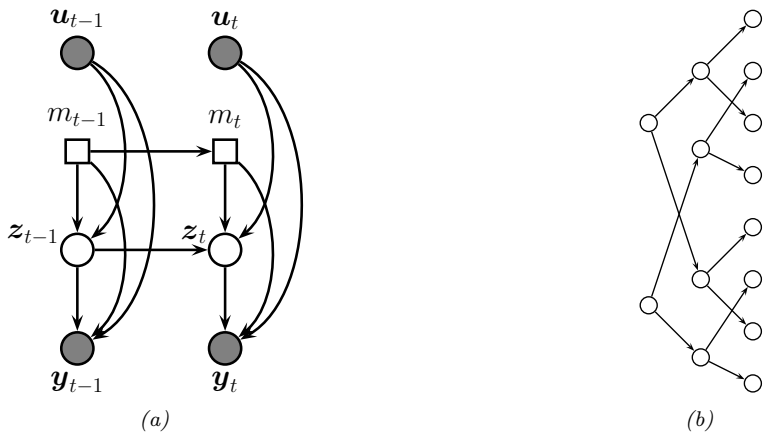


Figure 29.25: (a) A switching SSM. Squares represent discrete random variables, circles represent continuous random variables. (b) Illustration of how the number of modes in the belief state of a switching SSM grows exponentially over time. We assume there are two binary states.

See Figure 29.25a for the DPGM representation. It is straightforward to make a nonlinear version of this model.

29.9.2 Posterior inference

Unfortunately exact inference in such switching models is intractable, even in the linear Gaussian case. To see why, suppose for simplicity that the latent discrete switching variable m_t is binary, and that only the dynamics matrix $\mathbf{F}$ depend on m_t, not the observation matrix $\mathbf{H}$. Our initial belief state will be a mixture of 2 Gaussians, corresponding to $p(\boldsymbol{z}_1|\boldsymbol{y}_1, m_1 = 1)$ and $p(\boldsymbol{z}_1|\boldsymbol{y}_1, m_1 = 2)$. The one-step-ahead predictive density will be a mixture of 4 Gaussians $p(\boldsymbol{z}_2|\boldsymbol{y}_1, m_1 = 1, m_2 = 1)$, $p(\boldsymbol{z}_2|\boldsymbol{y}_1, m_1 = 1, m_2 = 2)$, $p(\boldsymbol{z}_2|\boldsymbol{y}_1, m_1 = 2, m_2 = 1)$, and $p(\boldsymbol{z}_2|\boldsymbol{y}_1, m_1 = 2, m_2 = 2)$, obtained by passing each of the prior modes through the 2 possible transition models. The belief state at step 2 will also be a mixture of 4 Gaussians, obtained by updating each of the above distributions with $\boldsymbol{y}_2$. At step 3, the belief state will be a mixture of 8 Gaussians. And so on. So we see there is an exponential explosion in the number of modes. Each sequence of discrete values corresponds to a different hypothesis (sometimes called a **track**), which can be represented as a tree, as shown in Figure 29.25b.

Various methods for approximate online inference have been proposed for this model, such as the following:

- Prune off low probability trajectories in the discrete tree. This is widely used in multiple hypothesis tracking methods (see Section 29.9.3).
- Use particle filtering (Section 13.2) where we sample discrete trajectories, and apply the Kalman filter to the continuous variables. See Section 13.4.1 for details.
- Use ADF (Section 8.10), where we approximate the exponentially large mixture of Gaussians with a smaller mixture of Gaussians. See Section 8.10.2 for details.

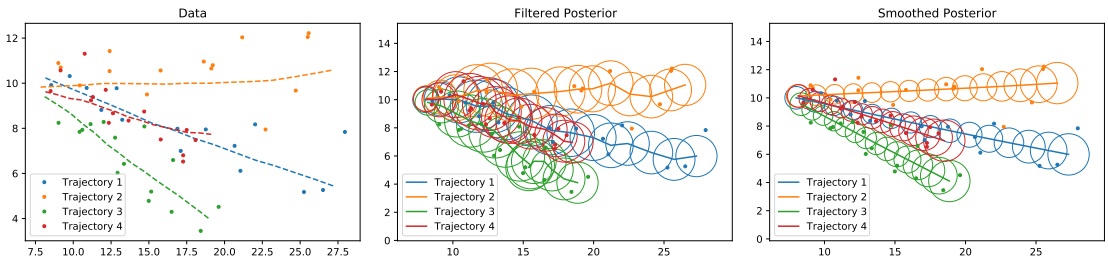


Figure 29.26: Illustration of Kalman filtering and smoothing for tracking multiple moving objects. Generated by kf_parallel.ipynb.

- Use structured variational inference, where we approximate the posterior as a product of chain-structured distributions, one over the discrete variables and one over the continuous variables, with variational "coupling" terms in between (see e.g., [GH98; PJD21; Wan+22].

29.9.3 Application: Multi-target tracking

The problem of **multi-target tracking** frequently arises in engineering applications (especially in aerospace and defence). This is a very large topic (see e.g. [BSF88; BSL93; Vo+15] for details), but in this section, we show how switching LDS models (or their nonlinear extensions) can be used to tackle the problem.

29.9.3.1 Warmup

In the simplest setting, we know there are N objects we want to track, and each one generates its own uniquely identified observation. If we assume the objects are independent, we can apply Kalman filtering and smoothing in parallel, as shown in Figure 29.26. (In this example, each object follows a linear dynamical model with different initial random velocities, as in Section 29.7.1.)

29.9.3.2 Data association

More generally, at each step we may observe M measurements e.g., "blips" on a radar screen. We can have $M < N$ due to occlusion or missed detections. We can have $M > N$ due to clutter or false alarms. Or we can have $M = N$. In any case, we need to figure out the **correspondence** between the M detections $\boldsymbol{x}_t^m$ and the N objects $\boldsymbol{z}_t^i$. This is called the problem of **data association**, and it arises in many application domains.

We can model this problem by augmenting the state space with discrete variables m_t that represent the association matrix between the observations, $\boldsymbol{y}_{t,1:M}$, and the sources, $\boldsymbol{z}_{t,1:N}$. See Figure 29.27 for an illustration, where we have $N = 2$ objects, but a variable number M_t of observations per time step.

As we mentioned in Section 29.9.2, inference in such hybrid (discrete-continouus) models is intractable, due to the exponential number of posterior modes. In the sections below, we briefly

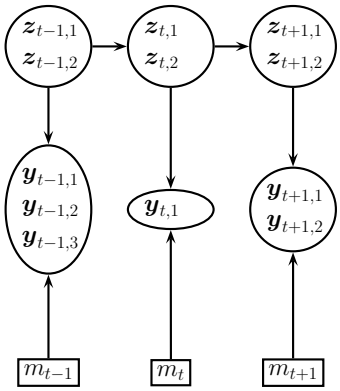


Figure 29.27: A model for tracking two objects in the presence of data-association ambiguity. We observe 3, 1 and 2 detections at time steps $t-1$, t and $t+1$. The m_t hidden variable encodes the association between the observations and the hidden causes.

mention a few approximate inference methods.

29.9.3.3 Nearest neighbor approximation using Hungarian algorithm

A common way to perform approximate inference in this model is to compute an $N \times M$ weight matrix, where W_{im} measures the "compatibility" between object i and measurement m, typically based on how close m is to where the model thinks i is (the so-called **nearest neighbor data association** heuristic).

We can make this into a square matrix by adding dummy background objects, which can explain all the false alarms, and adding dummy observations, which can explain all the missed detections. We can then compute the maximal weight bipartite matching using the **Hungarian algorithm**, which takes $O(\max(N, M)^3)$ time (see e.g., [BDM09]).

Conditional on knowing the assignments of measurements to tracks, we can perform the usual Bayesian state update procedure (e.g., based on Kalman filtering). Note that objects that are assigned to dummy observations do not perform a measurement update, so their state estimate is just based on forwards prediction from the dynamics model.

29.9.3.4 Other approximate inference schemes

The Hungarian algorithm can be slow (since it is cubic in the number of measurements), and can give poor results since it relies on hard assignment. Better performance can be obtained by using loopy belief propagation (Section 9.3). The basic idea is to approximately marginalize out the unknown assignment variables, rather than perform a MAP estimate. This is known as the **SPADA** method (sum-product algorithm for data association) [WL14b; Mey+18].

The cost of each iteration of the iterative procedure is $O(NM)$. Furthermore, [WL14b] proved this will always converge in a finite number of steps, and [Von13] showed that the corresponding solution will in fact be the global optimum. The SPADA method is more efficient, and more accurate, than earlier heuristic methods, such as **JPDA** (joint probabilistic data association) [BSWT11; Vo+15].

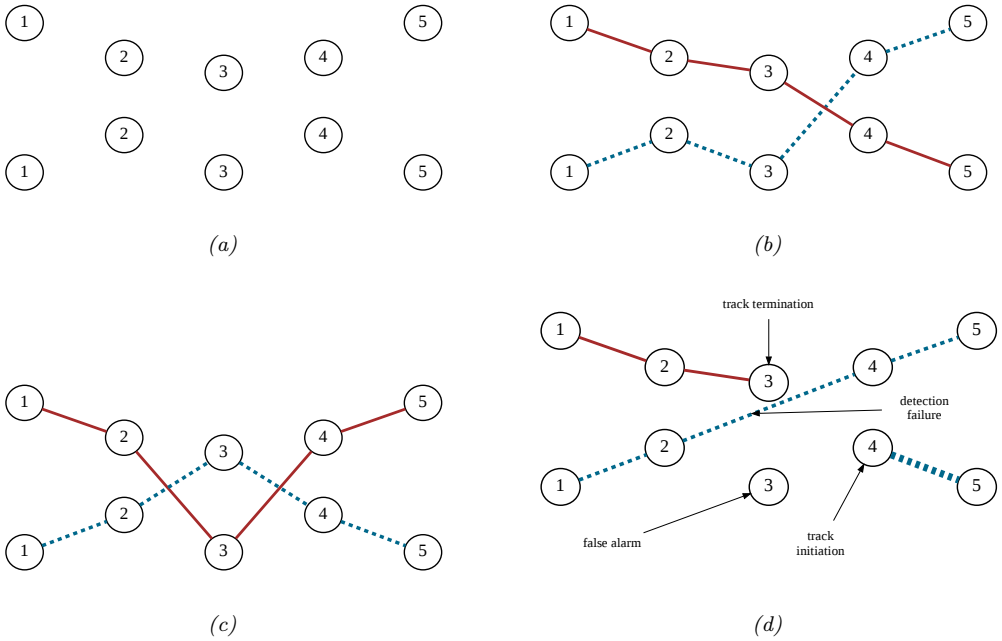


Figure 29.28: Illustration of multi-target tacking in 2d over 5 time steps. (a) We observe 2 measurements per time step. (b-c) Possible hypotheses about the underlying object tracks. (d) A more complex hypothesis in which the red track stops at step 3, the dashed red track starts at step 4 the dotted blue track has a detection failure at step 3, and one of the measurements at step 3 is a false alarm. Adapted from Figure 15.8 of [RN19].

It is also possible to use sequential Monte Carlo methods to solve data association and tracking. See Section 13.2 for a general discussion of SMC, and [RAG04; Wan+17b] for a review of specific techniques for this model family.

29.9.3.5 Handling an unknown number of targets

In general, we do not know the true number of targets N, so we have to deal with variable-sized state space. This is an example of an **open world** model (see Section 4.6.5), which differs from the standard **closed world assumption** where we know how many objects of interest there are.

For example, suppose at each time step we get two "blips" on our radar screen, representing the presence of an object at a given location. These measurements are not tagged with the source of the object that generated them, so the data looks like Figure 29.28(a). In Figure 29.28(b-c) we show two different hypotheses about the underlying object trajectories that could have generated this data. However, how can we know there are two objects? Maybe there are more, but some are just not detected. Maybe there are fewer, and some observations are false alarms due to background clutter. One such more complex hypothesis is shown in Figure 29.28(d). Figuring out what is going on in problems such as this is known as **multiple hypothesis tracking**.

A common approximate solution to this is to create new objects whenever an observation cannot be

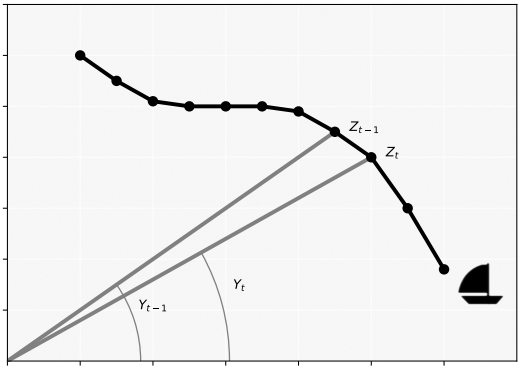


Figure 29.29: Illustration of a bearings-only tracking problem. Adapted from Figure 2.1 of [CP20b].

"explained" (i.e., generated with high likelihood) by any existing objects, and to prune out old objects that have not been detected in a while (in order to keep the computational cost bounded). Sets whose size and content are both random are called **random finite sets**. An elegant mathematical framework for dealing with such objects is described in [Mah07; Mah13; Vo+15].

29.10 Nonlinear SSMs

In this section, we consider SSMs with nonlinear transition and/or observation functions, and additive Gaussian noise. That is, we assume the model has the following form

$$\boldsymbol{z}_t = \boldsymbol{f}(\boldsymbol{z}_{t-1}, \boldsymbol{u}_t) + \boldsymbol{q}_t \tag{29.120}$$
$$\boldsymbol{q}_t \sim \mathcal{N}(\mathbf{0}, \mathbf{Q}_t) \tag{29.121}$$
$$\boldsymbol{y}_t = \boldsymbol{h}(\boldsymbol{z}_t, \boldsymbol{u}_t) + \boldsymbol{r}_t \tag{29.122}$$
$$\boldsymbol{r}_t \sim \mathcal{N}(\mathbf{0}, \mathbf{R}_t) \tag{29.123}$$

This is called a **nonlinear dynamical system** (**NLDS**), or **nonlinear Gaussian SSM** (**NLG-SSM**).

29.10.1 Example: object tracking and state estimation

In Section 8.5.5.1 we give an example of a 2d tracking problem where the motion model is nonlinear, but the observation model is linear.

Here we consider an example where the motion model is linear, but the observation model is nonlinear. In particular, suppose we use the same 2d linear dynamics as in Section 29.7.1, where the state space contains the position and velocity of the object, $\boldsymbol{z}_t = \begin{pmatrix} u_t & v_t & \dot{u}_t & \dot{v}_t \end{pmatrix}$. (We use u and v for the two coordinates, to avoid confusion with the state and observation variables.) Instead of directly observing the location, suppose we have a **bearings only tracking problem**, in which we

just observe the angle to the target:

$$y_t = \tan^{-1}\left(\frac{v_t - s_y}{u_t - s_x}\right) + r_t \tag{29.124}$$

where (s_x, s_y) is the position of the measurement sensor. See Figure 29.29 for an illustration. This nonlinear observation model prevents the use of the Kalman filter, but we can still apply approximate inference methods, as we discuss below.

29.10.2 Posterior inference

Inferring the states of an NLDS model is in general computationally difficult. Fortunately, there are a variety of approximate inference schemes that can be used, such as the extended Kalman filter (Section 8.5.2), the unscented Kalman filter (Section 8.7.2), etc.

29.11 Non-Gaussian SSMs

In this section, we consider SSMs in which the transition and observation noise is non-Gaussian. The transition and observation functions can be linear or nonlinear. We can represent this as a probabilistic model as follows:

$$p(\boldsymbol{z}_t|\boldsymbol{z}_{t-1}, \boldsymbol{u}_t) = p(\boldsymbol{z}_t|\boldsymbol{f}(\boldsymbol{z}_{t-1}, \boldsymbol{u}_t)) \tag{29.125}$$

$$p(\boldsymbol{y}_t|\boldsymbol{z}_t, \boldsymbol{u}_t) = p(\boldsymbol{y}_t|\boldsymbol{h}(\boldsymbol{z}_t, \boldsymbol{u}_t)) \tag{29.126}$$

This is called a **non-Gaussian SSM** (**NSSM**).

29.11.1 Example: Spike train modeling

In this section we discuss consider an SSM with linear-Gaussian latent dynamics and a Poisson likelihood. Such models are widely used in neuroscience for modeling **neural spike trains**. (see e.g., [Pan+10; Mac+11]). This is an example of an **exponential family state-space model** (see e.g., [Vid99; Hel17]).

