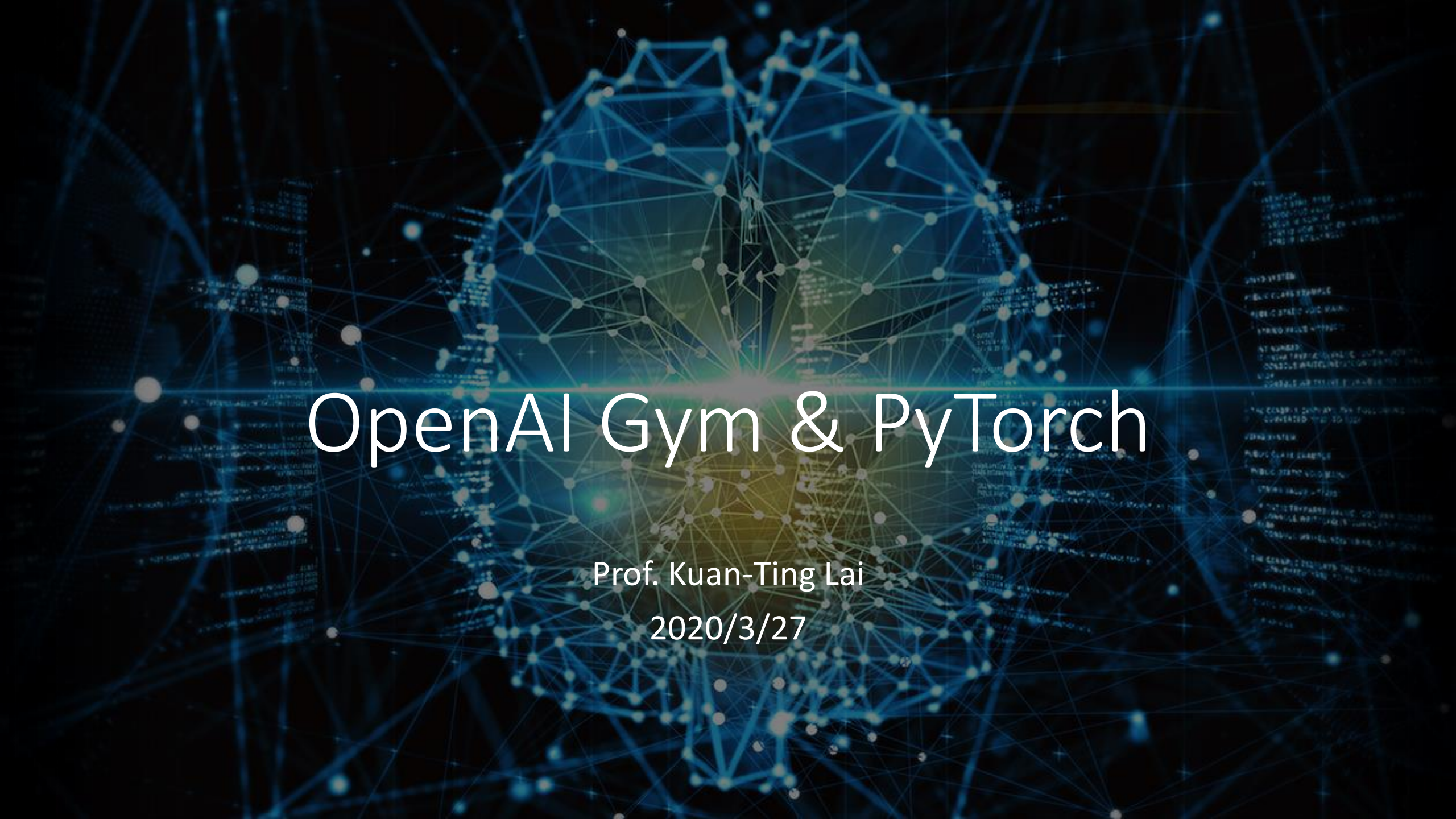


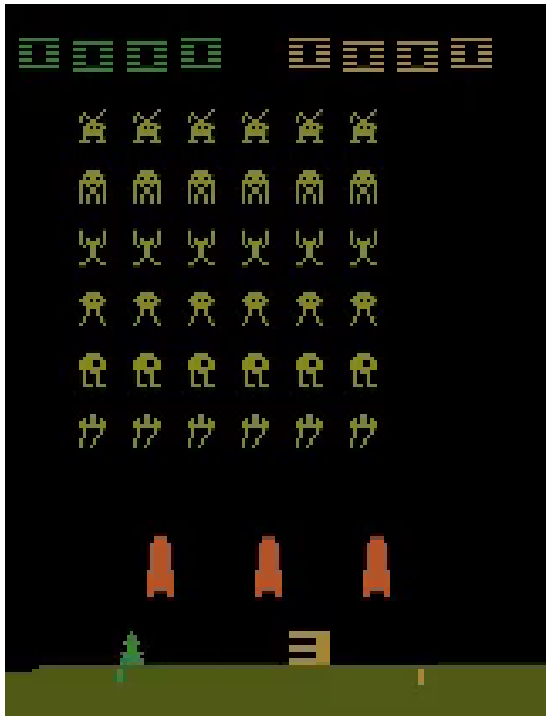
OpenAI Gym & PyTorch

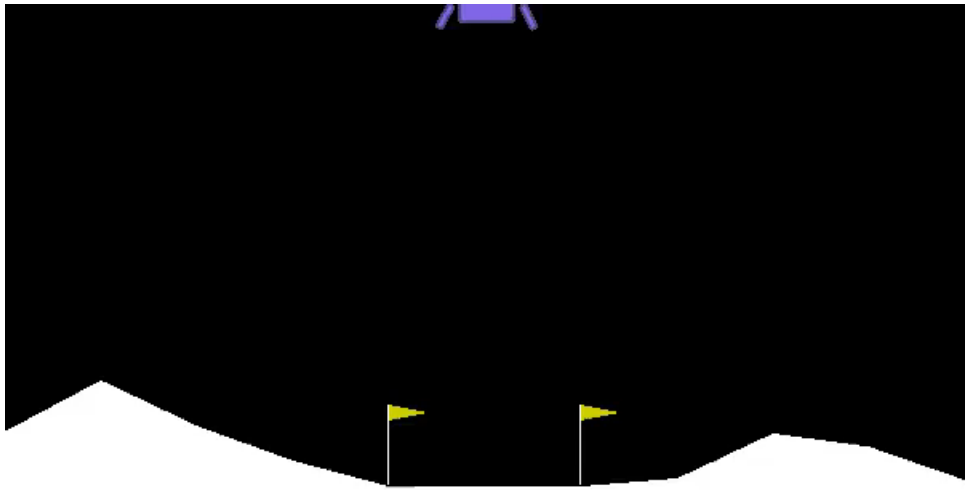
Prof. Kuan-Ting Lai

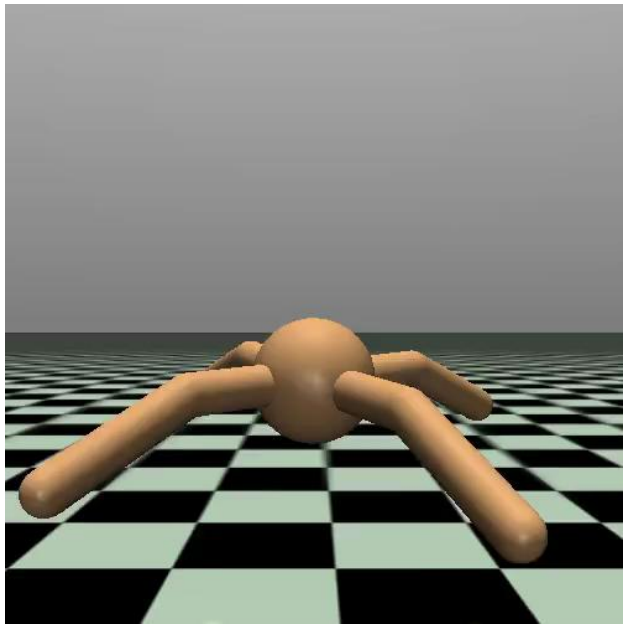
2020/3/27

OpenAI Gym

- https://gym.openai.com/



