Lernbasierte Computer Vision
Prof. Dr.-Ing. Andreas Geiger

University of Tübingen
MPI for Intelligent Systems
Autonomous Vision Group

Self-Driving Cars

Exercise 1 – Imitation Learning

Release date: Fri, 25. October 2019 - **Deadline for Homework: Wed, 20. November 2019 - 21:00**

For this exercise you need to submit a **.zip** folder containing your report as a **.pdf** file (up to 3 pages), your pre-trained model as a **.t7** file and your code (namely the files `network.py`, `training.py`, `imitations.py`). Please use the provided code templates for these exercises. The given `main.py` file will be used for evaluating your code.
Changes to this main file, to the gym environment or additionally installed packages will not be considered. Comment your code clearly, use docstrings and self-explanatory variable names and structure your code well.

1.1 Network design (1+2+2+2+2+1 Points)

Let $\mathcal{S}$ be the state space and $\mathcal{A}$ the action space. The gym environment encodes a transition function $T : \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{S}$ that maps a (state, action) - pair to a new state. It also provides a reward function $R : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$ which we will only use for evaluation this time.
The aim of this exercise is to design a network that learns a policy $\pi_\theta : \mathcal{S} \rightarrow \mathcal{A}$, parameterized by θ, to predict the best action for a given state. We formulate it as a supervised learning problem, where the objective is to follow the observed imitations of an "expert" driver.

a) Some imitations from an expert are given in `data/teacher`. Implement the function `load_imitations` in `imitations.py`. Given the folder of the imitations, it should load all `observation` and `action` files into two separate lists `observations` and `actions` which are returned.

```
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
"""
idx_file = os.path.join(data_folder, "count.npy")
assert os.path.exists(idx_file), "file doesn't exist: %s" % idx_file
idx = np.load(idx_file)

observations = []
actions = []
for i in range(idx):
if i % max(1, int(idx / 10)) == 0:
print("preloading data %d/%d" % (i, idx - 1))
observations.append(np.load(os.path.join(data_folder,
```

```
23          "observation_%05d.npy" % i)))
24          actions.append(np.load(os.path.join(data_folder,
25          "action_%05d.npy" % i)))
26          return observations, actions


('TP', 75)
('TN', 82)
```

b) The module `training.py` contains the training loop for the network. Read and understand its function `train`. Why is it necessary to divide the data into batches? What is an epoch? What do lines 43 to 48 do? Please answer shortly and precisely.

c) To start with, we formulate the problem as a classification network. Have a look at the actions provided by the expert imitations, there are three controls: steer, gas and brake. Which values do they take? Since they are not independent (accelerate and brake simultaneously does not make sense), it is reasonable to define classes of possible actions, like {steer_left}, {}, {steer_right and brake}, {gas} and so forth.
Define the set of action-classes you want to use and complete the class-methods `actions_to_classes` and `scores_to_action` in `network.py` The former maps every action to its class representation using a one-hot encoding and the latter retrieves an action from the scores predicted by the network. Lastly, implement the loss function `cross_entropy_loss` in `training.py` to calculate the training loss for a given pair of predicted and ground truth classes.

```
1  class ClassificationNetwork(torch.nn.Module):
2      def __init__(self):
3      """
4      1.1 d)
5      Implementation of the network layers. The image size of the input
6      observations is 96x96 pixels.
7      """
8      super().__init__()
9      gpu = torch.device('cuda')
10         self.classes = [[-1., 0., 0.],
11                         [-1., 0.5, 0.],
12                         [-1., 0., 0.8],
13                         [1., 0., 0.],
14                         [1., 0.5, 0.],
15                         [1., 0., 0.8],
16                         [0., 0., 0.],
17                         [0., 0.5, 0.],
18                         [0., 0., 0.8]]
19
20     def actions_to_classes(self, actions):
21     """
22     1.1 c)
23     For a given set of actions map every action to its corresponding
24     action-class representation. Assume there are C different classes, then
25     every action is represented by a C-dim vector which has exactly one
26     non-zero entry (one-hot encoding). That index corresponds to the class
27     number.
28     actions:        python list of N torch.Tensors of size 3
29     return          python list of N torch.Tensors of size C
30     """
```

```
    return [torch.Tensor([int(torch.prod(action == this_class))
            for this_class in torch.Tensor(self.classes)])
            for action in actions]

    def scores_to_action(self, scores):
    """
    1.1 c)
    Maps the scores predicted by the network to an action-class and returns
    the corresponding action [steer, gas, brake].
    C = number of classes
    scores:         python list of torch.Tensors of size C
    return          (float, float, float)
    """
    _, class_number = torch.max(scores[0], dim=0)
    steer, gas, brake = self.classes[class_number]
    return steer, gas, brake
```

```
def cross_entropy_loss(batch_out, batch_gt):
        """
        Calculates the cross entropy loss between the prediction of the network and
        the ground truth class for one batch.
        C = number of classes
        batch_out:      torch.Tensor of size (batch_size, C)
        batch_gt:       torch.Tensor of size (batch_size, C)
        return          float
        """
        epsilon = 0.000001
        loss = batch_gt * torch.log(batch_out + epsilon) + \
                (1 - batch_gt) * torch.log(1 - batch_out + epsilon)
        return -torch.mean(torch.sum(loss, dim=1), dim=0)
```