We consider a simple example where the model has 2 continuous latent variables, and we set the dynamics matrix $\mathbf{A}$ to a random rotation matrix. The observation model has the form

$$p(\boldsymbol{y}_t|\boldsymbol{z}_t) = \prod_{d=1}^{D} \text{Poi}(y_{td}|\exp(\boldsymbol{w}_d^\mathsf{T}\boldsymbol{z}_t)) \tag{29.127}$$

where $\boldsymbol{w}_d$ is a random vector, and we use $D = 5$ observations per time step. Some samples from this model are shown in Figure 29.30.

We can fit this model by using EM, where in the E step we approximate $p(\boldsymbol{y}_t|\boldsymbol{z}_t)$ using a Laplace approximation, after which we can use the Kalman smoother to compute $p(\boldsymbol{z}_{1:T}|\boldsymbol{y}_{1:T})$. In the M step, we optimize the expected complete data log likelihood, similar to Section 29.8.1. We show the result in Figure 29.31, where we compare the parameters $\mathbf{A}$ and the posterior trajectory $\mathbb{E}[\boldsymbol{z}_t|\boldsymbol{y}_{1:T}]$ using the true model and the estimated model. We see good agreement.

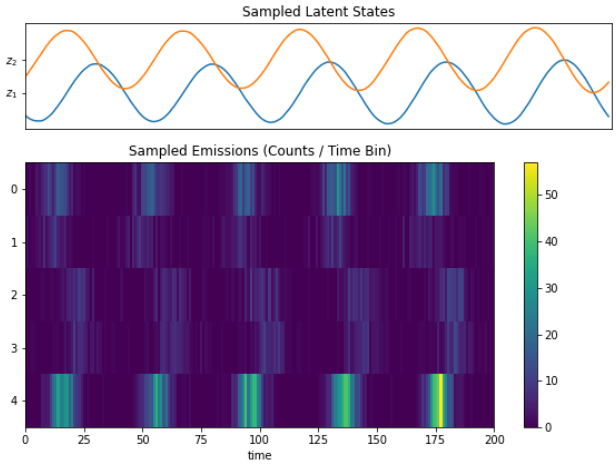


Figure 29.30: Samples from a 2d LDS with 5 Poisson likelihood terms. Generated by poisson_lds_example.ipynb.

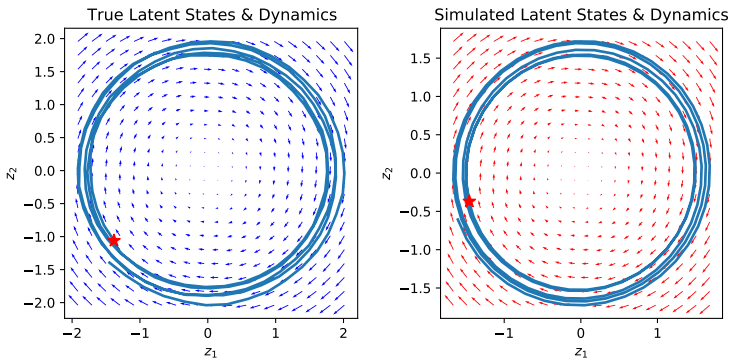


Figure 29.31: Latent state trajectory (blue lines) and dynamics matrix $\mathbf{A}$ *(arrows) for (left) true model and (right) estimated model. The star marks the start of the trajectory. Generated by poisson_lds_example.ipynb.*

29.11.2 Example: Stochastic volatility models

In finance, it is common to model the the **log-returns**, $y_t = \log(p_t/p_{t-1})$, where p_t is the price of some asset at time t. A common model for this problem, known as a **stochastic volatility model**,

(see e.g., [KSC98]), has the following form:

$$y_t = \boldsymbol{u}_t^\mathsf{T}\boldsymbol{\beta} + \exp(z_t/2)r_t \tag{29.128}$$
$$z_t = \mu + \rho(z_{t-1} - \mu) + \sigma q_t \tag{29.129}$$
$$r_t \sim \mathcal{N}(0,1) \tag{29.130}$$
$$q_t \sim \mathcal{N}(0,1) \tag{29.131}$$

We see that the dynamical model is a first-order autoregressive process. We typically require that $|\rho| < 1$, to ensure the system is stationary. The observation model is Gaussian, but can be replaced by a heavy-tailed distribution such as a Student.

We can capture longer range temporal correlation by using a higher order auto-regressive process. To do this, we just expand the state-space to contain the past K values. For example, if $K = 2$ we have

$$\begin{pmatrix} z_t - \mu \\ z_{t-1} - \mu \end{pmatrix} = \begin{pmatrix} \rho_1 & \rho_2 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} z_{t-1} - \mu \\ z_{t-2} - \mu \end{pmatrix} + \begin{pmatrix} q_t \\ 0 \end{pmatrix} \tag{29.132}$$

where $q_t \sim \mathcal{N}(0, \sigma_z^2)$. Thus we have

$$z_t = \mu + \rho_1(z_{t-1} - \mu) + \rho_2(z_{t-2} - \mu) + q_t \tag{29.133}$$

29.11.3 Posterior inference

Inferring the states of an NGSSM model is in general computationally difficult. Fortunately, there are a variety of approximate inference schemes that can be used, which we discuss in Chapter 8 and Chapter 13.

29.12 Structural time series models

In this section, we discuss **time series forecasting**, which is the problem of computing the predictive distribution over future observations given the data up until the present, i.e., computing $p(\boldsymbol{y}_{t+h}|\boldsymbol{y}_{1:t})$. (The model may optionally be conditioned on known future inputs, to get $p(\boldsymbol{y}_{t+h}|\boldsymbol{y}_{1:t}, \boldsymbol{x}_{1:t+h})$.) There are many approaches to this problem (see e.g., [HA21]), but in this section, we focus on **structural time series** (**STS**) models based on SSMs.

Many classical time series methods such as the **ARMA** (autoregressive moving average) method, can be represented as a a (linear-Gaussian) STS model (see e.g., [Har90; Sim06; CK07; Fra08; DK12; Sar13; PFW21; Tri21]). However, the STS approach has much more flexibility. For example, we can create nonlinear, non-Gaussian and even hierarchical extensions, as we discuss below.

29.12.1 Basics

The basic idea of an STS model is to represent the observed scalar time series as a sum of C individual **components**:

$$f(t) = f_1(t) + f_2(t) + \cdots + f_C(t) + \epsilon_t \tag{29.134}$$

where $\epsilon_t \sim \mathcal{N}(0, \sigma^2)$. For example, we might have a seasonal component that causes the observed values to oscillate up and down, and a growth component, that causes the observed values to get larger over time. Each latent process $f_c(t)$ is modeled by a linear Gaussian state-space model, which (in this context) is also called a **dynamic linear model** (**DLM**). Since these are linear, we can combine them altogether into a single LG-SSM. In particular, in the case of scalar observations, the model has the form

$$p(\boldsymbol{z}_t|\boldsymbol{z}_{t-1}, \boldsymbol{\theta}) = \mathcal{N}(\boldsymbol{z}_t|\mathbf{F}\boldsymbol{z}_{t-1}, \mathbf{Q}) \tag{29.135}$$

$$p(y_t|\boldsymbol{z}_t, \boldsymbol{\theta}) = \mathcal{N}(y_t|\mathbf{H}\boldsymbol{z}_t + \boldsymbol{\beta}^\mathsf{T}\boldsymbol{u}_t, \sigma_y^2) \tag{29.136}$$

where $\mathbf{F}$ and $\mathbf{Q}$ are block structured matrices, with one block per component. The vector $\mathbf{H}$ then adds up all the relevant pieces from each component to generate the overall mean. Note that the matrices $\mathbf{F}$ and $\mathbf{H}$ are fixed sparse matrices which can be derived from the form of the corresponding components of the model, as we discuss below. So the only model parameters are the variance terms, $\mathbf{Q}$ and σ_y^2, and the optional regression coefficients $\boldsymbol{\beta}$. [4]

Once we have specified the form of the model, we need to learn the model parameters, $\boldsymbol{\theta} = (\boldsymbol{\beta}, \sigma_y^2, \mathbf{Q})$. Common approaches are based on maximum likelihood estimation (see Section 29.8), and Bayesian inference (see Section 29.8.4). The latter approach is known as **Bayesian structural time series** or **BSTS** modeling and tends to give more reliable results (since small errors in estimating the noise variances can have a large impact on the forecast quality). Once the parameters have been estimated on an historical dataset, we can perform inference on a new time series to compute $p(\boldsymbol{z}_t|\boldsymbol{y}_{1:t}, \boldsymbol{\theta})$ using the Kalman filter (Section 8.3.2). Given the current posterior, we can then "roll forward" in time to forecast future observations h steps ahead by computing $p(\boldsymbol{y}_{t+h}|\boldsymbol{y}_{1:t}, \boldsymbol{\theta})$, as in Section 8.3.2.3.

29.12.2 Details

In this section, we discuss STS models in more detail.

29.12.2.1 Local level model

The simplest latent dynamical process is known as the **local level model**. It assumes the observations $y_t \in \mathbb{R}$ are generated by a Gaussian with (latent) mean μ_t, which evolves over time according to a random walk:

$$y_t = \mu_t + \epsilon_{y,t} \quad \epsilon_{y,t} \sim \mathcal{N}(0, \sigma_y^2) \tag{29.137}$$

$$\mu_t = \mu_{t-1} + \epsilon_{\mu,t}, \quad \epsilon_{\mu,t} \sim \mathcal{N}(0, \sigma_\mu^2) \tag{29.138}$$

We also assume $\mu_1 \sim \mathcal{N}(0, \sigma_\mu^2)$. Hence the latent mean at any future step has distribution $\mu_t \sim \mathcal{N}(0, t\sigma_\mu^2)$, so the variance grows with time. We can also use an autoregressive (AR) process, $\mu_t = \rho\mu_{t-1} + \epsilon_{\mu,t}$, where $|\rho| < 1$. This has the stationary distribution $\mu_\infty \sim \mathcal{N}(0, \frac{\sigma_\mu^2}{1-\rho^2})$, so the uncertainty grows to a finite asymptote instead of unboundedly.

4. In the statistics community, the notation is often slightly different, and often follow [DK12]. In particular, the dynamics are written as $\boldsymbol{\alpha}_t = \mathbf{T}_t\boldsymbol{\alpha}_{t-1} + \boldsymbol{c}_t\mathbf{R}_t\boldsymbol{\eta}_t$ and the observations are written as $y_t = \mathbf{Z}_t\boldsymbol{\alpha}_t + \boldsymbol{\beta}^\mathsf{T}\boldsymbol{x}_t + H_t\epsilon_t$, where $\boldsymbol{\eta}_t \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ and $\epsilon_t \sim \mathcal{N}(0, 1)$.

29.12.2.2 Local linear model

Many time series exhibit linear trends upwards or downwards, at least locally. We can model this by letting the level μ_t change by an amount δ_{t-1} (representing the slope of the line over an interval $\Delta t = 1$) at each step

$$\mu_t = \mu_{t-1} + \delta_{t-1} + \epsilon_{\mu,t} \tag{29.139}$$

The slope itself also follows a random walk,

$$\delta_t = \delta_{t-1} + \epsilon_{\delta,t} \tag{29.140}$$

and $\epsilon_{\delta,t} \sim \mathcal{N}(0, \sigma_\delta^2)$. This is called a **local linear trend** model.

We can combine these two processes by defining the following dynamics model:

$$\underbrace{\begin{pmatrix}\mu_t \\ \delta_t\end{pmatrix}}_{\boldsymbol{z}_t} = \underbrace{\begin{pmatrix}1 & 1 \\ 0 & 1\end{pmatrix}}_{\mathbf{F}} \underbrace{\begin{pmatrix}\mu_{t-1} \\ \delta_{t-1}\end{pmatrix}}_{\boldsymbol{z}_{t-1}} + \underbrace{\begin{pmatrix}\epsilon_{\mu,t} \\ \epsilon_{\delta,t}\end{pmatrix}}_{\boldsymbol{\epsilon}_t} \tag{29.141}$$

For the emission model we have

$$y_t = \underbrace{\begin{pmatrix}1 & 0\end{pmatrix}}_{\mathbf{H}} \underbrace{\begin{pmatrix}\mu_t \\ \delta_t\end{pmatrix}}_{\boldsymbol{z}_t} + \epsilon_{y,t} \tag{29.142}$$

We can also use an autoregressive model for the slope, i.e.,

$$\delta_t = D + \rho(\delta_{t-1} - D) + \epsilon_{\delta,t} \tag{29.143}$$

where D is the long run slope to which δ will revert. This is called a "**semilocal linear trend**" model, and is useful for longer term forecasts.

29.12.2.3 Adding covariates

We can easily include covariates $\boldsymbol{u}_t$ into the model, to increase prediction accuracy. If we use a linear model, we have

$$y_t = \mu_t + \boldsymbol{\beta}^\mathsf{T} \boldsymbol{u}_t + \epsilon_{y,t} \tag{29.144}$$

See Figure 29.32a for an illustration of the local level model with covariates. Note that, when forecasting into the future, we will need some way to predict the input values of future $\boldsymbol{u}_{t+h}$; a simple approach is just to assume future inputs are the same as the present, $\boldsymbol{u}_{t+h} = \boldsymbol{u}_t$.

29.12.2.4 Modelling seasonality

Many time series also exhibit **seasonality**, i.e., they fluctuate periodically. This can be modeled by creating a latent process consisting of a series offset terms, s_t. To model cyclicity, we ensure that these sum to zero (on average) over a complete cycle of S steps:

$$s_t = -\sum_{k=1}^{S-1} s_{t-k} + \epsilon_{s,t}, \;\; \epsilon_{s,t} \sim \mathcal{N}(0, \sigma_s^2) \tag{29.145}$$

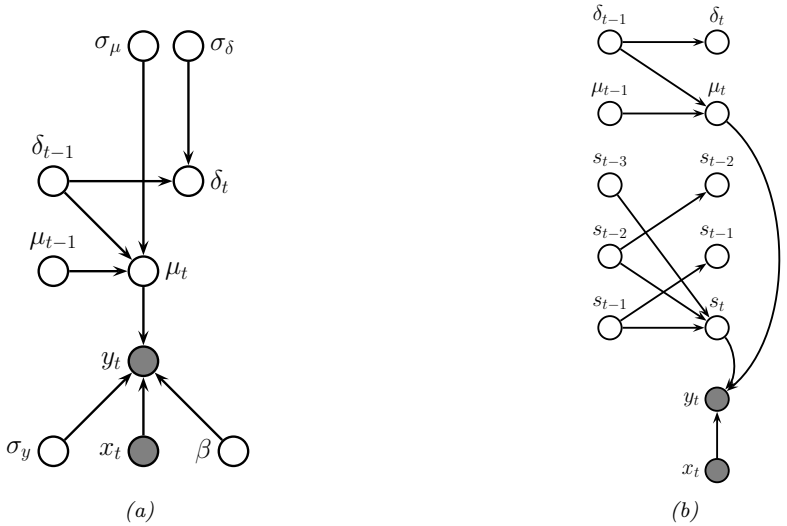


Figure 29.32: (a) A BSTS model with local linear trend and linear regression on inputs. The observed output is y_t. The latent state vector is defined by $\boldsymbol{z}_t = (\mu_t, \delta_t)$. The (static) parameters are $\boldsymbol{\theta} = (\sigma_y, \sigma_\mu, \sigma_\delta, \boldsymbol{\beta})$. The covariates are $\boldsymbol{u}_t$. (b) Adding a latent seasonal process (with $S = 4$ seasons). Parameter nodes are omitted for clarity.

For example, for $S = 4$, we have $s_t = -(s_{t-1} + s_{t-2} + s_{t-3}) + \epsilon_{s,t}$. We can convert this to a first-order model by stacking the last $S - 1$ seasons into the state vector, as shown in Figure 29.32b.

