Sep 25, 2016 · 11 min read

Write an AI to win at Pong from scratch with Reinforcement Learning

There's a huge difference between reading about Reinforcement Learning and actually implementing it.

In this post, you'll implement a Neural Network for Reinforcement Learning and see it learn more and more as it finally becomes good enough to beat the computer in Pong! You can play around with other such Atari games at the OpenAI Gym.

By the end of this post, you'll be able to do the following:

- Write a Neural Network from scratch.
- Implement a Policy Gradient with Reinforcement Learning.
- Build an AI for Pong that can beat the computer in less than 250 lines of Python.
- Use OpenAI gym.

Andrej Karpathy's final output

Sources

The code and the idea are all **tightly based on Andrej Karpathy's blog post**. The code in me_pong.py is intended to be a simpler to follow version of pong.py which was written by Dr. Karpathy.

Prerequisites and Background Reading

To follow along, you'll need to know the following:

- Basic Python
- Neural Network design and backpropogation
- Calculus and Linear Algebra

If you want a deeper dive into the material at hand, read the blog post on which all of this is based. This post is meant to be a simpler introduction to that material.

Great! Let's get started.

. . .

Setup

1. Open up the code to follow along.
2. Follow the instructions for installing OpenAI Gym. You may need to install cmake first.
3. Run **pip install -e .[atari]**
4. Go back to the root of this repository and open a new file called my_pong.py in your favorite editor.
5. Let's get to problem solving. Here's the problem:

Problem

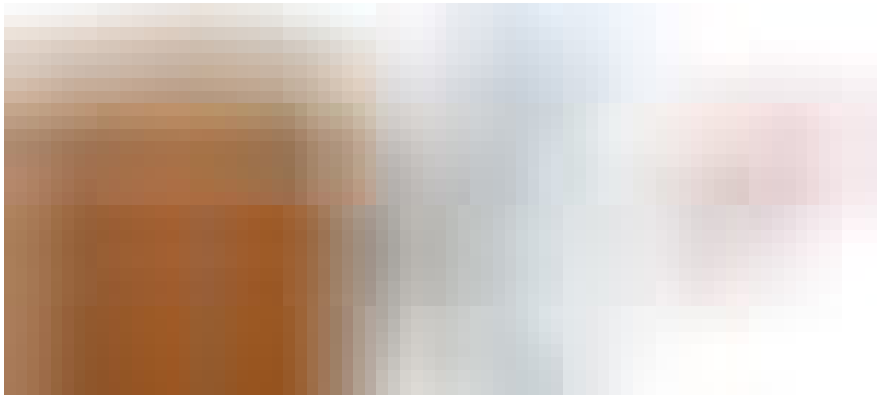
We are given the following:

- A sequence of images (frames) representing each frame of the Pong game.
- An indication when we've won or lost the game.
- An opponent agent that is the traditional Pong computer player.
- An agent we control that we can tell to move up or down at each frame.

Can we use these pieces to train our agent to beat the computer? Moreover, can we make our solution generic enough so it can be reused to win in games that aren't pong?

Solution

Indeed, we can! Andrej does this by building a Neural Network that takes in each image and outputs a command to our AI to move up or down.



The architecture Of Andrej's solution from his blog post.

We can break this down a bit more into the following steps:

Our Neural Network, based heavily on Andrej's solution, will do the following:

1. Take in images from the game and preprocess them (remove color, background, downsample etc.).
2. Use the Neural Network to compute a probability of moving up.
3. Sample from that probability distribution and tell the agent to move up or down.
4. If the round is over (you missed the ball or the opponent missed the ball), find whether you won or lost.
5. When the episode has finished(someone got to 21 points), pass the result through the backpropagation algorithm to compute the gradient for our weights.
6. After 10 episodes have finished, sum up the gradient and move the weights in the direction of the gradient.
7. Repeat this process until our weights are tuned to the point where we can beat the computer. That's basically it! Let's start looking at how our code achieves this.

Ok now that we've described the problem and its solution, let's get to writing some code!

. . .

Code

We're now going to follow the code in me_pong.py. Please keep it open and read along! The code starts here:

```
def main():
```

Initialization

First, let's use OpenAI Gym to make a game environment and get our very first image of the game.

```
1  env = gym.make("Pong-v0")
2  observation = env.reset() # This gets us the image
```

Next, we set a bunch of parameters based off of Andrej's blog post. We aren't going to worry about tuning them but note that you can probably get better performance by doing so. The parameters we will use are:

- *batch_size*: how many rounds we play before updating the weights of our network.
- *gamma*: The discount factor we use to discount the effect of old actions on the final result. (Don't worry about this yet)
- *decay_rate*: Parameter used in RMSProp algorithm. (Don't worry about this yet)
- *num_hidden_layer_neurons:* How many neurons are in our hidden layer.
- *learning_rate:* The rate at which we learn from our results to compute the new weights. A higher rate means we react more to results and a lower rate means we don't react as strongly to each result.

```
batch_size = 10 # how many episodes to wait before moving the weights
gamma = 0.99 # discount factor for reward
decay_rate = 0.99
num_hidden_layer_neurons = 200 # number of neurons
input_dimensions = 80 * 80 # dimension of our observation images
learning_rate = 1e-4
```

Then, we set counters, initial values, and the initial weights in our Neural Network.

