IIA project SF3: Machine Learning

Easter 2019
Project Leader: Gabor Csanyi (gc121)
Project academics: Richard Turner (ret26), J. Miguel Hernandez-Lobato (jmh233)
Project demonstrators: Erik Daxberger (ead54), Adrià Garriga Alonso (ag919)

Admin

Important deadlines

Final project report deadline: **Friday 7 June 2019, 4pm** (electronic submission via Moodle)

Interim report deadline: **Tuesday 21 May 2019, 4pm** (electronic submission via Moodle) (Interim report should contain report on tasks up to and including the linear modelling of the dynamics)

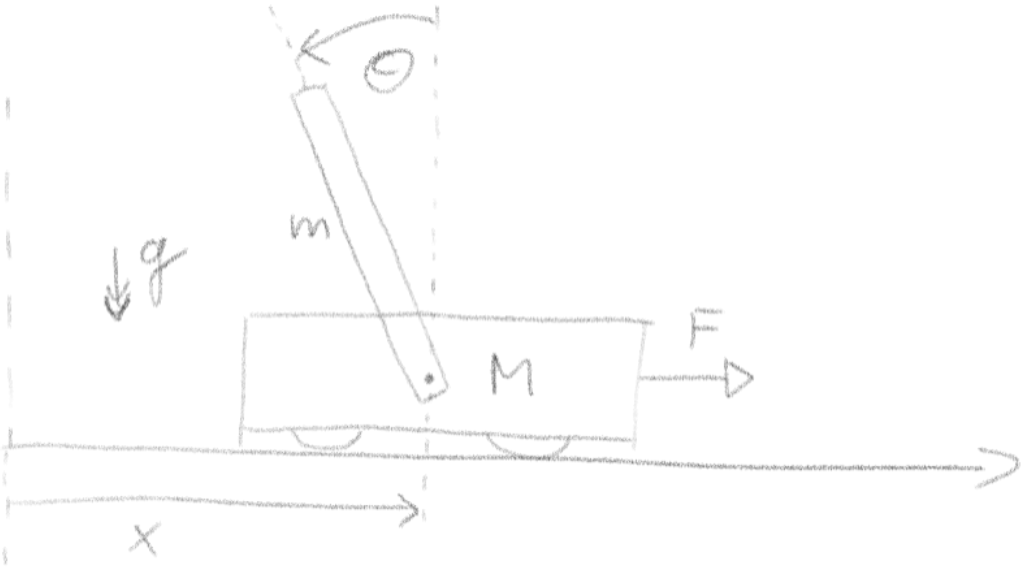
Project notes

- You should spend about 20 hours a week on the project.
- Project is to be carried out on your own (recommended) or departmental computers.
- Make your own schedule. Help is available from project leader and demonstrators during shceduled sessions: Tuesdays 9-11am, Tuesdays 2-5pm, Fridays 11-1pm all in Oatley 1 or Oatley 2 room. None of the subsequent sessions are compulsory, but you are strongly encouraged to seek verbal feedback after your interim report.
- Project carries 80 marks overall:
  - 20 marks for interim report
  - 60 marks for final report
- Project report
  - Should be clearly broken down by *Tasks* (see handout below), any notes you wish to make in how you structured and carried out the tasks, and most importantly your **results** in the form of completely labelled graphs, and **accompanying conclusions** you draw from your results.
  - Should be about than 14 pages (Interim report about 4 pages) when converted to a PDF (excluding appendices such as attached code, but *including* figures). The final report can be an extension of the interim report.
  - When deciding what to include in your report, how to organise it, and what to emphasize, please prioritise communicating understanding over formalities - I would like give you marks for doing the right thing, and showing that you did it and understand it. If I have to wade through pages of undigested data and graphs shown just because it was there, I will feel less generous. The length requirements are guidelines.
  - **All code** that you used during to project must be attached as an appendix to your reports. If you modified the `CartPole.py` file, include it.
  - Python notebooks are acceptable as a report, as long as it is "clean" (its main

section includes text and figures) and reads like a report, and is converted to a PDF.
    - Incude [cover sheets (http://teaching.eng.cam.ac.uk/node/3344)](http://teaching.eng.cam.ac.uk/node/3344) provided by the Teaching Office

Approximate Schedule

- Week 1: Software tools, simulation of cart-pole system
- Week 2: Linear and nonlinear modelling of dynamics
    - Interim report (up to linear modelling)
- Week 3: Control (linear and nonlinear)
- Week 4: Sensitivity analysis
    - Final report



Week 1: Dynamical simulation

Consider the inverted pendulum system ("cartpole"), familiar from the coursework of 3F2, with a freely moving cart and freely rotating pendulum attached to the cart, with gravity.