d) Design and implement an easy first network architecture in `network.py`. Start with 2 to 3 2D convolution layers on the images, followed by 2 or 3 fully connected layers (linear layers) to extract a 1D feature vector. Let each layer be followed by a ReLU as the non-linear activation (see code snippets below).
The output of the network should function as a controller for the car and predict one of the action-classes for each given state. At the end of the network, add a softmax layer to normalize the output. We can then interpret it as a probability distribution over the action-classes.

```
        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))
```

```
    def __init__(self):

    self.features_2d = torch.nn.Sequential(
            torch.nn.Conv2d(1, 2, 3, stride=1),
            torch.nn.LeakyReLU(negative_slope=0.2),  # 94x94
            torch.nn.Conv2d(2, 4, 3, stride=2),
```

```
            torch.nn.LeakyReLU(negative_slope=0.2),  # 46x46
            torch.nn.Conv2d(4, 8, 3, stride=2),
            torch.nn.LeakyReLU(negative_slope=0.2),  # 22x22
            ).to(gpu)

    self.scores = torch.nn.Sequential(
            torch.nn.Linear(8 * 22 * 22, 64),
            torch.nn.LeakyReLU(negative_slope=0.2),
            torch.nn.Linear(64, 32),
            torch.nn.LeakyReLU(negative_slope=0.2),
            torch.nn.Linear(32, 16),
            torch.nn.LeakyReLU(negative_slope=0.2),
            torch.nn.Linear(16, 9),
            torch.nn.Softmax(dim=1)
            ).to(gpu)
```

e) Implement the forward pass for your network, which is the function `forward` in `network.py`. Given an observation, it should return the probability distribution over the action-classes predicted by the network. You can decide whether you want to work with all 3 color channels or convert them to gray-scale beforehand. Motivate your choice briefly.
Train your network by running `python3 main.py train`. Afterwards, enjoy watching its performance by running `python3 main.py test` on your local machine. Can you achieve better results when changing the hyper-parameters? Can you explain this?

```
    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)
    """
    batch_size = observation.shape[0]
    # conversion to gray scale
    observation = observation[:, :, :, 0] * 0.2989 + \
            observation[:, :, :, 1] * 0.5870 + \
            observation[:, :, :, 2] * 0.1140

    obs = observation.reshape(batch_size, 1, 96, 96)
    features_2d = self.features_2d(obs).reshape(batch_size, -1)
    return self.scores(features_2d)
```

