Deep Reinforcement Learning

Visual DRL Control of Robots

P. Durdevic[1]

[1]Department of Energy Technology
Aalborg University

Wednesday 13th July, 2022

Table of Contents

Introduction

Contents

In this lecture we will look at a few State of the Art (SOTA) methods for controlling agents from pixel information.

- State of the Art
- Example One:
- Example Two:

NOTES

- The methods which will be introduced are dealing with 2D environments
- Non-dynamic environments
- real life problem with UAVs and robots will be much more complicated, specially in unknown generic environments, ad only using information available from the sensors mounted on-board (preferably from the camera)
- Training is a huge challenge, and realistic simulation environments could be a potential solution
  - Examples: Gazebo, NVIDIA Isaac Sim, Unity3D(1)

Challenges

When using NN in RL Introduce instability

When using NN in RL Need a lot of data fro training, even so not guaranteed to converge on the optimal value function

There are many methods to solve these issues in the Deep Q-Network, in this lecture we focus on two:

- Experience Replay
- Target Network

Source: `https://towardsdatascience.com/deep-q-network-dqn-ii-b6bf911b6b2c`

TODO

SOTA

SOTA

This could be done in the previous lecture, going from DRL to pixel DRL
TODO

Method 1: DQN

Example of deep Q-network **DQN** ((2))

this first section should be revised using Lauras book section 4

- The first model to work with raw visual inputs was the DQN proposed by (2) *(more than 15000 citations)*
- They use a deep convolutional neural network to approximate the optimal action-value function (Q-function):

$$Q^*(s,a) = \max_{\pi} \mathbb{E}[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots | s_t = t, a_t = a, \pi] \quad (1)$$

This is the maximum sum of the rewards r_t discounted by γ at each time-step t, achievable by a behavior policy $\pi = P(a|s)$, after making an observation s and taking an action a (2)

- A deep convolutional neural network (DCNN) is used to parametrize an approximate value function $Q(s,a;\theta)$.
- Where θ_i are the parameters (weights) of the Q-network at iteration i.

DQN
Introduction

In this section we analyze the milestone work done by (2)

- **deep Q-network** (DQN) *combines reinforcement learning with deep convolutional neural network (CNN)*

A DCNN is used to approximate the optimal action-value function:

$$Q^*(s,a) = \max_{\pi} \mathbb{E}\left[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots | s_t = s, a_t = a, \pi\right] \tag{2}$$

give detailed explanation by reading the methods section

$$Q(s,a,\theta_i) \tag{3}$$

An approximate value function is parametrized using a DCMM, with the weights θ_i of the Q-network at iteration i

- In the paper they use **experience replay** short introduction
  - This is done by storing the agents experiences $e_t = (s_t, a_t, {}_t, s_{t+1})$ at each time-step t in a data set $D_t = \{e_1, \ldots, e_t\}$

DQN

A sketch of this network was illustrated in (2), see below:

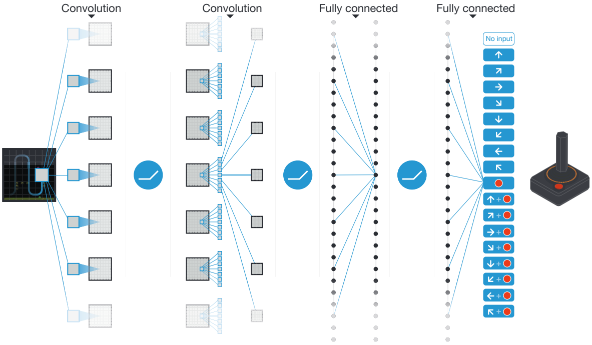


Figure: Schematic illustration of the convolutional neural network: The input to the neural network consists of an [84,84,4] image

DQN

Experience Replay

Introduction (TORRES)