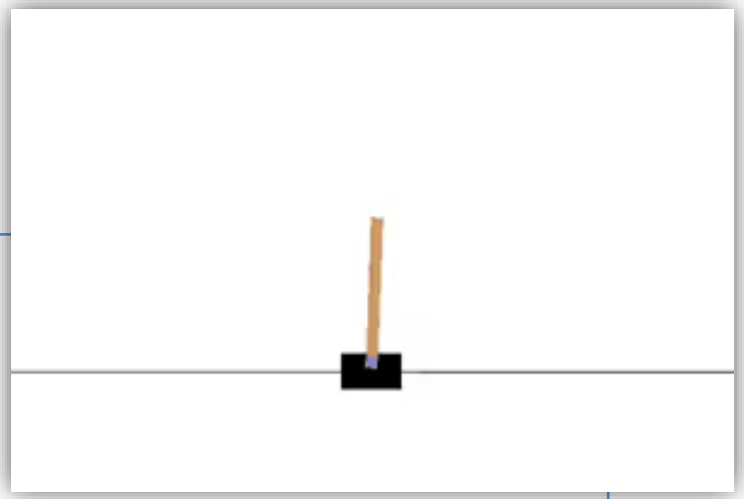




Install OpenAI Gym

- Install via pip (Python 3.5+)
  - pip install gym
- Building from Source
  - git clone https://github.com/openai/gym
  - cd gym
  - pip install -e .

Hello Gym Example

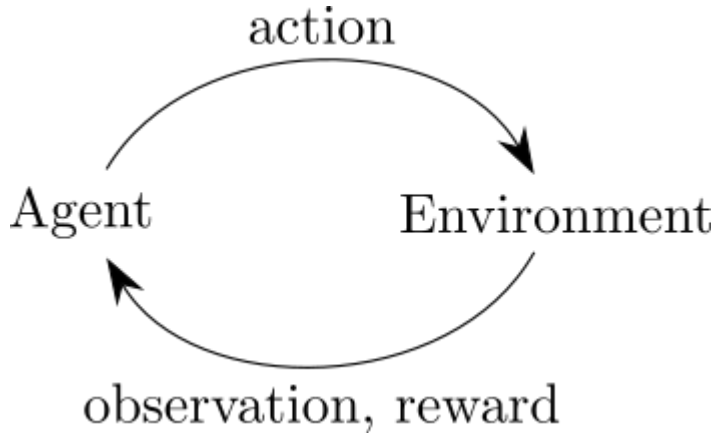


```
import gym
env = gym.make("CartPole-v1")
observation = env.reset()
for _ in range(1000):
  env.render()
  # your agent here (this takes random actions)
  action = env.action_space.sample()
  observation, reward, done, info = env.step(action)

  if done:
    observation = env.reset()

env.close()
```