29.12.2.5 Adding it all up

We can combine the various latent processes (local level, linear trend, and seasonal cycles) into a single linear-Gaussian SSM, because the sparse graph structure can be encoded by sparse matrices. More precisely, the transition model becomes

$$\underbrace{\begin{pmatrix} s_t \\ s_{t-1} \\ s_{t-2} \\ \mu_t \\ \delta_t \end{pmatrix}}_{\boldsymbol{z}_t} = \underbrace{\begin{pmatrix} -1 & -1 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}}_{\mathbf{F}} \underbrace{\begin{pmatrix} s_{t-1} \\ s_{t-2} \\ s_{t-3} \\ \mu_{t-1} \\ \delta_{t-1} \end{pmatrix}}_{\boldsymbol{z}_{t-1}} + \mathcal{N}\left(\mathbf{0}, \text{diag}([\sigma_s^2, 0, 0, \sigma_\mu^2, \sigma_\delta^2])\right) \tag{29.146}$$

Having defined the model, we can use the Kalman filter to compute $p(\boldsymbol{z}_t|\boldsymbol{y}_{1:t})$, and then make predictions forwards in time by rolling forward in latent space, and then predicting the outputs:

$$p(\boldsymbol{y}_{t+1}|\boldsymbol{y}_{1:t}) = \int p(\boldsymbol{y}_{t+1}|\boldsymbol{z}_{t+1}) p(\boldsymbol{z}_{t+1}|\boldsymbol{z}_t) p(\boldsymbol{z}_t|\boldsymbol{y}_{1:t}) d\boldsymbol{z}_t \tag{29.147}$$

This can be computed in closed form, as explained in Section 8.3.2.

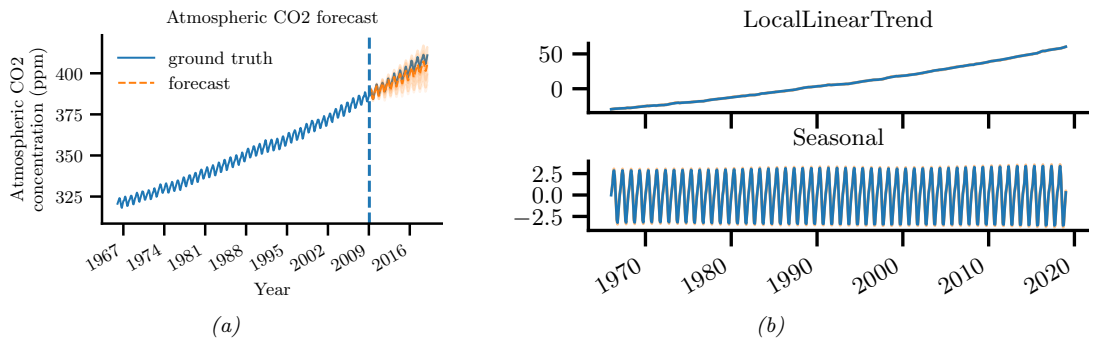


Figure 29.33: (a) CO_2 levels from Mauna Loa. In orange plot we show predictions for the most recent 10 years. (b) Underlying components for the STS mode which was fit to Figure 29.33a. Generated by sts.ipynb.

29.12.3 Example: forecasting CO_2 levels from Mauna Loa

In this section, we fit an STS model to the monthly atmospheric CO_2 readings from the Mauna Loa observatory in Hawaii.[5] The data is from January 1966 to February 2019. We combine a local linear trend model with a seasonal model, where we assume the periodicity is $S = 12$, since the data is monthly (see Figure 29.33a). We fit the model to all the data except for the last 10 years using variational Bayes. The resulting posterior mean and standard deviations for the parameters are $\sigma_y = 0.169 \pm 0.008$, $\sigma_\mu = 0.159 \pm 0.017$, $\sigma_\delta = 0.009 \pm 0.003$, $\sigma_s = 0.038 \pm 0.008$. We can sample 10 parameter vectors from the posterior and then plug them it to create a distribution over forecasts. The results are shown in orange in Figure 29.33a. Finally, in Figure 29.33b, we plot the posterior mean values of the two latent components (linear trend and current seasonal value) over time. We see how the model has successfully decomposed the observed signal into a sum of two simpler signals. (See also Section 18.8.1 where we model this data using a GP.)

29.12.4 Example: forecasting (real-valued) electricity usage

In this section, we consider a more complex example: forecasting electricity demand in Victoria, Australia, as a function of the previous value and the external temperature. (Remember that January is summer in Australia!) The hourly data from the first six weeks of 2014 is shown in Figure 29.34a.[6]

We fit an STS to this using 4 components: a seasonal hourly effect (period 24), a seasonal daily effect (period 7, with 24 steps per season), a linear regression on the temperature, and an autoregressive term on the observations themselves. We fit the model with variational inference. (This takes about a minute on a GPU.) We then draw 10 posterior samples and show the posterior predictive forecasts in Section 29.12.4. We see that the results are reasonable, but there is also considerable uncertainty.

We plot the individual components in Figure 29.35. Note that they have different vertical scales, reflecting their relative importance. We see that the regression on the external temperature is the most important effect. However, the hour of day effect is also quite significant, even after accounting for external temperature. The autoregressive effect is the most uncertain one, since it is responsible

5. For details, see https://blog.tensorflow.org/2019/03/structural-time-series-modeling-in.html.
6. The data is from https://github.com/robjhyndman/fpp2-package.

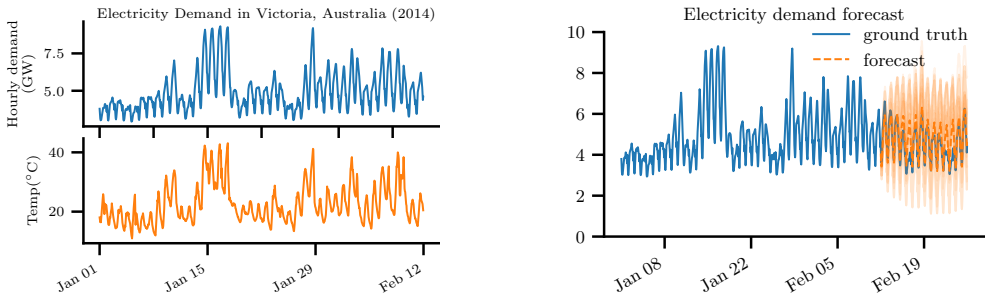


Figure 29.34: (a) Hourly temperature and electricity demand in Victoria, Australia in 2014. (b) Electricity forecasts using an STS. Generated by sts.ipynb.

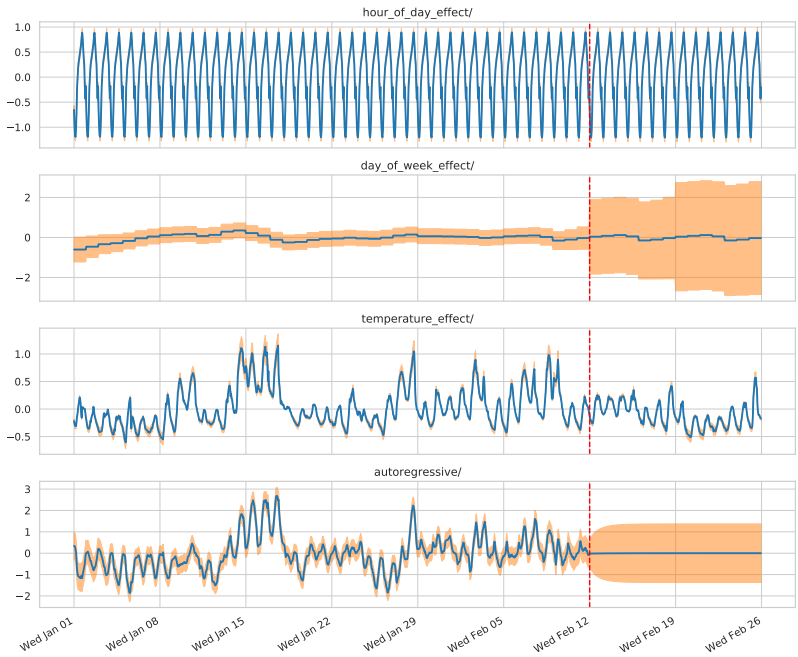


Figure 29.35: Components of the electricity forecasts. Generated by sts.ipynb.

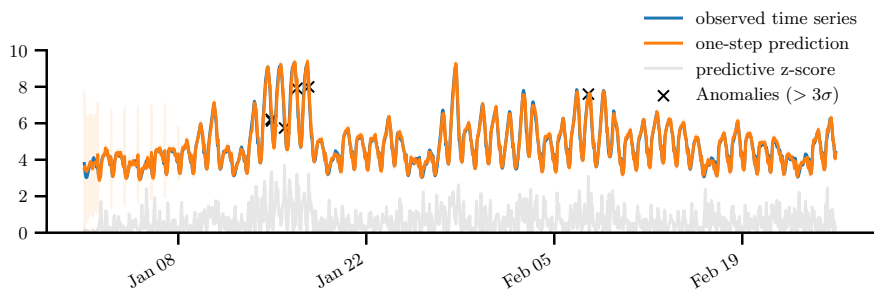


Figure 29.36: Anomaly detection in a time series. We plot the observed electricity data in blue and the predictions in orange. In gray, we plot the z-score at time t , given by $(y_t - \mu_t)/\sigma_t$, where $p(y_t|\boldsymbol{y}_{1:t-1}, \boldsymbol{u}_{1:t}) = \mathcal{N}(\mu_t, \sigma_t^2)$. Anomalous observations are defined as points where $z_t > 3$ and are marked with red crosses. Generated by sts.ipynb.

for modeling all of the residual variation in the data beyond what is accounted for by the observation noise.

We can also use the model for **anomaly detection**. To do this, we compute the one-step-ahead predictive distributions, $p(y_t|\boldsymbol{y}_{1:t-1}, \boldsymbol{u}_{1:t})$, for each timestep t, and then flag all timesteps where the observation is improbable. The results are shown in Figure 29.36.

29.12.5 Example: forecasting (integer valued) sales

In Section 29.12.4, we used a linear Gaussian STS model to forecast electricity demand. However, for some problems, we have integer valued observations, e.g., for neural spike data (see Section 29.11.1), RNA-Seq data [LJY19], sales data, etc. Here we focus on the case of sales data, where $y_t \in \{0, 1, 2, \ldots\}$ is the number of units of some item that are sold on a given day. Predicting future values of y_t is important for many businesses. (This problem is known as **demand forecasting**.)

We assume the observed counts are due to some latent demand, $z_t \in \mathbb{R}$. Hence we can use a model similar to Section 29.11.1, with a Poisson likelihood, except the linear dynamics are given by an STS model. In [SSF16; See+17], they consider a likelihood of the form $y_t \sim \text{Poi}(y_t|g(d_t^y))$, where $d_t = z_t + \boldsymbol{u}_t^\mathsf{T}\boldsymbol{w}$ is the instantaneous latent demand, $\boldsymbol{u}_t$ are the covariates that encode seasonal indicators (e.g., temporal distance from holidays), and $g(d) = e^d$ or $\log(1+e^d)$ is the transfer function. The dynamics model is a local random walk term, and $z_t = z_{t-1} + \alpha\mathcal{N}(0, 1)$, to capture serial correlation in the data.

However, sometimes we observe zero counts, $y_t = 0$, not because there is no demand, but because there is no supply (i.e., we are out of stock). If we do not model this properly, we may incorrectly infer that $z_t = 0$, thus underestimating demand, which may result in not ordering enough inventory for the future, further compounding the error.

One solution is to use a **zero-inflated Poisson** (**ZIP**) model [Lam92] for the likelihood. This is a mixture model of the form $p(y_t|d_t) = p_0\mathbb{I}(y_t = 0) + (1 - p_0)\text{Poi}(y_t|e^{d_t})$, where p_0 is the probability of the first mixture component. It is also common to use a (possibly zero-inflated) negative binomial model (Section 2.2.1.4) as the likelihood. This is used in [Cha14; Sal+19b] for the demand forecasting problem. The disadvantage of these likelihoods is that they are not log-concave for $d_t = 0$, which

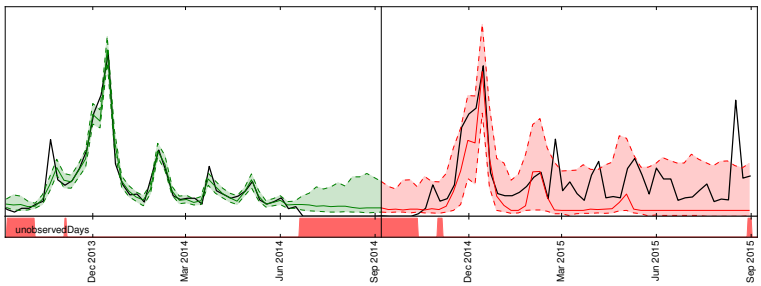


Figure 29.37: Visualization of a probabilistic demand forecast for a hypothetical product. Note the sudden spike near the Christmas holiday in December 2013. The black line denotes the actual demand. Green lines denote the model samples in the training range, while the red lines show the actual probabilistic forecast on data unseen by the model. The red bars at the bottom indicate out-of-stock events which can explain the observed zeros. From Figure 1 of [Bös+17]. Used with kind permission of Tim Januschowski.

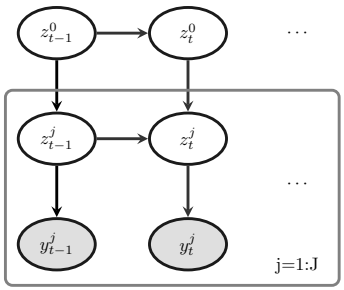


Figure 29.38: Illustration of a hierarchical state-space model.

complicates posterior inference. In particular, the Laplace approximation is a poor choice, since it may find a saddle point. In [SSF16], they tackle this using a log-concave **multi-stage likelihood**, in which $y_t = 0$ is emitted with probability $\sigma(d_t^0)$; otherwise $y_t = 1$ is emitted with probability $\sigma(d_t^1)$; otherwise $y_t =$ is emitted with probability $\text{Poi}(d_t^2)$. This generalizes the scheme in [SOB12].

29.12.6 Example: hierarchical SSM for electoral panel data

Suppose we perform a survey for the US presidential elections. Let N_t^j be the number of people who vote at time t in state j, and let y_t^j be the number of those people who vote Democrat. (We assume $N_t^j - y_t^j$ vote Republican.) It is natural to want to model the dependencies in this data both across time (longitudinally) and across space (this is an example of **panel data**).

We can do this using a hierarchical SSM, as illustrated in Figure 29.38. The top level Markov chain, z_t^0, models national-level trends, and the state-specific chains, z_t^j, model local "random effects". In practice we would usually also include covariates at the national level, $\boldsymbol{u}_t^0$ and state level, $\boldsymbol{u}_t^j$.

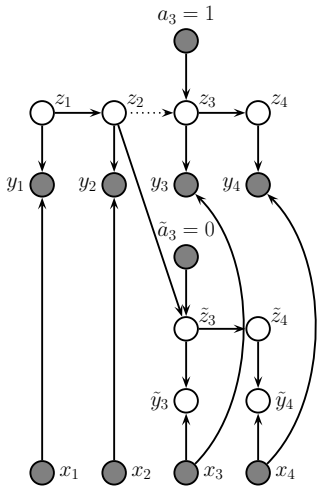


Figure 29.39: Twin network state-space model for estimating causal impact of an intervention that occurs just after time step $n = 2$. We have $m = 4$ actual observations, denoted $\boldsymbol{y}_{1:4}$. We cut the incoming arcs to $\boldsymbol{z}_3$ since we assume $\boldsymbol{z}_{3:T}$ comes from a different distribution, namely the post-intervention distribution. However, in the counterfactual world, shown at the bottom of the figure (with tilde symbols), we assume the distributions are the same as in the past, so information flows along the chain uninterrupted.

Thus the model becomes

$$y_t^j \sim \text{Bin}(y_t^j | \pi_t^j, N_t^j) \tag{29.148}$$

$$\pi_t^j = \sigma\left[(\boldsymbol{z}_t^0)^\mathsf{T}\boldsymbol{u}_t^0 + (\boldsymbol{z}_t^j)^\mathsf{T}\boldsymbol{u}_t^j\right] \tag{29.149}$$

$$\boldsymbol{z}_t^0 = \boldsymbol{z}_{t-1}^0 + \mathcal{N}(\boldsymbol{0}, \sigma^2\mathbf{I}) \tag{29.150}$$

$$\boldsymbol{z}_t^j = \boldsymbol{z}_{t-1}^j + \mathcal{N}(\boldsymbol{0}, \tau^2\mathbf{I}) \tag{29.151}$$

For more details, see [Lin13b].