DQN

Experience Replay

Experience Replay is a replay memory technique used in reinforcement learning where we store the agent's experiences at each time-step in a buffer

$$e_t = (s_t, a_t, s_{t+1}, r_{t+1}) \tag{4}$$

in a data-set

$$D = e_1, \ldots, e_N \tag{5}$$

- During learning, the Q-learning updates are applied on samples (or minibatches) of experience (s, a, r, s') $U(D)$, drawn uniformly at random from the pool of stored samples.
- This is different from the naive Q-learning algorithm which learns from the single most recent experience
- practically we use Python's built-in collections library deque

DQN

Experience Replay

- Code example of Experience replay buffer from DWN Jordi
-

```
Experience = collections.namedtuple(Experience,
           field_names=[state, action, reward,
           done, new_state])
class ExperienceReplay:
  def __init__(self, capacity):
      self.buffer = collections.deque(maxlen=capacity)
  def __len__(self):
      return len(self.buffer)
  def append(self, experience):
      self.buffer.append(experience)

  def sample(self, batch_size):
      indices = np.random.choice(len(self.buffer),
       batch_size,
               replace=False)
      states, actions, rewards, dones, next_states =
             zip([self.buffer[idx] for idx in indices
       ])
      return np.array(states), np.array(actions),
             np.array(rewards,dtype=np.float32),
             np.array(dones, dtype=np.uint8),
             np.array(next_states)
```

DQN

Target Network

- *Remember that in Q-Learning, we update a guess with a guess, and this can potentially lead to harmful correlations. The Bellman equation provides us with the value of Q(s, a) via Q(s', a') . However, both the states s and s' have only one step between them. This makes them very similar, and it's very hard for a Neural Network to distinguish between them.*
- *To make training more stable, there is a trick, called target network, by which we keep a copy of our neural network and use it for the Q(s', a') value in the Bellman equation.*
- *That is, the predicted Q values of this second Q-network called the target network, are used to backpropagate through and train the main Q-network. It is important to highlight that the target network's parameters are not trained, but they are periodically synchronized with the parameters of the main Q-network. The idea is that using the target network's Q values to train the main Q-network will improve the stability of the training.*

rewrite above to be more condense and also fit with the slides from MNIH paper i.e. θ_i^-

DQN

Experience Replay

The Q-learning update at iteration i uses the following loss function:

$$L_i(\theta_i) = \mathbb{E}_{(s,a,r,s')\ \mathsf{U}(D)} \left[\left(r + \gamma \max_{a'} Q(s', a'; \theta_i^-) - Q(s, a; \theta_i) \right)^2 \right] \tag{6}$$

where:

- γ the discount factor determining the agent's horizon
- θ_i the parameters of the Q-network at iteration i
- θ_i^- the network parameters used to compute the target at iteration i, *only updated with the Q-network parameters (θ_i) every C steps and are held fixed between individual updates*
  finish from methods section

Example of **DQN** ((2))

Evaluation

- In the paper the DQN agent is evaluated in a Atari 2600 environment
  - input is $(210 \times 160$ color video at 60 Hz)

Finish

More Details on **Minh's DQN**: Target Network

Finish this *https://towardsdatascience.com/deep-q-network-dqn-ii-b6bf911b6b2c
Finish this *https://torres.ai/deep-reinforcement-learning-explained-series/

NOTES

(2) learn to play games from raw pixel
https://www.cs.ubc.ca/~van/papers/2016-TOG-deepRL/index.html
https://www.bloomberg.com/features/2015-preschool-for-robots/
https://www.youtube.com/watch?v=oPGVsoBonLM https://www.davidsilver.uk/teaching/
check this one out http://karpathy.github.io/2016/05/31/rl/

Example in Python

Deep Q-network (DQN) - Example Using Pytorch and OpenAI Gym

Introduction

- An example of DQN using OpenAI Gym and Pytorch, based on ((PyTorch))
- Example is based on work done by (2)
- In the example the `CartPole-v0` task from OpenAI Gym is used to train a DQN policy from raw visual inputs

