

# Chapter 7

## Imitation Learning

### Reading

1. The DAGGER algorithm  
(<https://www.cs.cmu.edu/~sross1/publications/Ross-AIStats11-NoRegret.pdf>)
2. [https://www.youtube.com/watch?v=TUBBIgtQL\\_k](https://www.youtube.com/watch?v=TUBBIgtQL_k)
3. An Algorithmic Perspective on Imitation Learning  
(<https://arxiv.org/pdf/1811.06711.pdf>)

This is the beginning of Module 3 of the course. The previous two modules have been about how to estimate the state of the world around the robot (Module 1) and how to move the robot (or the world) to a desired state (Module 2). Both of these required that we maintain a model of the dynamics of the robot; this model may be inaccurate and we fudged over this inaccuracy by modeling the remainder as “noise” in Markov Decision Processes.

The next few lectures introduce different aspects of what is called Reinforcement Learning (RL). This is a very large field and you can think of using techniques from RL in many different ways.

- 1. Dynamic programming with function approximation.** If we are solving a dynamic programming problem, we can think of writing down the optimal cost-to-go  $J^*(x, t)$  as a function of some parameters, e.g., the cost-to-go is

$$J_\varphi(x, t) = \frac{1}{2} x(t)^\top \underbrace{\left( \text{some function of } A, B, Q, R \right)}_{\text{function of } \varphi} x(t)$$

for LQR. We know the stuff inside the brackets to be exactly  $P(t)$  but, if we did not, it could be written down as some generic function of parameters  $\varphi$ . We know that any cost-to-go that satisfies the Bellman

equation is the optimal cost-to-go, so we can now “fit” the candidate function  $J_\varphi(x, t)$  to satisfy the Bellman equation. Similarly, one may also express the optimal feedback control  $u(\cdot)$  using some parameters  $\theta$  as

$$u_\theta(\cdot).$$

We will see how to fit such functions in this chapter.

2. **Learning from data.** It may happen that we do not know very much about the dynamical system, e.g., we do not know a good model for what drives customers as they buy items in an online merchandise platform, or a robot traveling in a crowded area may not have a good model for how large crowds of people walk around it. One may collect data from these systems fit some model of the form  $\dot{x} = f(x, u)$  to the data and then go back to the techniques of Module 2. It is typically not clear how much data one should collect. RL gives a suite of techniques to learn the cost-to-go in these situations by collecting and assimilating the data *automatically*. These techniques go under the umbrella of policy gradients, on-policy methods etc. One may also simply “memorize” the data provided by an expert operator, this is called Imitation Learning and we will discuss it next.

**Some motivation** Imitation Learning is also called “learning from demonstrations”. This is in fact one of the earliest successful examples of using a neural network for driving. The ALVINN project at CMU by Dean Pomerleau in 1988 (<https://www.youtube.com/watch?v=2KMAAmkz9go>) used a two-layer neural network with 5 hidden neurons, about 1000 inputs from the pixels of a camera and 30 outputs. It successfully drove in different parts of the United States and Germany. Imitation learning has also been responsible for numerous other early-successes of RL, e.g., acrobatic maneuvers on an RC helicopter ([http://ai.stanford.edu/~acoates/papers/AbbeelCoatesNg\\_IJRR2010.pdf](http://ai.stanford.edu/~acoates/papers/AbbeelCoatesNg_IJRR2010.pdf)).

Imitation Learning seeks to record data from experts, e.g., humans, and reproduce these desired behaviors on robots. The key questions we should ask, and which we will answer in this chapter, are as follows.

1. Who should demonstrate (experts, amateurs, or novices) and how should we record data (what states, controls etc.)?
2. How should we learn from this data? e.g., fit a supervised regression model for the policy. How should one ignore bad behaviors in non-expert data?
3. And most importantly, what can we do if the robot encounters a situation which was not in the dataset.

## 7.1 A crash course in supervised learning

Nature gives us data  $X$  and targets  $Y$  for this data.

$$X \rightarrow Y.$$

Nature does not usually tell us what property of a datum  $x \in X$  results in a particular prediction  $y \in Y$ . We would like to learn to imitate Nature, namely predict  $y$  given  $x$ .

What does such learning mean? It is simply a notion of being able to identify patterns in the input data without explicitly programming a computer for prediction. We are often happy with a learning process that identifies correlations: if we learn correlations on a few samples  $(x^1, y^1), \dots, (x^n, y^n)$ , we may be able to predict the output for a new datum  $x^{n+1}$ . We may not need to know *why* the label of  $x^{n+1}$  was predicted to be so and so.

Let us say that Nature possesses a probability distribution  $P$  over  $(X, Y)$ . We will formalize the problem of machine learning as Nature drawing  $n$  independent and identically distributed samples from this distribution. This is denoted by

$$D_{\text{train}} = \{(x^i, y^i) \sim P\}_{i=1}^n$$

is called the “training set”. We use this data to identify patterns that help make predictions on some future data.

