CSE 571 - Robotics
Guided Project 1 - Cartpole

Due Monday, May 2nd @ 11:59pm

1 Overview

This is the **first out of two** guided projects, which will span 3-4 weeks. These projects are designed to be similar to the normal course projects in structure, but more restricted in scope and environment so students can explore more course topics in depth. For this project, you will train dynamics models with Gaussian Processes and learn a state-estimator with Convolutional Neural Networks.

The project will be completed in teams of 2 or 3. In addition to restricting scope and environment, teams MUST use PyTorch (as opposed to other deep learning frameworks) in order to implement their networks.

2 Setup

We highly suggest you install Anaconda (`https://www.anaconda.com/`) or Miniconda (`https://docs.conda.io/en/latest/miniconda.html`) to manage a Python virtual environment.

Download and unzip the codebase from `https://courses.cs.washington.edu/courses/cse571/22sp/projects/project1.zip`. Install Miniconda (`https://docs.conda.io/en/latest/miniconda.html`) and run the following commands.

```
cd project1
conda env create -f environment.yml
conda activate project1
```

The above commands create an anaconda environment with a python3.6 interpreter and activate the environment. It then installs necessary packages for this project assignment. After the commands have completed, you can run a Python script by typing `python cartpole_test.py`.

Python version. There are two major versions of Python: Python2 and Python3. Code written in Python2 may crash or produce incorrect results in Python3. In this project you must make sure your code is compatible with Python3.

3 Learning Dynamics and State-Estimation Models

This project is composed of two sections: learning dynamics with Gaussian Processes and learning state-estimation with CNNs and RNNs. The code to be modified for this project is in `cartpole_test.py` for Section 3.1, and `cartpole_data.py` & `cartpole_state_est.py` for Section 3.2

3.1 Learning dynamics with GPs

In this section, you will implement Gaussian processes to approximate the dynamics of a cartpole system. A cartpole is an underactuated system with a pendulum fixed to a cart and is controlled by applying a force on the cart. We represent the state x and control u of the cartpole system as follows:

```
Cartpole state:
     dθ        [rad/s] angular velocity of the pendulum
     dp        [m/s]   velocity of cart
     θ         [rad]   angle of the pendulum
     p         [m]     position of cart
Control:
     u         [N]     force on cart
```

Our task is to learn a model of the cartpole dynamics:

$$x_{t+1} = f(x_t, u_t)$$

from data using Gaussian processes. We do this by interacting with a cartpole simulator which, over time provides us with the proper training data needed for the GP. We make two important modifications to the learning problem:

- First, we use an augmented state ($\hat{x}$) instead of the actual state (x) as input to the GP. The augmented state replaces the pendulum angle θ with the pair $[sin(\theta), cos(\theta)]$ in order to avoid the wrap-around issue with angles.
- Second, we predict delta-values of the state (dx_t), rather than the state directly. This reduces the difficulty of the problem as the model needs to only capture changes in state, centers the predictions (approximately) around zero and prevents wrap around issues when predicting θ.

We now have the following inputs and outputs to the GP:

```
GP dynamics model input:
     [dθ_t, dp_t, sin(θ_t), cos(θ_t), p_t, u_t]
GP dynamics model prediction:
     [Δdθ_{t+1}, Δdp_{t+1}, Δθ_{t+1}, Δp_{t+1}] # delta-state, not next state
```

The input of the GP is $6D$ while the output is $4D$. Since a GP only predicts a 1-dimensional output, we will train 4 separate GPs for this problem.

As discussed in class, GPs use kernel functions to describe the covariance of random variables. You are free to choose a kernel function of your choice, but here some common kernels for reference:

- **Squared Exponential Kernel**: This is the kernel we discussed in class.

$$k(x_i, x_j) = \sigma_f^2 \exp(-\frac{(x_i - x_j)^T M (x_i - x_j)}{2})$$

  where σ_f is a scale factor for the kernel and M is a metric measuring distance between two input vectors. In the 1D case, $M = \frac{1}{l^2}$ where l is the length scale of the kernel.

- **Matern Kernel**: This kernel is used commonly in many machine learning applications.

$$k(x_i, x_j) = \frac{2^{1-\nu}}{\Gamma(\nu)} \left(\frac{\sqrt{2\nu} r}{l} \right)^{\nu} K_\nu \left(\frac{\sqrt{2\nu} r}{l} \right)$$

  where ν and l are (positive) parameters of the kernel and $r = |x_i - x_j|$. K_ν is a modified bessel function and Γ is the gamma function. Good parameters settings for ν are 0.25 - 3.