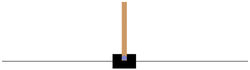


Figure: `CartPole-v0`

Description	Dimension	Value	Explanation
Action Space	Discrete(2)	$[0, 1]$	*[push left, push right]*
Observation Shape	(4,)	$[0, 1, 2, 3]$	*[Cart Position, Cart Velocity, Pole Angle, Pole Angular Velocity]*
Observation High		$[4.8, inf, 0.42, inf]$	
Observation Low		$[-4.8 - inf - 0.42 - inf]$	

1

[1] https://www.gymlibrary.ml/environments/classic_control/cart_pole/

Deep Q-network (DQN) - Example Using Pytorch and OpenAI Gym

Introduction

Rewards

- Since the goal is to keep the pole upright for as long as possible, a reward of $+14$ for every step taken, including the termination step, is allotted. The threshold for rewards is 475 for $v1$.

Starting State

- All observations are assigned a uniformly random value in (-0.05, 0.05)

Episode Termination The episode terminates if any one of the following occurs:

1. Pole Angle is greater than $\pm 12^{\circ}$
2. Cart Position is greater than ± 2.4 (center of the cart reaches the edge of the display)
3. Episode length is greater than $500 (200 \text{for } v0)$

Arguments

`gym.make('CartPole-v1')` [1]

[1] `https://www.gymlibrary.ml/environments/classic_control/cart_pole/`

Deep Q-network (DQN) - Example Using Pytorch and OpenAI Gym

Environment

- Instead of a tuple of systems **states** we use a frame from the scene
- The **state** is presented as the difference between current and previous frame
  - Thus, the agent will take the velocity of the pole into account from a single image.

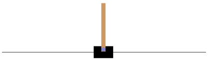


Figure: `CartPole-v0`

Deep Q-network (DQN) - Example Using Pytorch and OpenAI Gym

Environment

```python
# Import packages
import gym
import math
import random
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
from collections import namedtuple, deque
from itertools import count
from PIL import Image
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
import torchvision.transforms as T

# load the 'CartPole-v0' enviroment from GYM
env = gym.make('CartPole-v0').unwrapped

# set up matplotlib
is_ipython = 'inline' in matplotlib.get_backend()
if is_ipython:
    from IPython import display

# interactive mode will be on
# figures will automatically be shown
plt.ion()

# if gpu is to be used
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

Deep Q-network (DQN)

Replay Memory

- Store transition
- Sample randomly
- As mentioned earlier it: *stabilizes and improves the DQN training procedure*
- TWO CLASSES

Transition: Maps `(state, action)` pairs to their `(next_state, reward)` result.

ReplayMemory: **cyclic buffer** of **bounded size** that holds the **transitions observed recently**. It also implements a `.sample()` method for selecting a **random batch** of **transitions** for training.

Deep Q-network (DQN) - Example Using Pytorch and OpenAI Gym

Environment - CODE

```
Transition = namedtuple('Transition',
                        ('state', 'action', 'next_state', 'reward'))


class ReplayMemory(object):

    def __init__(self, capacity):
        self.memory = deque([],maxlen=capacity)

    def push(self, *args):
        """Save a transition"""
        self.memory.append(Transition(*args))

    def sample(self, batch_size):
        return random.sample(self.memory, batch_size)