**What is the task in machine learning?** Suppose  $D_{\text{train}}$  consists of  $n = 50$  RGB images of size  $100 \times 100$  of two kinds, ones with an orange inside them and ones without.  $10^4$  is a large number of pixels, each pixel taking any of the possible  $255^3$  values. Suppose we discover that one particular pixel, say at location  $(25, 45)$ , takes distinct values in all images inside our training set. We can then construct a predictor based on this pixel. This predictor, it is a binary classifier, perfectly maps the training images to their labels (orange: +1 or no orange: -1). If  $x_{ij}^k$  is the  $(ij)^{\text{th}}$  pixel for image  $x^k$ , then we use the function

$$f(x) = \begin{cases} y^k & \text{if } x_{ij}^k = x_{ij} \text{ for some } k = 1, \dots, n \\ -1 & \text{otherwise.} \end{cases}$$

This predictor certainly solves the task. It correctly works for all images in the training set. Does it work for images outside the training set?

Our task in machine learning is to learn a predictor that works *outside* the training set. The training set is only a source of information that Nature gives us to find such a predictor.

Designing a predictor that is accurate on  $D_{\text{train}}$  is trivial. A hash function that memorizes the data is sufficient. This is NOT our task in machine learning. We want predictors that generalize to new data outside  $D_{\text{train}}$ .

🔗 How many such binary classifiers are there at most?

76 **Generalization** If we never see data from outside  $D_{\text{train}}$  why should we hope  
 77 to do well on it? The key is the distribution  $P$ . Machine learning is formalized  
 78 as constructing a predictor that works well on new data that is also drawn  
 79 independently from the distribution  $P$ . We will call this set of data the “test  
 80 set”

$$D_{\text{test}}$$

81 and it is constructed similarly. This assumption is important. It provides  
 82 coherence between past and future samples: past samples that were used to  
 83 train and future samples that we will wish to predict upon. How to find such  
 84 predictors that work well on new data? The central idea in machine learning is  
 85 to restrict the set of possible binary functions that we consider.

We are searching for a predictor that generalizes well but only have the training data to select predictors.

86 The *right* class of functions  $f$  cannot be too large, otherwise we will find  
 87 our binary classifier above as the solution, and that is not very useful. The class  
 88 of functions cannot be too small either, otherwise we won't be able to predict  
 89 difficult images. If the predictor does not even work well on the training set,  
 90 there is no reason why we should expect it to work on the test set.

Finding this correct class of functions with the right balance is what machine learning is all about.

🔍 Can you now think how is machine learning different from other fields you might know such as statistics or optimization?

### 91 7.1.1 Fitting a machine learning model

92 Let us now solve a classification problem. We will again go around the model  
 93 selection problem and consider the class of linear classifiers. Assume binary  
 94 labels  $Y \in \{-1, 1\}$ . To keep the notation clear, we will use the trick of  
 95 appending a 1 to the data  $x$  and hide the bias term  $b$  in the linear classifier. The  
 96 predictor is now given by

$$\begin{aligned} f(x; w) &= \text{sign}(w^\top x) \\ &= \begin{cases} +1 & \text{if } w^\top x \geq 0 \\ -1 & \text{else.} \end{cases} \end{aligned} \quad (7.1)$$

97 We have used the sign function denoted as  $\text{sign}$  to get binary  $\{-1, +1\}$  outputs  
 98 from our real-valued prediction  $w^\top x$ . This is the famous perceptron model of  
 99 Frank Rosenblatt.

100 We want the predictions of the model to match those in the training data  
 101 and devise an objective to fit/train the perceptron.

$$\ell_{\text{zero-one}}(w) := \frac{1}{n} \sum_{i=1}^n \mathbf{1}_{\{y^i \neq f(x^i; w)\}}. \quad (7.2)$$

102 The indicator function inside the summation measures the number of mistakes  
 103 the perceptron makes on the training dataset. The objective here is designed to  
 104 find weights  $w$  that minimizes the average number of mistakes, also known as  
 105 the training error. Such a loss that measures the mistakes is called the zero-one  
 106 loss, it incurs a penalty of 1 for a mistake and zero otherwise.

❓ Can you think of some quantity other than the zero-one error that we may wish to optimize?

