Self-Driving Cars

Exercise 1 - Imitation Learning

Eshed Ohn-Bar

Autonomous Vision Group
MPI-IS / University of Tübingen

October 25, 2019

University of Tübingen
MPI for Intelligent Systems
Autonomous Vision Group

Exercise Setup

Download

- `01_imitation_learning_exercise.pdf`
- `main.py`
- `netw ork.py`, `training.py`, `imitations.py`
- `data`, a folder with imitations

Exercise Setup

Download

- `01_imitation_learning_exercise.pdf`
- `main.py`
- `netw ork.py`, `training.py`, `imitations.py`
- `data`, a folder with imitations

Submit

- a report as a `.pdf` file, up to 3 pages
- your code: `network.py`, `training.py`, `imitations.py` as a `.zip` file
- your pre-trained model as a `.t7` file

Deadline: **Wed, 21. November 2018 - 21:00**

Exercise Setup

Do's

- comment your code
- use docstrings
- use self-explanatory variable names
- structure your code well

Exercise Setup

Do's

- comment your code
- use docstrings
- use self-explanatory variable names
- structure your code well

Do not's

- change `main.py`, especially `calculate_score_for_leaderboard()`
- install more packages
- change the gym environment

Imitation Learning

Behavioral Cloning

Imitation Learning

Components:

- State: $s \in \mathcal{S}$ — may be partially observed (e.g., game screen)
- Action: $a \in \mathcal{A}$ — may be discrete or continuous (e.g., turn angle, speed)
- Policy: $\pi_\theta : \mathcal{S} \to \mathcal{A}$ — we want to learn the policy parameters θ
- Optimal action: $a^* \in \mathcal{A}$ — provided by expert demonstrator
- Optimal policy: $\pi^* : \mathcal{S} \to \mathcal{A}$ — provided by expert demonstrator
- State dynamics: $P(s_{i+1}|s_i, a_i)$ — simulator, typically not known to policy
  Often deterministic: $s_{i+1} = T(s_i, a_i)$ — deterministic mapping
- Rollout: Given s_0, sequentially execute $a_i = \pi_\theta(s_i)$ & sample $s_{i+1} \sim P(s_{i+1}|s_i, a_i)$
  yields trajectory $\tau = (s_0, a_0, s_1, a_1, \dots)$
- Loss function: $\mathcal{L}(a^*, a)$ — loss of action a given optimal action a^*

Imitation Learning

Components:

- State: $s \in \mathcal{S}$ — may be partially observed (e.g., game screen)
- Action: $a \in \mathcal{A}$ — may be discrete or continuous (e.g., turn angle, speed)
- Policy: $\pi_\theta : \mathcal{S} \rightarrow \mathcal{A}$ — we want to learn the policy parameters θ
- Optimal action: $a^* \in \mathcal{A}$ — provided by expert demonstrator
- Optimal policy: $\pi^* : \mathcal{S} \rightarrow \mathcal{A}$ — provided by expert demonstrator
- State dynamics: $P(s_{i+1}|s_i, a_i)$ — simulator, typically not known to policy

  Often deterministic: $s_{i+1} = T(s_i, a_i)$ — deterministic mapping
- Rollout: Given s_0, sequentially execute $a_i = \pi_\theta(s_i)$ & sample $s_{i+1} \sim P(s_{i+1}|s_i, a_i)$

  yields trajectory $\tau = (s_0, a_0, s_1, a_1, \dots)$
- Loss function: $\mathcal{L}(a^*, a)$ — loss of action a given optimal action a^*