The equations of motion of the system are

$$\begin{aligned} 3\ddot{x}\cos\theta + 2L\ddot{\theta} &= -3g\sin\theta \\ (m+M)\ddot{x} + \tfrac{1}{2}mL\ddot{\theta}\cos\theta - \tfrac{1}{2}ML\dot{\theta}^2\sin\theta &= F \end{aligned}$$

and that $\theta = 0$ and $\theta = \pi$ are stationary points. F is the external *action* (force) on the cart.

The state of the system is described by the following variables: position and velocity of the cart $x, \dot{x}$, angle and angular velocity of the pole $\theta, \dot{\theta}$, with the angle being periodic on $[-\pi, \pi]$. The center position of the cart corresponds to $x = 0$, and the pole hanging vertically down corresponds to $\theta = \pi$.

Task 1.1

Study the code in the `CartPole.py` file, which creates a Python class to describe the system. Note the variables that describe to the state of system, and the `performAction()` function that updates the state variables using the *Euler* algorithm (it does a small number of steps), using a given force (which is the 'action') on the cart. Passing a zero value for the force corresponds to free dynamics.

Write code to simulate a “rollout” (i.e. a run with specified initial condition simulated for a number of time steps) using the `performAction` funtion in a loop, starting from the stable equilibrium position and some nonzero initial cart velocity or angular velocity, and no applied force. Plot the resulting time evolution of the system variables. Vary the size of the initial velocities to realize different behaviours: simple oscillation around the stable equilibrium, and also the complete rotation of the pendulum. Useful ranges are as follows. Cart velocity: $[-10, 10]$, pole angle: $[-\pi, \pi]$, pole (angular) velocity: $[-15, 15]$.

You can plot all variables as a function of time, and also pairs of variables against one another (similar to phase portraits).

Note how the angle is used in the dynamics as a continous variable, rather than just in the range $[-\pi, \pi]$. There is a `remap_angle` function in the `CartPole` module that you can use to get the angle in the usual range. *This will be an important consideration later on when we develop models of the dynamics.*

Changes of state

You know from 3F2 that a simple linear controller works for this system, as long as you know where the stationary point is, and have access to the equations of motion so that you can linearise them. But in general, we do not know the equations behind the evolution of a physical system, and so we will take a different approach. What do have are *observations* of the time evolution of the system. So we will use the simulations like the ones you did above to gather data about the system, and develop a *model* for this time evolution.

We will want to build a *model* for the time evolution of the system. The model is a function $f(X)$ that takes the current state of the system, and maps it onto a new state, which is its prediction for the state at a later time. Let the state of the system be described by a vector X, given by

$$X = [x, \dot{x}, \theta, \dot{\theta}]$$

Given the current state X, let us call Y the state of the system after a single call to the `PerformAction` function (with 0.0 as the force argument).

Task 1.2

To investigate and visualise the functional relationship between X and Y, initialise the system using a random value for all state variables, and then scan through one of the state variable in a suitable range (don't forget to reset the state variables after each call to `PerformAction`), and plot Y as a function of your scan.

You will observe that the relationship between X and Y as defined above is nearly linear, which is not surprising because the change in one step is small.

We can take account of this and model the *change* in state vector, rather than taking the new state vector itself as the target of our model. So we define the new target for the modelling as $Y \equiv X(T) - X(0)$, where $X(t)$ represents the time evolution of the state under the dynamics, and T corresponds to a single call to `PerformAction`. Note that in principle we could model changes corresponding arbitrary time shifts, rather than a single call to `PerformAction`, but the longer the time shift, the more complex the model would have to be.

Explore this new functional relationship again using scans, or even contour plots where you take slices of the data in two of the variables while you keep the other two variables fixed (the `tricontourf` function of `matplotlib` is very useful). One of the variables has no effect on the next step - which one?

Week 2: Modelling

Task 2.1

By initialising the simulator in a completely random state (using suitable ranges) and running it for *one* step, gather data in the form of pairs of state vectors (X, Y), where X represents a state of the system at step n, so $X \equiv X(n)$ and Y represent the change in state after a single call to `performAction` (with zero force), so $Y \equiv X(n+1) - X(n)$. Start with 1000 data points.

Linear model

The simplest model is a linear one, where the target Y is assumed to be linear function of the current state X,

$$f(X) = \mathbf{C}X$$

where $\mathbf{C}$ is a 4×4 matrix of coefficients.

Task 2.2

Using your data set, do linear regression to find the optimal coefficient matrix. Test your predictions against the data. One way to plot the results is to put the input state variable on the horizontal axis and on the vertical axis put the predicted state variable (i.e. what should be the "next step") and the real next step. Another way is to put the target data (i.e. the actual "next step") on the horizontal axis and the predicted "next step" on the vertical axis. In this latter plot, a perfect prediction would correspond to a perfect straight line. You should also repeat the "scans" from the previous task, and plot simultaneously the real change in state with your predicted change in state as a function of your scan. Which variables are predicted well by the linear model and which ones are not? Why ?