107 **Surrogate losses** The zero-one loss is the clearest indication of whether the  
 108 perceptron is working well. It is however non-differentiable, so we cannot use  
 109 powerful ideas from optimization theory to minimize it. This is why surrogate  
 110 losses are constructed in machine learning. These are proxies for the loss  
 111 function, typically for the classification problems and look as follows. The  
 112 exponential loss is

$$\ell_{\text{exp}}(w) = e^{-y(w^\top x)}$$

113 or the logistic loss is

$$\ell_{\text{logistic}}(w) = \log(1 + e^{-yw^\top x}).$$

114 **Stochastic Gradient Descent (SGD)** SGD is a very general algorithm to  
 115 optimize objectives typically found in machine learning. We can use it so  
 116 long as we have a dataset and an objective that is differentiable. Consider an  
 117 optimization problem where we want to solve for

$$w^* = \underset{w}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^n \ell^i(w)$$

118 where the function  $\ell^i$  denotes the loss on the sample  $(x^i, y^i)$  and  $w \in \mathbb{R}^p$   
 119 denotes the weights of the classifier. Solving this problem using SGD corre-  
 120 sponds to iteratively updating the weights using

$$w^{t+1} = w^t - \eta \frac{d\ell^{\omega_t}(w)}{dw} \Big|_{w=w^t},$$

121 i.e., we compute the gradient one sample with index  $\omega_t$  in the dataset. The  
 122 index  $\omega_t$  is chosen uniformly randomly from

$$\omega_t \in \{1, \dots, n\}.$$

123 In practice, at each time-step  $t$ , we typically select a few (not just one) input  
 124 data  $\omega_t$  from the training dataset and average the gradient  $\frac{d\ell^{\omega_t}(w)}{dw} \Big|_{w=w^t}$  across  
 125 them; this is known as a “mini-batch”. The gradient of the loss  $\ell^{\omega_t}(w)$  with  
 126 respect to  $w$  is denoted by

$$\nabla \ell^{\omega_t}(w^t) := \frac{d\ell^{\omega_t}(w)}{dw} \Big|_{w=w^t} = \begin{bmatrix} \nabla_{w_1} \ell^{\omega_t}(w^t) \\ \nabla_{w_2} \ell^{\omega_t}(w^t) \\ \vdots \\ \nabla_{w_p} \ell^{\omega_t}(w^t) \end{bmatrix} \in \mathbb{R}^p.$$

127 The gradient  $\nabla \ell^{\omega_t}(w^t)$  is therefore a vector in  $\mathbb{R}^p$ . We have written

$$\nabla_{w_1} \ell^{\omega_t}(w^t) = \left. \frac{d\ell^{\omega_t}(w)}{dw_1} \right|_{w=w^t}$$

128 for the scalar-valued derivative of the objective  $\ell^{\omega_t}(w^t)$  with respect to the  
129 first weight  $w_1 \in \mathbb{R}$ . We can therefore write SGD as

$$w^{t+1} = w^t - \eta \nabla \ell^{\omega_t}(w^t). \quad (7.3)$$

130 The non-negative scalar  $\eta \in \mathbb{R}_+$  is called the step-size or the learning rate. It  
131 governs the distance traveled along the negative gradient  $-\nabla \ell^{\omega_t}(w^t)$  at each  
132 iteration.

### 133 7.1.2 Deep Neural Networks

134 The Perceptron in (7.1) is a linear model: it computes a linear function of  
135 the weights  $w^\top x$  and uses this function to make the predictions  $f(x; w) =$   
136  $\text{sign}(w^\top x)$ . Linear models try to split the data (say we have binary labels  
137  $Y = \{-1, 1\}$ ) using a hyper-plane with  $w$  denoting the normal to this hyper-  
138 plane. This does not work for all situations of course, as the figure below  
139 shows, there is no hyper-plane that cleanly separates the two classes (i.e.,  
140 achieves zero mis-prediction error) but there *is* a nonlinear function that can  
do the job.

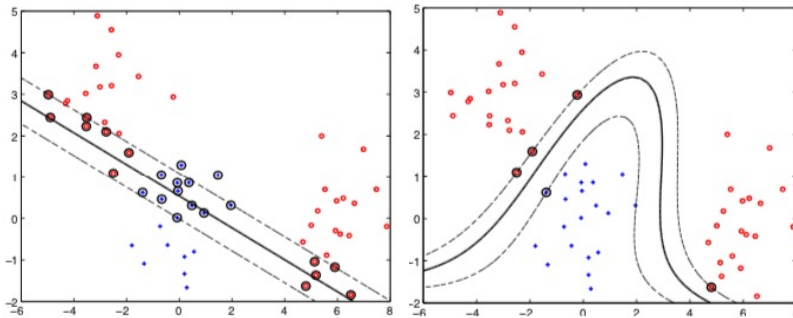


Figure 7.1

141 A deep neural network is one such nonlinear function. First consider a  
142 “two-layer” network  
143

$$f(x; v, S) = \text{sign}(v^\top \sigma(S^\top x))$$

144 where the matrix  $S \in \mathbb{R}^{d \times p}$  and a vector  $v \in \mathbb{R}^p$  are the parameters or  
145 “weights” of the classifier. The “nonlinearity”  $\sigma$  is usually set to be what is  
146 called a Rectified Linear Unit (ReLU)

$$\begin{aligned} \sigma(x) &:= \text{ReLU}(x) = |x|_+ \\ &= \max(0, x). \end{aligned} \quad (7.4)$$