1.1

Network Design

a) Load imitations

```python
def load_imitations(data_folder):
    """
    1.1 a)
    Given the folder containing the expert imitations, the data gets loaded and
    stored it in two lists: observations and actions.
                    N = number of (observation, action) - pairs
    data_folder:    python string, the path to the folder containing the
                    observation_%05d.npy and action_%05d.npy files
    return:
    observations:   python list of N numpy.ndarrays of size (96, 96, 3)
    actions:        python list of N numpy.ndarrays of size 3
    """
    pass
```

b) Understand training

```python
def train(data_folder, trained_network_file):
    """
    Function for training the network.
    """
    infer_action = ClassificationNetwork()
    optimizer = torch.optim.Adam(infer_action.parameters(), lr=1e-2)
    observations, actions = load_imitations(data_folder)
    observations = [torch.Tensor(observation) for observation in observations]
    actions = [torch.Tensor(action) for action in actions]

    batches = [batch for batch in zip(observations,
                                      infer_action.actions_to_classes(actions))]
    gpu = torch.device('cuda')

    nr_epochs = 100
    batch_size = 64
    nr_of_classes = 0  # needs to be changed
    start_time = time.time()

    for epoch in range(nr_epochs):
        random.shuffle(batches)
```

c) Classification Network

- expert imitations

  action = [steer, gas, brake]

  e.g. `[1., 0., 0.8]`
- define **action-classes**
  - {steer_left}
  - {}
  - {steer_right, brake}
  - {gas}

$\Rightarrow$ map `[1., 0., 0.8]` $\rightarrow$ `?`

c) Classification Network

- expert imitations

  action = [steer, gas, brake]

  e.g. `[1., 0., 0.8]`
- define **action-classes**
  - {steer_left}
  - {}
  - {steer_right, brake}
  - {gas}

$\Rightarrow$ map `[1., 0., 0.8]` $\rightarrow$ `[0, 0, 1, 0]`

c) Classification Network

- `actions_to_classes`
  expert imitations → action-classes
- `scores_to_action`
  score predicted by the network → action
- `cross_entropy_loss`
  loss function: gt vs. prediction

d) Implement network

```python
class ClassificationNetwork(torch.nn.Module):
    def __init__(self):
        """
        1.1 d)
        Implementation of the network layers. The image size of the input
        observations is 96x96 pixels.
        """
        super().__init__()
        gpu = torch.device('cuda')
```

- 2 to 3 convolution layers + ReLU
- 2 to 3 fully connected layers + ReLU
- Softmax

d) Implement network

```
torch.nn.Sequential(
    torch.nn.Conv2d(in_channels, out_channels, filter_size, stride=*arg),
    torch.nn.LeakyReLU(negative_slope=0.2))

torch.nn.Sequential(
    torch.nn.Linear(in_size, out_size),
    torch.nn.LeakyReLU(negative_slope=0.2))
```

- 2 to 3 convolution layers + ReLU
- 2 to 3 fully connected layers + ReLU
- Softmax

e) Forward pass, train and test

```python
    def forward(self, observation):
        """
        1.1 e)
        The forward pass of the network. Returns the prediction for the given
        input observation.
        observation:   torch.Tensor of size (batch_size, 96, 96, 3)
        return         torch.Tensor of size (batch_size, C)
        """
        pass
```

- color channels or gray-scale
- `python3 main.py train`
- `python3 main.py test`
- hyper-parameter tuning

e) Forward pass, train and test

`python3 main.py test`

```
def evaluate():
    """
    """
    infer_action = torch.load(trained_network_file)
    infer_action.eval()
    env = gym.make('CarRacing-v0')
    gpu = torch.device('cuda')

    for episode in range(5):
        observation = env.reset()

        reward_per_episode = 0
        for t in range(500):
            env.render()
            action_scores = infer_action(
                torch.Tensor(np.ascontiguousarray(observation[None])).to(gpu))

            steer, gas, brake = infer_action.scores_to_action(action_scores)
            observation, reward, done, info = env.step([steer, gas, brake])
            reward_per_episode += reward

        print('episode %d \t reward %f' % (episode, reward_per_episode))
```

f) Record own imitations

```python
def record_imitations(imitations_folder):
    """
    Function to record own imitations by driving the car in the gym car-racing
    environment.
    imitations_folder:  python string, the path to where the recorded imitations
                        are to be saved

    The controls are:
    arrow keys:         control the car; steer left, steer right, gas, brake
    ESC:                quit and close
    SPACE:              restart on a new track
    TAB:                save the current run
    """
    env = gym.make('CarRacing-v0').env
    status = ControlStatus()
    total_reward = 0.0

    while not status.quit:
        observations = []
        actions = []
```

f) Record own imitations

```
python3 main.py teach
```

```
python3 main.py test
```