https://gym.openai.com/docs/

Basics of OpenAI Gym

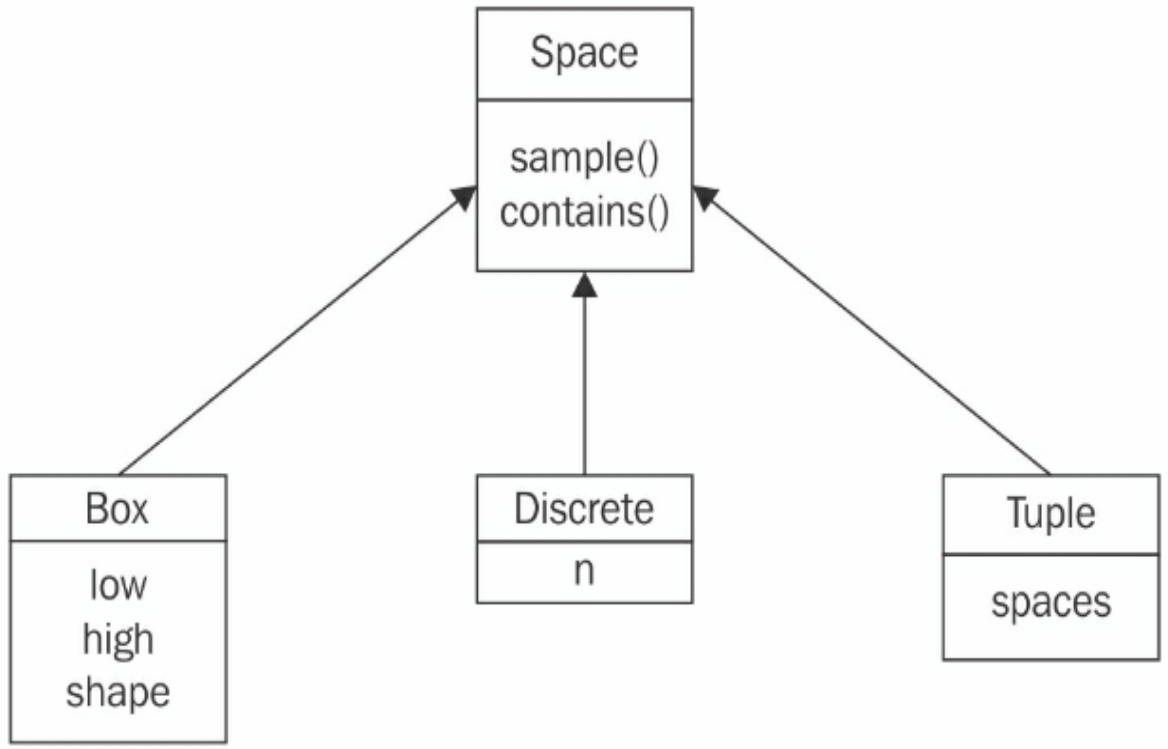
- observation (state S_t):
  - Observation of the environment. Ex: pixel data from a camera, joint angles and joint velocities of a robot, or the board state in a board game.
- reward (r_t)
  - amount of reward achieved by the previous action.
- done:
  - True indicates the episode has terminated
- info:
  - diagnostic information useful for debugging.



Action Space & Observation Space

- Action space (class Discrete)
  - Number of actions
- Observation Space (Box)
  - N-dimensional tensor

```
import gym env = gym.make('CartPole-v0')
print(env.action_space)
#> Discrete(2)
print(env.observation_space)
#> Box(4,)
```