3.1.1 Learning the dynamics model

Let us now look at the training process for the Gaussian Process dynamics model. The system is setup to train in epochs, using data from the cartpole simulator to train a model that becomes more accurate with each epoch. Each epoch proceeds as follows:

1. At the start of an epoch, a random initial state x_0 and a sequence of H controls are chosen (either randomly or via a preset policy) $U = \{u_0, u_1, \ldots, u_{H-1}\}$. The code propagates these controls through the cartpole simulator to generate a rollout (trajectory) $\{x_0, u_0, \ldots, x_{H-1}, u_{H-1}, x_H\}$.
2. During this epoch, we will compute our GP predictions $\hat{x}_{t+1} = f_{\text{GP}}(x_t, u_t)$ for every t, and (visually) compare this with the actual state x_{t+1} to evaluate how well our GP captures the dynamics of the cartpole. Additionally, the ground truth trajectory will be added to the training dataset for the GP **at the end of the current epoch**.
3. To (visually) evaluate how well our GP model is able to predict the dynamics of the cartpole, we make predictions using the GP model, conditioned on the simulator rollouts from previous epochs. ***You need to implement this step***. Alg 1 shows pseudocode for generating the rollout. We compute the mean and variance of the predicted next state using the GP and propagate the mean prediction across time. We call this a *Mean-Rollout* since we discard the actual next state distribution and rollout only the mean.
4. The *Mean-Rollout* captures how well the GP does in expectation. In this step, we would like to see how the variance of the GP behaves as we predict across a long time horizon. Unlike the previous case, we will now

Algorithm 1 Mean-Rollout using GP (f_{GP}) - TO BE IMPLEMENTED

Inputs: Initial state: x_0, Training inputs: X, Training targets: Y.
Output: Future states: $\{x_1, x_2, \ldots, x_H\}$
for $t = 0 : H - 1$ **do**
  $\hat{x}_t = g(x_t)$ ▷ Create augmented state
  for $k = 0 : 3$ **do** ▷ Predict using separate GP per output dimension
    $(\mu_t[k], \sigma_t[k]) = f_{GP}[k](\hat{x}_t, u_t)$ ▷ Predict delta-state mean and variance. $a[k] = k$th element of a
    $x_{t+1}[k] = x_t[k] + \mu_t[k]$ ▷ Compute next state's mean

make use of the complete Gaussian distribution predicted by the GP. Instead of using the mean delta-state, we will sample from the distribution around the mean, based on the predicted variance. This will result in trajectories that capture the uncertainties in the predictions over longer horizons. ***Once again, you need to implement this step***. Alg 2 shows the pseudocode for generating sampled rollouts. Ideally, to fully represent the uncertainty, we would generate multiple samples from the state distribution at each timestep of each trajectory. Unfortunately, this would result in the number of samples increasing exponentially over time. To avoid this, we only sample once from the distribution at each timestep and repeat this N times to generate the rollouts. In practice, these samples will be near the Mean-Rollout initially and slowly move away as the uncertainty increases over time.

Algorithm 2 Sampled Rollouts using GP (f_{GP}) - TO BE IMPLEMENTED

Inputs: Initial state: x_0, Training inputs: X, Training targets: Y.
Output: Sampled Future states (N samples per timestep): $\{x_1^0, x_2^0, \ldots, x_H^0\}, \{x_1^1, x_2^1, \ldots, x_H^1\}, \ldots, \{x_1^{N-1}, x_2^{N-1}, \ldots, x_H^{N-1}\}$
for $j = 0 : N - 1$ **do** ▷ Generate N sampled trajectories
  for $t = 0 : H - 1$ **do**
    $\hat{x}_t^j = g(x_t^j)$ ▷ Create augmented state. $\forall j,\ x_0^j = x_0$
    for $k = 0 : 3$ **do** ▷ Predict using separate GP per output dimension
      $(\mu_t^j[k], \sigma_t^j[k]) = f_{GP}[k](\hat{x}_t^j, u_t)$ ▷ Predict delta-state mean and variance. $a[k] = k$th element of a
      $s \sim \mathcal{N}(\mu_t^j[k], \sigma_t^j[k])$ ▷ Sample a delta-state from the predicted Gaussian
      $x_{t+1}^j[k] = x_t^j[k] + s$ ▷ Compute next sampled state

5. Finally, at the end of each epoch, we compare the predictions from the GP (Mean/Samples) with the ground-truth from the cartpole simulator. The system displays the mean trajectory, the N rolled out trajectories and plots the GP and simulator predictions with uncertainties. Also, the predictions from the cartpole simulator are added to the training data from the previous epoch. We use the augmented state and control pair $[\hat{x}_t, u_t]$ as the training inputs with the delta-states dx_t as the targets.

3.1.2 Remarks:

A few points to note:

- The hyper-parameters (for each GP) are given to you for this task. You are encouraged to modify the hyper-parameters and see how the system behaves.
- With a successful implementation, you will see that the system does quite poorly at the start (the rollouts are all over the place) and starts to improve very fast. The predictions should match well against the

simulator with low variance. One case where the system fails to do well even with a lot of training data is when the pendulum is upright as even a small force there can cause large changes in the state.

- When sampling from a Gaussian to sample GP rollouts, please use `np.random.RandomState.normal()`. See the code for more details. This will allow our autograders to fix the random seed and grade your results accurately.
- When implementing mathematical code in Python/Numpy, try to minimize the amount of for loops. If a for loop can be replaced with vectorized computation, it will greatly increase the efficiency of the code. Here is a nice tutorial about this and other advantages of Numpy: `https://realpython.com/numpy-array-programming`.