29.12.7 Causal impact of a time series intervention

In this section, we discuss how to perform **counterfactual reasoning** about the effect on an intervention given some observational (non experimental) time series data. (We discuss counterfactuals in more detail in Section 4.7.4.) For example, suppose y_t is the click through rate (CTR) of the web page of some company at time t. The company launches an ad campaign at time n, and observes outcomes $\boldsymbol{y}_{1:n}$ before the intervention and $\boldsymbol{y}_{n+1:m}$ after the intervention. A natural question to ask is: what would the CTR have been had the company not run the ad campaign?

To answer this question, we will use a structural time series (STS) model (see Section 29.12 for details). An STS model is a linear-Gaussian state-space model, where arrows have a natural causal interpretation in terms of the **arrow of time**; thus a STS is a kind of structural equation model, and hence a structural causal model (see Section 4.7). The use of an SCM allows us to infer the latent state of the noise variables given the observed data; we can then "roll back time" to the point

of intervention, where we explore an alternative "fork in the road" from the one we actually took by "rolling forward in time" in a new version of the model, using the **twin network** approach to counterfactual inference (see Section 4.7.4). This approach is known as "**causal impact**", and was developed by econometricians at Google [Bro+15].

29.12.7.1 Basic idea

To explain the idea in more detail, consider the twin network in Figure 29.39. The intervention occurs after time $n = 2$, and there are $m = 4$ observations in total. We observe 2 data points before the intervention, $\boldsymbol{y}_{1:2}$, and 2 data points afterwards, $\boldsymbol{y}_{3:4}$. We assume observations are generated by latent states $\boldsymbol{z}_{1:4}$, which evolve over time. The states are subject to exogeneous noise terms, which can represent any set of unmodeled factors, such as the state of the economy. In addition, we have exogeneous covariates, $\boldsymbol{x}_{1:m}$.

To predict what would have happened if we had not performed the intervention, (an event denoted by $\tilde{a} = 0$), we replicate the part of the model that occurs after the intervention, and use it to make forecasts. The goal is to compute the counterfactual distribution, $p(\tilde{\boldsymbol{y}}_{n+1:m}|\boldsymbol{y}_{1:n}, \boldsymbol{x}_{1:m})$, where $\tilde{y}_t$ represents counterfactual outcomes if the action had been $\tilde{a} = 0$. We can compute this counterfactual distribution as follows:

$$p(\tilde{\boldsymbol{y}}_{n+1:m}|\boldsymbol{y}_{1:n}, \boldsymbol{x}_{1:m}) = \int p(\tilde{\boldsymbol{y}}_{n+1:m}|\tilde{\boldsymbol{z}}_{n+1:m}, \boldsymbol{x}_{n+1:m}, \boldsymbol{\theta}) p(\tilde{\boldsymbol{z}}_{n+1:m}|\boldsymbol{z}_n, \boldsymbol{\theta}) \times \tag{29.152}$$

$$p(\boldsymbol{z}_n, \boldsymbol{\theta}|\boldsymbol{x}_{1:n}, \boldsymbol{y}_{1:n}) d\boldsymbol{\theta} d\boldsymbol{z}_n d\tilde{\boldsymbol{z}}_{n+1:m} \tag{29.153}$$

where

$$p(\boldsymbol{z}_n, \boldsymbol{\theta}|\boldsymbol{x}_{1:n}, \boldsymbol{y}_{1:n}) = p(\boldsymbol{z}_n|\boldsymbol{x}_{1:n}, \boldsymbol{y}_{1:n}, \boldsymbol{\theta}) p(\boldsymbol{\theta}|\boldsymbol{x}_{1:n}, \boldsymbol{y}_{1:n}) \tag{29.154}$$

For linear Gaussian SSMs, the term $p(\boldsymbol{z}_n|\boldsymbol{x}_{1:n}, \boldsymbol{y}_{1:n}, \boldsymbol{\theta})$ can be computed using Kalman filtering (Section 8.3.2), and the term $p(\boldsymbol{\theta}|\boldsymbol{y}_{1:n}, \boldsymbol{x}_{1:n})$, can be computed using MCMC or variational inference.

We can use samples from the above posterior predictive distribution to compute a Monte Carlo approximation to the distribution of the **treatment effect** per time step, $\tau_t^i = y_t - \tilde{y}_t^i$, where the i index refers to posterior samples. We can also approximate the distribution of the cumulative causal impact using $\sigma_t^i = \sum_{s=n+1}^{t} \tau_t^i$. (There will be uncertainty in these quantities arising both from epistemic uncertainty, about the true parameters controlling the model, and aleatoric uncertainty, due to system and observation noise.)

The validity of the method is based on 3 assumptions: (1) Predictability: we assume that the outcome can be adequately predicted by our model given the data at hand. (We can check this by using **backcasting**, in which we make predictions on part of the historical data.) (2) Unaffectedness: we assume that the intervention does not change future covariates $\boldsymbol{x}_{n+1:m}$. (We can potentially check this by running the method with each of the covariates as an outcome variable.) (3) Stability: we assume that, had the intervention not taken place, the model for the outcome in the pre-treatment period would have continued in the post-treatment period. (We can check this by seeing if we predict an effect if the treatment is shifted earlier in time.)

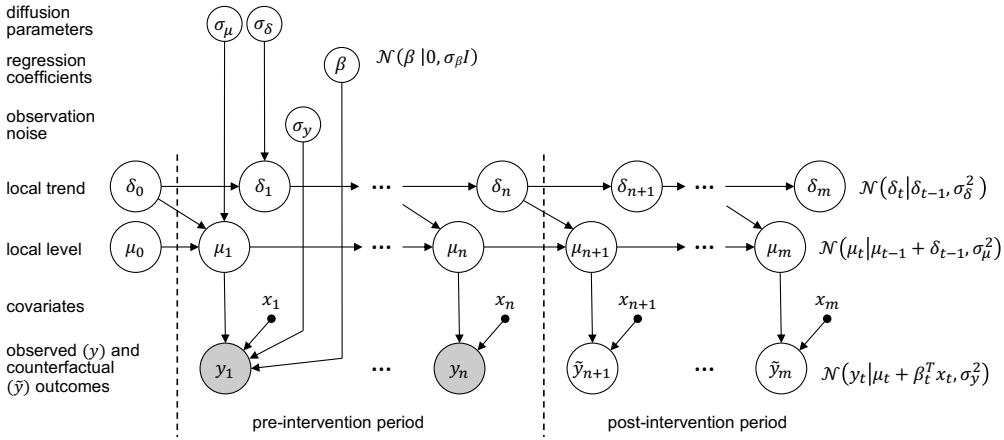


Figure 29.40: A graphical model representation of the local level causal impact model. The dotted line represents the time n at which an intervention occurs. Adapted from Figure 2 of [Bro+15]. Used with kind permission of Kay Brodersen.

29.12.7.2 Example

As a concrete example, let us assume we have a local level model and we use linear regression to model the dependence on the covariates, as in Section 29.12.2.3. That is,

$$y_t = \mu_t + \boldsymbol{\beta}^\mathsf{T}\boldsymbol{x}_t + \mathcal{N}(0, \sigma_y^2) \tag{29.155}$$

$$\mu_t = \mu_{t-1} + \delta_{t-1} + \mathcal{N}(0, \sigma_\mu^2) \tag{29.156}$$

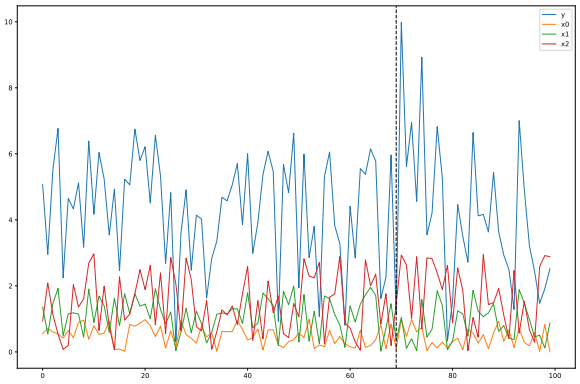
$$\delta_t = \delta_{t-1} + \mathcal{N}(0, \sigma_\delta^2) \tag{29.157}$$

See the graphical model in Figure 29.40. The static parameters of the model are $\boldsymbol{\theta} = (\boldsymbol{\beta}, \sigma_y^2, \sigma_\mu^2, \sigma_\delta^2)$, the other terms are state or observation variables. (Note that we are free to use any kind of STS model; the local level model is just a simple default.)

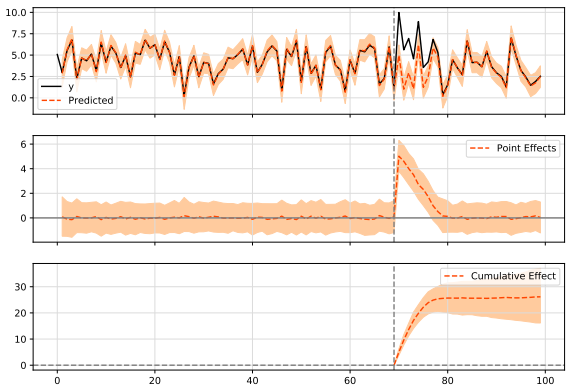
The use of a linear combination of other "**donor**" time series is similar in spirit to the concept of a "**synthetic control**" [Aba; Shi+21]. However we do not restrict ourselves to a convex combination of donors. Furthermore, when we have many covariates, we can use a spike-and-slab prior (Section 15.2.4) or horseshoe prior (Section 15.2.6) to select the relevant ones.

We now give a simple example using synthetic data. We create 3 random sequences of covariates, $\boldsymbol{x}_t \in \mathbb{R}^3$, and define the observed output to be given by Equation (29.155), with regression weights $\boldsymbol{\beta} = (2, 3, 0)$. At time step $n = 70$, we perform an artificial intervention by adding Δ_t to y_t, where Δ_t starts off at 5 and drops down to 0 over a period of 10 steps. This simulates the kind of transient lift one often sees when performing marketing campaigns. The data is shown in Figure 29.41(a). We see that there seems to be a small increase in the blue curve y at around time step 70, but it is hard to tell because of the noise.

We fit a local level STS model using variational inference, and then make the counterfactual forecast shown in the top row of Figure 29.41(b). Now we see more clearly that, had the process



(a)



(b)

Figure 29.41: (a) Some simulated time series data which we use to estimate the causal impact of some intervention, which occurs at time $n = 70$, illustrated by the dotted line. The blue curve are the observed outcomes y, the other curves are covariates (inputs). (b) Output of causal inference. Top row: observed vs predicted outcomes. Middle row: Estimate of causal effect τ_t at each time step. Bottom row: Cumulative causal effect, σ_t, up to each time step. Generated by causal_impact.ipynb.

continued without the intervention, the counterfactual outcome would have been smaller. We can therefore estimate the instantaneous and cumulative causal impact, shown in the middle and bottom rows of Figure 29.41(b). Furthermore, we find that the posterior mean estimate for the regression coefficient is $\overline{\boldsymbol{\beta}} = (1.4927632, 1.945029, -0.10319362)$. We see that the third input variable is mostly ignored, as desired.

29.12.8 Prophet

Prophet [TL18a] is a popular time series forecasting library from Facebook. It fits a generalized additive model of the form

$$y(t) = g(t) + s(t) + h(t) + \boldsymbol{w}^{\mathsf{T}}\boldsymbol{x}(t) + \epsilon_t \tag{29.158}$$

where $g(t)$ is a trend function, $s(t)$ is a seasonal fluctuation (modeled using linear regression applied to a sinusoidal basis set), $h(t)$ is an optional set of sparse "holiday effects", $\boldsymbol{x}(t)$ are an optional set of (possibly lagged) covariates, $\boldsymbol{w}$ are the regression coefficients, and $\epsilon(t)$ is the residual noise term, assumed to be iid Gaussian.

Prophet is a regression model, not an auto-regressive model, since it predicts the time series $\boldsymbol{y}_{1:T}$ given the time stamp t and the covariates $\boldsymbol{x}_{1:T}$, but without conditioning on past observations of y. To model the dependence on time, the trend function is assumed to be a piecewise linear trend with S changepoints, uniformly spaced in time. (See Section 29.5.6 for a discussion of changepoint detection.) That is, the model has the form

$$g(t) = (k + \boldsymbol{a}(t)^T\boldsymbol{\delta})t + (m + \boldsymbol{a}(t)^{\mathsf{T}}\boldsymbol{\gamma}) \tag{29.159}$$

where k is the growth rate, m is the offset, $a_j(t) = \mathbb{I}(t \geq s_j)$, where s_j is the time of the j'th changepoint, $\delta_t \sim \text{Laplace}(\tau)$ is the magnitude of the change, and $\gamma_j = -s_j\delta_j$ to make the function continuous. The Laplace prior on δ ensures the MAP parameter estimate is sparse, so the difference across change point boundaries is usually 0.

For an interactive visualization of how prophet works, see http://prophet.mbrouns.com/.

29.12.9 Neural forecasting methods

Classical time series methods work well when there is little data (e.g., short sequences, or few covariates). However, in some cases, we have a lot of data. For example, we might have a single, but very long sequence, such as in anomaly detection from real-time sensors [Ahm+17]. Or we may have multiple, related sequences, such as sales of related products [Sal+19b]. In both cases, larger data means we can afford to fit more complex parametric models. Neural networks are a natural choice, because of their flexibility. Until recently, their performance in forecasting tasks was not competitive with classical methods, but this has recently started to change, as described in [Ben+20; LZ20].

A common benchmark in the univariate time series forecasting literature is the **M4 forecasting competition** [MSA18], which requires participants to make forecasts on many different kinds of (univariate) time series (without covariates). This was recently won by a neural method [Smy20]. More precisely, the winner of the 2019 M4 competition was a *hybrid* RNN-classical method called **ES-RNN** [Smy20]. The exponential smoothing (ES) part allows data-efficient adaptation to the observed past of the current time series; the recurrent neural network (RNN) part allows for learning

of nonlinear components from multiple related timeseries. (This is known as a **local+global** model, since the ES part is "trained" just on the local timeseries, whereas the RNN is trained on a global dataset of related time series.)

In [Ran+18] they adopt a different approach for combining RNNs and classical methods, called **DeepSSM**. In particular, they train a single RNN to predict the parameters of a state-space model (see Section 29.1). In more detail, let $\boldsymbol{x}_{1:T}^n$ represent the n'th time series, and let $\boldsymbol{\theta}_t^n$ represent the non-stationary parameters of a linear-trend SSM model (see Section 29.12.1). We train an RNN to compute $\boldsymbol{\theta}_t^n = f(\boldsymbol{c}_{1:T}^n; \boldsymbol{\phi})$, where $\boldsymbol{\phi}$ are the RNN parameters shared across all sequences. We can use the predicted parameters to compute the log likelihood of the sequence, $L_n = \log p(\boldsymbol{x}_{1:T}^n | \boldsymbol{c}_{1:T}^n, \boldsymbol{\theta}_{1:T}^n)$, using the Kalman filter. These two modules can be combined to allow for end-to-end training of $\boldsymbol{\phi}$ to maximize $\sum_{n=1}^N L_n$.

In [Wan+19c], they propose a different hybrid model known as **Deep Factors**. The idea is to represent each time series (or its latent function, for non-Gaussian data) as a weighted sum of a global time series, coming from a neural model, and a stochastic local model, such as an SSM or GP. The **DeepGLO** (global-local) approach of [SYD19] proposes a related hybrid method, where the global model uses matrix factorization to learn shared factors. This is then combined with temporal convolutional networks.

It is also possible to train a purely neural model, without resorting to classical methods. For example, the **N-BEATS** model of [Ore+20] trains a residual network to predict the weights of a set of basis functions, corresponding to a polynomial trend and a periodic signal. The weights for the basis functions are predicted for each window of input using the neural network. Another approach is the **DeepAR** model of [Sal+19b], which fits a single RNN to a large number of time series. The original paper used integer (count) time series, modeled with a negative binomial likelihood function. This is a unimodal distribution, which may not be suitable for all tasks. More flexible forms, such as mixtures of Gaussians, have also been proposed [Muk+18]. A popular alternative is to use **quantile regression** [Koe05], in which the model is trained to predict quantiles of the distribution. For example, [Gas+19] proposed **SQF-RNN**, which uses splines to represent the quantile function. They used **CRPS** or **continuous-ranked probability score** as the loss function. This is a proper scoring rule, but is less sensitive to outliers, and is more "distance aware", than log loss.

