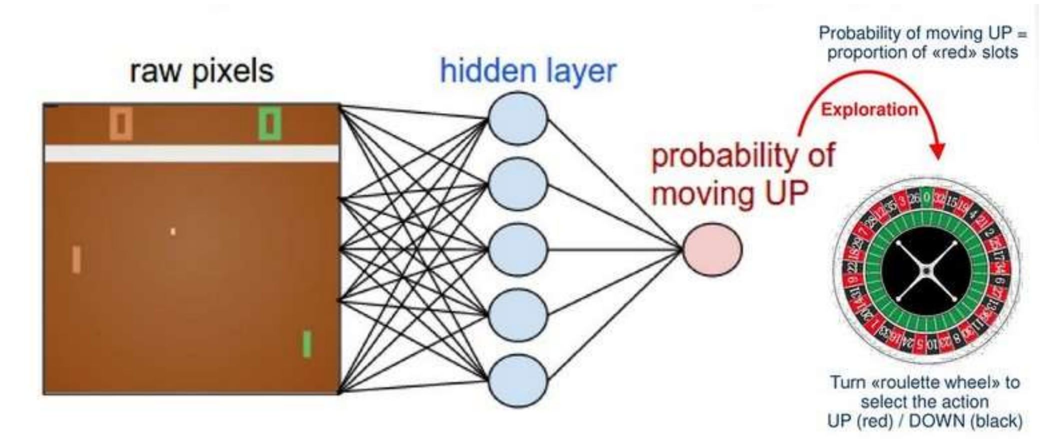
Ping Pong game

Jinrui Liu
Dana Tokmurzina
EIT Data Science

The following code allows playing the Pong game from a sequence of screen snapshots without providing explicitly any game-specific rules or general knowledge of the game.


raw pixels
hidden layer
probability of moving UP
Probability of moving UP = proportion of «red» slots
Exploration
Turn «roulette wheel» to select the action UP (red) / DOWN (black)

Game Description

The orange paddle is the anI opponent. The green is our agent. When the ball goes past the opponent, the agent gets a reward of +1 and if ball passes our agent then we get -1. A score of -21 happens when the opponent wins every game in an episode, +21 is the result when the trained agent wins all the games. We can move paddle UP or DOWN. Our goal was to add the convolutional layers (1D or 2D) before a dense layer and then compare the efficiency of the code.

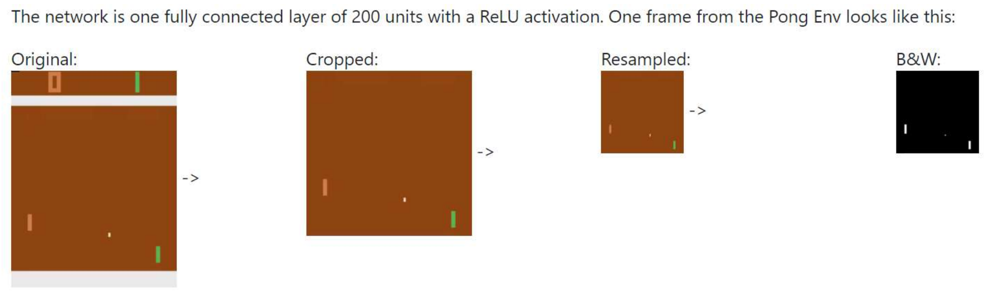
Pong is based on Markov Decision Process (MDP). So the game can be represented as the graph, where each node is one game and each edge is a possible (in general probabilistic) transition. Each edge also gives a reward, and the goal is to compute the optimal way of acting in any state to maximize rewards.
The input to our DRL system is 80 by 80 pixel information, and assume that pi and the score after each served the ball. The system learns a policy described as ***π(a|s; w)*** without supervision, it learns itself what action ***a*** are preferred *in a state* ***s****.* The learning is obtained by adjusting the neural network weights **W1 and W2***.*