Task 2.3

The true test of the model is not how it matches with the gathered data it was fit to, but whether it can predict the time evolution of the physical system. Iterate the model to predict the time evolution of the system, and compare using various initial conditions how accurate the predictions are with respect to the true dynamics started from the same initial conditions. (Note that the model is being used deterministically, with no noise added)

Since your models above predict the *change* in the state variable, the iterated time evolution is

$$X_{n+1} \leftarrow X_n + f(X_n)$$

Plot the true time evolution of the system as well as that of your fitted models for many cycles, and for different initial conditions, including ones where the pole makes a full circle.

Angle range If you leave the angle without remapping, your solution with the iterated model will diverge. Why is that? Ensure that you remap the angle during the above iterations. (Note how remapping is not needed in the true dynamics, since that is nonlinear, and the angle only appears inside trigonometric functions that are periodic anyway).

Nonlinear model

As you observed above, the linear model is not particularly good. In order to do better, we need nonlinear modelling. Next you are going to do build a nonlinear model using a linear regression with nonlinear basis functions. Given a data set of (X,Y) pairs, the model function is given by

$$f(X) = \sum_i \alpha_i K(X, X_i)$$

where the sum runs over the basis functions, α_i are the corresponding coefficients, and K is a *kernel function* that is used to define the nonlinear basis. The kernel function takes two arguments, the first one X is the state vector where you evaluate the basis function, and the second argument, X_i is another state vector which we use to place the basis function somewhere in the state space. To make the basis functions relevant, we take the set of locations $\{X_i\}$ to be a subset of the gathered data points.

For the present problem, let us use a Gassian kernel function,

$$K(X, X') = e^{-\sum_j \frac{\left(X^{(j)} - X'^{(j)}\right)^2}{2\sigma_j^2}}$$

Here $X^{(j)}$ refers to the jth component of the state vector. There is one caveat for using this kernel function in our current situation: one of our state vector components, θ is periodic. It helps quite a bit if we introduce this periodicity in our kernel function, and we can do that by using $\sin^2\left((\theta - \theta')/2\right)$ in place of $\left(\theta - \theta'\right)^2$ in the part of the kernel function that corresponds to the angle variable.

Suppose we collected N pairs of (X, Y) data pairs, and we choose a subset M of the X locations to serve as basis function centres. Then the vector of linear coefficients of the fitted model are given by

$$\alpha_M^{(j)} = (K_{MN}K_{NM} + \lambda K_{MM})^{-1} K_{MN} Y_N^{(j)}$$

where K_{MM} is an $M \times M$ matrix with elements that are given by the kernel function evaluated between the X locations selected as basis locations, K_{MN} is an $M \times N$ matrix, analogous but with elements corresponding to the M basis locations and all the N data points, Y_N are the target function values. We have to have four separate models for the four components of the state vector, these are indexed by j. The above solution is the *regularised least squares* solution, with regularisation strength λ. You need to experiment with different values of λ (e.g. between 1E-6 and 1E-1, on a log scale). You will also need to select the length scale parameters σ_j for each state variable. A good start is the standard deviation of the state variable in your dataset.

Note: You should not use the `np.linalg.inv` function to invert the matrix in the above formula, because that can be numerically unstable for such ill-conditioned matrices, but instead use `np.linalg.lstsq`, which solves equations of the form $Ax = b$ in a least-squares sense.

Task 2.4

Fit a model of the system using the data you gathered earlier. Your target function could be either again the change in state after one step as before, or the *error of your linear model* in the change in state. Verify using scatterplots that the nonlinear model indeed fits the data. Study the convergence of the model (i.e. the systematic reduction in error) as a function of increasing data amount, and the increasing number of basis functions (e.g. start with $M = 10$ and increase by factors of 2, select the data locations for the basis randomly from the data). Also plot 2D slices of your target function and the fit, as well as do roll-outs to see how closely the iterated model matches the real dynamics for a wide range of sensible initial conditions.

An alternative to random data locations is to use *quasirandom sequences*, these provide a set of locations in an arbitrary dimensional unit cube that are "nicely" spaced out for sampling, better than random draws. You can use the `sobol_seq` module to generate such locations (read the on-line example, but each new location is obtained with a call to the `vec, seed = sobol_seq.i4_sobol(4, seed)` function for 4 dimensions, with the seed provided by the previous call)

Week 3: Control

Having developed a good model for the dynamics, now it is time to control the system. When you call the `performAction` routine, it takes a signed scalar which is interpreted as an external force on the cart. The value is passed through the `tanh` function before being interpreted as a force, this prevents the application of excessively large forces (the transformation is controlled by the `max_force` variable inside the `CartPole` class)

So the first thing you will need do is to modify your models (both the linear and nonlinear) to take account of this new state variable (i.e. your system now has 5 inputs, including the force F, and 4 outputs after a call to `performAction`).