The above methods all predict a single output (per time step). If there are multiple simultaneous observations, it is best to try to model their interdependencies. In [Sal+19a], they use a (low-rank) **Gaussian copula** for this, and in [Tou+19], they use a **nonparametric copula**.

In [Wen+17], they simultaneously predict quantiles for multiple steps ahead using dilated causal convolution (or an RNN). They call their method **MQ-CNN**. In [WT19], they extend this to predict the full quantile function, taking as input the desired quantile level α, rather than prespecifying a fixed set of levels. They also use a copula to learn the dependencies among multiple univariate marginals.

29.13 Deep SSMs

Traditional state-space model assume linear dynamics and linear observation models, both with additive Gaussian noise. This is obviously very limiting. In this section, we allow the dynamics and/or observation model to be modeled by nonlinear and/or non-Markovian deep neural networks; we call these **deep state-space model**, also known as **dynamical variational autoencoders**.

(To be consistent with the literature on VAEs, we denote the observations by $\boldsymbol{x}_t$ instead of $\boldsymbol{y}_t$.) For a detailed review, see [Ged+20; Gir+21].

29.13.1 Deep Markov models

Suppose we create a SSM in which we use a deep neural network for the dynamics model and/or observation model; the result is called a **deep Markov model** [KSS17] or **stochastic RNN** [BO14; Fra+16]. (This is not quite the same as a variational RNN, which we explain in Section 29.13.4.)

We can fit a DMM using SVI (Section 10.3.1). The key is to infer the posterior over the latents. From the first-order Markov properties, the exact posterior is given by

$$p(\boldsymbol{z}_{1:T}|\boldsymbol{x}_{1:T}) = \prod_{t=1}^{T} p(\boldsymbol{z}_t|\boldsymbol{z}_{t-1}, \boldsymbol{x}_{1:T}) = \prod_{t=1}^{T} p(\boldsymbol{z}_t|\boldsymbol{z}_{t-1}, \cancel{\boldsymbol{x}_{1:t-1}}, \boldsymbol{x}_{t:T}) \tag{29.160}$$

where we define $p(\boldsymbol{z}_1|\boldsymbol{z}_0, \boldsymbol{x}_{1:T}) = p(\boldsymbol{z}_1|\boldsymbol{x}_{1:T})$, and the cancelation follows since $\boldsymbol{z}_t \perp \boldsymbol{x}_{1:t-1}|\boldsymbol{z}_{t-1}$, as pointed out in [KSS17].

In general, it is intractable to compute $p(\boldsymbol{z}_{1:T}|\boldsymbol{x}_{1:T})$, so we approximate it with an inference network. There are many choices for q. A simple one is a fully factorized model, $q(\boldsymbol{z}_{1:T}) = \prod_t q(\boldsymbol{z}_t|\boldsymbol{x}_{1:t})$. This is illustrated in Figure 29.42a. Since $\boldsymbol{z}_t$ only depends on past data, $\boldsymbol{x}_{1:t}$ (which is accumulated in the RNN hidden state $\boldsymbol{h}_t$), we can use this inference network at run time for online inference. However, for training the model offline, we can use a more accurate posterior by using

$$q(\boldsymbol{z}_{1:T}|\boldsymbol{x}_{1:T}) = \prod_{t=1}^{T} q(\boldsymbol{z}_t|\boldsymbol{z}_{t-1}, \boldsymbol{x}_{1:T}) = \prod_{t=1}^{T} q(\boldsymbol{z}_t|\boldsymbol{z}_{t-1}, \cancel{\boldsymbol{x}_{1:t-1}}, \boldsymbol{x}_{t:T}) \tag{29.161}$$

Note that the dependence on past observation $\boldsymbol{x}_{1:t-1}$ is already captured by $\boldsymbol{z}_{t-1}$, as in Equation (29.160). The dependencies on future observations, $\boldsymbol{x}_{t:T}$, can be summarized by a backwards RNN, as shown in Figure 29.42b. Thus

$$q(\boldsymbol{z}_{1:T}, \boldsymbol{h}_{1:T}|\boldsymbol{x}_{1:T}) = \prod_{t=T}^{1} \mathbb{I}\left(\boldsymbol{h}_t = f(\boldsymbol{h}_{t+1}, \boldsymbol{x}_t)\right) \prod_{t=1}^{T} q(\boldsymbol{z}_t|\boldsymbol{z}_{t-1}, \boldsymbol{h}_t) \tag{29.162}$$

Given a fully factored $q(\boldsymbol{z}_{1:T})$, we can compute the ELBO as follows.

$$\log p(\boldsymbol{x}_{1:T}) = \log \left[\sum_{\boldsymbol{z}_{1:T}} p(\boldsymbol{x}_{1:T}|\boldsymbol{z}_{1:T}) p(\boldsymbol{z}_{1:T})\right] \tag{29.163}$$

$$= \log \mathbb{E}_{q(\boldsymbol{z}_{1:T})}\left[p(\boldsymbol{x}_{1:T}|\boldsymbol{z}_{1:T}) \frac{p(\boldsymbol{z}_{1:T})}{q(\boldsymbol{z}_{1:T})}\right] \tag{29.164}$$

$$= \log \mathbb{E}_{q(\boldsymbol{z}_{1:T})}\left[\prod_{t=1}^{T} \frac{p(\boldsymbol{x}_t|\boldsymbol{z}_t) p(\boldsymbol{z}_t|\boldsymbol{z}_{t-1})}{q(\boldsymbol{z}_t)}\right] \tag{29.165}$$

$$\geq \mathbb{E}_{q(\boldsymbol{z}_{1:T})}\left[\sum_{t=1}^{T} \log p(\boldsymbol{x}_t|\boldsymbol{z}_t) + \log p(\boldsymbol{z}_t|\boldsymbol{z}_{t-1}) - \log q(\boldsymbol{z}_t)\right] \tag{29.166}$$

$$= \sum_{t=1}^{T} \mathbb{E}_{q(\boldsymbol{z}_t)}\left[\log p(\boldsymbol{x}_t|\boldsymbol{z}_t)\right] - \mathbb{E}_{q(\boldsymbol{z}_{t-1})}\left[D_{\mathbb{KL}}\left(q(\boldsymbol{z}_t) \parallel p(\boldsymbol{z}_t|\boldsymbol{z}_{t-1})\right)\right] \tag{29.167}$$

(a) *(b)*

Figure 29.42: Inference networks for deep Markov model. (a) Fully factorized causal posterior $q(\boldsymbol{z}_{1:T}|\boldsymbol{x}_{1:T}) = \prod_t q(\boldsymbol{z}_t|\boldsymbol{x}_{1:t})$. The past observations $\boldsymbol{x}_{1:t}$ are stored in the RNN hidden state $\boldsymbol{h}_t$. (b) Markovian posterior $q(\boldsymbol{z}_{1:T}|\boldsymbol{x}_{1:T}) = \prod_t q(\boldsymbol{z}_t|\boldsymbol{z}_{t-1}, \boldsymbol{x}_{t:T})$. The future observations $\boldsymbol{x}_{t:T}$ are stored in the RNN hidden state $\boldsymbol{h}_t$.

(a) *(b)*

Figure 29.43: Recurrent state-space models. (a) Prior is first-order Markov, $p(\boldsymbol{z}_t|\boldsymbol{z}_{t-1})$, but observation model is not Markovian, $p(\boldsymbol{x}_t|\boldsymbol{h}_t) = p(\boldsymbol{x}_t|\boldsymbol{z}_{1:t})$, where $\boldsymbol{h}_t$ summarizes $\boldsymbol{z}_{1:t}$. (b) Prior model is no longer first-order Markov either, $p(\boldsymbol{z}_t|\boldsymbol{h}_{t-1}) = p(\boldsymbol{z}_t|\boldsymbol{z}_{1:t-1})$. Diamonds are deterministic nodes, circles are stochastic.

If we assume that the variational posteriors are jointly Gaussian, we can use the reparameterization trick to use posterior samples to compute stochastic gradients of the ELBO. Furthermore, since we assumed a Gaussian prior, the KL term can be computed analytically.

29.13.2 Recurrent SSM

In a DMM, the observation model $p(\boldsymbol{x}_t|\boldsymbol{z}_t)$ is first-order Markov, as is the dynamics model $p(\boldsymbol{z}_t|\boldsymbol{z}_{t-1})$. We can modify the model so that it captures long-range dependencies by adding deterministic hidden states as well. We can make the observation model depend on $\boldsymbol{z}_{1:t}$ instead of just $\boldsymbol{z}_t$ by using $p(\boldsymbol{x}_t|\boldsymbol{h}_t)$, where $\boldsymbol{h}_t = f(\boldsymbol{h}_{t-1}, \boldsymbol{z}_t)$, so $\boldsymbol{h}_t$ records all the stochastic choices. This is illustrated in Figure 29.43a. We can also make the dynamical prior depend on $\boldsymbol{z}_{1:t-1}$ by replacing $p(\boldsymbol{z}_t|\boldsymbol{z}_{t-1})$ with $p(\boldsymbol{z}_t|\boldsymbol{h}_{t-1})$. as is illustrated in Figure 29.43b. This is known as a **recurrent SSM**.

We can derive an inference network for an RSSM similar to the one we used for DMMs, except now we use a standard forwards RNN to compute $q(\boldsymbol{z}_t|\boldsymbol{x}_{1:t-1}, \boldsymbol{x}_{1:t})$.

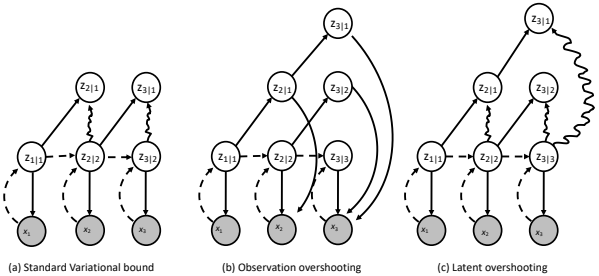


Figure 29.44: Unrolling schemes for SSMs. The labels $z_{i|j}$ is shorthand for $p(\boldsymbol{z}_i|\boldsymbol{x}_{1:j})$. Solid lines denote the generative process, dashed lines the inference process. Arrows pointing at shaded circles represent log-likelihood loss terms. Wavy arrows indicate KL divergence loss terms. (a) Standard 1 step reconstruction of the observations. (b) Observation overshooting tries to predict future observations by unrolling in latent space. (c) Latent overshooting predicts future latent states and penalizes their KL divergence, but does need to care about future observations. Adapted from Figure 3 of [Haf+19].

29.13.3 Improving multi-step predictions

In Figure 29.44(a), we show the loss terms involved in the ELBO. In particular, the wavy edge $z_{t|t} \rightarrow z_{t|t-1}$ corresponds to $\mathbb{E}_{q(\boldsymbol{z}_{t-1})}\left[D_{\mathbb{KL}}\left(q(\boldsymbol{z}_t) \parallel p(\boldsymbol{z}_t|\boldsymbol{z}_{t-1})\right)\right]$, and the solid edge $z_{t|t} \rightarrow \boldsymbol{x}_t$ corresponds to $\mathbb{E}_{q(\boldsymbol{z}_t)}\left[\log p(\boldsymbol{x}_t|\boldsymbol{z}_t)\right]$. We see that the dynamics model, $p(\boldsymbol{z}_t|\boldsymbol{z}_{t-1})$, is only ever penalized in terms of how it differs from the one-step-ahead posterior $q(\boldsymbol{z}_t)$, which can hurt the ability of the model to make long-term predictions.

One solution to this is to make multi-step forward predictions using the dynamics model, and use these to reconstruct future observations, and add these errors as extra loss terms. This is called **observation overshooting** [Amo+18], and is illustrated in Figure 29.44(b).

A faster approach, proposed in [Haf+19], is to apply a similar idea but in latent space. More precisely, let us compute the multi-step prediction model, by repeatedly applying the transition model and integrating out the intermediate states to get $p(\boldsymbol{z}_t|\boldsymbol{z}_{t-d})$. We can then compute the ELBO for this as follows:

$$\log p_d(\boldsymbol{x}_{1:T}) \triangleq \log \int \prod_{t=1}^{T} p(\boldsymbol{z}_t|\boldsymbol{z}_{t-d}) p(\boldsymbol{x}_t|\boldsymbol{z}_t) d\boldsymbol{z}_{1:T} \tag{29.168}$$

$$\geq \sum_{t=1}^{T} \mathbb{E}_{q(\boldsymbol{z}_t)}\left[\log p(\boldsymbol{x}_t|\boldsymbol{z}_t)\right] - \mathbb{E}_{p(\boldsymbol{z}_{t-1}|\boldsymbol{z}_{t-d})q(\boldsymbol{z}_{t-d})}\left[D_{\mathbb{KL}}\left(q(\boldsymbol{z}_t) \parallel p(\boldsymbol{z}_t|\boldsymbol{z}_{t-1})\right)\right] \tag{29.169}$$

To train the model so it is good at predicting at different future horizon depths d, we can average the above over all $1 \leq d \leq D$. However, for computational reasons, we can instead just average the KL terms, using weights β_d. This is called **latent overshooting** [Haf+19], and is illustrated in

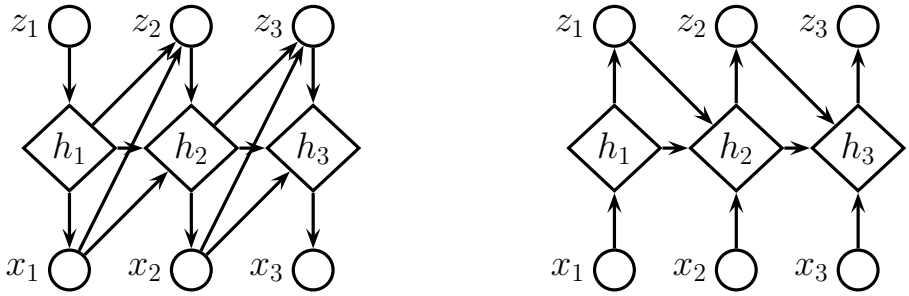


Figure 29.45: Variational RNN. (a) Generative model. (b) Inference model. The diamond-shaped nodes are deterministic.

Figure 29.44(c). The new objective becomes

$$\frac{1}{D}\sum_{d=1}^{D}\log p_d(\boldsymbol{x}_{1:T}) \geq \sum_{t=1}^{T}\mathbb{E}_{q(\boldsymbol{z}_t)}\left[\log p(\boldsymbol{x}_t|\boldsymbol{z}_t)\right] \tag{29.170}$$

$$-\frac{1}{D}\sum_{d=1}^{D}\beta_d\mathbb{E}_{p(\boldsymbol{z}_{t-1}|\boldsymbol{z}_{t-d})q(\boldsymbol{z}_{t-d})}\left[D_{\mathbb{KL}}\left(q(\boldsymbol{z}_t)\parallel p(\boldsymbol{z}_t|\boldsymbol{z}_{t-1})\right)\right] \tag{29.171}$$

29.13.4 Variational RNNs

A **variational RNN** (**VRNN**) [Chu+15] is similar to a recurrent SSM except the hidden states are generated conditional on all past hidden states *and* all past observations, rather than just the past hidden states. This is a more expressive model, but is slower to use for forecasting, since unrolling into the future requires generating observations $\boldsymbol{x}_{t+1}, \boldsymbol{x}_{t+2}, \ldots$ to "feed into" the hidden states, which controls the dynamics. This makes the model less useful for forecasting and model-based RL (see Section 35.4.5.2).