f) `imitations.py` provides some code to record new imitations. Complete the function `save_imitations` and start driving yourself by running `python3 main.py teach` on your local machine. What is 'good' training data? Is there any problem with only perfect imitations?

```
def save_imitations(data_folder, actions, observations):
        """
        1.1 f)
        Save the lists actions and observations in numpy .npy files that can ↙
            ↳ be read
        by the function load_imitations.
        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
        observations:   python list of N numpy.ndarrays of size (96, 96, 3)
        actions:        python list of N numpy.ndarrays of size 3
        """
        idx_file = os.path.join(data_folder, "count.npy")
        if os.path.exists(idx_file):
                idx = np.load(idx_file)
        else:
                idx = 0

        for i in range(int(len(actions))):
                print("Saving %d/%d" % (10 * i, len(actions)))
                np.save(os.path.join(data_folder, "observation_%05d.npy" %
                        (idx + i)), observations[10 * i])
                np.save(os.path.join(data_folder, "action_%05d.npy" %
                        (idx + i)), np.array(actions[10 * i]))
        np.save(idx_file, idx + int(len(actions) / 10))
```

1.2 Network Improvements (2+2+2+2+2 Points)

Now that your network is up and running, it's time to increase its performance! Each of the following tasks adds to its architecture. It is up to you to choose which of them you use for participating in the competition. However, all subtasks need to be answered. Evaluate and compare different methods always on the same training data (no matter whether that is the provided or self-recorded data or a mix of both).

a) Observations. The training data of the network actually contains more information than just the image from the car in the environment. Look at the class method `extract_sensor_values` in `network.py`. What does it do? Incorporate it into your network architecture. How does the performance change?

It provides bars for true speed, four ABS sensors, steering wheel position and gyroscope.

```
def forward(self, observation):
    batch_size = observation.shape[0]
    # extract sensor values
    speed, abs_sensors, steering, gyroscope = \
            self.extract_sensor_values(observation, batch_size)

    # conversion to gray scale
    observation = observation[:, :, :, 0] * 0.2989 + \
            observation[:, :, :, 1] * 0.5870 + \
            observation[:, :, :, 2] * 0.1140

    # crop and reshape observations to 84 x 96
    obs = observation[:, :84, :].reshape(batch_size, 1, 84, 96)

    # get features
    features_2d = self.features_2d(obs).reshape(batch_size, -1)
    features_1d = self.features_1d(features_2d)

    fused_features = torch.cat((
            speed,              # batch_size x 1
            abs_sensors,        # batch_size x 4
            steering,           # batch_size x 1
```

```
            gyroscope,          # batch_size x 1
            features_1d), 1)    # batch_size x 16
    return self.fused_features(fused_features)
```