Why adding a convolutional makes the algorithm more efficient?
The input that comes from the pong game is pixel information. The
RL algorithm should be able to perform a generalization task and be efficient (take less time and use less memory). So they should try to reduce as much as possible the complexity of the state space. Fully connected feed-forward layers require that every neuron of one layer is connected to another unit in the next layer. Every unit should have connections. So a huge amount of neurons (each pixel is a single neuron with 80 by 80 input and assuming each pixel represents 2 possible states, so 2^ 6400 states) with all the connections will result in a big convergence time
Also, the pixel information varies due to correlation, neighborhood pixels are correlated much more. And convolution applied to NN restricts that connections to be local. And a unit of the first layer hidden layer is a function of a small patch of the input image. So,

the connections between each image patch and the corresponding unit in the next layer are restricted to be equal over the whole image, which again reduces the number of parameters needed - more efficient [2].
Most of the time, an additional activation function is executed after the convolution operation to transform the elements of the feature map to the activations of the next layer. Multiple learned kernels can be used to produce multiple feature maps for detecting different features in the image. Stacking convolutional layers
enables the network to learn a hierarchical form of dependencies between pixels in distant regions of the original input image, which has greatly reduced dimensionality.

First, we're going to define a *policy network* that implements our player (or "agent"). This network will take the state of the game and decide what we should do.Policy gradients are preferred in that implementation as they are end-to-end. An explicit policy that directly optimizes the expected reward is provided.
s - input vector, п(a|s; w) - policy function, which predicts action a, given a state s and the parametrized weight w. The predicted y from the policy network indicates the best action according to the policy п. [1]

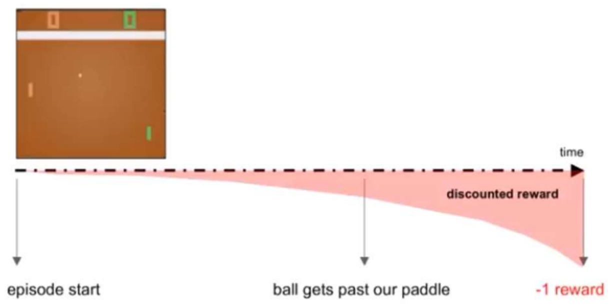
Pre-processing: `210x160x3 uint8 frame to 6400 (80x80) 1D float vector`


The network is one fully connected layer of 200 units with a ReLU activation. One frame from the Pong Env looks like this:

In general, we take images from the game and preprocess them: (remove color, background, downsample).
 I = I[35:195] taking only this image info, the only important pixels are the balls and the paddle
 I = I[::2,::2,0] - downsampling by a factor of 2, reducing the complexity of the image, taking alternating pixels and having the image resolution.
  I[I == 144] = 0 in these two lines we remove the background color (#1 and #2 backgrounds)
  I[I == 109] = 0
  I[I != 0] = 1 convert to black and white picture.
return I.astype(np.float).ravel() - return continuous flattened array.
Our input is the difference between the current frame and the last frame (used to express things like the direction of the ball).

Reward shaping: We want to train our agent in such a way that actions taken closer to the end of an episode more heavily influence our learning than actions taken at the start (so they have bigger discounted more). For example, determine whether or not paddle reaches the ball and how your paddle hits the ball. We use a discount factor of 0.99.

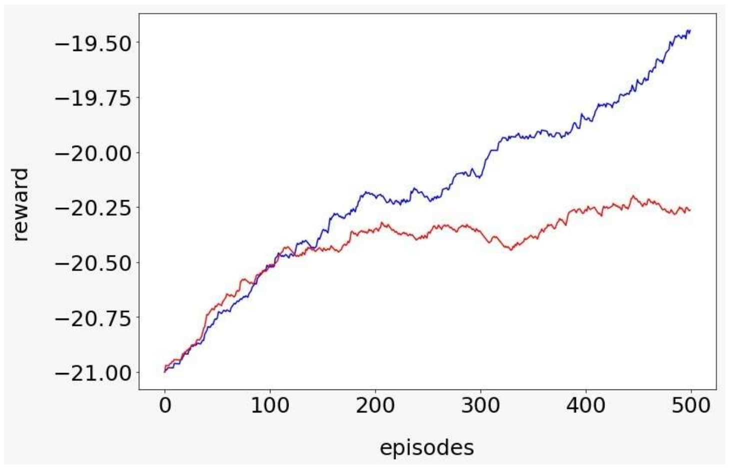
For our *discount_rewards*: transforms the list of rewards so that even actions that remotely lead to positive rewards are encouraged.