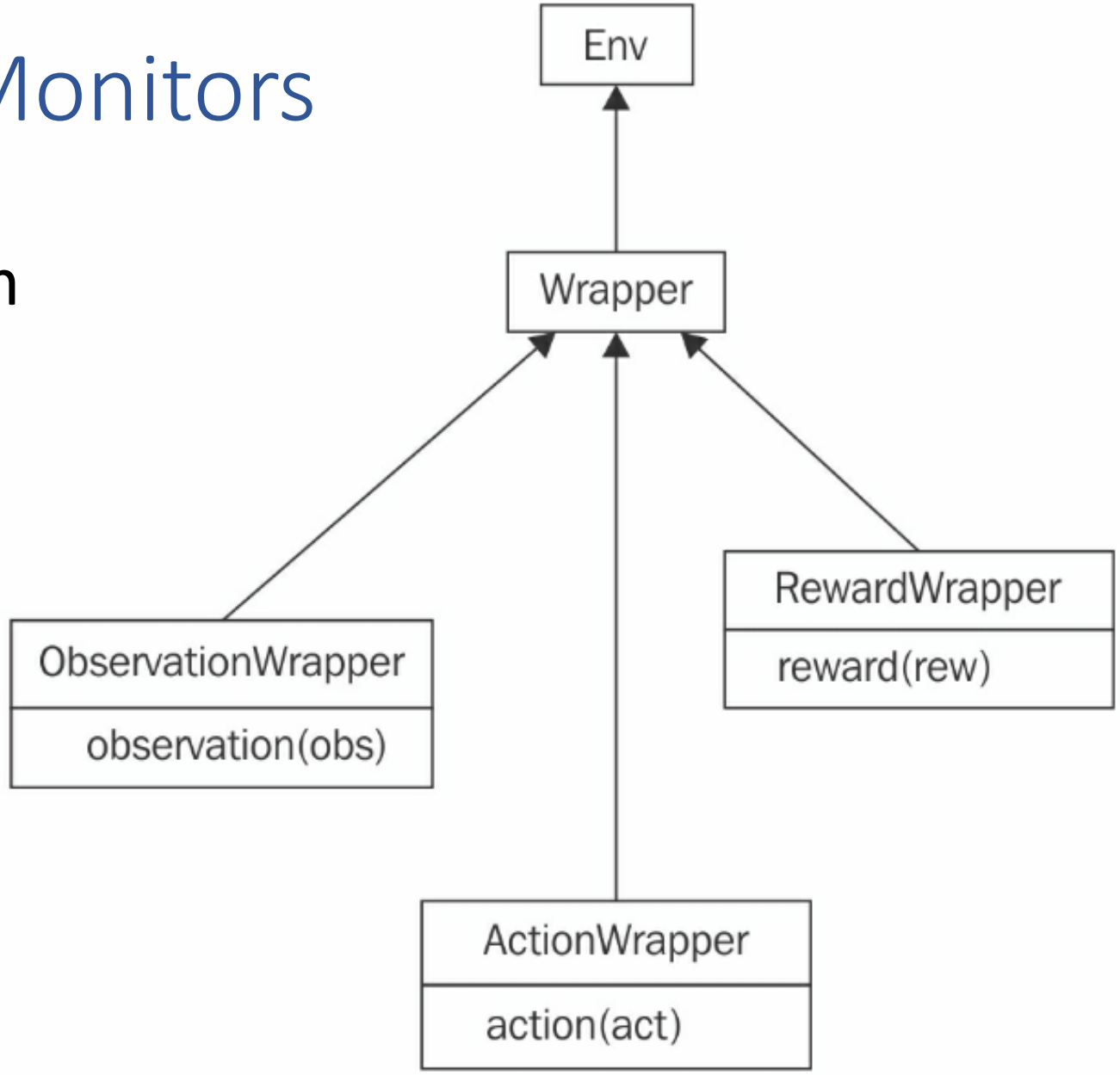


Available Environments

- Classic control and toy text
  - Small & famous RL tasks (Cart Pole, Mountain Car, …)
- Algorithmic
  - perform computations such as addition or reversing sequences.
  - These tasks are easy for a computer, the challenge is to learn these algorithms purely from examples.
- Atari
  - play classic Atari games.
- 2D and 3D robots
  - control a robot in simulation. These tasks use the MuJoCo physics engine.

Gym Wrappers and Monitors

- Class Wrapper inherits from Env, and allows users to override specific functions
  - ObservationWrapper
  - RewardWrapper
  - ActionWrapper



Implement ε greedy

https://github.com/PacktPublishing/Deep-Reinforcement-Learning-Hands-On/blob/master/Chapter02/03_random_actionwrapper.py

```
import gym
import random

class RandomActionWrapper(gym.ActionWrapper):
    def __init__(self, env, epsilon=0.1):
        super(RandomActionWrapper, self).__init__(env)
        self.epsilon = epsilon

    def action(self, action):
        if random.random() < self.epsilon:
            print("Random!")
            return self.env.action_space.sample()
        return action

if __name__ == "__main__":
    env = RandomActionWrapper(gym.make("CartPole-v0"))
```

Monitor

- Record the history of learning (requires FFMPEG!)

```
if __name__ == "__main__":
    env = gym.make("CartPole-v0")
    env = gym.wrappers.Monitor(env, "recording")
```