b) MultiClass prediction. Design a second network architecture that encodes a multi-class approach by defining 4 binary classes that represent the 4 arrow keys on a keyboard and stand for: steer right, steer left, accelerate and brake. Since those don't all exclude each other, let the network learn to predict 0 or 1 for each class independently.
You will need to implement another loss function and might find a sigmoid-activation function useful. Again, compare the results to the previous classification approach.

```
class MultiClassNetwork(torch.nn.Module):
        def __init__(self):
                """
                The image size of the input is 96x96 pixels.
                """
                super().__init__()
                gpu = torch.device('cuda')

                self.features_2d = torch.nn.Sequential(
                        torch.nn.Conv2d(1, 2, 3, stride=1),
                        torch.nn.BatchNorm2d(2),
                        torch.nn.LeakyReLU(negative_slope=0.2),  # 94x94
                        torch.nn.Conv2d(2, 4, 3, stride=2),
                        torch.nn.BatchNorm2d(4),
                        torch.nn.LeakyReLU(negative_slope=0.2),  # 46x46
                        torch.nn.Conv2d(4, 8, 3, stride=2),
                        torch.nn.BatchNorm2d(8),
                        torch.nn.LeakyReLU(negative_slope=0.2)  # 22x22
                        ).to(gpu)

                self.scores = torch.nn.Sequential(
                        torch.nn.Linear(8 * 22 * 22, 64),
                        torch.nn.BatchNorm1d(64),
                        torch.nn.LeakyReLU(negative_slope=0.2),
                        torch.nn.Linear(64, 32),
                        torch.nn.BatchNorm1d(32),
                        torch.nn.LeakyReLU(negative_slope=0.2),
                        torch.nn.Linear(32, 16),
                        torch.nn.BatchNorm1d(16),
                        torch.nn.LeakyReLU(negative_slope=0.2),
                        torch.nn.Linear(16, 8),
                        torch.nn.BatchNorm1d(8),
                        torch.nn.LeakyReLU(negative_slope=0.2),
                        torch.nn.Linear(8, 4),
                        torch.nn.BatchNorm1d(4),
                        torch.nn.LeakyReLU(negative_slope=0.2),
                        torch.nn.Sigmoid(dim=1)
                        ).to(gpu)

        def forward(self, observation):
                B = observation.shape[0]
                # conversion to gray scale
                observation = observation[:, :, :, 0] * 0.2989 + \
```

```
                        observation[:, :, :, 1] * 0.5870 + \
                        observation[:, :, :, 2] * 0.1140

                obs = observation.reshape(B, 1, 96, 96)
                features_2d = self.features_2d(obs).reshape(B, -1)
                return self.scores(features_2d)

        def class_to_action(self, action_scores):
                steer_difference = action_scores[0][0] - action_scores[0][1]
                steer = -1.0 if steer_difference > 0.5 else 0.0
                steer = 1.0 if steer_difference < -0.5 else steer
                gas = 0.5 if action_scores[0][2] > 0.5 else 0.0
                brake = 0.8 if action_scores[0][3] > 0.5 else 0.0
                return steer, gas, brake

        def actions_to_classes(self, actions):
                actions = torch.Tensor(actions)
                return [[int(action[0] < 0.), int(action[0] > 0.),
                int(action[1] > 0.), int(action[2] > 0.)]
                for action in actions]
```

c) Classification vs. regression. Formulate the current problem as a regression network. Which loss function is appropriate? What are the advantages / drawbacks compared to the classification networks? Is it reasonable to use a regression approach given our training data?
Hint: You can control the top speed of your car by changing the `acceleration` variable in `ControlStatus` from `imitations.py`. But be aware that this also changes the actions you are recording.

d) Data augmentation. As discussed in the lecture, the more versatile the training data is, the better generally. Investigate two ways to create more training data with synthetically modified data by augmenting the (observation, action) - pairs the simulator provides. Does the overall performance change?

- Mirror input images, agent is driving more left than right turns.
- Rotate the images just slightly.

e) Fine-tuning. What other tricks can be used to improve the performance of the network? You could think of trying different network architectures, learning rate adaptation, dropout-, batch- or instance normalization, different optimizers or class imbalance of the training data. Please try at least two ideas, explain your motivation for trying them and whether they improved the result.

1.3 Competition (0 Points)

With each exercise sheet you are welcome to participate in a non-graded competition! Run `python3 main.py score` using your current model and submit your score to https://docs.google.com/forms/d/1drpfCu3k2-LegpCfCQBgthbdYlcmqLvsvWieNUJ1MMQ. You can see the rankings here: https://docs.google.com/spreadsheets/d/1um5xUlLlGsw9UE6FdmmnAmsic-QKvfp83Kdce7VbVF0/edit?usp=sharing. The winners for each of the 3 exercise sessions will be asked to present their approach in the very last lecture.
The evaluation works as follows: the models are tested on a set of validation-tracks. For each track, the reward after 600 frames is used as performance measure. And the mean reward from all validation tracks then forms the overall score. For the final ranking, we will run that evaluation on a set of secret tracks for every submission. The reward is -0.1 every frame and +1000/N for every track tile visited, where N is the total number of tiles in track. Good luck!

1.4 References

[1] https://papers.nips.cc/paper/95-alvinn-an-autonomous-land-vehicle-in-a-neural-network.pdf
[2] https://arxiv.org/pdf/1604.07316.pdf