Algorithms: Use the Neural Network to compute a probability of moving up.
Our model (in the code Model1):
model.add(Reshape((1,80,80), input_shape=(input_dim,)))
model.add(Convolution2D(32, 9, 9, subsample=(4, 4), border_mode='same', activation='relu', init='he_uniform'))
model.add(Flatten()) because we used 2D input
model.add(Dense(16, activation='relu', init='he_uniform'))
model.add(Dense(number_of_inputs, activation='softmax'))- softmax activation is used for the output layer as we must obtain the probabilities of choosing the action (going UP). Because n=2, the softmax = sigmoid.
opt = Adam(lr=learning rate)
Adam is an optimization algorithm that can used instead of the classical stochastic gradient descent procedure to update network weights iterative based in training data. We set the lr to 0.001.
model.compile(loss='categorical_crossentropy', optimizer=opt)-standard loss for classification problems
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.

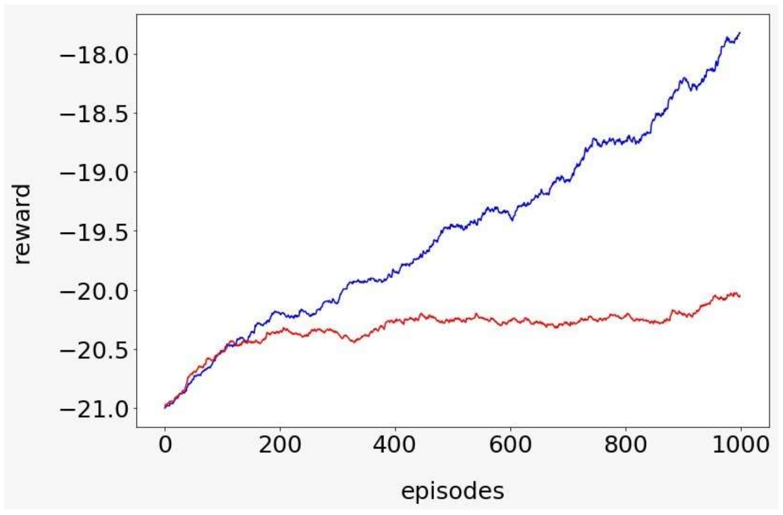
Obtained results:

Plotting the data for 500, 1000, 1500, 2000 iterations respectively.
Blue line: Model0-only dense layer.
Red line: Model1-2D convolutional layer before the dense layer.
Below are the reults of our iterations.
(1) 500 iterations



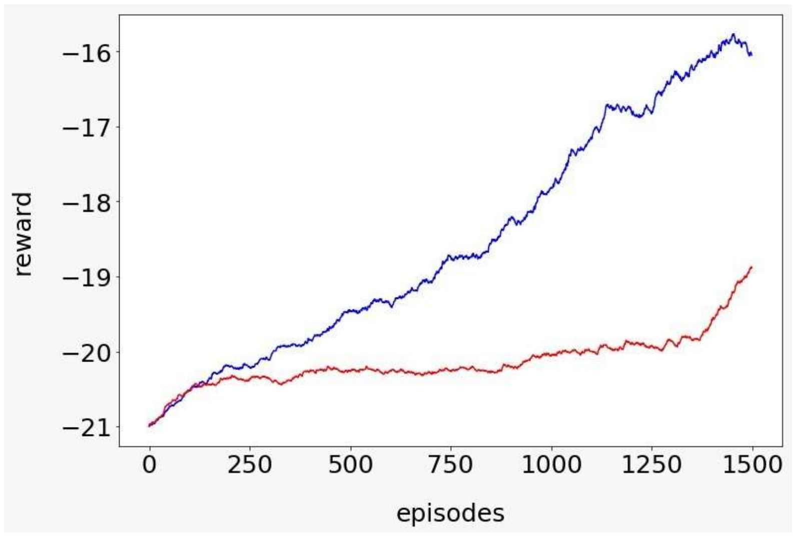
During the first 150 episodes, red and blue lines are very close to each other. After that, the blue line keeps a steady increasing trend, and 2D layers one tends to be flat.