Weights are stored in matrices. Layer 1 of our Neural Network is a 200 x 6400 matrix representing the weights for our hidden layer. For layer 1, element w1_ij represents the weight of neuron i for input pixel j in layer 1.

Layer 2 is a 200 x 1 matrix representing the weights of the output of the hidden layer on our final output. For layer 2, element w2_i represents the weights we place on the activation of neuron i in the hidden layer.

We initialize each layer's weights with random numbers for now. We divide by the square root of the number of the dimension size to normalize our weights.

```
weights = {
  '1': np.random.randn(num_hidden_layer_neurons,input_dimensions) /np.sqrt(input_dimensions),
  '2': np.random.randn(num_hidden_layer_neurons) / np.sqrt(num_hidden_layer_neurons)
}
```

Next, we set up the initial parameters for RMSProp (a method for updating weights that we will discuss later). Don't worry too much about understanding what you see below. I'm mainly bringing it up here so we can continue to follow along the main code block.

```
# To be used with rmsprop algorithm (http://sebastianruder.com/optimizing-gradient-descent/index.html#rmsprop)
expectation_g_squared = {}
g_dict = {}
for layer_name in weights.keys():
  expectation_g_squared[layer_name] = np.zeros_like(weights[layer_name])
  g_dict[layer_name] = np.zeros_like(weights[layer_name])
```

We'll need to collect a bunch of observations and intermediate values across the episode and use those to compute the gradient at the end based on the result. The below sets up the arrays where we'll collect all that information.

```
episode_hidden_layer_values, episode_observations, episode_gradient_log_ps, episode_rewards = [], [], [], []
```

Ok we're all done with the setup! If you were following, it should look something like this:

```
env = gym.make("Pong-v0")
observation = env.reset() # This gets us the image

# hyperparameters
episode_number = 0
batch_size = 10
gamma = 0.99 # discount factor for reward
decay_rate = 0.99
num_hidden_layer_neurons = 200
input_dimensions = 80 * 80
learning_rate = 1e-4

episode_number = 0
reward_sum = 0
running_reward = None
prev_processed_observations = None

weights = {
    '1': np.random.randn(num_hidden_layer_neurons, input_dimensions) / np.sqrt(input_dimensions),
    '2': np.random.randn(num_hidden_layer_neurons) / np.sqrt(num_hidden_layer_neurons)
}

# To be used with rmsprop algorithm (http://sebastianruder.com/optimizing-gradient-descent/index.html#rmsprop)
expectation_g_squared = {}
g_dict = {}
for layer_name in weights.keys():
    expectation_g_squared[layer_name] = np.zeros_like(weights[layer_name])
    g_dict[layer_name] = np.zeros_like(weights[layer_name])

episode_hidden_layer_values, episode_observations, episode_gradient_log_ps, episode_rewards = [], [], [], []
```

Phew. Now for the fun part!

Figuring out how to move

The crux of our algorithm is going to live in a loop where we continually make a move and then learn based on the results of the move. We'll put everything in a while block for now but in reality you might set up a break condition to stop the process.

The first step to our algorithm is processing the image of the game that OpenAI Gym passed us. We really don't care about the entire image - just certain details. We do this below:

```
while True:
```


```
while True:
    env.render()
    processed_observations, prev_processed_observations = preprocess_observations(observation, prev_processed_observations, input_dimensions)
```


Let's dive into preprocess_observations to see how we convert the image OpenAI Gym gives us into something we can use to train our Neural Network. The basic steps are:

1. Crop the image (we just care about the parts with information we care about).
2. Downsample the image.
3. Convert the image to black and white (color is not particularly important to us).
4. Remove the background.
5. Convert from an 80 x 80 matrix of values to 6400 x 1 matrix (flatten the matrix so it's easier to use).
6. Store just the difference between the current frame and the previous frame if we know the previous frame (we only care about what's changed).

```
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
```

Now that we've preprocessed the observations, let's move on to actually sending the observations through our neural net to generate the probability of telling our AI to move up. Here are the steps we'll take:

```
hidden_layer_values, up_probability = apply_neural_nets(processed_observations, weights)
episode_observations.append(processed_observations)
episode_hidden_layer_values.append(hidden_layer_values)
```