More precisely, the generative model is as follows:

$$p(\boldsymbol{x}_{1:T}, \boldsymbol{z}_{1:T}, \boldsymbol{h}_{1:T}) = \prod_{t=1}^{T} p(\boldsymbol{z}_t|\boldsymbol{h}_{t-1}, \boldsymbol{x}_{t-1})\mathbb{I}\left(\boldsymbol{h}_t = f(\boldsymbol{h}_{t-1}, \boldsymbol{x}_{t-1}, \boldsymbol{z}_t)\right) p(\boldsymbol{x}_t|\boldsymbol{h}_t) \tag{29.172}$$

where $p(\boldsymbol{z}_1|\boldsymbol{h}_0, \boldsymbol{x}_0) = p(\boldsymbol{z}_0)$ and $\boldsymbol{h}_1 = f(\boldsymbol{h}_0, \boldsymbol{x}_0, \boldsymbol{z}_1) = f(\boldsymbol{z}_1)$. Thus $\boldsymbol{h}_t = (\boldsymbol{z}_{1:t}, \boldsymbol{x}_{1:t-1})$ is a summary of the past observations and past and current stochastic latent samples. If we marginalize out these deterministic hidden nodes, we see that the dynamical prior on the stochastic latents is $p(\boldsymbol{z}_t|\boldsymbol{h}_{t-1}, \boldsymbol{x}_{t-1}) = p(\boldsymbol{z}_t|\boldsymbol{z}_{1:t-1}, \boldsymbol{x}_{1:t-1})$, whereas in a DMM, it is $p(\boldsymbol{z}_t|\boldsymbol{z}_{t-1})$, and in an RSSM, it is $p(\boldsymbol{z}_t|\boldsymbol{z}_{1:t-1})$. See Figure 29.45a for an illustration.

We can train VRNNs using SVI. In [Chu+15], they use the following inference network:

$$q(\boldsymbol{z}_{1:T}, \boldsymbol{h}_{1:T}|\boldsymbol{x}_{1:T}) = \prod_{t=1}^{T}\mathbb{I}\left(\boldsymbol{h}_t = f(\boldsymbol{h}_{t-1}, \boldsymbol{z}_{t-1}, \boldsymbol{x}_t)\, q(\boldsymbol{z}_t|\boldsymbol{h}_t)\right) \tag{29.173}$$

Thus $\boldsymbol{h}_t = (\boldsymbol{z}_{1:t-1}, \boldsymbol{x}_{1:t})$. Marginalizing out these deterministic nodes, we see that the filtered posterior has the form $q(\boldsymbol{z}_{1:T}|\boldsymbol{x}_{1:T}) = \prod_t q(\boldsymbol{z}_t|\boldsymbol{z}_{1:t-1}, \boldsymbol{x}_{1:t})$. See Figure 29.45b for an illustration. (We can also optionally replace $\boldsymbol{x}_t$ with the output of a bidirectional RNN to get the smoothed posterior, $q(\boldsymbol{z}_{1:T}|\boldsymbol{x}_{1:T}) = \prod_t q(\boldsymbol{z}_t|\boldsymbol{z}_{1:t-1}, \boldsymbol{x}_{1:T})$.)

This approach was used in [DF18] to generate simple videos of moving objects (e.g., a robot pushing a block); they call their method **stochastic video generation** or **SVG**. This was scaled up in [Vil+19], using simpler but larger architectures.

30 Graph learning

30.1 Introduction

Graphs are a very common way to represent data. In this chapter we discuss probability models for graphs. In Section 30.2, we assume the graph structure G is known, but we want to "explain" it in terms of a set of meaningful latent features; for this we use various kinds of latent variable models. In Section 30.3, we assume the graph structure G is unknown and needs to be inferred from correlated data, $\boldsymbol{x}_n \in \mathbb{R}^D$; for this, we will use probabilistic graphical models with unknown topology. (See also Section 16.3.6, where we discuss graph neural networks, for performing supervised learning using graph-structured data.)

30.2 Latent variable models for graphs

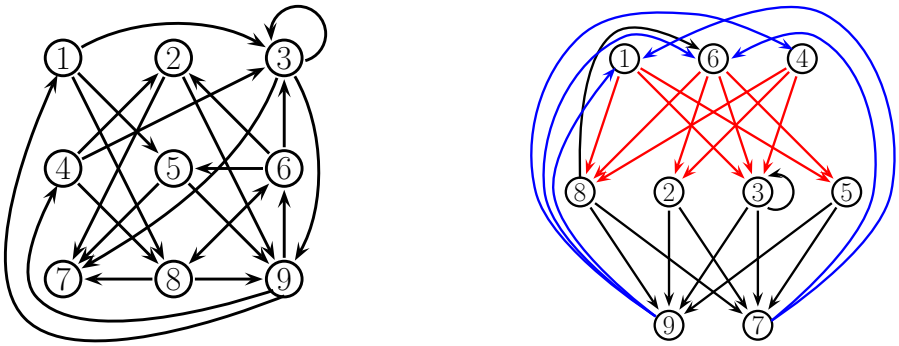
Graphs arise in many application areas, such as modeling social networks, protein-protein interaction networks, or patterns of disease transmission between people or animals. There are usually two primary goals when analysing such data: first, try to discover some "interesting structure" in the graph, such as clusters or communities; second, try to predict which links might occur in the future (e.g., who will make friends with whom). In this section, we focus on the former. More precisely, we will consider a variety of latent variable models for observed graphs.

30.2.1 Stochastic block model

In Figure 30.1(a) we show a directed graph on 9 nodes. There is no apparent structure. However, if we look more deeply, we see it is possible to partition the nodes into three groups or blocks, $B_1 = \{1, 4, 6\}$, $B_2 = \{2, 3, 5, 8\}$, and $B_3 = \{7, 9\}$, such that most of the connections go from nodes in B_1 to B_2, or from B_2 to B_3, or from B_3 to B_1. This is illustrated in Figure 30.1(b).

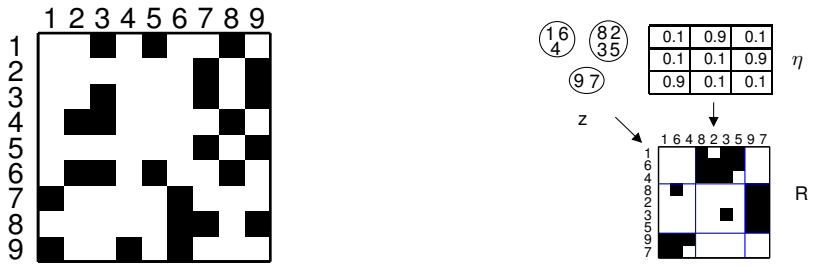
The problem is easier to understand if we plot the adjacency matrices. Figure 30.2(a) shows the matrix for the graph with the nodes in their original ordering. Figure 30.2(b) shows the matrix for the graph with the nodes in their permuted ordering. It is clear that there is block structure.

We can make a generative model of block structured graphs as follows. First, for every node, sample a latent block $q_i \sim \text{Cat}(\boldsymbol{\pi})$, where π_k is the probability of choosing block k, for $k = 1 : K$. Second, choose the probability of connecting group a to group b, for all pairs of groups; let us denote this probability by $\eta_{a,b}$. This can come from a beta prior. Finally, generate each edge R_{ij} using the



(a) (b)

Figure 30.1: (a) A directed graph. (b) The same graph, with the nodes partitioned into 3 groups, making the block structure more apparent.



(a) (b)

Figure 30.2: (a) Adjacency matrix for the graph in Figure 30.1(a). (b) Rows and columns are shown permuted to show the block structure. We also show how the stochastic block model can generate this graph. From Figure 1 of [Kem+06]. Used with kind permission of Charles Kemp.

following model:

$$p(R_{ij} = r|q_i = a, q_j = b, \boldsymbol{\eta}) = \text{Ber}(r|\eta_{a,b}) \tag{30.1}$$

This is called the **stochastic block model** [NS01]. Figure 30.4(a) illustrates the model as a DGM, and Figure 30.2 illustrates how this model can be used to cluster the nodes in our example.

Note that this is quite different from a conventional clustering problem. For example, we see that all the nodes in block 3 are grouped together, even though there are no connections between them. What they share is the property that they "like to" connect to nodes in block 1, and to receive connections from nodes in block 2. Figure 30.3 illustrates the power of the model for generating many different kinds of graph structure. For example, some social networks have hierarchical structure, which can be modeled by clustering people into different social strata, whereas others consist of a set of cliques.

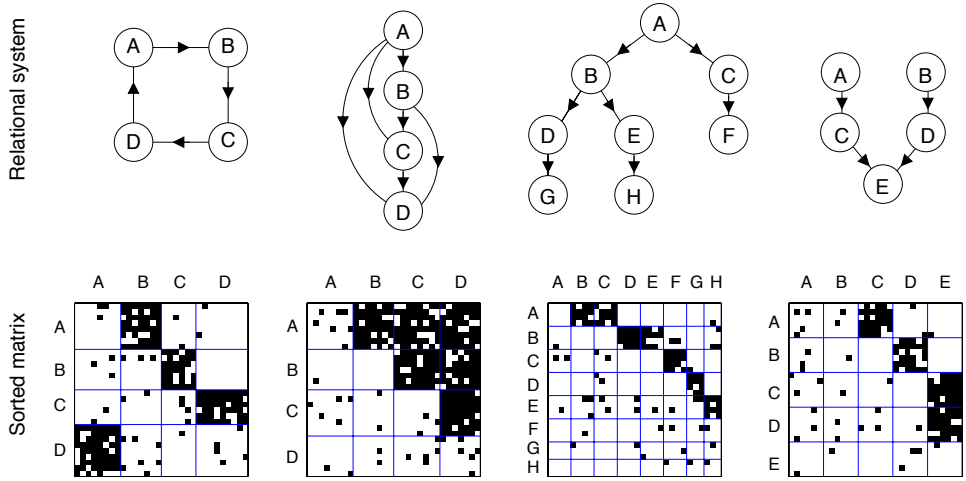


Figure 30.3: Some examples of graphs generated using the stochastic block model with different kinds of connectivity patterns between the blocks. The abstract graph (between blocks) represent a ring, a dominance hierarchy, a common-cause structure, and a common-effect structure. From Figure 4 of [Kem+10]. Used with kind permission of Charles Kemp.

Unlike a standard mixture model, it is not possible to fit this model using exact EM, because all the latent q_i variables become correlated. However, one can use variational EM [Air+08], collapsed Gibbs sampling [Kem+06], etc. We omit the details (which are similar to the LDA case).

In [Kem+06], they lifted the restriction that the number of blocks K be fixed, by replacing the Dirichlet prior on $\boldsymbol{\pi}$ by a Dirichlet process (see Section 31.2). This is known as the infinite relational model. See Section 30.2.3 for details.

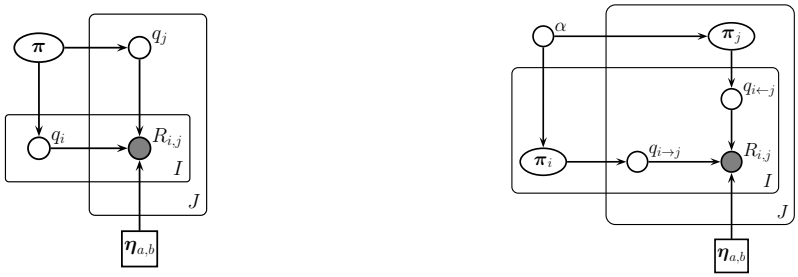
If we have features associated with each node, we can make a discriminative version of this model, for example by defining

$$p(R_{ij} = r|q_i = a, q_j = b, \boldsymbol{x}_i, \boldsymbol{x}_j, \boldsymbol{\theta}) = \text{Ber}(r|\boldsymbol{w}_{a,b}^T f(\boldsymbol{x}_i, \boldsymbol{x}_j)) \tag{30.2}$$

where $f(\boldsymbol{x}_i, \boldsymbol{x}_j)$ is some way of combining the feature vectors. For example, we could use concatenation, $[\boldsymbol{x}_i, \boldsymbol{x}_j]$, or elementwise product $\boldsymbol{x}_i \otimes \boldsymbol{x}_j$ as in supervised LDA. The overall model is like a relational extension of the mixture of experts model.

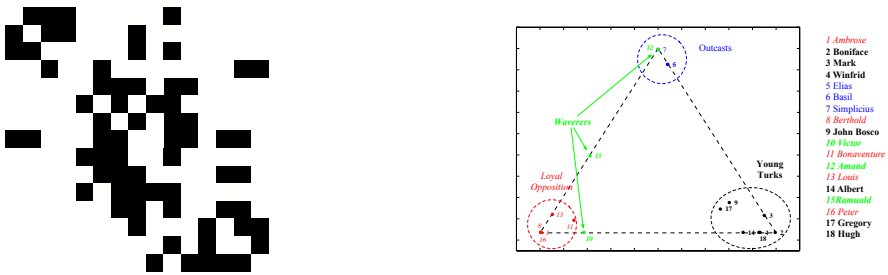
30.2.2 Mixed membership stochastic block model

In [Air+08], they lifted the restriction that each node only belong to one cluster. That is, they replaced $q_i \in \{1, \ldots, K\}$ with $\boldsymbol{\pi}_i \in S_K$. This is known as the **mixed membership stochastic block model**, and is similar in spirit to **fuzzy clustering** or **soft clustering**. Note that π_{ik} is not the same as $p(z_i = k|\mathcal{D})$; the former represents **ontological uncertainty** (to what degree does each object belong to a cluster) whereas the latter represents **epistemological uncertainty** (which cluster does an object belong to). If we want to combine epistemological and ontological uncertainty, we can compute $p(\boldsymbol{\pi}_i|\mathcal{D})$.



(a) *(b)*

Figure 30.4: (a) Stochastic block model. (b) Mixed membership stochastic block model.



(a) *(b)*

Figure 30.5: (a) Who-likes-whom graph for Sampson's monks. (b) Mixed membership of each monk in one of three groups. From Figures 2-3 of [Air+08]. Used with kind permission of Edo Airoldi.

In more detail, the generative process is as follows. First, each node picks a distribution over blocks, $\boldsymbol{\pi}_i \sim \text{Dir}(\boldsymbol{\alpha})$. Second, choose the probability of connecting group a to group b, for all pairs of groups, $\eta_{a,b} \sim \beta(\alpha, \beta)$. Third, for each edge, sample two discrete variables, one for each direction:

$$q_{i\rightarrow j} \sim \text{Cat}(\boldsymbol{\pi}_i), \; q_{i\leftarrow j} \sim \text{Cat}(\boldsymbol{\pi}_j) \tag{30.3}$$

Finally, generate each edge R_{ij} using the following model:

$$p(R_{ij} = 1 | q_{i\rightarrow j} = a, q_{i\leftarrow j} = b, \boldsymbol{\eta}) = \eta_{a,b} \tag{30.4}$$

See Figure 30.4(b) for the DGM.

Unlike the regular stochastic block model, each node can play a different role, depending on who it is connecting to. As an illustration of this, we will consider a data set that is widely used in the social networks analysis literature. The data concerns who-likes-whom amongst of group of 18 monks. It was collected by hand in 1968 by Sampson [Sam68] over a period of months. (These days, in the era of social media such as Facebook, a social network with only 18 people is trivially small, but the methods we are discussing can be made to scale.) Figure 30.5(a) plots the raw data, and Figure 30.5(b) plots $\mathbb{E}[\boldsymbol{\pi}]_i$ for each monk, where $K = 3$. We see that most of the monks belong

to one of the three clusters, known as the "young turks", the "outcasts" and the "loyal opposition". However, some individuals, notably monk 15, belong to two clusters; Sampson called these monks the "waverers". It is interesting to see that the model can recover the same kinds of insights as Sampson derived by hand.

One prevalent problem in social network analysis is missing data. For example, if $R_{ij} = 0$, it may be due to the fact that person i and j have not had an opportunity to interact, or that data is not available for that interaction, as opposed to the fact that these people don't want to interact. In other words, *absence of evidence is not evidence of absence*. We can model this by modifying the observation model so that with probability ρ, we generate a 0 from the background model, and we only force the model to explain observed 0s with probability $1 - \rho$. In other words, we robustify the observation model to allow for outliers, as follows:

$$p(R_{ij} = r | q_{i \to j} = a, q_{i \leftarrow j} = b, \boldsymbol{\eta}) = \rho \delta_0(r) + (1 - \rho) \text{Ber}(r | \eta_{a,b}) \tag{30.5}$$

See [Air+08] for details.

30.2.3 Infinite relational model

The stochastic block model is defined for graphs, in which each pair of edges may or may not have an edge. We can easily extend this to hyper-graphs, which is useful for modeling relational data. For example, suppose we want to model a family tree. We might write $R_1(i, j, k) = 1$ if adults i and j are the parents of child k, where R_1 is the "parent-of" relation. Here i and j are entities of type T^1 (adults), and j is an entity of type T^2 (child), so the type signature of R_1 is $T^1 \times T^1 \times T^2 \to \{0, 1\}$.

To define the probability of relations holding between entities, we can associate a latent cluster variable $q_i^t \in \{1, \ldots, K_t\}$ with each entity i of each type t. We then define the probability of the relation holding between specific entities by looking up the probability of the relation holding between the corresponding entity clusters. Continuing our example above, we have

$$p(R_1(i, j, k) | q_i^1 = a, q_j^1 = b, q_k^2 = c, \boldsymbol{\eta}) = \text{Ber}(R_1(i, j, k) | \eta_{a,b,c}) \tag{30.6}$$

We can also have real-valued relations, where each edge has a weight. For example, we can write

$$p(R_1(i, j, k) | q_i^1 = a, q_j^1 = b, q_k^2 = c, \boldsymbol{\mu}) = \mathcal{N}(R_1(i, j, k) | \mu_{a,b,c}, \sigma^2), \tag{30.7}$$

where $\mu_{a,b,c}$ captures the average response for that group of clusters. We can also add entity-specific offset terms:

$$p(R_1(i, j, k) | q_i^1 = a, q_j^1 = b, q_k^2 = c, \boldsymbol{\mu}) = \mathcal{N}(R_1(i, j, k) | \mu_{a,b,c} + \mu_i + \mu_j + \mu_k, \sigma^2), \tag{30.8}$$

This model was proposed in [BBM07], who fit the model using an alternating minimization procedure.

If we allow the number of clusters K_t for each type of entity to be unbounded, by using a Dirichlet process, the model is called the **infinite relational model** (IRM) [Kem+06], also known as an **infinite hidden relational model** (IHRM) [Xu+06]. We can fit this model with variational Bayes [Xu+06; Xu+07] or collapsed Gibbs sampling [Kem+06]. Rather than go into algorithmic detail, we just sketch some interesting applications.

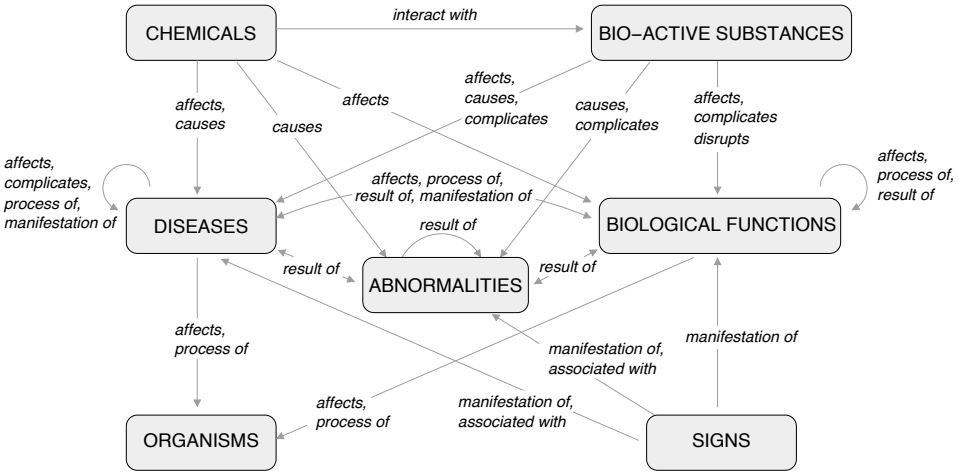


Figure 30.6: Illustration of an ontology learned by IRM applied to the Unified Medical Language System. The boxes represent 7 of the 14 concept clusters. Predicates that belong to the same cluster are grouped together, and associated with edges to which they pertain. All links with weight above 0.8 have been included. From Figure 9 of [Kem+10]. Used with kind permission of Charles Kemp.

30.2.3.1 Learning ontologies

An **ontology** refers to an organisation of knowledge. In AI, ontologies are often built by hand (see e.g., [RN10]), but it is interesting to try and learn them from data. In [Kem+06], they show how this can be done using the IRM.

The data comes from the Unified Medical Language System [McC03], which defines a semantic network with 135 concepts (such as "disease or syndrome", "diagnostic procedure", "animal"), and 49 binary predicates (such as "affects", "prevents"). We can represent this as a ternary relation $R : T^1 \times T^1 \times T^2 \to \{0, 1\}$, where T^1 is the set of concepts and T^2 is the set of binary predicates. The result is a 3d cube. We can then apply the IRM to partition the cube into regions of roughly homogeneous response. The system found 14 concept clusters and 21 predicate clusters. Some of these are shown in Figure 30.6. The system learns, for example, that biological functions affect organisms (since $\eta_{a,b,c} \approx 1$ where a represents the biological function cluster, b represents the organism cluster, and c represents the affects cluster).

30.2.3.2 Clustering based on relations and features

We can also use IRM to cluster objects based on their relations and their features. For example, [Kem+06] consider a political dataset (from 1965) consisting of 14 countries, 54 binary predicates representing interaction types between countries (e.g., "sends tourists to", "economic aid"), and 90 features (e.g., "communist", "monarchy"). To create a binary dataset, real-valued features were thresholded at their mean, and categorical variables were dummy-encoded. The data has 3 types: T^1 represents countries, T^2 represents interactions, and T^3 represents features. We have two relations: $R^1 : T^1 \times T^1 \times T^2 \to \{0, 1\}$, and $R^2 : T^1 \times T^3 \to \{0, 1\}$. (This problem therefore combines aspects

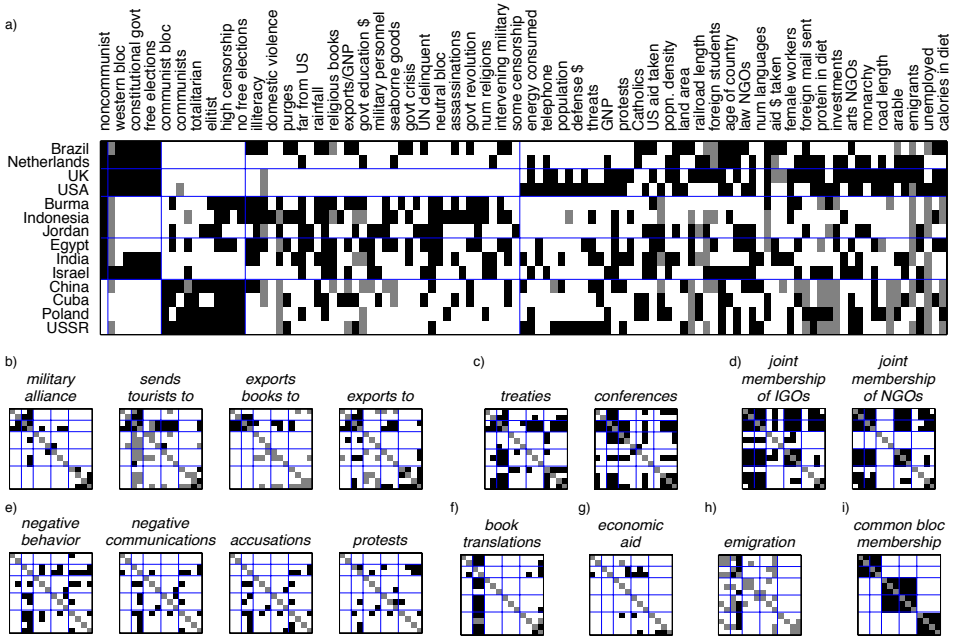


Figure 30.7: Illustration of IRM applied to some political data containing features and pairwise interactions. Top row (a): the partition of the countries into 5 clusters and the features into 5 clusters. Every second column is labelled with the name of the corresponding feature. Small squares at bottom (b-i): these are 8 of the 18 clusters of interaction types. From Figure 6 of [Kem+06]. Used with kind permission of Charles Kemp.

of both the biclustering model and the ontology discovery model.) When given multiple relations, the IRM treats them as conditionally independent. In this case, we have

$$p(\mathbf{R}^1, \mathbf{R}^2 | \boldsymbol{q}^1, \boldsymbol{q}^2, \boldsymbol{q}^3, \boldsymbol{\theta}) = p(\mathbf{R}^1 | \boldsymbol{q}^1, \boldsymbol{q}^2, \boldsymbol{\theta}) p(\mathbf{R}^2 | \boldsymbol{q}^1, \boldsymbol{q}^3, \boldsymbol{\theta}) \tag{30.9}$$

The results are shown in Figure 30.7. The IRM divides the 90 features into 5 clusters, the first of which contains "noncommunist", which captures one of the most important aspects of this Cold-War era dataset. It also clusters the 14 countries into 5 clusters, reflecting natural geo-political groupings (e.g., US and UK, or the Communist Bloc), and the 54 predicates into 18 clusters, reflecting similar relationships (e.g., "negative behavior and "accusations").

30.3 Graphical model structure learning

In this section, we discuss how to learn the structure of a probabilistic graphical model given sample observations of some or all of its nodes. That is, the input is an $N \times D$ data matrix, and the output is a graph G (directed or undirected) with N_G nodes. (Usually $N_G = D$, but we also consider the case where we learn extra latent nodes that are not present in the input.)

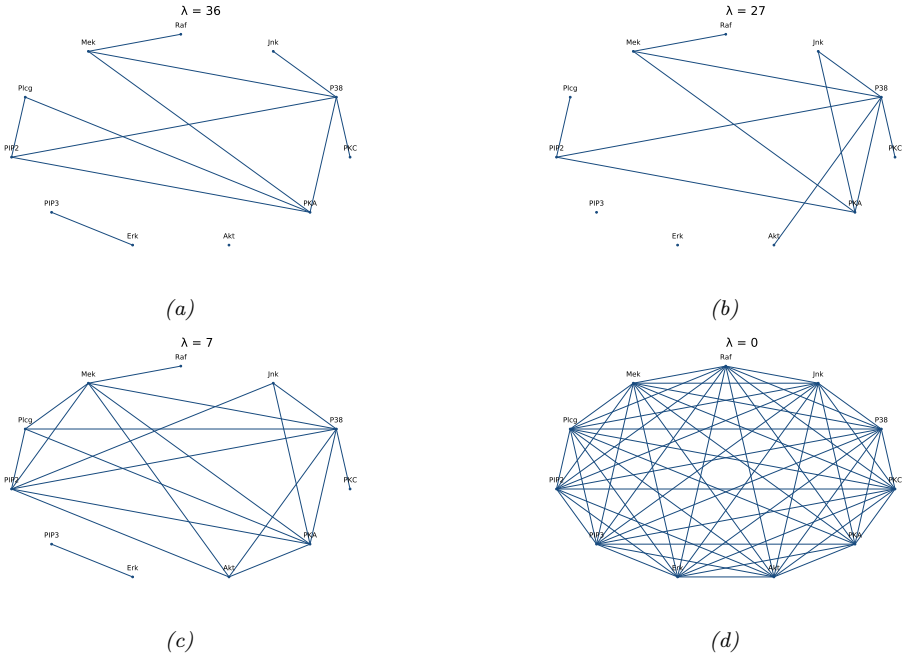


Figure 30.8: A sparse undirected Gaussian graphical model learned using graphical lasso applied to some flow cytometry data (from [Sac+05]), which measures the phosphorylation status of 11 proteins. The sparsity level is controlled by λ. (a) $\lambda = 36$. (b) $\lambda = 27$. (c) $\lambda = 7$. (d) $\lambda = 0$. Adapted from Figure 17.5 of [HTF09]. Generated by ggm_lasso_demo.ipynb.

30.3.1 Applications

There are three main reasons to perform structure learning for PGMs: understanding, prediction, and causal inference (which involves both understanding and prediction), as we summarize below.

Learning sparse PGMs can be useful for gaining an understanding of multiple interacting variables. For example, consider a problem that arises in systems biology: we measure the phosphorylation status of some proteins in a cell [Sac+05] and want to infer how they interact. Figure 30.8 gives an example of a graph structure that was learned from this data, using a method called graphical lasso [FHT08; MH12], which is explained in Supplementary Section 30.3.2. As another example, [Smi+06] showed that one can recover the neural "wiring diagram" of a certain kind of bird from multivariate time-series EEG data. The recovered structure closely matched the known functional connectivity of this part of the bird brain.

In some cases, we are not interested in interpreting the graph structure, we just want to use it to make predictions. One example of this is in financial portfolio management, where accurate models of the covariance between large numbers of different stocks is important. [CW07] show that by learning a sparse graph, and then using this as the basis of a trading strategy, it is possible to outperform (i.e., make more money than) methods that do not exploit sparse graphs. Another example is predicting traffic jams on the freeway. [Hor+05] describe a deployed system called JamBayes for predicting

traffic flow in the Seattle area, using a directed graphical model whose structure was learned from data.

Structure learning is also an important pre-requisite for causal inference. In particular, to predict the effects of interventions on a system, or to perform counterfactual reasoning, we need to know the structural causal model (SCM), as we discuss in Section 4.7. An SCM is a kind of directed graphical model where the relationships between nodes are deterministic (functional), except for stochastic root (exogeneous) variables. Consequently one can use techniques for learning DAG structures as a way to learn SCMs, if we make some assumptions about (lack of) confounders. This is called **causal discovery**. See Supplementary Section 30.4 for details.

30.3.2 Methods

There are many different methods for learning PGM graph structures. See Supplementary Chapter 30 for details.

31 Non-parametric Bayesian models

This chapter is written by Vinayak Rao.

31.1 Introduction

The defining characteristic of a **parametric model** is that the objects being modeled, whether regression or classification functions, probability densities, or something more modern like graphs or shapes, are indexed by a finite-dimensional parameter vector. For instance, neural networks have a fixed number of parameters, independent of the dataset. In a **parametric Bayesian model**, a prior probability distribution on these parameters is used to define a prior distribution on the objects of interest. By contrast, in a **Bayesian nonparametric** (**BNP**) model (also called a **non-parametric Bayesian** model) we directly place prior distributions on the objects of interest, such as functions, graphs, probability distributions, etc. This is usually done via some kind of **stochastic process**, which is a probability distribution over a potentially infinite set of random variables.

One example is a **Gaussian process**. As explained in Chapter 18, this defines a probability distribution over an unknown function $f : \mathcal{X} \to \mathbb{R}$, such that the joint distribution of $f(\mathbf{X}) = (f(\boldsymbol{x}_1), \ldots, f(\boldsymbol{x}_N))$ is jointly Gaussian for any finite set of values $\mathbf{X} = \{\boldsymbol{x}_n \in \mathcal{X}\}_{n=1}^N$ i.e., $p(f(\mathbf{X})) = \mathcal{N}(f(\mathbf{X})|\boldsymbol{\mu}(\mathbf{X}), \mathbf{K}(\mathbf{X}))$ where $\boldsymbol{\mu}(\mathbf{X}) = [\mu(\boldsymbol{x}_1), \ldots, \mu(\boldsymbol{x}_N)]$ is the mean, $\mathbf{K}(\mathbf{X}) = [\mathcal{K}(\boldsymbol{x}_i), \mathcal{K}(\boldsymbol{x}_j)]$ is the $N \times N$ Gram matrix, and $\mathcal{K}$ is a positive definite kernel function. The complexity of the posterior over functions can grow with the amount of data, avoiding underfitting, since we maintain a full posterior distribution over the infinite set of unknown "parameters" (i.e., function evaluations at all points $\boldsymbol{x} \in \mathcal{X}$). But by taking a Bayesian approach, we avoid overfitting this infinitely flexible model. Despite involving infinite-parameter objects, practitioners are often only interested in inferences on a finite training dataset and predictions on a finite test dataset. This often allows these models to be surprisingly tractable. We can also define probability distributions over probability distributions, as well as other kinds of objects, as we discuss in the sections below. For more details, see e.g., [Hjo+10; GV17].

31.2 Dirichlet processes

A **Dirichlet process** (DP) is a nonparametric probability distribution over probability distributions, and is useful as a flexible prior for unsupervised learning tasks like clustering and density modeling [Fer73]. We give more details in the sections below.

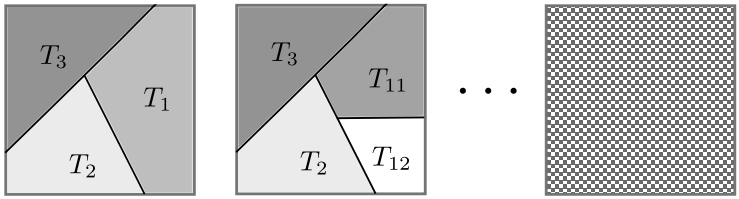


Figure 31.1: Partitions of the unit square. (left) One possible partition into $K = 3$ regions, and (center) A refined partition into $K = 4$ regions. In both figures, the shading of cell T_k is proportional $G(T_k)$, resulting from the same realization of a Dirichlet process. (right) An 'infinite partition' of the unit square. The Dirichlet process can informally be viewed as an infinite-dimensional Dirichlet distribution defined on this.