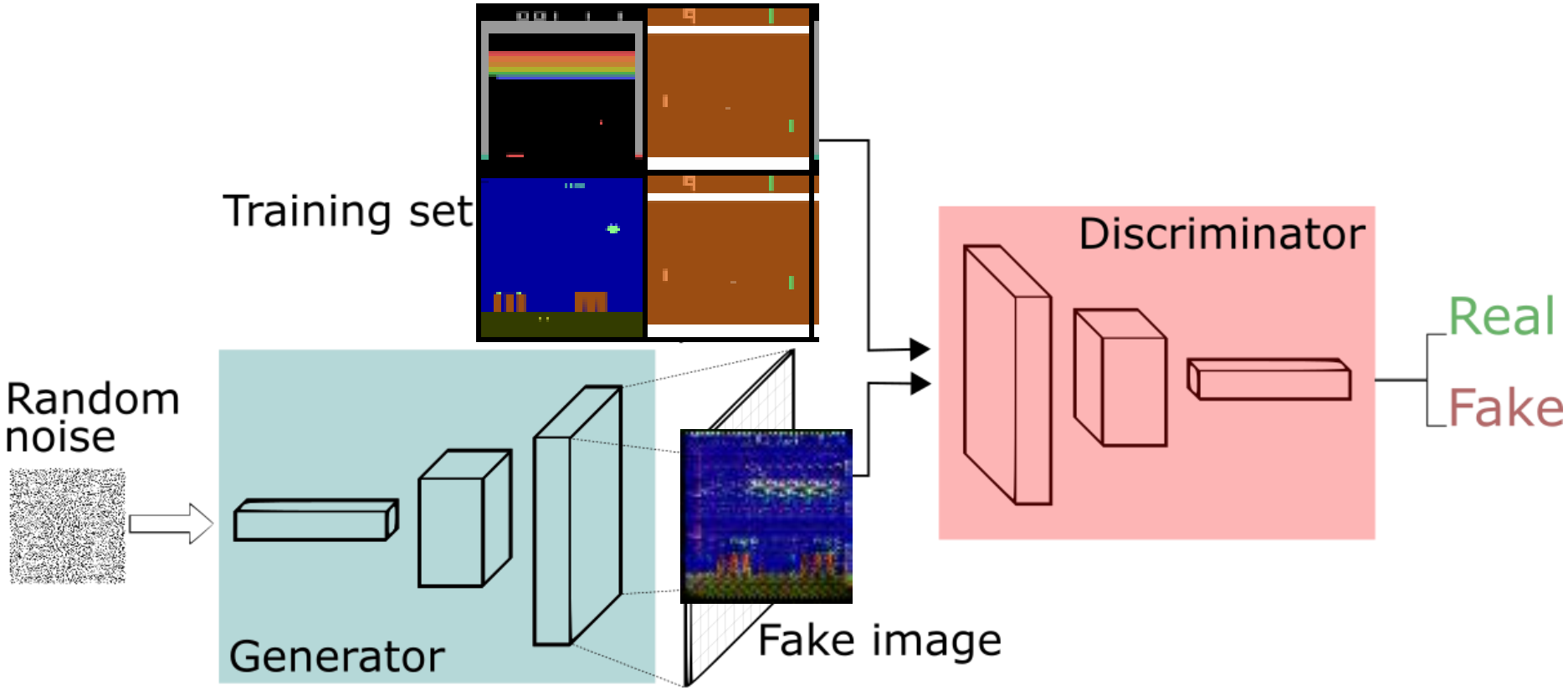
PyTorch Basics

- Tensors
- Gradients
- NN Building Blocks
- Custom Layers
- Loss functions and optimizers
- Monitoring with TensorBoard
- Example: GAN on Atari Images

GAN on Atari Images

- Using OpenAI Gym Atari observations to train a GAM generator

https://github.com/PacktPublishing/Deep-Reinforcement-Learning-Hands-On/blob/master/Chapter03/03_atari_gan.py



Generative Adversarial Network Architecture

Input Wrapper

```
class InputWrapper(gym.ObservationWrapper):
    """
    Preprocessing of input numpy array:
    1. resize image into predefined size
    2. move color channel axis to a first place
    """
    def __init__(self, *args):
        super(InputWrapper, self).__init__(*args)
        assert isinstance(self.observation_space, gym.spaces.Box)
        old_space = self.observation_space
        self.observation_space = gym.spaces.Box(self.observation(old_space.low),
                                                self.observation(old_space.high), dtype=np.float32)

    def observation(self, observation):
        # resize image
        new_obs = cv2.resize(observation, (IMAGE_SIZE, IMAGE_SIZE))
        # transform (210, 160, 3) -> (3, 210, 160)
        new_obs = np.moveaxis(new_obs, 2, 0)
        return new_obs.astype(np.float32)
```

