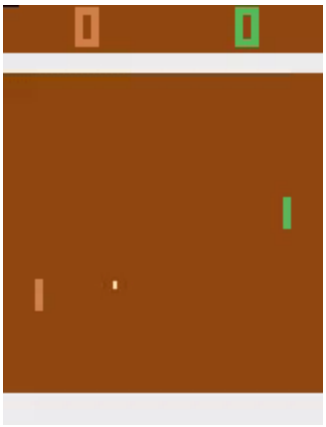
Instructions for project

Learning to play Pong with DQN

In this project, you will be implementing the Deep Q-Network (DQN) introduced by Mnih et al. (2015) (Links to an external site.) and eventually learn to play Pong. The project consists of three phases: understanding the algorithm, implementing and testing the algorithm on a simple problem, and using it to learn to play Pong. Please note that learning to play Pong takes several hours and you may need to run many experiments, so it is important that you start early.



Deadline for project report: May 23.

Group work: You will work in groups with 4 people.

Questions about the project

Make use of consultation hours. You can also send an e-mail to john.moberg@it.uu.se

Deliverables

Your submission for this project should be your code and a short report. The report should include

1. Short description of DQN. Special focus on what new ideas that were used to make it work.
2. Results of your experiments.
3. Discussion.

Step 0: Getting ready for the project

Step 0.1: Join a group

Join a group. You find the groups under "Persons/Groups" on Studium. If you want help with finding a group, you can add yourself to the group "Help with finding a group".

Discuss with your group how you will communicate during the project. Each group is assigned a page on Studium, with a discussion forum etc, but if you prefer to use other ways to communicate you are of course allowed to do that.

Step 0.2: Read the papers

The first step is to read and understand the method that you will implement. It was first introduced in a 2013 paper (Links to an external site.) and further improved and elaborated upon in the Nature DQN paper (Links to an external site.) in 2015. We suggest reading both. In your final report, we want you to briefly describe how the Deep Q-learning method works and discuss the new ideas that makes the algorithm work.

Step 0.3: Make sure that you have the Atari-environments installed

We will use OpenAI gyms Atari-environments (Links to an external site.). To test that your installation include these you can use

```
import gym
```

```
env = gym.make('Pong-v0')
```

If this does not work, you can install it with

```
pip install gym[atari]
```

Step 0.4: Familiarize yourselves with PyTorch

In this project, you will use PyTorch (Links to an external site.) which is a popular library used for deep learning. Instructions for installing PyTorch can be found here (Links to an external site.). If you have a supported GPU, we recommend you to install PyTorch with CUDA. Otherwise you choose CPU (in this case we recommend you to use Google Colab to learn Pong, see Step 0.5).
PyTorch enables fast tensor operations on GPU, including automatic differentiation which is used to train neural networks. We provide most of the neural network code, but you will have to write code using PyTorch functions. If you are unfamiliar with PyTorch, please read this basic tutorial on tensors (Links to an external site.), which is primarily what you will be working with.

Overall, PyTorch is quite similar to NumPy but with some important differences. In particular, tensors can be either on CPU or GPU. In the provided code, you will find

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

which specifies a device that data can be stored on. In this case GPU if it is available, and CPU otherwise. When you create a tensor,

```
a = torch.tensor([1, 2, 3])
```

it will by default be stored on CPU unless you specify otherwise, either by

```
a = torch.tensor([1, 2, 3]).to(device)
```

or

```
a = torch.tensor([1, 2, 3], device=device)
```

For performance reasons, it is important that you do not move data too often. For example, it is much faster to have the full replay memory stored on GPU rather than having to move transitions to GPU each batch.

Step 0.5: Try Google Colab (Recommended)

If you do not have access to a GPU, we recommend you to use Google Colab (Links to an external site.) when you learn Pong in Step 3 (without a GPU this will take very long time).
With Google Colab, you can get free access to a GPU by going to `Runtime > Change runtime type` and selecting GPU. Note that notebooks aren't permanent and may be shut down, in which case you will lose your progress and saved models. For this reason, it may be a good idea to mount a Google Drive folder and save models there, see e.g., this answer on StackOverflow (Links to an external site.).

Step 1: Implement and test DQN

Download code skeleton download

DQN can be tricky to implement because it's difficult to debug and sensitive to the choice of hyperparameters. For this reason, it is advisable to start testing on a simple environment where it is clear if it works within minutes rather than hours.