    def __len__(self):
        return len(self.memory)
```

Deep Q-network (DQN) - Example Using Pytorch and OpenAI Gym

Deterministic Version of the DQN Algortihm

Simplified to a deterministic case

$$R_{t_0} = \sum_{t=t_0}^{\infty} \gamma^{t-t_0} r_t \tag{7}$$

- Where R_{t_0} is the return and $\gamma = [0, 1]$ is the discount factor (constant) prioritizing more recent rewards.

Main Idea

- Having a function $Q^* State \times Action \rightarrow \mathbb{R}$ that given an action in a given state could tell us what our return would be.
- Then a policy can be constructed to maximize our rewards

$$\pi^*(s) = \arg\max_a Q * (s, a) \tag{8}$$

Issue $Q*$ is unknown as it would require knowledge of all the states in the world

Solution Train **NNs** (universal function approximators) to resemble Q^*

Deep Q-network (DQN) - Example Using Pytorch and OpenAI Gym

Deterministic Version of the DQN Algortihm

For our training update rule, we'll use a fact that every Q function for some policy obeys the Bellman equation:

$$Q^{\pi}(s,a) = r + \gamma Q^{\pi}(s', \pi(s')) \tag{7}$$

The difference between the two sides of the equality is known as the temporal difference (TD) error, δ:

$$\delta = Q(s,a) - (r + \gamma \max_a Q(s', s')) \tag{8}$$

Next we wish to minimize the error δ, using the **Huber** loss. **Calculate** over a **batch** of **transitions**, B, sampled from the replay memory:

$$\mathcal{L} = \frac{1}{|B|} \sum_{s,a,s',r \in B} \mathcal{L}(\delta) \tag{9}$$

Where the piecewise loss function is defined as:

$$\mathcal{L}(\delta) = \begin{cases} \frac{1}{2}\delta^2 & \text{for } |\delta| \leq 1 \\ |\delta| - \frac{1}{2} & \text{otherwise} \end{cases} \tag{10}$$

Deep Q-network (DQN) - Example Using Pytorch and OpenAI Gym

Deterministic Version of the DQN Algortihm

Where the piecewise loss function is defined as:

$$\mathcal{L}(\delta) = \begin{cases} \frac{1}{2}\delta^2 & \text{for } |\delta| \leq 1 \\ |\delta| - \frac{1}{2} & \text{otherwise} \end{cases} \tag{7}$$

For **small** values of δ, the function is **quadratic** and **linear**for **large**.

Deep Q-network (DQN) - Example Using Pytorch and OpenAI Gym

Q-network

```
class DQN(nn.Module):

    def __init__(self, h, w, outputs):
        super(DQN, self).__init__()
        self.conv1 = nn.Conv2d(3, 16, kernel_size=5, stride=2)
        self.bn1 = nn.BatchNorm2d(16)
        self.conv2 = nn.Conv2d(16, 32, kernel_size=5, stride=2)
        self.bn2 = nn.BatchNorm2d(32)
        self.conv3 = nn.Conv2d(32, 32, kernel_size=5, stride=2)
        self.bn3 = nn.BatchNorm2d(32)

        # Number of Linear input connections depends on output of conv2d layers
        # and therefore the input image size, so compute it.
        def conv2d_size_out(size, kernel_size = 5, stride = 2):
            return (size - (kernel_size - 1) - 1) // stride  + 1
        convw = conv2d_size_out(conv2d_size_out(conv2d_size_out(w)))
        convh = conv2d_size_out(conv2d_size_out(conv2d_size_out(h)))
        linear_input_size = convw * convh * 32
        self.head = nn.Linear(linear_input_size, outputs)