Discriminator

```
class Discriminator(nn.Module):
    def __init__(self, input_shape):
        super(Discriminator, self).__init__()
        # this pipe converges image into the single number
        self.conv_pipe = nn.Sequential(
            nn.Conv2d(in_channels=input_shape[0], out_channels=DISCR_FILTERS,
                      kernel_size=4, stride=2, padding=1),
            nn.ReLU(),
            nn.Conv2d(in_channels=DISCR_FILTERS, out_channels=DISCR_FILTERS*2,
                      kernel_size=4, stride=2, padding=1),
            nn.BatchNorm2d(DISCR_FILTERS*2),
            nn.ReLU(),
            nn.Conv2d(in_channels=DISCR_FILTERS * 2, out_channels=DISCR_FILTERS * 4,
                      kernel_size=4, stride=2, padding=1),
            nn.BatchNorm2d(DISCR_FILTERS * 4),
            nn.ReLU(),
            nn.Conv2d(in_channels=DISCR_FILTERS * 4, out_channels=DISCR_FILTERS * 8,
                      kernel_size=4, stride=2, padding=1),
            nn.BatchNorm2d(DISCR_FILTERS * 8),
            nn.ReLU(),
            nn.Conv2d(in_channels=DISCR_FILTERS * 8, out_channels=1,
                      kernel_size=4, stride=1, padding=0),
            nn.Sigmoid()
        )
    def forward(self, x):
        conv_out = self.conv_pipe(x)
        return conv_out.view(-1, 1).squeeze(dim=1)
```

Generator

```
class Generator(nn.Module):
    def __init__(self, output_shape):
        super(Generator, self).__init__()
        # pipe deconvolves input vector into (3, 64, 64) image
        self.pipe = nn.Sequential(
            nn.ConvTranspose2d(in_channels=LATENT_VECTOR_SIZE, out_channels=GENER_FILTERS * 8,
                               kernel_size=4, stride=1, padding=0),
            nn.BatchNorm2d(GENER_FILTERS * 8),
            nn.ReLU(),
            nn.ConvTranspose2d(in_channels=GENER_FILTERS * 8, out_channels=GENER_FILTERS * 4,
                               kernel_size=4, stride=2, padding=1),
            nn.BatchNorm2d(GENER_FILTERS * 4),
            nn.ReLU(),
            nn.ConvTranspose2d(in_channels=GENER_FILTERS * 4, out_channels=GENER_FILTERS * 2,
                               kernel_size=4, stride=2, padding=1),
            nn.BatchNorm2d(GENER_FILTERS * 2),
            nn.ReLU(),
            nn.ConvTranspose2d(in_channels=GENER_FILTERS * 2, out_channels=GENER_FILTERS,
                               kernel_size=4, stride=2, padding=1),
            nn.BatchNorm2d(GENER_FILTERS),
            nn.ReLU(),
            nn.ConvTranspose2d(in_channels=GENER_FILTERS, out_channels=output_shape[0],
                               kernel_size=4, stride=2, padding=1),
            nn.Tanh()
        )
    def forward(self, x):
        return self.pipe(x)
```