Task 3.1

Change your code so that the state vector now includes the action taken. Collect new data, again using random initial conditions or quasi-random sequences and one step, but this time include the action. Verify using scatter-plots, 1D and 2D scans and roll-outs that your models can predict the change in the state variables.

Policies

A *policy* is a function $p(X)$ that defines what the action should be given the other state variables. The ultimate goal is to find policy functions that when enacted, give rise to some desired behaviour, in this case the pole being balanced around its unstable equilibrium position.

We start with defining a linear policy,

$$p(X) = \mathbf{p} \cdot \mathbf{X}$$

with unknown coefficent vector $\mathbf{p}$.

Objective

In order to optimize a policy, we need to define what we want. That is encapsulated in an *objective function* (also called *loss function*), a measure of how close we are to the desired behaviour. In the present case, we want the pole to be upright, and the pole velocity to be small at least when the pole is near vertical. E.g. we could use the *loss function*

$$l(X) = -\cos\theta$$

Since it takes its minimum value (-1) when the pole is upright. But we can also desire that the pole should not be moving too fast when it is upright, so a combined loss could be

$$l(X) = -\cos\theta + \dot{\theta}^2 e^{-\alpha(1-\cos\theta)^2}$$

where the exponential factor in the second term turns off the pole rotation part of the loss when the pole is downwards.

The above loss functions are for a given state. We wish to keep the pole upright continously, so the total loss should be a time integral (sum, really) of the pointwise loss over some interval,

$$L = \sum_{i=1}^{N} l(X_i)$$

The `CartPole` class contains functions to evaluate the $l(x)$, you can use that or make up your own.

Task 3.2

Write code to evaluate the loss function for the trajectory of a rollout - use a short time horizon, but enough to capture 1-2 oscillation periods. Before any optimisation, *visualise* the loss function as you vary some parameters in $\mathbf{p}$, using similar 1D and 2D scans that you did for Task 1, i.e. varying just one or two elements of $\mathbf{p}$ and plotting the loss as a function of those elements.

Optimise the unknowns in the policy to minimise the loss function. Since you do not have too many variables, you can do this by looking at how the loss changes for small changes in policy variables. Feel free to use off-she-shelf optimizers, e.g. from the `scipy.optimize` package. For low dimensional problems without derivatives, you can use the `Nelder-Mead` method.

Do no expect the loss function to be simple, or to have only a few minima - it is likely that you need to explore a variety of initial conditions. But there is not just one solution!

Alternativey, you can use the `autograd` module to obtain a *gradient* of the loss function with respect to the policy variables, and perform *gradient descent* optimisation. This is much better for high dimensional problems. Note that in order to use `autograd` , you need to import `autograd.numpy` instead of the regular `numpy` and there are limitations on what functions and constructs you can use in the function that you want to take the gradient of.

When using autograd, calculate the rollout and the loss *using your model* rather than running the real dynamics. The idea is that we mimic a situation in which you don't have access to the underlying dynamical equations, so no access to their gradients either

When you use such model-rollouts you need to limit the time horizon (number of steps) to where you think your model is still accurate. Use the nonlinear model developed earlier, rather than the linear model, regardless of whether your policy is linear or not.

Your linear policy should be able to keep the pole upright, if it is started near the unstable position. How close do you have to start? Plot the time evolution of the variables under the policy to demonstrate that the pole is kept upright.

Task 3.3

Define a *nonlinear* policy, using a similar construction that we used for modelling the system, but this time we are modelling the `force` variable as a function of the system state variables. Use between 5-20 Gaussian basis functions, and you can optimise not only the magnitude of each, but also the location of Gaussian centres and the widths in each dimension. Optimise this nonlinear policy, and try to obtain a policy that can keep the pole upright starting from the stable equilibrium (down) position.

Here, you can write a function that iterates your model forward with the current action function and computes the loss, and `autograd` allows you to take the derivative of this loss with respect to model parameters.

When you do model-rollouts, you can start the system from the stable equilibrium position, but you can also include starting configurations that are elsewhere in configuration space, this lets you probe larger parts of the space still with a limited time horizon of the model-rollouts.

Week 4: Sensitivity

- Introduce noise in the observed dynamics
  - Refit the models, observe degradation in predictions
  - Add known amount of noise to the actual dynamics during policy training
  - Refit models on noisy data, taking account of the known amount of noise
- Introduce nonzero friction in the model, refit the model and the policy

Task 4

Study the sensitivity of your policies to the above variables.

In [ ]: