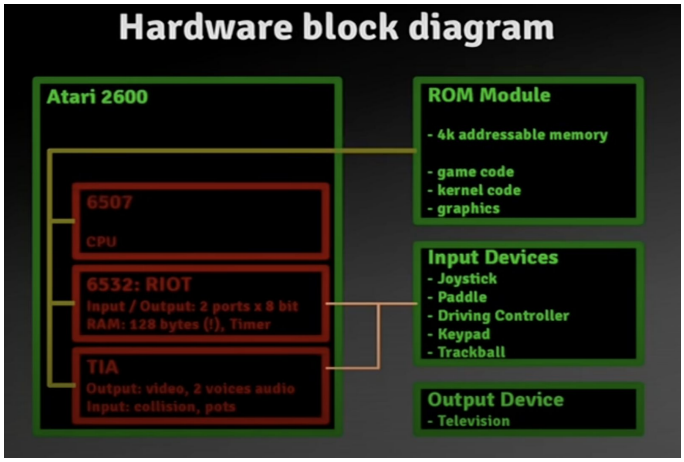
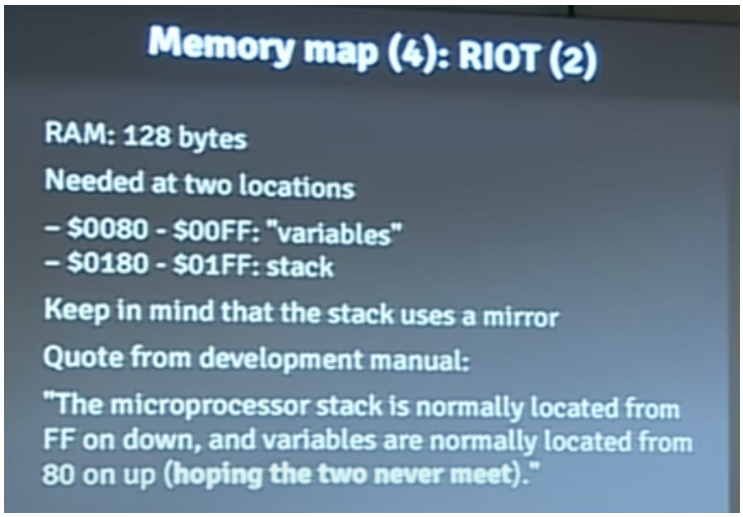
Atari 2600 Overview

This is a great overview of “retro” computing for Atari 2600. No need to watch it all just jump to the “highlights” below.

At 8:35 you see what the Atari machine looks like.
At 26:00 you see the hardware overview here is a key diagram. Note the RAM is just 128 bytes (not kilobytes, not megabytes). The ROM (Read ONLY Memory) is where the program is stored which is in a cartridge (last century RAM was very expensive). For the purpose of the first question in the assignment RAM refers to the 128 bytes on the Atari console.



At 32:03 you see an overview of the 128 Bytes of RAM (see if you get the joke from the manual)
The relevant screen shot is here:



Note at 34:00 they discuss ways to add more RAM onto the ROM module as a work-around. This is ingenious as the ROM cartridge had no read/write line **but ignore this for the assignment** and assume you only have 128 Bytes of memory if you use that as a state.

At 38:04 they discuss how to write a program and in particular how to write to the screen. This will give you an idea what the RAM is used for.

An Overview of The Logic Of Our AI Agent In The Starter Code

We've implemented the basic approach to deep Q-learning as in the Google Deepmind paper (see Deep Learning Module) in run_dqn_pong.py. The approach is straight forward (I've copied the code below for your reference).

1) You learn from num_frames "episodes" before exiting set to 1M by default and that should be enough to win close to 20 to 0 (see plot downstream).
2) We've implemented a classic exploration vs exploitation strategy set in epsilon_by_frame which explores initially a lot and exploits in later frames.
3) A major innovation from the DeepMind paper was the notion of a replay buffer. That is you don't iterate play-learn-play-learn ... instead:
    a) You play enough frames (10K as per the code) to generate a replay buffer. Only after that point do you start to learn (i.e. back-propagate) weight updates.
    b) You learn from sampling from the replay buffer to get training "episodes" which are just a frame and the next frame
    c) You learn from these two frames
    d) See the paper to explain why this is desirable.
4) Note \gamma is set to 0.99 because a point may take hundreds of frames to play so the reward can be well down the track from the current action.

Code Snippet From run_dqn_pong.py

```
for frame_idx in range(1, num_frames + 1):
    #print("Frame: " + str(frame_idx))

    epsilon = epsilon_by_frame(frame_idx)
    action = model.act(state, epsilon)

    next_state, reward, done, _ = env.step(action)
    replay_buffer.push(state, action, reward, next_state, done)

    state = next_state
    episode_reward += reward

    if done:
        state = env.reset()
        all_rewards.append((frame_idx, episode_reward))
        episode_reward = 0
```

```
if len(replay_buffer) > replay_initial:
    loss = compute_td_loss(model, target_model, batch_size, gamma, replay_buffer)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
    losses.append((frame_idx, loss.data.cpu().numpy()))

if frame_idx % 10000 == 0 and len(replay_buffer) <= replay_initial:
    print('#Frame: %d, preparing replay buffer' % frame_idx)

if frame_idx % 10000 == 0 and len(replay_buffer) > replay_initial:
    print('#Frame: %d, Loss: %f' % (frame_idx, np.mean(losses, 0)[1]))
    print('Last-10 average reward: %f' % np.mean(all_rewards[-10:], 0)[1])

if frame_idx % 50000 == 0:
    target_model.copy_from(model)
```

Quick Start Guide To Programming Gym

You are free to investigate more about gym, but for the purposes of this project you don't need to know too much about the architecture etc. You will have to understand the syntax of how to generate q-values etc. Unfortunately, the gym overview articles (documentation) seem to have been deleted from their website. We are in the process of creating our own and will put them here. We will go over some details to get you started during the section on Monday.