How exactly does apply_neural_nets take observations and weights and generate a probability of going up? This is just the forward pass of the Neural Network. Let's look at the code below for more information:

```
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['1'], observation_matrix)
    hidden_layer_values = relu(hidden_layer_values)
    output_layer_values = np.dot(hidden_layer_values, weights['2'])
    output_layer_values = sigmoid(output_layer_values)
    return hidden_layer_values, output_layer_values
```

As you can see, it's not many steps at all! Let's go step by step:

1. Compute the unprocessed hidden layer values by simply finding the dot product of the weights[1] (weights of layer 1) and the

observation_matrix. If you remember, weights[1] is a 200 x 6400 matrix and observations_matrix is a 6400 x 1 matrix. So the dot product will give us a matrix of dimensions 200 x 1. We have 200 neurons so each row represents the output of one neuron.

2. Next, we apply a non linear thresholding function on those hidden layer values - in this case just a simple ReLU. At a high level, this introduces the nonlinearities that makes our network capable of computing nonlinear functions rather than just simple linear ones.

3. We use those hidden layer activation values to calculate the output layer values. This is done by a simple dot product of hidden_layer_values (200 x 1) and weights['2'] (1 x 200) which yields a single value (1 x 1).

4. Finally, we apply a sigmoid function on this output value so that it's between 0 and 1 and is therefore a valid probability (probability of going up).

Let's return to the main algorithm and continue on. Now that we have obtained a probability of going up, we need to now record the results for later learning and choose an action to tell our AI to implement:

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

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

We choose an action by flipping an imaginary coin that lands "up" with probability up_probability and down with 1 - up_probability. If it lands up, we choose tell our AI to go up and if not, we tell it to go down. We also

Having done that, we pass the action to OpenAI Gym via **env.step(action)**.

Ok we've covered the first half of the solution! We know what action to tell our AI to take. If you've been following along, your code should look like this:

```
while True:
    env.render()
    processed_observations, prev_processed_observations = preprocess_observations(observation, prev_processed_observations, input_dimensions)
    hidden_layer_values, up_probability = apply_neural_nets(processed_observations, weights)

    episode_observations.append(processed_observations)
    episode_hidden_layer_values.append(hidden_layer_values)

    action = choose_action(up_probability)

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

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

```
10
11      # carry out the chosen action
```

Now that we've made our move, it's time to start learning so we figure out the right weights in our Neural Network!

Learning

Learning is all about seeing the result of the action (i.e. whether or not we won the round) and changing our weights accordingly. The first step to learning is asking the following question:

- How does changing the output probability (of going up) affect my result of winning the round?

Mathematically, this is just the derivative of our result with respect to the outputs of our final layer. If **L** is the value of our result to us and **f** is the function that gives us the activations of our final layer, this derivative is just $\partial \mathbf{L}/\partial \mathbf{f}$.

In a binary classification context (i.e. we just have to tell the AI one of two actions, up or down), this derivative turns out to be

$$\partial L_i/\partial f_j = y_{ij} - \sigma(f_j)$$

Note that σ in the above equation represents the sigmoid function. Read the **Attribute Classification** section here for more information about how we get the above derivative. We simplify this further below:

```
∂L/∂f = true_label(0 or 1) − predicted_label(0 or 1)
```

After one action(moving the paddle up or down), we don't really have an idea of whether or not this was the right action. So we're going to cheat and treat the action we end up sampling from our probability as the correct action.

Our predicion for this round is going to be the probability of going up we calculated. Using that, we have that **∂L/∂f** can be computed by

```
# 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)
```

Awesome! We have the gradient per action.

The next step is to figure out how we learn after the end of an episode (i.e. when we or our opponent miss the ball and someone gets a point). We do this by computing the policy gradient of the network at the end of each episode. The intuition here is that if we won the round, we'd like our network to generate more of the actions that led to us winning. Alternatively, if we lose, we're going to try and generate less of these actions.

OpenAI Gym provides us the handy **done** variable to tell us when an episode finishes (i.e. we missed the ball or our opponent missed the ball). When we notice we are done, the first thing we do is compile all our observations and gradient calculations for the episode. This allows us to apply our learnings over all the actions in the episode.

