Report Summary

Description:

Pong is a table tennis-themed arcade video game featuring simple two-dimensional graphics, manufactured by Atari and originally released in 1972. In Pong, one player scores if the ball passes by the other player. An episode is over when one of the players reaches 21 points.

In the OpenAI Gym framework version of Pong, there are three actions; an Agent (player) can take within the Pong Environment: remaining stationary, vertical translation up, and vertical translation down. However, if we use the method action_space.n we can realize that the Environment has 6 actions:
['NOOP', 'FIRE', 'RIGHT', 'LEFT', 'RIGHTFIRE', 'LEFTFIRE']

Input space:

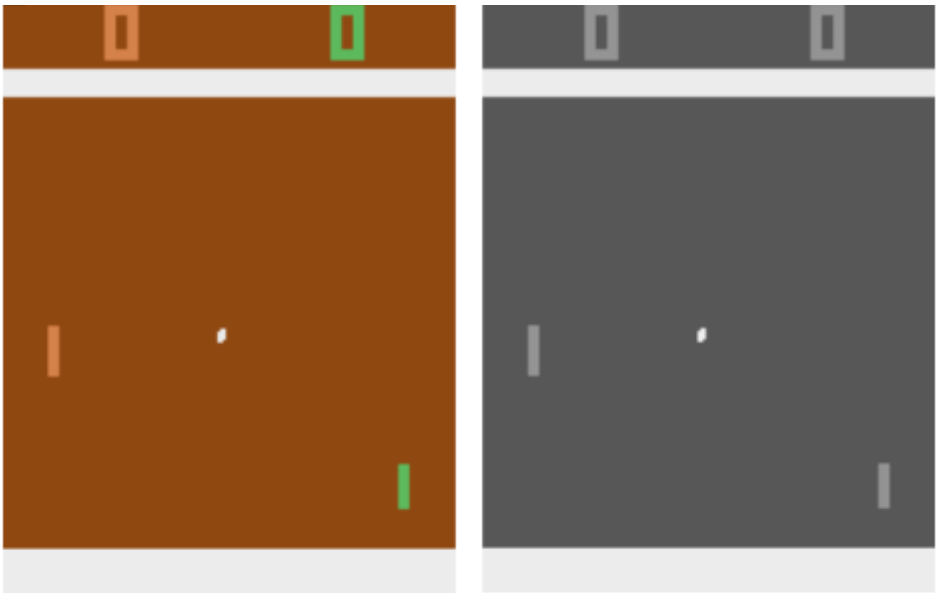
Atari games are displayed at a resolution of 210 by 60 pixels, with 128 possible colors for each pixel.

A detailed description of the implementation:

Working directly with raw Atari frames, which are 210 × 160 (in our case it depends on pygame screen) pixel images with a 128 color palette, can be computationally demanding, so we apply a basic preprocessing step aimed at reducing the input dimensionality.

1- We take the maximum of every pixel in the last two frames and use it as an observation. Some Atari games have a flickering effect and selecting two frames only fixes the problem.

2- Several transformations (as the already introduced the conversion of the frames to grayscale, and scale them down to a square 84 by 84 pixel block) are applied to the Atari platform interaction in order to improve the speed and convergence of the method.
3- We stack 4 subsequent frames together and use them as the observation at every state. For this reason, the preprocessing stacks four frames together resulting in a final state space size of 84 by 84 by 4.
4- We then input these frames into a Dueling DQN.



(a) Original Image

(b) Grayscale version of original

(c) Resized and cropped version of grayscale image

(d) 4 recent images of type (c) stacked on top of each other

figure x: output after each pre-processing step

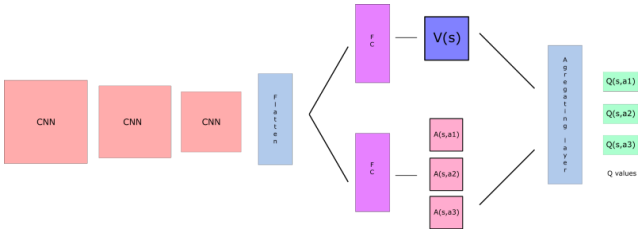
The learning algorithms:

- We use the dueling DQN algorithm to ameliorate the normal DQN.
  This algorithm is Deep Q learning:

Initialize network Q
Initialize target network $\hat{Q}$
Initialize experience replay memory D
Initialize the *Agent* to interact with the Environment
while *not converged* **do**
  `/* Sample phase`
  $\epsilon \leftarrow$ setting new epsilon with ϵ-decay
  Choose an action a from state s using policy ϵ-greedy(Q)
  Agent takes action a, observe reward r, and next state s'
  Store transition $(s, a, r, s', done)$ in the experience replay memory D
  if *enough experiences in* D **then**
    `/* Learn phase`
    Sample a random *minibatch* of N transitions from D
    for *every transition* $(s_i, a_i, r_i, s'_i, done_i)$ *in minibatch* **do**
      if $done_i$ then
        $y_i = r_i$
      else
        $y_i = r_i + \gamma \max_{a' \in \mathcal{A}} \hat{Q}(s'_i, a')$
      end
    end
    Calculate the loss $\mathcal{L} = 1/N \sum_{i=0}^{N-1} (Q(s_i, a_i) - y_i)^2$
    Update Q using the SGD algorithm by minimizing the loss $\mathcal{L}$
    Every C steps, copy weights from Q to $\hat{Q}$
  end
end

Like the standard DQN architecture, we have convolutional layers to process game-play frames. From there, we split the network into two separate streams, one for estimating the state-value (**V(s)**) and the other for estimating state-dependent action advantages (**A(a,s)**).
After the two streams, the last module of the network combines the state-value and advantage outputs.

The image below details the neural network architecture.



Now, we combine the two layers, process them in an isolated environment, then sum them back to the Q function.
The Dueling DQN objective is to split the Q function into two separate ones. V(a,s) and A(a,s)
The alpha, beta and theta variables are added to extract back the V(s) and the A(a,s) from the Q function; after splitting the latter.
From:

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + A(s,a;\theta,\alpha)$$

To:

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + (A(s,a;\theta,\alpha) - \frac{1}{|\mathcal{A}|}\sum_{a'} A(s,a;\theta,\alpha))$$

The agent hyperparameters:

gamma: 0.99 Discount factor gamma used for Q-learning update
epsilon: 1.0 The eps-greedy variable
mem_size: 50000 How many memories to store in the replay memory
eps_min: 0.1 Constant = the epsilon value threshold

batch_size: 32 How many transitions to sample each time experience is replayed.
replace: 10000 How many experience replay rounds to perform between each update to the target Q network.
eps_dec: 1e-5 The epsilon decay used to iteratively decrement the epsilon

The model architecture and hyperparameters:

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1          [-1, 32, 20, 20]           8,224
            Conv2d-2            [-1, 64, 9, 9]          32,832
            Conv2d-3            [-1, 64, 7, 7]          36,928
            Linear-4                [-1, 1024]       3,212,288
            Linear-5                 [-1, 512]         524,800
            Linear-6                   [-1, 1]             513
            Linear-7                   [-1, 6]           3,078
================================================================
Total params: 3,818,663
Trainable params: 3,818,663
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.11
Forward/backward pass size (MB): 0.17
Params size (MB): 14.57
Estimated Total Size (MB): 14.85
----------------------------------------------------------------
```

n_games: 300 (300 episodes)
n_steps: 18000 (maximum step number)
and the agent hyperparameters are included.

The analysed result:

It is clear that the model starts to perform well starting from 400k training steps.