_future_state[:] = self._memory_future_state[ind,:,:,:]

    rewards = self._rewards[ind]
    is_terminal = self._is_terminal[ind]
    actions = self._actions[ind]

    return self._mini_batch_state, self._mini_batch_future_state, actions, rewards,
is_terminal
```

Final Training Algorithm

In [ ]:

```
def run_episode(self, training=True):
    state = self.reset_environment()
    is_terminal = False
    total_reward = 0
    steps = 0

    while not is_terminal:
        if not training:
            self.render() # Speeds up training to not display it

        action = self.get_action(state, training)
        next_state, reward, is_terminal = self.step(action)

        if training:
            clipped_reward = np.clip(reward ,-self.reward_clip ,self.reward_clip)
            self.replay_memory.add_example(state=state,
                                           future_state=next_state,
                                           action=action,
                                           reward=clipped_reward,
                                           is_terminal=is_terminal)
            self.training_step()

        state = next_state
        total_reward += reward
        steps += 1

    return total_reward, steps # for logging
```

Training Results



Thank You

- Slides will be available on http://srome.github.io
- The (functional) framework code is available at https://github.com/srome/ExPyDQN.
- Check out TechGirlz!

References

1. Playing Atari with Deep Reinforcement Learning
2. Wikipedia: Reinforcement Learning
3. Human-level control through deep reinforcement learning
4. Wikipedia: Banach fixed-point theorem
5. Frame Skipping and Preprocessing for Deep Q Networks on Atari 2600
6. Wikipedia: Multi-Armed Bandit
7. Google Image Search
8. Scholarpedia: Reinforcement Learning
9. Richard Bellman on the Birth of Dynamic Programming