```
# 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)
```

Next, we want to learn in such a way that actions taken towards the end of an episode more heavily influence our learning than actions

taken at the beginning. This is called discounting.

Think about it this way - if you moved up at the first frame of the episode, it probably had very little impact on whether or not you win. However, closer to the end of the episode, your actions probably have a much larger effect as they determine whether or not your paddle reaches the ball and how your paddle hits the ball.

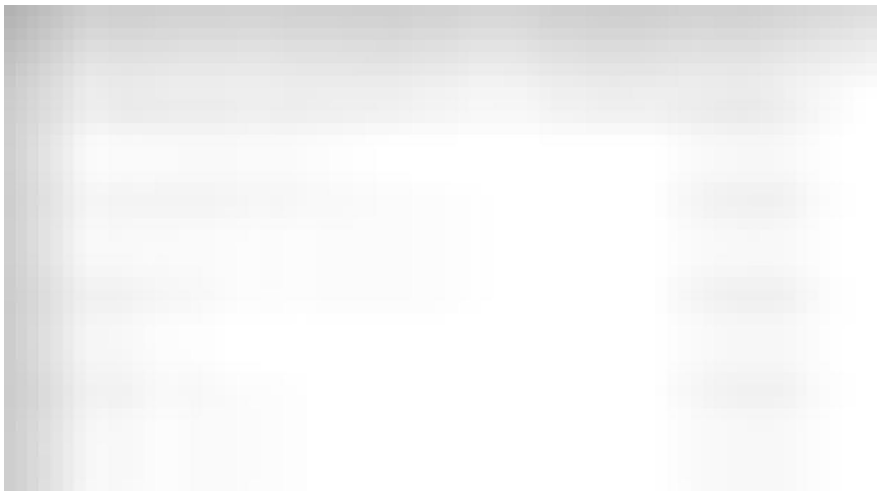
We're going to take this weighting into account by discounting our rewards such that rewards from earlier frames are discounted a lot more than rewards for later frames. After this, we're going to finally use backpropagation to compute the gradient (i.e. the direction we need to move our weights to improve).

```
# 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)
gradient = compute_gradient(
    episode_gradient_log_ps_discounted,
    episode_hidden_layer_values,
    episode_observations,
    weights
)
```

Let's dig in a bit into how the policy gradient for the episode is computed. This is one of the most important parts of Reinforcement Learning as it's how our agent figures out how to improve over time.

To begin with, if you haven't already, read this excerpt on backpropagation from Michael Nielsen's excellent free book on Deep Learning.

As you'll see in that excerpt, there are four fundamental equations of backpropogation, a technique for computing the gradient for our weights.



Four fundamental equations of backpropagation. Source: Michael Nielsen

Our goal is to find **∂C/∂w1 (BP4)**, the derivative of the cost function with respect to the first layer's weights, and **∂C/∂w2**, the derivative of the cost function with respect to the second layer's weights. These gradients will help us understand what direction to move our weights in for the greatest improvement.

To begin with, let's start with **∂C/∂w2**. If **a^l2** is the activations of the hidden layer (layer 2), we see that the formula is:

$$\frac{\delta C}{\delta w_2} = a^{l_2} \cdot \delta^L$$

Indeed, this is exactly what we do here:

```
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()
```

Next, we need to calculate **∂C/∂w1**. The formula for that is:

$$\frac{\delta C}{\delta w_1} = a^{l_1} \cdot \delta^{l_2}$$

and we also know that a^l1 is just our observation_values.

So all we need now is δ^l2. Once we have that, we can calculate $\partial C / \partial w_1$ and return. We do just that below:

```
delta_l2 = np.outer(delta_L, weights['2'])
delta_l2 = relu(delta_l2)
dC_dw1 = np.dot(delta_l2.T, observation_values)
return {
    '1': dC_dw1,
    '2': dC_dw2
}
```

If you've been following along, your function should look like this:

```
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['2'])
    delta_l2 = relu(delta_l2)
    dC_dw1 = np.dot(delta_l2.T, observation_values)
    return {
        '1': dC_dw1,
        '2': dC_dw2
    }
```

Computing The Gradient

With that, we've finished backpropagation and computed our gradients!

After we have finished batch_size episodes, we finally update our weights for our Neural Network and implement our learnings.

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

Update The Weights Every Batch Episodes

To update the weights, we simply apply RMSProp, an algorithm for updating weights described by Sebastian Reuder here.

We implement this below:

```
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
```

Updating Weights Using RMSProp

This is the step that tweaks our weights and allows us to get better over time.

This is basically it! Putting it altogether it should look like this.

You just coded a full Neural Network for playing Pong! Uncomment env.render() and run it for 3–4 days to see it finally beat the computer! You'll need to do some pickling as done in Andrej Karpathy's solution to be able to visualize your results when you win.

Performance

According to the blog post, this algorithm should take around 3 days of training on a Macbook to start beating the computer.

Consider tweaking the parameters or using Convolutional Neural Nets to boost the performance further.

Further Reading

If you want a further primer into Neural Networks and Reinforcement Learning, there are some great resources to learn more (I work at Udacity as the Director of Machine Learning programs):

- Andrej Karpathy's Original Blog Post
- Udacity's Free Deep Learning Course By Google
- Udacity's Free Supervised Learning Course By Georgia Tech
- Michael Neilsen's Deep Learning Book
- Sutton and Barto's Reinforcement Learning Book