Ian MacDonald

ESE 650

Summary 2 (Chapters 6-9)

Chapter 6: Linear Quadratic Regulators (LQR)

Linear Quadratic Regulators are feedback controllers that can be used to solve dynamic programming problems, including those where the state/control space are infinite. LQRs and their variants can be used to solve many real-world control problems.

Discrete Time LQR

The discrete time LQR is a dynamic programming solution for a linear dynamical system that finds a sequence of control inputs that minimize the function below.

$$J(x_0; u_0, u_1, \ldots, u_{T-1}) = \frac{1}{2} x_T^\top Q_f x_T + \frac{1}{2} \sum_{k=0}^{T-1} \left(x_k^\top Q x_k + u_k^\top R u_k \right) \tag{6.3}$$

The optimal controller at each time step is a linear function of the state at that time step. The Kalman gain, which is used to calculate the optimal control, also changes with each time step, but along with the cost-to-go matrix does not depend on the state, and can be calculated beforehand if the time horizon is known.

Hamilton-Jacobi-Bellman Equation

The Hamilton-Jacobi-Bellman Equation is used for continuous-time optimal control problems, where the goal is to solve the following equation.

$$J^*(x_0) = \min_{u(t),\ t \in [0,T]} \left\{ q_f(x(T)) + \int_0^T q(x(t), u(t))\, \mathrm{d}t \right\} \tag{6.6}$$

The equation can be nicely incorporated when looking to find a control trajectory that minimizes a cost function for the continuous-time LQR problem, modifying the bracketed part of the equation above to be the following.

$$\frac{1}{2}\, x(T)^\top Q_f\, x(T) + \frac{1}{2} \int_0^T x(t)^\top Q\, x(t) + u(t)^\top R\, u(t)\, \mathrm{d}t.$$

Stochastic LQR

Stochastic LQR looks to solve for the optimal control sequence when given a linear dynamical system with standard Gaussian noise. Surprisingly though, the addition of noise does not have an effect on the optimal controls, as the optimal controller that we should pick in this situation is identical to the one given by the standard LQR problem. The fact that this optimal control sequence is the same

whether the system has noise or is deterministic is a special property of the LQR problem, and is not true for other dynamical systems (nonlinear/non-Gaussian noise, for example) or other costs.

Linear Quadratic Gaussian (LQG)

The Linear Quadratic Gaussian method is applied when you want to compute the optimal control but only know the states through observations. The LQG essentially solves this by running a Kalman Filter in parallel with the deterministic LQR controller, so that even if we don't observe the state exactly we can still get the optimal control. This solution only applies for linear dynamical systems with linear observations, Gaussian noise in both the dynamics and the observations, and quadratic terminal and run-time costs.

Iterative LQR (iLQR)

The Iterative LQR is used to solve for the optimal control sequence for nonlinear dynamical systems. It works by starting with an initial control sequence, and then iteratively updates it by linearizing about the current state trajectory, solving for a new control trajectory to replace the previous one, and then using that control trajectory to get a new state trajectory for the next iteration. While it is not guaranteed to find the optimal solution, it does work well in practice.

Chapter 7: Imitation Learning

This chapter provides a general introduction to supervised machine learning and deep learning methods, as well as introducing the concept of Imitation Learning. Imitation Learning is the process of learning from demonstrations, which is one technique under the broader concept of learning from data, which is the focus of Reinforcement Learning, which is discussed in the following chapters.

Supervised Learning Overview

The goal of supervised learning is to find a function that can accurately predict a target value for any given input value. This is done by using a set of data known as the training set, which consists of a number of input values and their corresponding labels. The training set is then fed into a model, which will learn a function that can map the training data to their corresponding labels. Ultimately, you want to find a model that learns from the training set but still generalizes well outside of it. How well the model will perform outside of the training set can be estimated by using a separate set of data known as the test set to evaluate the performance. If the model performs well on the training set but poorly on the testing set, then it is likely overfitting, learning functions that work very well specifically for the training set but work poorly outside of it. On the other end of the spectrum is underfitting, where the model doesn't learn from the training data enough and ends up performing poorly because of it. The ideal supervised machine learning model is the one that strikes the right balance between them.