147 Just like the case of a Perceptron, we can use an objective  $\frac{1}{n} \sum_{i=1}^n \ell^i(v, S)$

148 that depends on both  $v, S$  to fit this classifier on training data. A deep neural  
 149 network takes the idea of a two-layer network to the next step and has multiple  
 150 “layers”, each with a different weight matrix  $S_1, \dots, S_L$ . The classifier is  
 151 therefore given by

$$f(x; v, S_1, \dots, S_L) = \text{sign} \left( v^\top \sigma \left( S_L^\top \dots \sigma \left( S_2^\top \sigma \left( S_1^\top x \right) \dots \right) \right) \right). \quad (7.5)$$

152 We call each operation of the form  $\sigma \left( S_k^\top \dots \right)$ , as a *layer*. Consider the  
 153 second layer: it takes the features generated by the first layer, namely  $\sigma \left( S_1^\top x \right)$ ,  
 154 multiplies these features using its feature matrix  $S_2^\top$  and applies a nonlinear  
 155 function  $\sigma(\cdot)$  to this result element-wise before passing it on to the third layer.

A deep network creates new features by composing older features.

156 This composition is very powerful. Not only do we not have to pick a  
 157 particular feature vector, we can create very complex features by sequentially  
 158 combining simpler ones. For example Figure 7.2 shows the features (more  
 159 precisely, the kernel) learnt by a deep neural network. The first layer of features  
 160 are called Gabor-like, and incidentally they are similar to the features learned  
 161 by the human brain in the first part of the visual cortex (the one closest to the  
 162 eyes). These features are *combined* linearly along with a nonlinear operation  
 163 to give richer features (spirals, right angles) in the middle panel. The third  
 164 layer combines the lower features to get even more complex features, these  
 165 look like patterns (notice a soccer ball in the bottom left), a box on the bottom  
 166 right etc.

167 **Deep networks are universal function approximators** The multi-layer  
 168 neural network is a powerful class of classifiers: depending upon how many  
 169 layers we have and what is the dimensionality of the the weight matrices  
 170  $S_k$  at each layer, we can fit *any* training data. In fact, this statement, which  
 171 is called the *universal approximation property* holds even for a two-layer  
 172 neural network  $v^\top \sigma \left( S^\top x \right)$  if the number of columns in  $S$  is big enough. This  
 173 property is the central reason why deep networks are so widely applicable, we  
 174 can model complex machine learning problems if we choose a big enough  
 175 deep network.

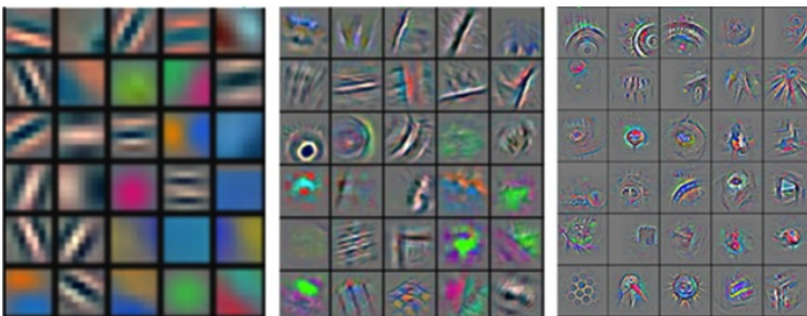


Figure 7.2

176 **Logits for multi-class classification.** The output

$$\hat{y} = v^\top \sigma(S_L^\top \dots \sigma(S_2^\top \sigma(S_1^\top x)) \dots)$$

177 is called the logits corresponding to the different classes. This name comes  
178 from logistic regression where logits are the log-probabilities of an input datum  
179 belonging to one of the two classes. A deep network provides an easy way to  
180 solve a multi-class classification problem, we simply set

$$v \in \mathbb{R}^{p \times C}$$

181 where  $C$  is the total number of classes in the data. Just like logistic regression  
182 predicts the logits of the two classes, we would like to *interpret* the vector  $\hat{y}$  as  
183 the log-probabilities of an input belonging to one of the classes.

184 **Weights** It is customary to not differentiate between the parameters of dif-  
185 ferent layers of a deep network and simply say *weights* when we want to refer  
186 to all parameters. The set

$$w := \{v, S_1, S_2, \dots, S_L\}$$

187 is the set of *weights*. This set is typically stored in PyTorch as a set of matrices,  
188 one for each layer. Using this new notation, we will write down a deep neural  
189 network classifier as simply

$$f(x, w) \tag{7.6}$$

190 and fitting the deep network to a dataset involves the optimization problem

$$w^* = \operatorname{argmin}_w \frac{1}{n} \sum_{i=1}^n \ell(y^i, \hat{y}^i). \tag{7.7}$$

191 We will also sometimes denote the loss of the  $i^{\text{th}}$  sample as

$$\ell^i(w) := \ell(y^i, \hat{y}^i).$$

192 **Backpropagation** The Backpropagation algorithm is a method to compute  
193 the gradient of the objective while fitting a deep network using SGD, i.e., it  
194 computes  $\nabla_w \ell^i(w)$ . For the purposes of this course, the details of how this is  
195 done are not essential, so we will skip them.

196 **PyTorch** We will use a library called PyTorch (<https://pytorch.org>) to code  
197 up deep neural networks for the reinforcement learning part of this course.  
198 You can find some excellent tutorials for it at  
199 <https://pytorch.org/tutorials/beginner/basics/intro.html>. For the purposes of  
200 this course, you do not need to know the intricacies of PyTorch, we will  
201 give you enough code to work with deep networks so that you can focus on  
202 implementing the reinforcement learning-specific parts.

🔗 What would the shape of  $w$  be if you were performing regression using a deep network?



## 7.2 Behavior Cloning

With that background, we are ready to tackle what is potentially the simplest problem in RL. We will almost exclusively deal with discrete-time systems for RL. Let us imagine that we are given access to  $n$  trajectories each of length  $T + 1$  time-steps from an expert demonstrator for our system. We write this as a training dataset

$$D = \{(x_t^i, u_t^i)_{t=0,1,\dots,T}\}_{i=1,\dots,n}$$

At each step, we record the state  $x_t^i \in \mathbb{R}^d$  and the control that the expert took at that state  $u_t^i$ . We would like to learn a deterministic feedback control for the robot that is parametrized by parameters  $\theta$

$$u_\theta(x) : X \mapsto U \subset \mathbb{R}^m.$$

using the training data. The idea is that if  $u_\theta(x^i(t)) \approx u^i(t)$  for all  $i$  and all times  $t$ , then we can simply run our learned controller  $u_\theta(x)$  on the robot instead of having the expert. A simple example is a baby deer learning to imitate how its mother in how to run.

**Parameterizing the controller** Our function  $u_\theta$  may represent many different families of controllers. For example,  $u_\theta(x) = \theta x$  where  $\theta \in \mathbb{R}^{d \times p}$  is a linear controller; this is much like the control for LQR except that we can fit  $\theta$  to the expert's data instead of solving the LQR problem to find the Kalman gain. We could also think of some other complicated function, e.g., a two-layer neural network,

$$u_\theta(x) = v \sigma(S^\top x)$$

where  $S \in \mathbb{R}^{d \times p}$  and  $v \in \mathbb{R}^{m \times p}$  and  $\sigma : \mathbb{R}^m \mapsto \mathbb{R}^m$  is some nonlinearity, say ReLU. As we did above, we will use

$$\theta := (v, S)$$

to denote all the weights of this two-layer neural network. Multi-layer neural networks are also another possible avenue. In general, we want the parameterization of the controller to be rich enough to fit some complex controller that the expert may have used on the system.

**How to fit the controller?** Given our chosen model for  $u_\theta(x)$ , say a two-layer neural network with weights  $\theta$ , fitting the controller involves finding the best value for the parameters  $\theta$  such that  $u_\theta(x_t^i) \approx u_t^i$  for data in our dataset. There are many ways to do this, e.g., we can solve the following optimization problem

$$\hat{\theta} = \underset{\theta}{\operatorname{argmin}} \ell(\theta) := \frac{1}{n} \sum_{i=1}^n \underbrace{\frac{1}{T+1} \sum_{t=0}^T \|u_t^i - u_\theta(x_t^i)\|_2^2}_{\ell^i(\theta)} \quad (7.8)$$

233 The difficulty of solving the above problem depends upon how difficult the  
 234 model  $u_\theta(x)$  is, for instance, if the model is linear  $\theta x$ , we can solve (7.8)  
 235 using ordinary least squares. If the model is a neural network, one would have  
 236 to use SGD to solve the optimization problem above. After fitting this model,  
 237 we have a new controller

$$u_{\hat{\theta}}(x) \in \mathbb{R}^m$$

238 that we can use *anywhere* in the domain  $X \subset \mathbb{R}^d$ , even at places where we had  
 239 no expert data. This is known as Behavior Cloning, i.e., cloning the controls  
 240 of the expert into a parametric model.

241 **Generalization performance of behavior cloning** Note that the data pro-  
 242 vided by the expert is not iid, of course the state  $x_{t+1}^i$  in the expert's trajectory  
 243 depends upon the previous state  $x_t^i$ . Standard supervised learning makes the  
 244 assumption that Nature gives training data that is independent and identically  
 245 distributed from the distribution  $P$ . While it is still reasonable to fit the re-  
 246 gression loss in (7.8) for such correlated data, one should remember that if  
 247 the expert trajectories do not go to all parts of the state-space, the learned  
 248 controller fitted on the training data may not work outside these parts. Of  
 249 course, if we behavior clone the controls taken by a generic driver, they are  
 250 unlikely to be competitive for racing, and vice-versa. It is very important to  
 251 realize that this does *not* mean that BC does not generalize. Generalization in  
 252 machine learning is a concept that suggests that the model should work well  
 253 on data *from the same distribution*. What does the the “distribution” of the  
 254 expert mean, in this case, it simply refers to the distribution of the states that  
 255 the expert's trajectories typically visit, e.g, a race driver typically drives at the  
 256 limits of tire friction and throttle, this is different from a usual city-driver who  
 257 would rather maximize the longevity of their tires and engine-life.

❗ Discuss generalization performance in behavior cloning.

## 258 7.2.1 Behavior cloning with a stochastic controller

259 In this case, we have always chosen feedback feedback controllers that are  
 260 deterministic, i.e., there is a single value of control  $u$  that is taken at the state  $x$ .  
 261 Going forward, we will also talk about stochastic controllers, i.e., controllers  
 262 which sample a control from a distribution. There can be a few reasons of  
 263 using such a controller. First, we will see in later lectures how this may help  
 264 in training a reinforcement learning algorithm; this is because in situations  
 265 where you do not know the system dynamics precisely, it helps to “hedge” the  
 266 feedback to take a few different control actions instead of simply the one that  
 267 the value function deems as the maximizing one. This is not very different  
 268 from having a few different stocks in your portfolio. Second, we benefit from  
 269 this hedging even at test-time when we run a stochastic feedback control, e.g.,  
 270 in situations where the limited training data may not want to always pick the  
 271 best control (because the best control was computed using an imprecise model  
 272 of the system dynamics and could be wrong), but rather hedge our bets by  
 273 choosing between a few different controls.

274 A stochastic feedback control is denoted by

$$u \sim u_\theta(\cdot | x) = P(\cdot | x)$$

275 notice that  $u_\theta(\cdot | x)$  is a probability distribution on the control space  $U$  that  
 276 depends on the state  $x$ , and in this case the parameters  $\theta$ . The control taken at a  
 277 state  $x$  is a sample drawn from this probability distribution. The deterministic  
 278 controller is a special case of this setup where

$$u_\theta(u | x) = \delta_{u_\theta(x)}(u) \equiv u_\theta(x)$$

279 is a Dirac-delta distribution at  $u_\theta(x)$ . If the control space  $U$  is discrete,  
 280 then  $u_\theta(\cdot | x)$  could be a categorical distribution. If the control space  $U$  is  
 281 continuous, then you may wish to think of the controls being sampled from a  
 282 Gaussian distribution with some mean  $\mu_\theta(x)$  and variance  $\sigma_\theta^2(x)$

$$\mathbb{R}^m \ni u \sim u_\theta(\cdot | x) = N(\mu_\theta(x), \Sigma_\theta(x)).$$

283 **Maximum likelihood estimation** Let's pick a particular stochastic con-  
 284 troller, say a Gaussian. How should we fit the parameters  $\theta$  for this? We would  
 285 like to find parameters  $\theta$  that make the expert's data in our dataset very likely.  
 286 The log-likelihood of each datum is

$$\log u_\theta(u_t^i | x_t^i)$$

287 and maximizing the log-likelihood of the entire dataset amounts to solving

$$\hat{\theta} = \underset{\theta}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^n \underbrace{\frac{1}{T+1} \sum_{t=0}^T -\log u_\theta(u_t^i | x_t^i)}_{\ell^i(\theta)}. \quad (7.9)$$

288 **Fitting BC with a Gaussian controller** Notice that if we use a Gaussian  
 289 distribution

$$u_\theta(\cdot | x) = N(\mu_\theta(x), I)$$

290 as our stochastic controller, the objective in (7.9) is the same as that in (7.8).

$$u_\theta(\cdot | x) = N(\mu_\theta(x), \sigma_\theta^2(x)I)$$

291 we have that

$$-\log u_\theta(u | x) = \frac{\|\mu_\theta(x) - u\|_2^2}{\sigma_\theta^2(x)} + 2cp \log \sigma_\theta(x).$$

292 where  $c$  is a constant.

## 293 7.2.2 KL-divergence form of Behavior Cloning

294 **Background on KL divergence** The Kullback-Leibler (KL) divergence is a  
 295 quantity to measure the distance between two probability distributions. There  
 296 are many similar distances, for example, given two probability distributions

297  $p(x)$  and  $q(x)$  supported on a discrete set  $X$ , the total variation distance  
298 between them is

$$\text{TV}(p, q) = \frac{1}{2} \sum_{x \in X} |p(x) - q(x)|.$$

299 Hellinger distance ([https://en.wikipedia.org/wiki/Hellinger\\_distance](https://en.wikipedia.org/wiki/Hellinger_distance)),  $f$ -divergences  
300 (<https://en.wikipedia.org/wiki/F-divergence>) and the Wasserstein metric  
301 ([https://en.wikipedia.org/wiki/Wasserstein\\_metric](https://en.wikipedia.org/wiki/Wasserstein_metric)) are a few other examples  
302 of ways to measure how different two probability distributions are from each  
303 other.