Batch Data Generation

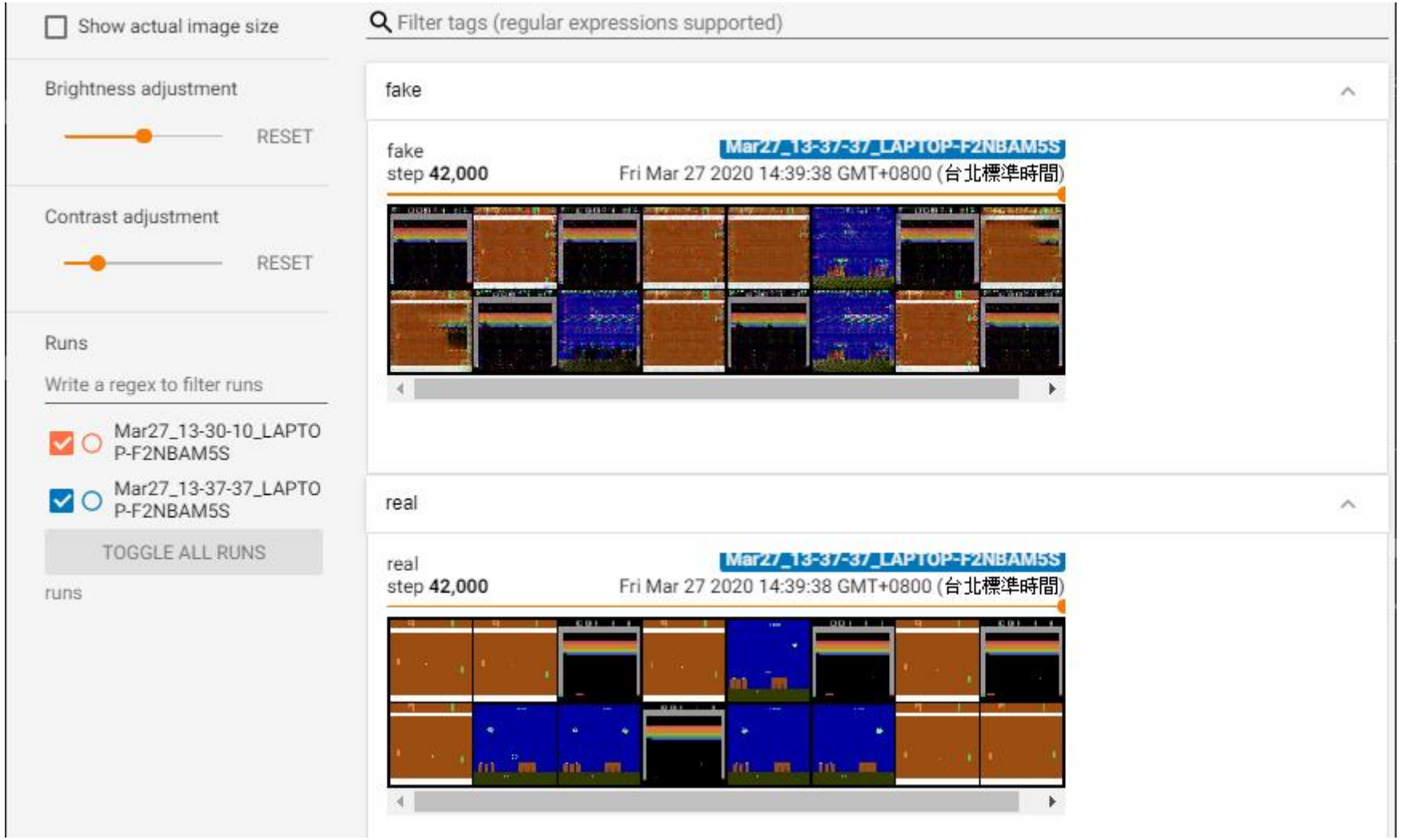
```
def iterate_batches(envs, batch_size=BATCH_SIZE):
    batch = [e.reset() for e in envs]
    env_gen = iter(lambda: random.choice(envs), None)

    while True:
        e = next(env_gen)
        obs, reward, is_done, _ = e.step(e.action_space.sample())
        if np.mean(obs) > 0.01:
            batch.append(obs)
        if len(batch) == batch_size:
            # Normalising input between -1 to 1
            batch_np = np.array(batch, dtype=np.float32) * 2.0 / 255.0 - 1.0
            yield torch.tensor(batch_np)
            batch.clear()
        if is_done:
            e.reset()
```

Training GAN

```
for batch_v in iterate_batches(envs):
        # generate extra fake samples, input is 4D: batch, filters, x, y
        gen_input_v = torch.FloatTensor(BATCH_SIZE, LATENT_VECTOR_SIZE, 1, 1).normal_(0, 1).to(device)
        batch_v = batch_v.to(device)
        gen_output_v = net_gener(gen_input_v)

        # train discriminator
        dis_optimizer.zero_grad()
        dis_output_true_v = net_discr(batch_v)
        dis_output_fake_v = net_discr(gen_output_v.detach())
        dis_loss=objective(dis_output_true_v, true_labels_v)+objective(dis_output_fake_v,fake_labels_v)
        dis_loss.backward()
        dis_optimizer.step()
        dis_losses.append(dis_loss.item())

        # train generator
        gen_optimizer.zero_grad()
        dis_output_v = net_discr(gen_output_v)
        gen_loss_v = objective(dis_output_v, true_labels_v)
        gen_loss_v.backward()
        gen_optimizer.step()
        gen_losses.append(gen_loss_v.item())
```

Experimental Results

- > tensorboard --logdir runs --host localhost



Reference

- https://gym.openai.com/docs/
- Maxim Lapan, “Deep Reinforcement Learning Hands On,” Chapter 2 and Chapter 3