(2) 1000



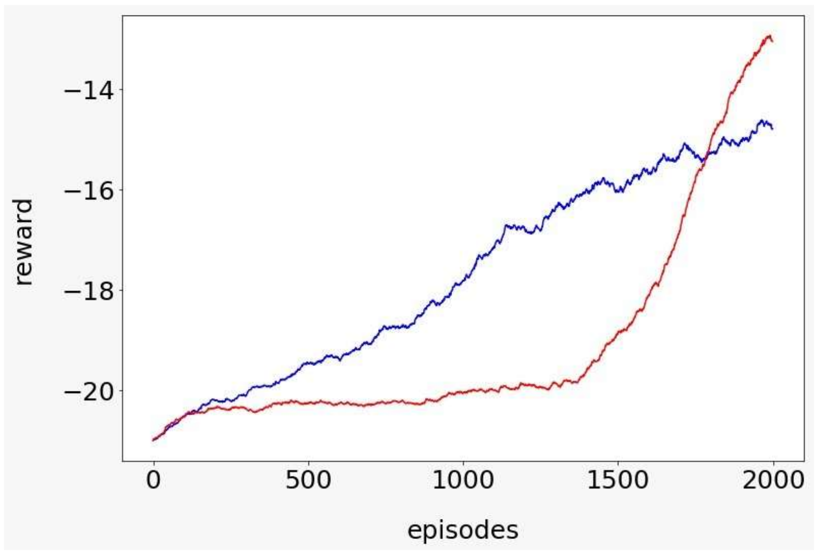
From this picture, we can see that the model with only a dense layer performs better than the model with 2D layer, which shows a convergent (stable) result.

(3) 1500 iterations



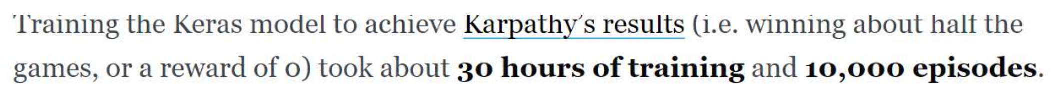
The red one is keeping flat during 150 episodes to 1300 episodes. And after 1300 episodes, it is obvious that red one climbs at a very fast speed. with only dense layer is still increasing constantly.

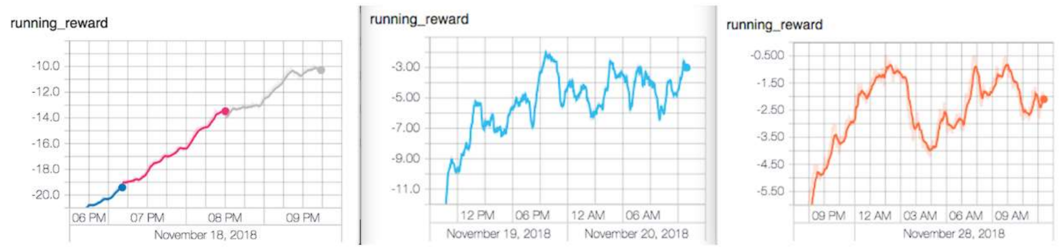
(4) 2000



Red line passed blue line after 1700 episodes and keeps increasing rapidly, while blue one stabilizes.

Conclusion:
Model without 2D convolution layer trains faster at the start of training and has a constant speed.
The model with 2D convolutional layer before dense layer starts slowly but after trained for enough episodes, it increases very fast and over the model with only the dense layer.
That can mean that the convolutional layer took time to learn the features from the images however later it outperforms other model.
We can compare our results with the Karpathy’s.


Training the Keras model to achieve Karpathy's results (i.e. winning about half the games, or a reward of 0) took about **30 hours of training** and **10,000 episodes**.



He needed 10000 episodes to train. Due to technical problems we can only run max of 2000, but could compare the performance of two models.
Code for testing was taken from the lecture.

Reference:
[1] *http://karpathy.github.io/2016/05/31/rl/*

[2]https://arxiv.org/pdf/1807.08452.pdf
[3]https://towardsdatascience.com/intro-to-reinforcement-learning-pong-92a94aa0f84d
[4]https://blog.floydhub.com/spinning-up-with-deep-reinforcement-learning/
[5]https://hollygrimm.com/rl_pg