304 The Kullback-Leibler divergence (KL) between two distributions is given  
305 by

$$\text{KL}(p \parallel q) = \sum_{x \in X} p(x) \log \frac{p(x)}{q(x)}. \quad (7.10)$$

306 This is a distance and not a metric, i.e., it is always non-zero and zero if and  
307 only if the two distributions are equal, but the KL-divergence is not symmetric  
308 (like a metric has to be). Also, the above formula is well-defined only if for all  
309  $x$  where  $q(x) = 0$ , we also have  $p(x) = 0$ . Notice that it is not symmetric

$$\text{KL}(q \parallel p) = \sum_{x \in X} q(x) \log \frac{q(x)}{p(x)} \neq \text{KL}(p \parallel q).$$

310 The funny notation  $\text{KL}(p \parallel q)$  was invented by Shun-ichi Amari  
311 ([https://en.wikipedia.org/wiki/Shun%27ichi\\_Amari](https://en.wikipedia.org/wiki/Shun%27ichi_Amari)) to emphasize the fact that  
312 the KL-divergence is asymmetric. The KL-divergence is always positive: you  
313 can show this using an application of Jensen's inequality. For distributions  
314 with continuous support, we integrate over the entire space  $X$  and define KL  
315 divergence as

$$\text{KL}(p \parallel q) = \int_{\mathcal{X}} p(x) \log \frac{p(x)}{q(x)} dx.$$

316 **Behavior Cloning** Let us now imagine the expert is also a parametric  
317 stochastic feedback controller  $u_{\theta^*}(\cdot \mid x)$ . Our data is therefore drawn by  
318 running this controller for  $n$  trajectories,  $T$  time-steps on the system. This  
319 dataset now consists of samples from

$$p_{u_{\theta^*}}(x, u)$$

320 which is the joint distribution on the state-space  $X$  and the control-space  $U$ .  
321 We have denoted the parameters of the feedback controller which creates this  
322 distribution as the subscript  $u_{\theta^*}$ . Our behavior cloning controller creates a  
323 similar distribution  $p_{u_{\theta}}(x, u)$  and the general version of the objective in (7.9)  
324 is therefore

$$\hat{\theta} = \underset{\theta}{\operatorname{argmin}} \text{KL}(p_{u_{\theta^*}} \parallel p_{u_{\theta}}); \quad (7.11)$$

325 The objective in (7.9) corresponds to this for Gaussian stochastic controllers,  
326 but we can just as easily imagine some other distribution for the stochastic  
327 controller of the expert and the robot.

Written this way, BC can be understood as finding a controller  $\hat{\theta}$  whose distribution on the states and controls is close to the distribution of states and controls of the expert.

### 328 7.2.3 Some remarks on Behavior Cloning

329 **Worst-case performance** Performance of Behavior Cloning can be quite bad  
 330 in the worst case. The authors in “Efficient reductions for imitation learning”  
 331 (<https://www.cs.cmu.edu/~sross1/publications/Ross-AIStats11-NoRegret.pdf>)  
 332 show that if the learned controller  $u_{\hat{\theta}}$  differs from the control taken by the  
 333 expert controller  $u_{\theta^*}$  with a probability  $\epsilon$  at each time-step, over a horizon of  
 334 length  $T$  time-steps, it can be  $\mathcal{O}(T^2\epsilon)$  off from the cost-to-go of the expert *as*  
 335 *averaged over states that the learned controller visits*. This is because once the  
 336 robot makes a mistake and goes away from the expert’s part in the state-space,  
 337 future states of the robot and the expert can be very different.

❗ Draw a picture of the amplifying errors of running behavior cloning in real-time.

338 **Model-free nature of BC** Observe that our learned controller  $u_{\hat{\theta}}(\cdot | x)$  is a  
 339 feedback controller and works for entire state-space  $X$ . We did not need to  
 340 know the dynamics of the system to build this controller. The data from the  
 341 expert is conceptually the same as the model  $\dot{x} = f(x, u)$  of the dynamics,  
 342 and you can learn controllers from both. Do you however notice a catch?

## 343 7.3 DAgger: Dataset Aggregation

344 The expert’s dataset in Behavior Cloning determines the quality of the con-  
 345 troller learned. If we collected very few trajectories from the expert, they may  
 346 not cover all parts of the state-space and the behavior cloned controller has no  
 347 data to fit the model in those parts.

348 Let us design a simple algorithm, of the same spirit as iterative-LQR, to  
 349 mitigate this. We start with a candidate controller, say  $u_{\theta^{(0)}}(x)$ ; one may also  
 350 start with a stochastic controller  $u_{\theta^{(0)}}(\cdot | x)$  instead.