Deep Neural Networks

While much of supervised learning has to do with learning linear functions, often times there exists no such linear function that works for the data. In these cases, deep neural networks may be a good fit, as they fit nonlinear functions to the data. Deep neural networks consist of multiple successive layers, with the output of each intermediate layer passed to the next layer in a sequential manner. Essentially, the network creates it's own features at each step through composition of older features. The weights used to calculate the outputs at each layer are updated using the backpropagation algorithm once the final output is calculated.

Behavior Cloning

Behavior Cloning is a form of Imitation Learning where the controller learns by copying the actions of an expert demonstrator. This type of model can have varying degrees of success. If the expert operates in an environment similar to the one that the trained controller will operate in, then it may do quite well. However, in the worst-case scenario where the two environments are very different, than the learned controller will likely do poorly.

DAgger: Dataset Aggregation

The DAgger Algorithm is a way to iteratively update the data set that the controller is learning from. It starts by fitting a controller to the original expert dataset. Then, at each iteration, it updates the dataset with some examples that the learned controller has seen, uses this updated dataset to calculate an updated controller, and then repeats.

Chapter 8: Policy Gradient Methods

Policy gradient methods are a subset of Reinforcement Learning that look to learn the optimal controller by minimizing a given cost function over the trajectories of an unknown dynamical system.

Standard RL Problem Overview

The standard RL problem looks to find the optimal controller that maximizes the average discounted return of some reward function over some trajectory. Mathematically, it can be formulated as finding the value below.

$$\widehat{\theta} = \underset{\theta}{\operatorname{argmax}}\, J(\theta) := \underset{\tau \sim p(\tau)}{\mathrm{E}} \left[R(\tau) \mid x_0 \right].$$

Additionally, the average return is calculated using the formula below.

$$\widehat{J}(\theta) \approx \frac{1}{n} \sum_{i=0}^{n} \sum_{k=0}^{T} \gamma^k r(x_k^i, u_k^i)$$

Cross-Entropy Method

The optimal controller is usually found by iteratively updating it's weights using gradient descent as shown below.

$$\theta^{k+1} = \theta^k + \eta \ \nabla J(\theta)$$

Instead of calculating the gradient directly, in this method we instead use a finite-difference approximation, shown mathematically below.

$$(\widehat{\nabla} J(\theta))_i = \frac{J(\theta + \epsilon\, e_i) - J(\theta - \epsilon\, e_i)}{2\epsilon} \approx \frac{\widehat{J}(\theta + \epsilon\, e_i) - \widehat{J}(\theta - \epsilon\, e_i)}{2\epsilon}.$$

This can be more efficiently computed by using the Cross-Entropy Method, which involves sampling a few stochastic controllers centered on the current controller, and then updating the weights of the controller in the direction that leads to an increase in the average return.

Policy Gradients

The Policy Gradient allows us to calculate the objective of the gradient without having to do any of the finite-difference methods above. It works by sampling some trajectories from the system and averaging the returns of each trajectory, weighted by the gradient of the likelihood of taking each trajectory, with the gradients being calculated using the backpropagation algorithm for a neural network. The calculation of the policy gradient is shown mathematically below.

$$
\begin{aligned}
\nabla_\theta \mathop{\mathbf{E}}_{\tau\sim p_\theta}[R(\tau)] &= \nabla_\theta \int R(\tau) p_\theta(\tau) \mathrm{d}\tau \\
&= \int R(\tau)\, \nabla_\theta\, p_\theta(\tau)\, \mathrm{d}\tau \\
&\text{(move the gradient inside, integral is over trajectories } \tau \text{ which do not depend on} \theta \text{ themselves)} \\
&= \int R(\tau) p_\theta(\tau) \frac{\nabla\, p_\theta(\tau)}{p_\theta(\tau)} \mathrm{d}\tau \\
&= \int R(\tau) p_\theta(\tau)\, \nabla \log p_\theta(\tau) \mathrm{d}\tau \\
&- \mathop{\mathbf{E}}_{\tau\sim p_\theta(\tau)}[R(\tau)\, \nabla \log p_\theta(\tau)] \\
&\approx \frac{1}{n} \sum_{i=1}^{n} R(\tau^i)\, \nabla \log p_\theta(\tau^i)
\end{aligned}
\tag{8.8}
$$

Unfortunately, the variance of this calculated policy gradient is usually relatively large. However, the variance can be reduced through the use of a baseline, where we subtract a constant value from the return. There are a couple of different methods for choosing what this value should be. A simple but effective method is to use the average returns of a mini-batch as the baseline. A better but more complicated approach is to use a weighted average of the returns using the log-likelihood of the trajectory. This is the best baseline value, and will result in the largest variance reduction.