    # Called with either one element to determine next action, or a batch
    # during optimization. Returns tensor([[left0exp,right0exp]...]).
    def forward(self, x):
        x = x.to(device)
        x = F.relu(self.bn1(self.conv1(x)))
        x = F.relu(self.bn2(self.conv2(x)))
        x = F.relu(self.bn3(self.conv3(x)))
        return self.head(x.view(x.size(0), -1))
```

- Use a CNN as the model structure
- Takes in the difference between the current and previous screen patches (s)
- Outputs
  - $Q(s, \text{left})$
  - $Q(s, \text{right})$

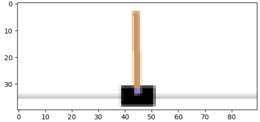
Deep Q-network (DQN) - Example Using Pytorch and OpenAI Gym

Utilities - Input Extraction

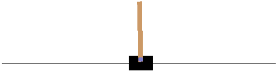
```python
resize = T.Compose([T.ToPILImage(),
                    T.Resize(40, interpolation=Image.CUBIC),
                    T.ToTensor()])
def get_cart_location(screen_width):
    world_width = env.x_threshold * 2
    scale = screen_width / world_width
    return int(env.state[0] * scale + screen_width / 2.0)  # MIDDLE OF CART
def get_screen():
    # Returned screen requested by gym is 400x600x3, but is sometimes larger
    # such as 800x1200x3. Transpose it into torch order (CHW).
    screen = env.render(mode='rgb_array').transpose((2, 0, 1))
    # Cart is in the lower half, so strip off the top and bottom of the screen
    _, screen_height, screen_width = screen.shape
    screen = screen[:, int(screen_height*0.4):int(screen_height * 0.8)]
    view_width = int(screen_width * 0.6)
    cart_location = get_cart_location(screen_width)
    if cart_location < view_width // 2:
        slice_range = slice(view_width)
    elif cart_location > (screen_width - view_width // 2):
        slice_range = slice(-view_width, None)
    else:
        slice_range = slice(cart_location - view_width // 2,
                            cart_location + view_width // 2)
    # Strip off the edges, so that we have a square image centered on a cart
    screen = screen[:, :, slice_range]
    # Convert to float, rescale, convert to torch tensor
    # (this doesn't require a copy)
    screen = np.ascontiguousarray(screen, dtype=np.float32) / 255
    screen = torch.from_numpy(screen)
    # Resize, and add a batch dimension (BCHW)
    return resize(screen).unsqueeze(0)
env.reset()
plt.figure()
plt.imshow(get_screen().cpu().squeeze(0).permute(1, 2, 0).numpy(),
           interpolation='none')
plt.title('Example extracted screen')
plt.show()
```

- Extracting and processing rendered images from the environment.
- `Torchvision` makes it easy to compose image transforms.
- Example:

Example extracted from the screen



Original Environment



Deep Q-network (DQN) - Example Using Pytorch and OpenAI Gym

Training - 1

- Set up hyper-parameters
- Set-up the size of the network based on the screen size and action space
- Initialize the optimizer
- Set the size of the deque for the replay memory
- finish explaining optim.RMSprop(policy_net.parameters())https://pytorch.org/docs/stable/generated/torch.optim.RMSprop.html

```
# Hyperparameters
BATCH_SIZE = 128
GAMMA = 0.999
EPS_START = 0.9
EPS_END = 0.05
EPS_DECAY = 200
TARGET_UPDATE = 10

# Get screen size so that we can initialize layers correctly based on shape
# returned from AI gym. Typical dimensions at this point are close to 3x40x90
# which is the result of a clamped and down-scaled render buffer in get_screen()
init_screen = get_screen()
_, _, screen_height, screen_width = init_screen.shape

# Get number of actions from gym action space
n_actions = env.action_space.n
# setup network size (from screen size and input size)
policy_net = DQN(screen_height, screen_width, n_actions).to(device)
target_net = DQN(screen_height, screen_width, n_actions).to(device)
# Load models parameter dictionary using a deserialized state_dict.
target_net.load_state_dict(policy_net.state_dict())
#  evaluate target_net
target_net.eval()
# Initialize optimizer
optimizer = optim.RMSprop(policy_net.parameters())
memory = ReplayMemory(10000) # size of the deque

steps_done = 0
```

Deep Q-network (DQN) - Example Using Pytorch and OpenAI Gym

Training - 1

Example of `policy_net.state_dict()`

```python
# Print model's state_dict
print("Model's state_dict:")
for param_tensor in policy_net.state_dict():
    print(param_tensor, "\t", policy_net.state_dict()[param_tensor].size())
```

Will print the model parameters