The code to be modified for this part of the project is in `cartpole_test.py`.

You can find a video of a correct implementation for the first 12 epochs at `https://drive.google.com/file/d/1N-KXEfa5hPSxcQ-mOPsEvv7KIxOBpfhW/view?usp=sharing`. Your results might be slightly different based on how you sample from the gaussian for the 10 rollouts. The results should have the general trend of high variance for the first few epochs (evidenced by the rollouts - transparent cartpoles moving all over) and lower variance as you get more data (tighter clumping of the rollouts).

3.2 Learning state-estimation with CNNs

Now that we have a learnt dynamics model, we can get started with state-estimation. Note that while training GPs in Section 3.1, we used ground-truth positions and velocities from the simulator. In most real-world settings we don't have access to perfect state information, so we often train state-estimators to map raw observations like camera images to states like position and velocity.

In this section, you will train a CNN to estimate the state of the cartpole: $[d\theta_t, dp_t, sin(\theta_t), cos(\theta_t), p_t]$ given a sequence of past image observations. A single observation is insufficient to predict velocities, so you will have to train RNNs (Recurrent Neural Networks) or stack several frames to encode past observations.

To get started, `project1.zip` includes some boilerplate code to generate a dataset and train a model with PyTorch:

- `cartpole_data.py`: A script that rolls out a policy that switches between random and swing-up maneuvers. The script will dump compressed numpy files (`npz`) containing image and state pairs. You can modify this script to include any additional preprocessing to help with training.
- `cartpole_state_est.py`: This file contains code for the dataloader, model architecture, training & eval loops. Modify `DynamicsDataset` to preprocess and batchify the dataset. The network architecture needs to be implement inside `StateEstimator`. Choose a good loss function for regressing to continuous states. Finally, update the training loop to suit your needs.

You will need a GPU for training the model. If you don't have access to one, use a Colab notebook (see Section 5). This section is also more open-ended than Section 3.1 and you free to make your own design choices while constructing networks, choosing architectures, and hyperparameters. You can even explore alternatives to CNN-RNNs like Vision-Transformers if you are familiar with getting them to work.

For reference, checkout PyTorch tutorials for training:
CNNs: `https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html` and
RNNs: `https://pytorch.org/tutorials/intermediate/char_rnn_classification_tutorial.html`

3.2.1 Tips for training

- Visualizing your dataset will save you a lot of debugging time. Set `VISUALIZE=True` in `cartpole_data.py` to see the rollouts during data generation.
- Start off by overfitting your network to a small training dataset. The training loss should go to zero if your model can memorize the data. If not, there is probably a bug in your code.
- Normalize the input data before feeding it into your model.
- Efficient batching can greatly reduce the training time.
- Use monitoring frameworks like Tensorboard or Weights&Biases to babysit your training and evaluation curves.
- Checkout more tips from Stanford's CS231N: `https://cs231n.github.io/neural-networks-3/`

3.3 Bonus

Once you have a reliable state-estimator, plug-it back into your GP dynamics model by replacing the ground-truth state information with position and velocity estimates from your CNN model. How well does the dynamics model perform with estimated states?

4 Deliverables

You will be using Gradescope `https://www.gradescope.com` to submit the project.

Video: Include rollout videos from the GP dynamics model with ground-truth state. `cartpole_test.py` should automatically record MP4 files. *Bonus*: Include a video from the GP dynamics with CNN state-estimation.

Code: Zip the `project1` folder and include a README (if needed) so that the TAs can replicate your results. Upload the zip file to the assignment named **Guided Project 1 Programming** on Gradescope.

Report [2 Pages]: Please submit the report as a PDF to the assignment named **Guided Project 1** on Gradescope. In the PDF, answer the following questions:

- **Both**: What worked and didn't work (if anything).
- **Dynamics Model**: What if we replaced GPs with a feed forward network. What are the benefits of GPs over neural-networks and vice-versa?
- **State-Estimator**: Describe your network architecture and any hyperparameters.
- **State-Estimator**: Report the total test-set errors for $[d\theta_t, dp_t, sin(\theta_t), cos(\theta_t), p_t]$.

5 Resources

- If you don't have access to a machine with a GPU, you can use Google Colab (access to a free GPU for up to 12 hours per session). You only need a Google account. Colab operates as notebooks, very similarly to Jupyter Notebooks if you have used them. PyTorch/TensorFlow is pre-installed in the Colab notebooks, so no need for manual installation. You can find more information here: `https://colab.research.google.com/notebooks/intro.ipynb`
- PyTorch: A framework that allows you to do Numpy-esque computations on GPU. It also has a lot of support for autodifferentation and neural networks, making it one of the most popular deep learning frameworks. You are required to use this framework for the guided projects. See tutorials here: `https://pytorch.org/tutorials/`