Actor-Critic Methods

Additionally, the policy gradient formula can be expressed in terms of the Q-function, and this new expression allows for some more complicated algorithms, including Actor-Critic methods. Actor-Critic methods work by fitting two deep networks – one with weights that parameterize a stochastic controller, and one with weights that parameterize the Q-function of that controller. The first network is known as the “actor”, since it is the one computing the controls, and the second network is known as the “critic”, since it is evaluating the controls chosen by the “actor”. This evaluation is iteratively performed, with the policy gradient calculated at each step and used to update the parameters of the “actor” network. However, this policy gradient also has a relatively large variance, which can be addressed by using the advantage function.

Chapter 9: Q-Learning

All of the previous methods were on-policy methods, where the current controller is used to generate the data that is then used to update the controller’s parameters. However, this requires generating an entire new set of data at every iteration, which is then immediately discarded after updating the controller. This chapter covers off-policy methods, which use past data along with new

data to update the controller at each iteration. These methods typically requires less total data compared to on-policy methods.

Tabular Q-Learning

Tabular Q-learning can be used when you have both a discrete state and control space, which allows you to create a table where each row is a state, each column is a control, and each entry in the table is the corresponding value for taking that control at that state. Then, Tabular Q-Learning can be done by solving the following optimization problem.

$$\min_{q} \frac{1}{n(T+1)} \sum_{i=1}^{n} \sum_{k=0}^{T} \|q(x_k^i, u_k^i) - r(x_k^i, u_k^i) - \gamma \max_{u' \in U} q(x_{k+1}^i, u')\|_2^2. \tag{9.1}$$

Unfortunately though, Tabular Q-learning methods have a lot of limitations. For one, they cannot handle any sort of continuous control spaces or continuous state spaces. Additionally, even for discrete spaces, if the number of controls or states is exceptionally large, then Q-Learning will be unfeasible from a storage perspective, as there won't be enough memory to hold the necessary value table.

Function Approximation (Deep Q Networks)

The limitations on Tabular Q-Learning essentially make it unusable for most real-world problems. However, it can be modified by replacing the large table of Q-values with a deep neural network that parameterizes the Q-function. The goal of this resultant Deep Q network is to find the weights that satisfy the equation below.

$$\begin{aligned} \hat{\varphi} &= \operatorname*{argmin}_{\varphi} \frac{1}{n(T+1)} \sum_{i=1}^{n} \sum_{t=1}^{T} \left(q_\varphi(x_t^i, u_t^i) - \underbrace{r(x_t^i, u_t^i) - \gamma \left(1 - \mathbf{1}_{\{x_{t+1}^i \text{ is terminal}\}}\right) \max_{u'} q_\varphi(x_{t+1}^i, u')}_{\text{target}(x'; \varphi)} \right)^2 \\ &\equiv \operatorname*{argmin}_{\varphi} \frac{1}{n(T+1)} \sum_{i=1}^{n} \sum_{t=1}^{T} \left(q_\varphi(x_t^i, u_t^i) - \text{target}(x_{t+1}^i; \varphi) \right)^2 \end{aligned} \tag{9.4}$$

In practice though, getting RL models to work can often be tricky. Below, are some points that help when implementing these models in practice.

- Pick mini-batches from different trajectories in SGD
- Use a replay buffer
    - Continuously add to this replay buffer using trajectories drawn from the controller using an epsilon-greedy policy
    - Start with a larger epsilon, slowly reduce it's value as training progresses
- Use robust regression to fit the Q-Function
    - Huber Loss is a good implementation
- Use improved targets -
    - Delayed targets
    - Update target using exponential averaging
    - Double Q-Learning Trick
    - Train two Q-Functions

Q-Learning For Continuous Control Spaces

While the methods above all work for discrete control spaces, there are times where controls are chosen from a continuous space and thus a different method must be applied. One such way is to instead use a deterministic policy gradient, after which on-policy or off-policy actor-critic methods can be applied, known as “on-policy SARSA” and “deep deterministic policy gradient” algorithms, respectively.