Battle of Pong: DQN vs PG

Yash, Matt, Kenny, John

Problem Background

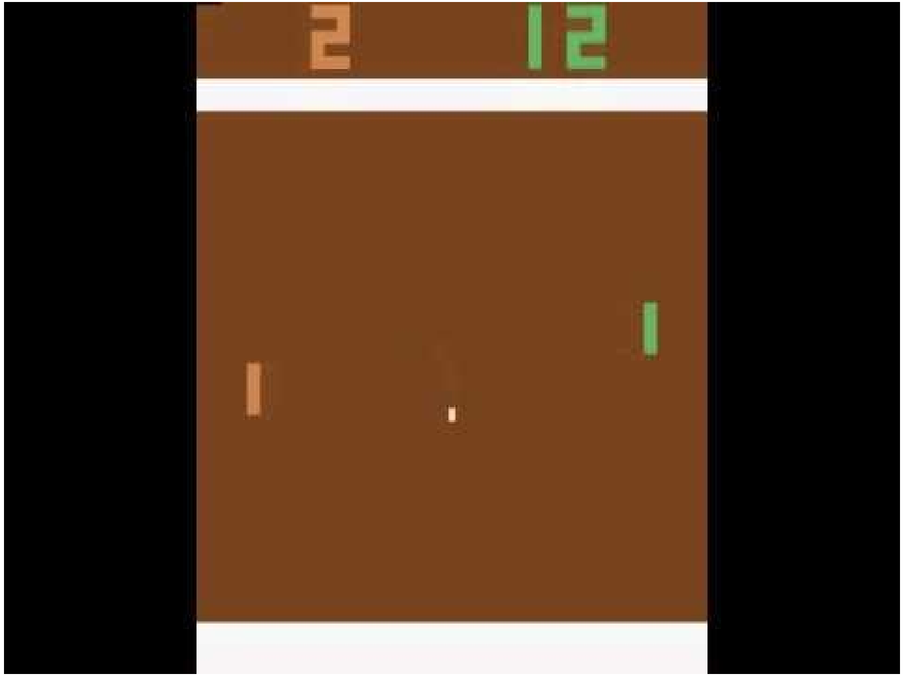
Given

1. Pixels from a game of pong
2. An indication of wins and losses
3. An opponent agent

Can we create an agent that could beat a human player?

Problem Motivation

- PONG is a classic game with an early, simple AI
- Games are an easy way to test theories
- Wanted to find the BEST training algorithm for this game



Goal

To see what performs better in a basic game of pong

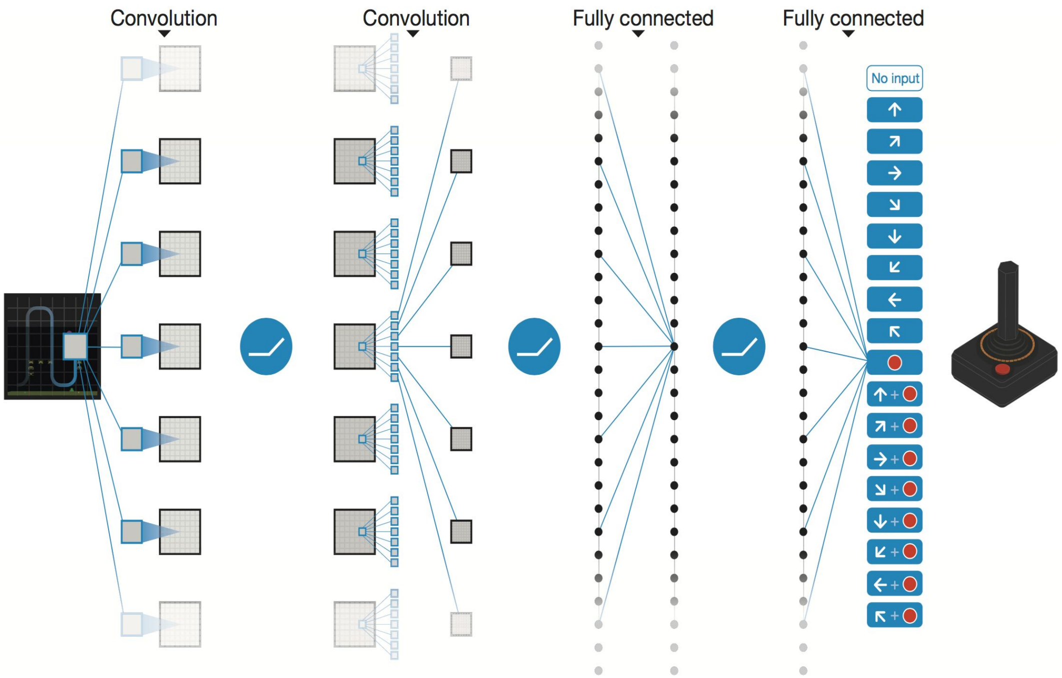
1. Deep Q-Learning Network (DQN)
    a. Deep Q-Learning (DQN): "Human-level control through deep reinforcement learning". 2015.
        i. DeepMind, AlphaGo
2. Policy Gradient (PG)
    a. Policy Gradient (PG): "Asynchronous Methods for Deep Reinforcement Learning". 2016

Algorithm: Deep Q-Learning Network

- Choose an action based on the best Q-value available
- Network architecture: Convolutional Neural Network to process images

Architecture: Deep Q-Learning Network

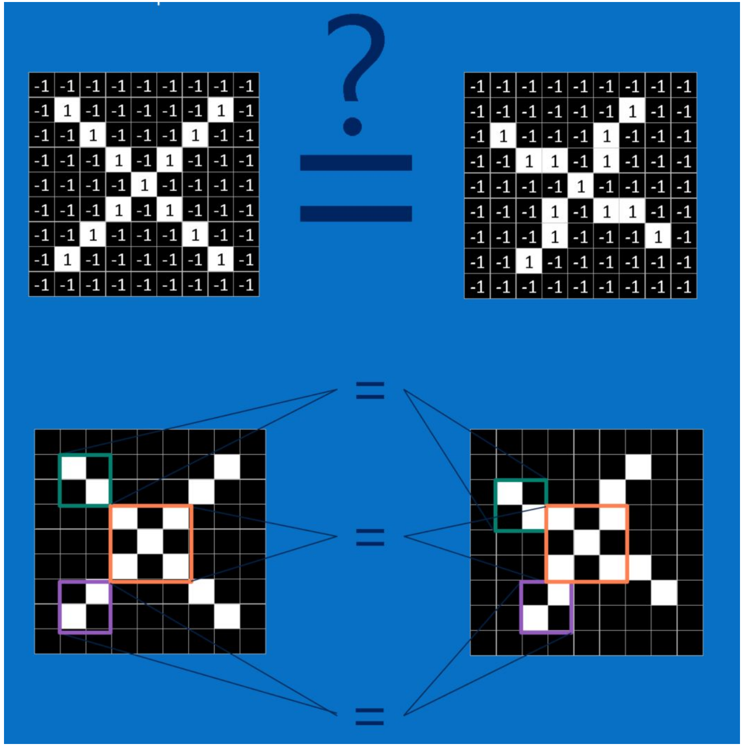
- **Convolutional neural network:**
  - Overlapping nodes
  - Captures visual input in small chunks
  - Final output comes from a regular neural net



https://github.com/devsisters/DQN-tensorflow/blob/master/assets/model.png

Architecture: Deep Q-Learning Network

- Chunking the input image and look for patterns
- Match sub-sections (features) against known outputs in similar spatial regions
- Does this by trying to match the feature at every possible location
- For PONG, we look only for features we care about, such as the paddles and the ball



https://brohrer.github.io/how_convolutional_neural_networks_work.html

Algorithm: Deep Q-Learning Network

- Explore vs Exploit
    - Ɛ - choosing a random action
- Execute the action

Algorithm: Deep Q-Learning Network

- Observe & record in replay memory
    - State
    - Action
    - Reward
    - New State

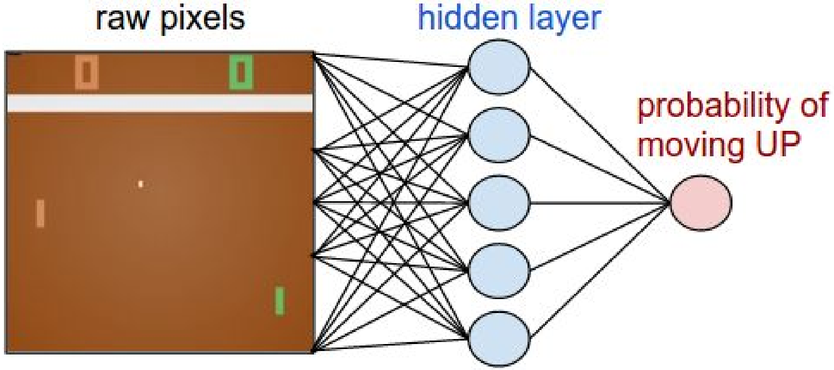
Algorithm: Deep Q-Learning Network

- Sample from replay memory
  - Back propagate error
  - Reduces correlations between updates
- Repeat process
  - Unless game has ended

Algorithm: Deep Q-Learning Network

- Previous Uses
  - DeepMind - https://deepmind.com/research/dqn/

Policy Gradient Architecture



Adopted from Dr. Andrej Karpathy

- Take in images from the game and preprocess them.
- Use the Neural Network to compute a probability of moving up.
- Sample from that probability distribution and tell the agent to move up/down.
- If the round is over, find whether you won or lost.
- When the episode has finished (someone got to 21 points), pass the result through the backpropagation algorithm to compute gradient for the weights.
- After 10 episodes have finished, sum up the gradient and move the weights in the direction of the gradient.
- Repeat this process until weights are tuned enough.

Hyperparameters/initializations

```python
batch_size = 10  # every how many episodes to do a param update?
gamma = 0.99  # discount factor for reward (i.e later rewards are exponentially less important)
decay_rate = 0.99  # decay factor for RMSProp leaky sum of grad^2
num_hidden_layer_neurons = 200  # number of hidden layer neurons
learning_rate = 1e-4  # for convergence (too low- slow to converge, too high,never converge)
resume = True  # resume from previous checkpoint?
render = False
```

- Weight initialization

```python
input_dimensions = 80 * 80  # input dimensionality: 80x80 grid (the pong world)
weights = {}  # initialize weights
# By using Xavier initialization, we make sure that the weights are not too small but not too big to
# propagate accurately the signals.
weights['W1'] = np.random.randn(num_hidden_layer_neurons, input_dimensions) / np.sqrt(
    input_dimensions)  # "Xavier" initialization
weights['W2'] = np.random.randn(num_hidden_layer_neurons) / np.sqrt(num_hidden_layer_neurons)
```

FORWARD PASS THROUGH THE NEURAL NET

- Compute hidden_layer_values to get initial activation values.
- ReLU(hidden_layer_values).
- Compute output_layer_value to get the probability of going up.
- p = sigmoid(output_layer_value) to make sure the probability is between 0 and 1.

```python
def apply_neural_nets(observation_matrix, weights):
    """ Based on the observation_matrix and weights,
    compute the new hidden layer values and the new output layer values. """
    hidden_layer_values = np.dot(weights['W1'], observation_matrix)
    hidden_layer_values = relu(hidden_layer_values)
    output_layer_values = np.dot(hidden_layer_values, weights['W2'])
    output_layer_values = sigmoid(output_layer_values)
    return hidden_layer_values, output_layer_values
```

Choose an action and move

```python
def choose_action(probability):
    random_value = np.random.uniform()
    if random_value < probability:
        # signifies up in openai gym
        return 2
    else:
        # signifies down in openai gym
        return 3
```

```python
action = choose_action(up_probability)

# carry out the chosen action
observation, reward, done, info = env.step(action)

reward_sum += reward
episode_rewards.append(reward)
```

LEARNING: GRADIENT PER ACTION

- How does changing the output probability (of going up) affect my result of winning the round?
- Binary Classification: $\partial L_i / \partial f_j = y_{ij} - \sigma(f_j)$
- Cheat: Treat the action we end up sampling from our probability as the correct action. **(PG Magic!)**

```
# see here: http://cs231n.github.io/neural-networks-2/#losses
fake_label = 1 if action == 2 else 0
loss_function_gradient = fake_label - up_probability
episode_gradient_log_ps.append(loss_function_gradient)
```

LEARNING AT THE END OF AN EPISODE

- Collect all observations and gradient calculations for episode.

```
if done:  # an episode finished
    episode_number += 1
    # Combine the following values for the episode
    episode_hidden_layer_values = np.vstack(episode_hidden_layer_values)
    episode_observations = np.vstack(episode_observations)
    episode_gradient_log_ps = np.vstack(episode_gradient_log_ps)
    episode_rewards = np.vstack(episode_rewards)
```

- Discount rewards

```
# Tweak the gradient of the log_ps based on the discounted rewards
episode_gradient_log_ps_discounted = discount_with_rewards(episode_gradient_log_ps, episode_rewards, gamma)
```

  - Actions taken towards the end of an episode more heavily influence our learning than actions taken at the beginning.