How Good Will My AI Get

Here you see a plot of training "episodes/frames" (x-axis) against how well the AI performed over 10 games in terms of score. You see initially it's hopeless losing 0 to 20 but after 1M frames it wins nearly 20 to 0.

I've just implemented a very basic agent, nothing fancy at all. It took about 1 hour to train on my GPU server and will take you about 4-6 hours.

Importantly, the performance does not increase monotonically. This is due to a number of reasons the easiest being that the proof of convergence only talks about Q-values not performance.

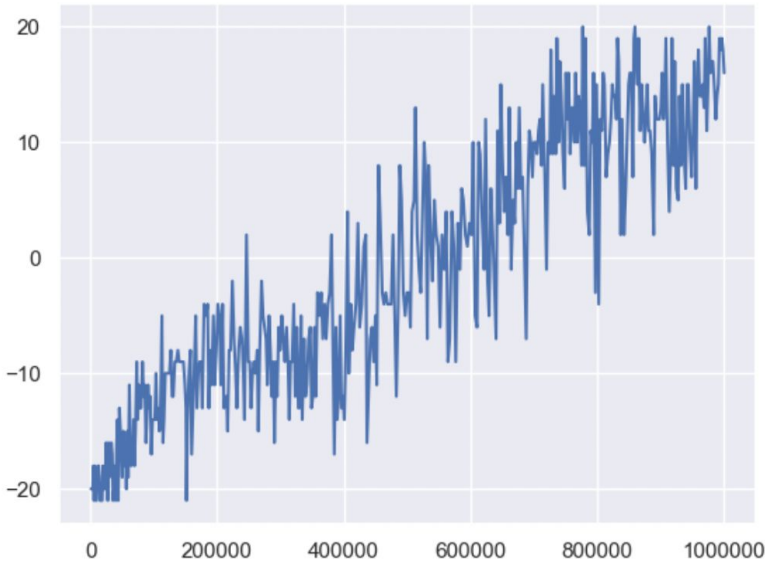


Fig: Training Frame vs Mean Difference in Scores (yours - opponent) Over 10 Games

A basic Python, Numpy, Pytorch introduction

Python data types:
Please check out the link.
'Dictionary' is a {key: value ...} structure, given the 'key', you can access the value by dict[key].

```
# enumerate the keys of the dictionary
for i, k in enumerate(example_dict):
    print(i, k, example_dict[k])

# enumerate through both keys and values this is the way
for i, (k, v) in enumerate(example_dict.items()):
    print(i, k, v)
```

One useful feature for debugging is string formattin.

```
>>> print('We are the {} who say "{}!"'.format('knights', 'Ni'))
We are the knights who say "Ni!"
```

Numpy structures:
NumPy is a library for the Python programming language, adding support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays.
You can read the link to get familiar with basic numpy operations.

The feature that makes numpy so powerful is **broadcasting.** The term broadcasting describes how numpy treats arrays with different shapes during arithmetic operations. Here is a brief introduction to it.

Sometimes you need to add a newaxis to a numpy array or squeeze the array. You can try:

```
# example is an existing array
# add a new axis
example[:, np.newaxis]
# resize the array
example.resize((2, 1))
# squeeze the array
np.squeeze(example)
```

Pytorch tensor types:
A torch.Tensor is a multi-dimensional matrix containing elements of a single data type.
The tensor types in pytorch are defined here. Note that the data type of tensors in pytorch could not be the same as the data type of numpy arrays.

Pytorch numpy bridge:

```
# tensor to numpy
a = torch.ones(5)
b = a.numpy()
# numpy to tensor
a = np.ones(5)
b = torch.from_numpy(a)
```

Cuda semantics:
To use high performance Gpus, please check out the documents.
Remember, a tensor on cpu is not the same type as a tensor on gpu.
When training or testing a model, you have to move both the model and the data to the same gpu device.

```
model = NeuralNetwork().to(device)
inputs, labels = data[0].to(device), data[1].to(device)
```

You can move it back to cpu by

```
model.cpu()
```

Saving & Loading models:
One recommended way to save & load models is

```
torch.save(model.state_dict(), PATH)

model = TheModelClass(*args, **kwargs)
model.load_state_dict(torch.load(PATH))
```

When training or testing (evaluating) the model, you have to set the model into training/evaluating modes respectively.

```
model.train()
or
model.eval()
```

Before testing, remember to set the model into “no gradient mode”

```
with torch.no_grad():
    for data in testloader:
        ……
```

You can also remove some tensor from the gradient computation graph by ‘detach’ which stops the gradients from flowing through the tensor.

Finally, here is a simple deep-q-learning tutorial with pytorch.

Troubleshooting with graphical displays:

Normally you will SSH to the machine but the gym environment will give you an error because it can’t write to the display.

The easiest solution is to install an X-server on your machine and then ssh -x to the machine. On my macbook Air M1 I installed xQuartz and then ssh -x and it worked fine.