You will be implementing DQN to solve CartPole-v0 (Links to an external site.).

To install the required packages, run `pip install -r requirements.txt`.

You will then have to fill in the missing code in the code skeleton, and make sure that it work. When you have completed the code skeleton, you run your code with

```
python train.py --env CartPole-v0
```

When training is done you can use

`python evaluate.py --path models/"Name of model.pt"`

to evaluate the agent you have trained.

Your final agent should have a mean return of +200.0 over several episodes. If you want, you can also try the slightly more difficult CartPole-v1 environment which has longer episodes.

The provided hyperparameters (in `config.py`) should work, but we would like you to investigate the impact of hyperparameter choice. Try small and large values of `target_update_frequency` and different exploration parameters and describe your findings in the report.

In report: We want to see plots with episode returns during the course of training. Also include a discussion on how the training was affected by the choice of hyperparameters.

Step 2: Learn to play Pong

Your final task is to train DQN to play Pong-v0 (Links to an external site.). This will require adding some code that wasn't necessary for CartPole, e.g., stacking observations to account for temporal information. We provide some code snippets and hints below. Note that learning to play Pong with DQN takes **4-6 hours** using a GPU so it is important that you start early.

You are not required to produce a perfect agent, but it must be clear that it has learned something.

In the report: At least a plot with episode returns during the course of training. Discussion on your experience with implementing and using DQN for training a Pong-playing agent.

Preprocessing frames

A convenient way to deal with preprocessing is to wrap the environment with `AtariPreprocessing` from `gym.wrappers` as follows:

```
env = AtariPreprocessing(env, screen_size=84, grayscale_obs=True, frame_skip=1, noop_max=30)
```

You should also rescale the observations from 0-255 to 0-1.

Stacking observations

The current frame doesn't provide any information about the velocity of the ball, so DQN takes multiple frames as input. At the start of each episode, you can initialize a frame stack tensor

```
obs_stack = torch.cat(obs_stack_size * [obs]).unsqueeze(0).to(device)
```

When you receive a new observation, you can update the frame stack with

```
next_obs_stack = torch.cat((obs_stack[:, 1:, ...], obs.unsqueeze(1)), dim=1).to(device)
```

and store it in the replay buffer as usual.

Policy network architecture

We recommend using the convolutional neural network (CNN) architecture described in the Nature DQN paper (Links to an external site.). The layers can be initialized with

```
self.conv1 = nn.Conv2d(4, 32, kernel_size=8, stride=4, padding=0)

self.conv2 = nn.Conv2d(32, 64, kernel_size=4, stride=2, padding=0)

self.conv3 = nn.Conv2d(64, 64, kernel_size=3, stride=1, padding=0)

self.fc1 = nn.Linear(3136, 512)

self.fc2 = nn.Linear(512, self.n_actions)
```

and we use ReLU activation functions as previously. `nn.Flatten()` may be helpful to flatten the outputs before the fully-connected layers.

Hyperparameters

We suggest starting with the following hyperparameters:

Observation stack size: 4

Replay memory capacity: 10000

Batch size: 32

Target update frequency: 1000

Training frequency: 4

Discount factor: 0.99

Learning rate: 1e-4

Initial epsilon: 1.0

Final epsilon: 0.01

Anneal length: 10**6

While these should work, they are not optimal and you may play around with hyperparameters if you want.

Mapping actions (optional)

Since all Atari environments use the same action space, Pong has some redundant actions. Concretely, actions 2/4 and 3/5 makes the paddle go down and up respectively,

while action 0 & 1 do nothing. Learning will be significantly faster if you make use of this knowledge by letting the DQN have just two outputs that represent the Q-values of actions 2 and 3.

Step 2b: Try other Atari games (not mandatory)

Ideally, your method should be able to learn all Atari games with the same hyperparameters!

If you have time, it may be fun to try to learn how to play other Atari games! Try for example Breakout-v0.

Step 3: Write your report

See "Deliverables" above for more information about this part.

Frequently asked questions

There's an undefined "obs_stack" variable in evaluate.py, what is this?

This is a typo, it should just be "obs". However, the Pong environment requires observation stacking, and so for step 2 you will have to make changes also in evaluate.py even though there are no explicit TODO's there.

PreviousNext