DRUM ROLL………

...BACKPROPAGATION: COMPUTING GRADIENTS

```python
def compute_gradient(gradient_log_p, hidden_layer_values, observation_values, weights):
    """ See here: http://neuralnetworksanddeeplearning.com/chap2.html"""
    delta_L = gradient_log_p
    dC_dw2 = np.dot(hidden_layer_values.T, delta_L).ravel()
    delta_l2 = np.outer(delta_L, weights['W2'])
    delta_l2 = relu(delta_l2)
    dC_dw1 = np.dot(delta_l2.T, observation_values)
    return {
        'W1': dC_dw1,
        'W2': dC_dw2
    }
```

Summary: the equations of backpropagation

$$\delta^L = \nabla_a C \odot \sigma'(z^L) \qquad \text{(BP1)}$$

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l) \qquad \text{(BP2)}$$

$$\frac{\partial C}{\partial b^l_j} = \delta^l_j \qquad \text{(BP3)}$$

$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j \qquad \text{(BP4)}$$

Four fundamental equations of backpropagation. Source: Michael Nielsen

UPDATING WEIGHTS USING RMSProp

```python
if episode_number % batch_size == 0:
    update_weights(weights, expectation_g_squared, g_dict, decay_rate, learning_rate)
```

```python
def update_weights(weights, expectation_g_squared, g_dict, decay_rate, learning_rate):
    """ See here: http://sebastianruder.com/optimizing-gradient-descent/index.html#rmsprop"""
    epsilon = 1e-5
    for layer_name in weights.keys():
        g = g_dict[layer_name]
        expectation_g_squared[layer_name] = decay_rate * expectation_g_squared[layer_name] + (1 - decay_rate) * g ** 2
        weights[layer_name] += (learning_rate * g) / (np.sqrt(expectation_g_squared[layer_name] + epsilon))
        g_dict[layer_name] = np.zeros_like(weights[layer_name])  # reset batch gradient buffer
```

Previous uses of policy gradients

- Underwater cable tracking - PG with initial example policy from computer simulation
- Robotic Motion
    - Tee Ball - optimized motor task planning
    - Swimming, hopping - robust performance on a wide variety of tasks: learning simulated robotic swimming, hopping, and walking gaits; and playing Atari games using images of the screen as input.

Advantages/Challenges with policy gradient

- Advantages:
    - Can easily defeat a human in a game that prioritizes short term frequent rewards, i.e: Pong, Flappy bird.
- Challenges:
    - Have to actually experience the reward function.
    - Humans can figure out what is likely to give rewards without ever actually experiencing the rewarding or unrewarding transition.

Demo

Group Dynamics

DQN

- Kenny, John

PG

- Yash, Matt

Metrics

Training

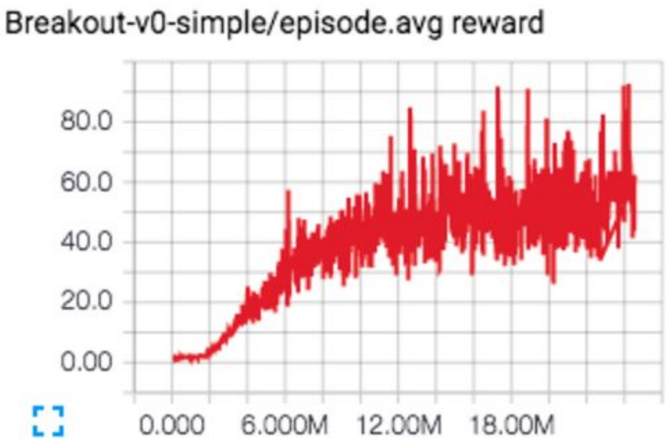
- 5000 Episodes

DQN vs PG

- 100 episodes
  - Episodes won
  - Running mean of rewards

Results

- In progress



Challenges & Reflection

- Having time to test agents and debug
- Training takes a long while
- The two different architectures were complicated and difficult to learn
- Things to try differently:
    - Policy Gradient
        - change hyperparameters
        - use convolutional neural net for quicker learning

Questions

Pre-Process the observations

```python
def preprocess_observations(input_observation, prev_processed_observation, input_dimensions):
    """ convert the 210x160x3 uint8 frame into a 6400 float vector """
    processed_observation = input_observation[35:195]  # crop
    processed_observation = downsample(processed_observation)
    processed_observation = remove_color(processed_observation)
    processed_observation = remove_background(processed_observation)
    processed_observation[processed_observation != 0] = 1  # everything else (paddles, ball) just set to 1
    # Convert from 80 x 80 matrix to 1600 x 1 matrix
    processed_observation = processed_observation.astype(np.float).ravel()

    # subtract the previous frame from the current one so we are only processing on changes in the game
    if prev_processed_observation is not None:
        input_observation = processed_observation - prev_processed_observation
    else:
        input_observation = np.zeros(input_dimensions)
    # store the previous frame so we can subtract from it next time
    prev_processed_observations = processed_observation
    return input_observation, prev_processed_observations
```

Supervised vs. PG