1.2

Network Improvements

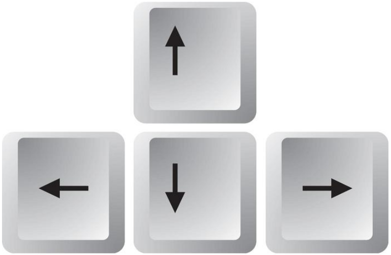
a) Observations

a) Observations

```
    def extract_sensor_values(self, observation, batch_size):
        """
        observation:    python list of batch_size many torch.Tensors of size
                        (96, 96, 3)
        batch_size:     int
        return          torch.Tensors of size (batch_size, 1),
                        torch.Tensors of size (batch_size, 4),
                        torch.Tensors of size (batch_size, 1),
                        torch.Tensors of size (batch_size, 1)
        """
        speed_crop = observation[:, 84:94, 12, 0].reshape(batch_size, -1)
        speed = speed_crop.sum(dim=1, keepdim=True) / 255
        abs_crop = observation[:, 84:94, 18:25:2, 2].reshape(batch_size, 10, 4)
        abs_sensors = abs_crop.sum(dim=1) / 255
        steer_crop = observation[:, 88, 38:58, 1].reshape(batch_size, -1)
        steering = steer_crop.sum(dim=1, keepdim=True)
        gyro_crop = observation[:, 88, 58:86, 0].reshape(batch_size, -1)
        gyroscope = gyro_crop.sum(dim=1, keepdim=True)
        return speed, abs_sensors.reshape(batch_size, 4), steering, gyroscope
```

b) MultiClass Prediction

- 4 binary classes:
  steer left, steer right, accelerate, brake
- multi-class network
- actions $\rightarrow$ action-classes
- sigmoid activation function
- new loss function



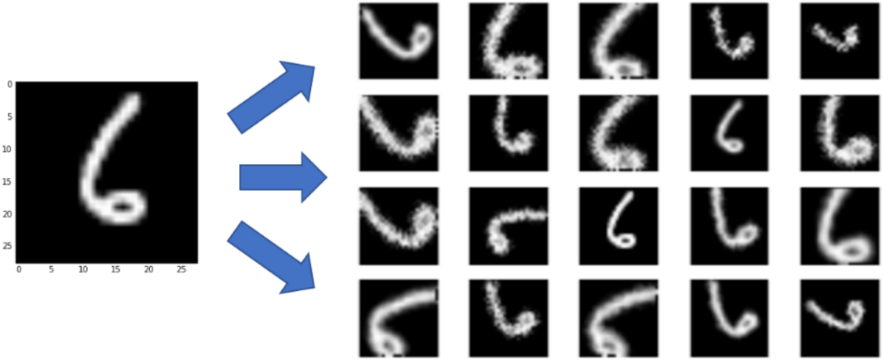
c) Classification vs. regression

Regression network

- Which loss function?
- Advantages / drawbacks compared to the classification networks?
- Is it a reasonable approach given our training data?

d) Data augmentation

Create more training data by synthetically modifying the data.



e) Fine-tuning

- different network architectures
- learning rate adaptation
- dropout-, batch- or instance normalization
- different optimizers
- class imbalance

- try at least two ideas
- explain the motivation and the outcome

Competition

Competition

`main.py`

```python
def calculate_score_for_leaderboard():
    """
    Evaluate the performance of the network. This is the function to be used for
    the final ranking on the course-wide leader-board, only with a different set
    of seeds. Better not change it.
    """
    infer_action = torch.load(trained_network_file, map_location='cpu')
    infer_action.eval()
    env = gym.make('CarRacing-v0')
    # you can set it to torch.device('cuda') in case you have a gpu
    device = torch.device('cpu')

    seeds = [22597174, 68545857, 75568192, 91140053, 86018367,
             49636746, 66759182, 91294619, 84274995, 31531469]
    total_reward = 0

    for episode in range(10):
        env.seed(seeds[episode])
        observation = env.reset()

        reward_per_episode = 0
        for t in range(600):
```

Questions?