```
conv1.weight     torch.Size([16, 3, 5, 5])
conv1.bias       torch.Size([16])
bn1.weight       torch.Size([16])
bn1.bias         torch.Size([16])
bn1.running_mean         torch.Size([16])
bn1.running_var          torch.Size([16])
bn1.num_batches_tracked          torch.Size([])
conv2.weight     torch.Size([32, 16, 5, 5])
conv2.bias       torch.Size([32])
bn2.weight       torch.Size([32])
bn2.bias         torch.Size([32])
bn2.running_mean         torch.Size([32])
bn2.running_var          torch.Size([32])
bn2.num_batches_tracked          torch.Size([])
conv3.weight     torch.Size([32, 32, 5, 5])
conv3.bias       torch.Size([32])
bn3.weight       torch.Size([32])
bn3.bias         torch.Size([32])
bn3.running_mean         torch.Size([32])
bn3.running_var          torch.Size([32])
bn3.num_batches_tracked          torch.Size([])
head.weight      torch.Size([2, 512])
head.bias        torch.Size([2])
```

Deep Q-network (DQN) - Example Using Pytorch and OpenAI Gym

Training - Select Action

- select_action select an action accordingly to an epsilon greedy policy.
- EPS_START The probability of choosing a random action will start at EPS_START and will decay exponentially towards EPS_END.
- EPS_DECAY controls the rate of the decay.

```
def select_action(state):
    global steps_done
    sample = random.random()
    eps_threshold = EPS_END + (EPS_START - EPS_END) * math.exp(-1. * steps_done / EPS_DECAY)
    steps_done += 1
    if sample > eps_threshold:
        with torch.no_grad():
            # t.max(1) will return largest column value of each row.
            # second column on max result is index of where max element was
            # found, so we pick action with the larger expected reward.
            return policy_net(state).max(1)[1].view(1, 1)
    else:
        return torch.tensor([[random.randrange(n_actions)]], device=device, dtype=torch.long)
```

Disabling gradient calculation is useful for inference, when you are sure that you will not call `Tensor.backward()`. It will reduce memory consumption for computations that would otherwise have `requires_grad=True`. (Pytorch)

would be nice to plot eps_threshold during training to show how it changes

Deep Q-network (DQN) - Example Using Pytorch and OpenAI Gym

Training - Plot Durations

- plot_durations a helper for plotting the durations of episodes, along with an average over the last 100 episodes (the measure used in the official evaluations). The plot will be underneath the cell containing the main training loop, and will update after every episode.

```
episode_durations = []

def plot_durations():
    plt.figure(2)
    plt.clf()
    durations_t = torch.tensor(episode_durations, dtype=torch.float)
    plt.title('Training...')
    plt.xlabel('Episode')
    plt.ylabel('Duration')
    plt.plot(durations_t.numpy())
    # Take 100 episode averages and plot them too
    if len(durations_t) >= 100:
        means = durations_t.unfold(0, 100, 1).mean(1).view(-1)
        means = torch.cat((torch.zeros(99), means))
        plt.plot(means.numpy())

    plt.pause(0.001)  # pause a bit so that plots are updated
    if is_ipython:
        display.clear_output(wait=True)
        display.display(plt.gcf())
```

Deep Q-network (DQN) - Example Using Pytorch and OpenAI Gym

Training - Training loop - 1

- The `optimize_model` function performs a single step of the optimization.
  - It first samples a batch, concatenates all the tensors into a single one, computes:

```python
def optimize_model():
    if len(memory) < BATCH_SIZE:
        return
    transitions = memory.sample(BATCH_SIZE)
    # Transpose the batch, this converts batch-array of Transitions to Transition of batch-arrays.
    batch = Transition(*zip(*transitions))

    # Compute a mask of non-final states and concatenate the batch elements
    # (a final state would've been the one after which simulation ended)
    non_final_mask = torch.tensor(tuple(map(lambda s: s is not None,
                                          batch.next_state)), device=device, dtype=torch.bool)
    non_final_next_states = torch.cat([s for s in batch.next_state
                                                if s is not None])
    state_batch = torch.cat(batch.state)
    action_batch = torch.cat(batch.action)
    reward_batch = torch.cat(batch.reward)
```

`optimize_model` is continued on next page...

Deep Q-network (DQN) - Example Using Pytorch and OpenAI Gym

Training - Training loop - 2

Next the Q function is computed using the `policy_net()` method and then the values are gathered along an axis with dimension specified by `action_batch` using `torch.gather` (5).

```

    # Compute Q(s_t, a) - the model computes Q(s_t), then we select the
    # columns of actions taken. These are the actions which would've been taken
    # for each batch state according to policy_net
    state_action_values = policy_net(state_batch).gather(1, action_batch)
```

Where `policy_net()` is our policy network, defined earlier as:

```
policy_net = DQN(screen_height, screen_width, n_actions).to(device)
```

BELOW IS TEMP, it is text directly form the webpage

- $Q(s_t, a_t)$ and $V(s_{t+1}) = \max_a Q(s_{t+1}, a)$, and combines them into the loss.
- By definition we set $V(s) = 0$ is s is a terminal state.
- We also use a target network to compute $V(s_{t+1})$ for added stability.
- The target network has its weights kept frozen most of the time, but is updated with the policy network's weights every so often. This is usually a set number of steps but we shall use episodes for simplicity.

Deep Q-network (DQN) - Example Using Pytorch and OpenAI Gym

Training - Training loop - 3

From earlier we found the value function Write the value function
This is the maximum sum of the rewards r_t discounted by γ at each time-step t, achievable by a behavior policy $\pi = P(a|s)$, after making an observation s and taking an action a (2)

- explain Huber loss in code shortly as I have the theory earlier. (7)
- where does **param** come from?, explain this

```
    # Compute V(s_{t+1}) for all next states.
    # Expected values of actions for non_final_next_states are computed based
    # on the "older" target_net; selecting their best reward with max(1)[0].
    # This is merged based on the mask, such that we'll have either the expected
    # state value or 0 in case the state was final.
    next_state_values = torch.zeros(BATCH_SIZE, device=device)
    next_state_values[non_final_mask] = target_net(non_final_next_states).max(1)[0].detach()
```

We use the function `torch.max(input, dim, keepdim=False, *, out=None) -> Tensor` (6) the maximum value of all elements in the input tensor are found.
this explanation is not final, must wok more on the explanation

Deep Q-network (DQN) - Example Using Pytorch and OpenAI Gym

Training - Main Training Loop - 1

Reset the environment and initialize the state Tensor

```
num_episodes = 1000
for i_episode in range(num_episodes):
    # Initialize the environment and state
    env.reset()
    last_screen = get_screen()
```

Sample an action, execute it, observe the next screen and the reward (always 1) and optimize our model once

```
    current_screen = get_screen()
    state = current_screen - last_screen
    for t in count():
        # Select and perform an action
        action = select_action(state)
        _, reward, done, _ = env.step(action.item())
        reward = torch.tensor([reward], device=device)

        # Observe new state
        last_screen = current_screen
        current_screen = get_screen()
        if not done:
            next_state = current_screen - last_screen
        else:
            next_state = None
```

Continued on next page

Deep Q-network (DQN) - Example Using Pytorch and OpenAI Gym

Training - Main Training Loop - 2

When the episode ends (our model fails), we restart the loop.

```python
        # Store the transition in memory
        memory.push(state, action, next_state, reward)

        # Move to the next state
        state = next_state

        # Perform one step of the optimization (on the policy network)
        optimize_model()
        if done:
            episode_durations.append(t + 1)
            plot_durations()
            break
    # Update the target network, copying all weights and biases in DQN
    if i_episode % TARGET_UPDATE == 0:
        target_net.load_state_dict(policy_net.state_dict())
```

Render the environment