**DAgger:** Let the dataset  $D^{(0)}$  be the data collected from the expert.

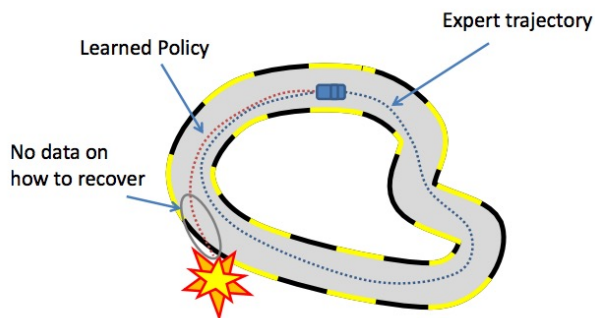
Initialize  $u_{\theta^{(0)}} = u_{\hat{\theta}}$  to be the BC controller learned using data  $D^{(0)}$ . At iteration  $k$

1. The robot queries the expert for a fraction  $p$  of the time-steps and uses its learned controller  $u_{\theta^{(k-1)}}$  for the other time-steps. If the expert corresponds to some controller  $u_{\theta^*}$ , then the robot controller at a state  $x$  is

$$u \sim p \delta_{u_{\theta^*}(x)} + (1 - p) \delta_{u_{\theta^{(k-1)}}(x)}.$$

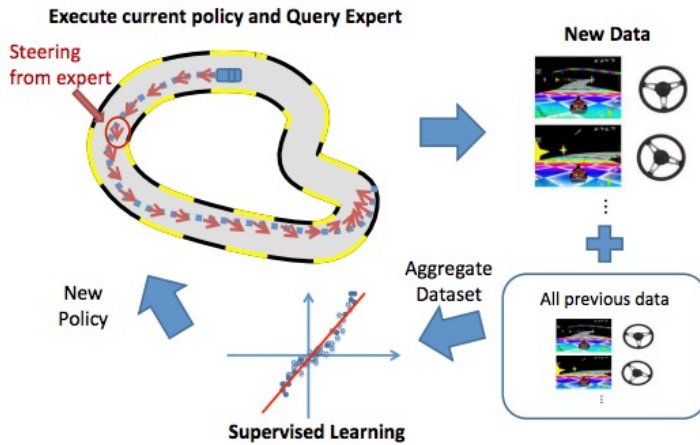
2. Use  $u(x)$  to collect a dataset  $D = \{(x_t^i, u_t^i)_{t=0, \dots, T}\}_{i=1, \dots, n}$  with  $n$  trajectories.
3. Set the new dataset to be  $D^{(k)} = D^{(k-1)} \cup D$
4. Fit a controller  $u_{\theta^{(k)}}$  using behavior cloning to the new dataset  $D^{(k)}$ .

351 The above algorithm iteratively updates the BC controller  $u_{\hat{\theta}}$  by drawing  
 352 new data from the expert. The robot first bootstraps off the expert's data, this  
 353 simply means that it uses the expert's data to fit its controller  $u_{\theta^{(0)}}(x)$ . As we  
 354 discussed above, this controller may veer off the expert's trajectory if the robot  
 355 starts at states that are different from the dataset, or even if it takes a slightly  
 356 different control than the expert midway through a trajectory.



357

358 To fix this, the robot collects more data at each iteration. It uses a combina-  
 359 tion of the expert and its controller to collect such data. This, allows *collecting*  
 360 *a dataset of expert's controls in states that the robot visits* and iteratively  
 361 expands the dataset  $D^{(k)}$ .



362

363 In the beginning we may wish to be close to the expert's data and use a large  
 364 value of  $p$ , as the fitted controller  $u_{\theta_{k+1}}$  becomes good, we can reduce the  
 365 value of  $p$  and rely less on the expert.

366 DAgger is an iterative algorithm which expands the controller to handle  
 367 larger and larger parts of the state-space. Therefore, the cost-to-go of the  
 368 controller learned via DAgger is  $\mathcal{O}(T)$  off from the cost-to-go of the expert *as*  
 369 *averaged over states that the learned controller visits.*

370 **DAgger with expert annotations at each step** DAgger is a conceptual  
 371 framework where the expert is queried repeatedly for new control actions.  
 372 This is obviously problematic because we need to expert on hand at each  
 373 iteration. We can also cook up a slightly version of DAgger where we start  
 374 with the BC controller  $u_{\theta^{(k)}} = u_{\hat{\theta}}$  and at each step, we run the controller on  
 375 the real system and ask the expert to relabel the data after that run. The dataset  
 376  $D^{(k)}$  collected by the algorithm expands at each iteration and although the  
 377 states  $x_t^i$  are those visited by our controller, their annotations are those given  
 378 by the expert. This is a much more natural way of implementing DAgger.

🔗 What criterion can we use to stop these iterations? We can stop when the incremental dataset collected  $D_k$  is not that different from the cumulative dataset  $D$ , we know that the new controllers are not that different. We can also stop when the parameters of our learned controller are  $\theta^{(k+1)} \approx \theta^{(k)}$ .