```python
print('Complete')
env.render()
env.close()
plt.ioff()
plt.show()
```

Actions are chosen either randomly or based on a policy, getting the next step sample from the gym environment. We record the results in the replay memory and also run optimization step on every iteration. Optimization picks a random batch from the replay memory to do training of the new policy. "Older" `target_net` is also used in optimization to compute the expected Q values; it is updated occasionally to keep it current.

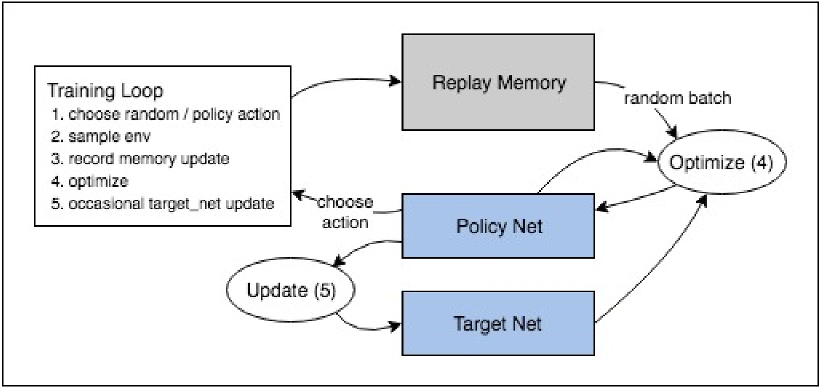


Figure: Todo

Pytorch Extras

Loading and Saving a Model

When it comes to saving and loading models, there are three core functions to be familiar with:

`torch.save` Saves a serialized object to disk. This function uses Python's pickle utility for serialization. Models, tensors, and dictionaries of all kinds of objects can be saved using this function.

`torch.load` Uses pickle's unpickling facilities to deserialize pickled object files to memory. This function also facilitates the device to load the data into (see Saving & Loading Model Across Devices).

`torch.nn.Module.load_state_dict` Loads a model's parameter dictionary using a deserialized `state_dict`. For more information on `state_dict`.

What is a `state_dict`?
In PyTorch, the learnable parameters (i.e. weights and biases) of an `torch.nn.Module` model are contained in the model's parameters (accessed with `model.parameters()`). A `state_dict` is simply a Python dictionary object that maps each layer to its parameter tensor. Note that only layers with learnable parameters (convolutional layers, linear layers, etc.) and registered buffers (batchnorm's `running_mean`) have entries in the model's `state_dict`. Optimizer objects (`torch.optim`) also have a `state_dict`, which contains information about the optimizer's state, as well as the hyperparameters used.
Because `state_dict` objects are Python dictionaries, they can be easily saved, updated, altered, and restored, adding a great deal of modularity to PyTorch models and optimizers.

https://pytorch.org/tutorials/beginner/saving_loading_models.html

Pytorch Extras

Loading and Saving a Model: **Example**

To save the model we then use the following code:

If we wish to load a model and use it to perform actions witouth leraning we can use the following modified main loop

https://pytorch.org/tutorials/beginner/saving_loading_models.html

References I

[1] Juliani, A., Berges, V.-P., Teng, E., Cohen, A., Harper, J., Elion, C., Goy, C., Gao, Y., Henry, H., Mattar, M., et al. (2018). Unity: A general platform for intelligent agents. *arXiv preprint arXiv:1809.02627*.

[2] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. (2015). Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533.

[Pytorch] Pytorch. no grad. `https://pytorch.org/docs/stable/generated/torch.no_grad.html`.

[PyTorch] PyTorch. Reinforcement learning (dqn) tutorial. https://pytorch.org/tutorials/intermediate/reinforcement$_{q}{}_{l}earning.html$.

[5] Pytorch. Torch gather. `https://pytorch.org/docs/stable/generated/torch.gather.html`.

[6] Pytorch. Torch max. `https://pytorch.org/docs/stable/generated/torch.max.html`.

[7] Pytorch. Torch smoothl1loss. `https://pytorch.org/docs/stable/generated/torch.nn.SmoothL1Loss.html`.

[TORRES] TORRES, J. Deep q-network (dqn)-ii. `https://towardsdatascience.com/deep-q-network-dqn-ii-b6bf911b6b2c`.