31.2.1 Definition of a DP

Let G be a probability distribution or a probability measure (we will use the latter terminology in this chapter) on some space Θ. Recall that a probability measure is a function that assigns values to subsets $T \subseteq \Theta$ satisfying the usual axioms of probability: $0 \leq G(T) \leq 1$, $G(\Theta) = \int_\Theta G(\theta)d\theta = 1$, and for disjoint subsets $T_1, \ldots, T_K$ of Θ, $G(T_1 \cup \ldots \cup T_K) = \sum_{k=1}^K G(T_k)$. Bayesian unsupervised learning now seeks to place a prior on the probability measure G.

We have already seen examples of *parametric* priors over probability measures. As a simple example, consider a Gaussian distribution $\mathcal{N}(\theta|\mu, \sigma^2)$: this is a probability measure on Θ, and by placing priors on the parameters μ and σ^2, we have a **parametric prior** on probability measures. Mixture models form more flexible priors, allowing multimodality and asymmetry, and are parametrized by the probabilities of the mixture components, as well as their parameters. DPs directly define a probability on probability measures G.

A Dirichlet Process is specified by a positive real number α, called the **concentration parameter**, and a probability measure H, called the **base measure**. We write a random measure drawn from a DP as $G \sim \mathrm{DP}(\alpha, H)$. H is typically a standard probability measure on Θ, and forms the mean of the Dirichlet process. That is, if $G \sim \mathrm{DP}(\alpha, H)$, then for any subset T of Θ, $\mathbb{E}[G(T)] = H(T)$. The parameter α measures how concentrated the Dirichlet process is around H, with $\mathbb{V}[G(T)] = \frac{H(T)(1-H(T))}{1+\alpha}$. If Θ is $\mathbb{R}^2$, then setting H to the bivariate normal $\mathcal{N}(0, I_2)$ and α to a large value implies a prior belief that G sampled from $\mathrm{DP}(\alpha, H)$ is close to the normal, whereas a small α represents a relatively uninformative prior.

We now define the Dirichlet process more precisely. Let $(T_1, \ldots, T_K)$ be a finite partition of Θ, that is, $(T_1, \ldots, T_K)$ are disjoint sets whose union is Θ. For a probability measure G, let $(G(T_1), \ldots, G(T_K))$ be the vector of probabilities of the elements of this partition. Then $\mathrm{DP}(\alpha, H)$ is a prior over probability measures G satisfying the following requirement: for any finite partition, the associated vector of probabilities has the following joint Dirichlet distribution:

$$(G(T_1), \ldots, G(T_K)) \sim \mathrm{Dir}(\alpha H(T_1), \ldots, \alpha H(T_K)). \tag{31.1}$$

Just like the Gaussian process, the DP is defined implicitly though a set of finite-dimensional distributions, in this case through the distribution of G projected onto any finite partition. The finite-dimensional distributions are **consistent** in the following sense: if T_{11} and T_{12} form a partition

of T_1, then one can sample $G(T_1)$ in two ways: directly, by sampling

$$(G(T_1), \ldots, G(T_K)) \sim \text{Dir}(\alpha H(T_1), \ldots, \alpha H(T_K)) \tag{31.2}$$

or, indirectly, by sampling

$$(G(T_{11}), G(T_{12}), \ldots, G(T_K)) \sim \text{Dir}(\alpha H(T_{11}), \alpha H(T_{12}), \ldots, \alpha H(T_K)) \tag{31.3}$$

and then setting $G(T_1) = G(T_{11}) + G(T_{12})$. From the properties of the Dirichlet distribution, $G(T_1)$ sampled either way follows the same distribution. This consistency property implies, via Kolmogorov's extension theorem [Kal06], that underlying all finite-dimensional probability vectors for different partitions is a single infinite-dimensional vector that we could informally write as

$$G(d\theta_1), \ldots, G(\theta_\infty) \sim \text{Dir}(\alpha H(d\theta_1), \ldots, \alpha H(d\theta_\infty)). \tag{31.4}$$

Very roughly, this 'infinite-dimensional Dirichlet distribution' is the Dirichlet Process. Figure 31.1 sketches this out.

Why is the Dirichlet process, defined in this indirect fashion, useful to practitioners? The answer has to do with conjugacy properties that it inherits from the Dirichlet distribution. One of the simplest unsupervised learning problems seeks to learn an unknown probability distribution G from i.i.d. samples $\{\overline{\theta}_1, \ldots, \overline{\theta}_N\}$ drawn from it. Consider placing a DP prior on the unknown G. Then given the data, one is interested in the posterior distribution over G, representing the updated probability distribution over G. For a partition $(T_1, \ldots, T_K)$ of Θ, an observation falls into the cell z following a multinoulli distribution:

$$z \sim \text{Cat}(G(T_1), \ldots, G(T_K)). \tag{31.5}$$

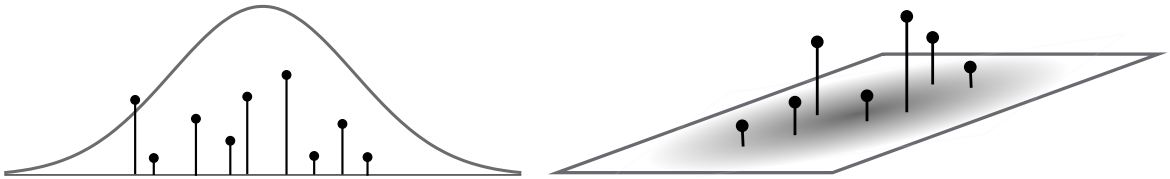
Under a DP prior on G, $(G(T_1), \ldots, G(T_K))$ follows a Dirichlet distribution (equation (31.1)). From the Dirichlet-multinomial conjugacy, the posterior for $(G(T_1), \ldots, G(T_K))$ given the observations is

$$(G(T_1), \ldots, G(T_K))|\{\overline{\theta}_1, \ldots, \overline{\theta}_N\} \sim \text{Dir}(G(T_1) + \sum_{i=1}^{N} \mathbb{I}\left(\overline{\theta}_i \in T_1\right), \ldots, G(T_K) + \sum_{i=1}^{N} \mathbb{I}\left(\overline{\theta}_i \in T_K\right)) \tag{31.6}$$

This is true for any finite partition, so that following our earlier definition, the posterior over G itself is a Dirichlet process, and it is easy to see that:

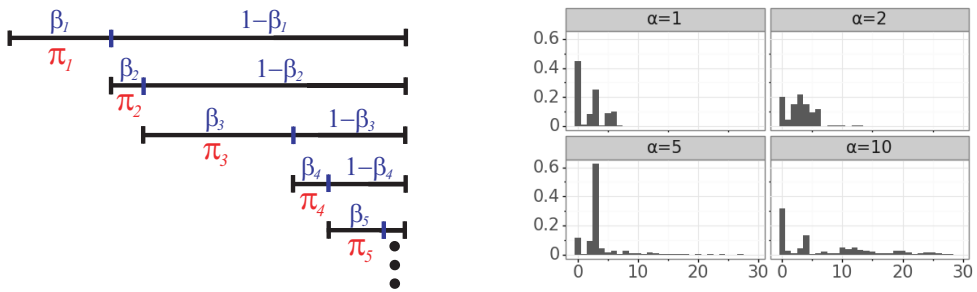
$$G|\overline{\theta}_1, \ldots, \overline{\theta}_N, \alpha, H \sim \text{DP}\left(\alpha + N, \frac{1}{\alpha + N}\left(\alpha H + \sum_{i=1}^{N} \delta_{\theta_i}\right)\right). \tag{31.7}$$

Thus we see that the DP prior on G is a conjugate prior for i.i.d. observations from G, with the posterior distribution over G also a Dirichlet process with concentration parameter $\alpha + N$, and base measure a convex combination of the original base measure H and the empirical distribution of the observations. Note that as N increases, the influence of the original base measure H starts to wane, and the posterior base measure becomes closer and closer to the empirical distribution of the observations. At the same time, the concentration parameter increases, suggesting that the posterior distribution concentrates around the empirical distribution.



(a) (b)

Figure 31.2: Realizations from a Dirichlet process when Θ is (a) the real line, and (b) the unit square. Also shown are the base measures αH. In reality, the number of atoms is infinite for both cases.



(a) (b)

Figure 31.3: Illustration of the stick breaking construction. (a) We have a unit length stick, which we break at a random point β_1; the length of the piece we keep is called π_1; we then recursively break off pieces of the remaining stick, to generate $\pi_2, \pi_3, \ldots$. From Figure 2.22 of [Sud06]. Used with kind permission of Erik Sudderth. (b) Samples of π_k from this process for different values of α. Generated by stick_breaking_demo.ipynb.

31.2.2 Stick breaking construction of the DP

Our discussion so far has been very abstract, with no indication of how to either sample the random measure G or how to sample observations from G. We address the first question, giving a constructive definition for the DP known as the **stick-breaking construction** [Set94].

We first mention that probability measures G sampled from a DP are **discrete with probability one** (see Figure 31.2), taking the form

$$G(\theta) = \sum_{k=1}^{\infty} \pi_k \delta_{\theta_k}(\theta). \tag{31.8}$$

Thus G consists of an infinite number of **atoms**, the kth atom located at θ_k, and having weight π_k. Informally, this follows from Equation (31.4), which represents the DP as an infinite-dimensional but infinitely-sparse Dirichlet distribution (recall that as its parameters become smaller, a Dirichlet distribution concentrates on sparse distributions that are dominated by a few components).

For a DP, the locations θ_k of the atoms are drawn independently from the base measure H, whereas the concentration parameter α controls the distribution of the weights π_k. Observe that the infinite sequence of weights $(\pi_1, \pi_2, \ldots)$ must add up to one, since G is a probability measure. The weights

can be simulated by the following process sketched in Figure 31.3, and known as the **stick-breaking process**. Start with a stick of length 1 representing the total probability mass, and sequentially break off a random Beta$(1, \alpha)$ distributed fraction of the remaining stick. The kth break forms π_k. In equations, for $k = 1, 2, \ldots,$

$$\beta_k \sim \text{Beta}(1, \alpha), \qquad \theta_k \sim H, \tag{31.9}$$

$$\pi_k = \beta_k \prod_{l=1}^{k-1}(1 - \beta_l) = \beta_k(1 - \sum_{l=1}^{k-1} \pi_l) \tag{31.10}$$

Then, setting $G(\theta) = \sum_{k=1}^{\infty} \pi_k \delta_{\theta_k}(\theta)$, one can show that $G \sim \text{DP}(\alpha, H)$. The distribution over the weights is often denoted by

$$\pi \sim \text{GEM}(\alpha), \tag{31.11}$$

where **GEM** stands for Griffiths, Engen and McCloskey (this term is due to [Ewe90]). Some samples from this process are shown in Figure 31.3.

We note that since the number of atoms in infinite, one cannot exactly simulate from a DP in finite time. However, the sequence of weights from the GEM distribution are **stochastically ordered**, having decreasing averages, and the truncation error resulting from terminating after a finite number of steps quickly becoming neglible [IJ01]. Nevertheless, we will see in the next section that it is possible to simulate samples and make predictions from a DP-distributed probability measure G without any truncation error. This exploits the conjugacy propery of the DP.

31.2.3 The Chinese restaurant process (CRP)

Consider a single observation $\overline{\theta}_1$ from a DP-distributed probability measure G. The probability that $\overline{\theta}_1$ lies within a set $T \subseteq \Theta$, marginalizing out the random G, is $\mathbb{E}[G(T)] = H(T)$, the equality following from the definition of the DP. This holds for arbitrary T, which implies that the first observation $\overline{\theta}_1$ is distributed as the base measure of the DP:

$$p(\overline{\theta}_1 = \theta | \alpha, H) = H(\theta). \tag{31.12}$$

Given N observations $\overline{\theta}_1, \ldots, \overline{\theta}_N$, the updated distribution over G is still a DP, but now modified as in Equation (31.7). Repeating the same argument, it follows that the $(N+1)$st observation is distributed as the base measure of the posterior DP, given by

$$p(\overline{\theta}_{N+1} = \theta | \overline{\theta}_{1:N}, \alpha, H) = \frac{1}{\alpha + N}\left(\alpha H(\theta) + \sum_{k=1}^{K} N_k \delta_{\overline{\theta}_k}(\theta)\right) \tag{31.13}$$

where N_k is the number of observations equal to $\overline{\theta}_k$. The previous two equations form the basis of what is called the **Pólya urn** or **Blackwell-MacQueen** sampling scheme [BM+73]. This provides a way to exactly produce samples from a DP-distributed random probability measure.

It is often more convenient to work with discrete variables $(z_1, \ldots, z_N)$, with z_i specifying which value of θ_k the ith sample takes. In particular, for the ith observation, $\overline{\theta}_i = \theta_{z_i}$. This allows us to decouple the cluster or partition structure of the dataset (controlled by α) and the cluster parameters

(controlled by H). Let us assign the first observation to cluster 1, i.e. $z_1 = 1$. The second observation can either belong to the same cluster as observation 1, or belong to a new cluster, which we call cluster 2. In the former event, $z_2 = 1$, after which z_3 can equal 1 or 2. In the latter event, $z_2 = 2$, and z_3 can equal $1, 2$ or 3. Based on the Equation (31.13), we have

$$p(z_{N+1} = z|\boldsymbol{z}_{1:N}, \alpha) = \frac{1}{\alpha + N}\left(\alpha \mathbb{I}\left(z = K + 1\right) + \sum_{k=1}^{K} N_k \mathbb{I}\left(z = k\right)\right), \tag{31.14}$$

assuming the first N observations have been assigned to K clusters. This is called the **Chinese restaurant process** or **CRP**, based on the following analogy: observations are customers in a restaurant with an infinite number of tables, each corresponding to a different cluster. Each table has a dish, corresponding to the parameter θ of that cluster. When a customer enters the restaurant, they may choose to join an existing table with probability proportional to the number of people already sitting at this table (i.e. they join table k with probability proportional to N_k); otherwise, with probability proportional to α, they choose to sit at a new table, ordering a new dish by sampling from the base measure H.

The sequence $Z = (z_1, \ldots, z_N)$ of cluster assignments is **partition of the integers** 1 to N, and the CRP is a distribution over such partitions. The probability of creating a new table diminishes as the number of observations increases, but is always non-zero, and one can show that the number of occupied tables K approaches $\alpha \log(N)$ as $N \to \infty$ almost surely. The fact that currently occupied tables are more likely to get new customers is sometimes called a **rich get richer** phenomenon. It is important to recognize that despite being defined as a sequential process, the CRP is an **exchangeable process**, with partition probabilities that are independent of the observation indices. Indeed, it is easy to show that the probability of a partition of N integers into K clusters with sizes $N_1, \ldots, N_K$ is

$$p(N_1, \ldots, N_k) = \frac{\alpha^{K-1}}{[\alpha + 1]_1^N} \prod_{k=1}^{K} N_k! \tag{31.15}$$

Here, $[\alpha + 1]_1^N = \prod_{i=0}^{N-1}(\alpha + 1 + i)$ is the rising factorial. Equation (31.15) depends only on the cluster sizes, and is called the Ewens sampling formula [Ewe72]. Exchangeability implies that the probability that the first two customers sit at the same table is the same as the probability that the first and last sit at the same table. Similarly all customers have the same probability of ending up in a cluster of size S. The fact that the first customer can only belong to cluster 1 (i.e. that $z_1 = 1$) does not contradict exchangeability and reflects the fact that the cluster indices are chosen arbitrarily. This disappears if we index clusters by their associated parameter θ_k.

31.3 Dirichlet process mixture models

Real-world datasets are often best modeled by continuous probability densities. By contrast, a sample G from a DP is discrete with probability one, and sampling observations from G will result in repeated values, making it inappropriate for many applications. However, the discrete structure of G is useful in clustering applications, as a prior for the latent variables underlying the observed datapoints. In particular, z_i and $\overline{\boldsymbol{\theta}}_i$ can represent the cluster assignment and cluster parameter of

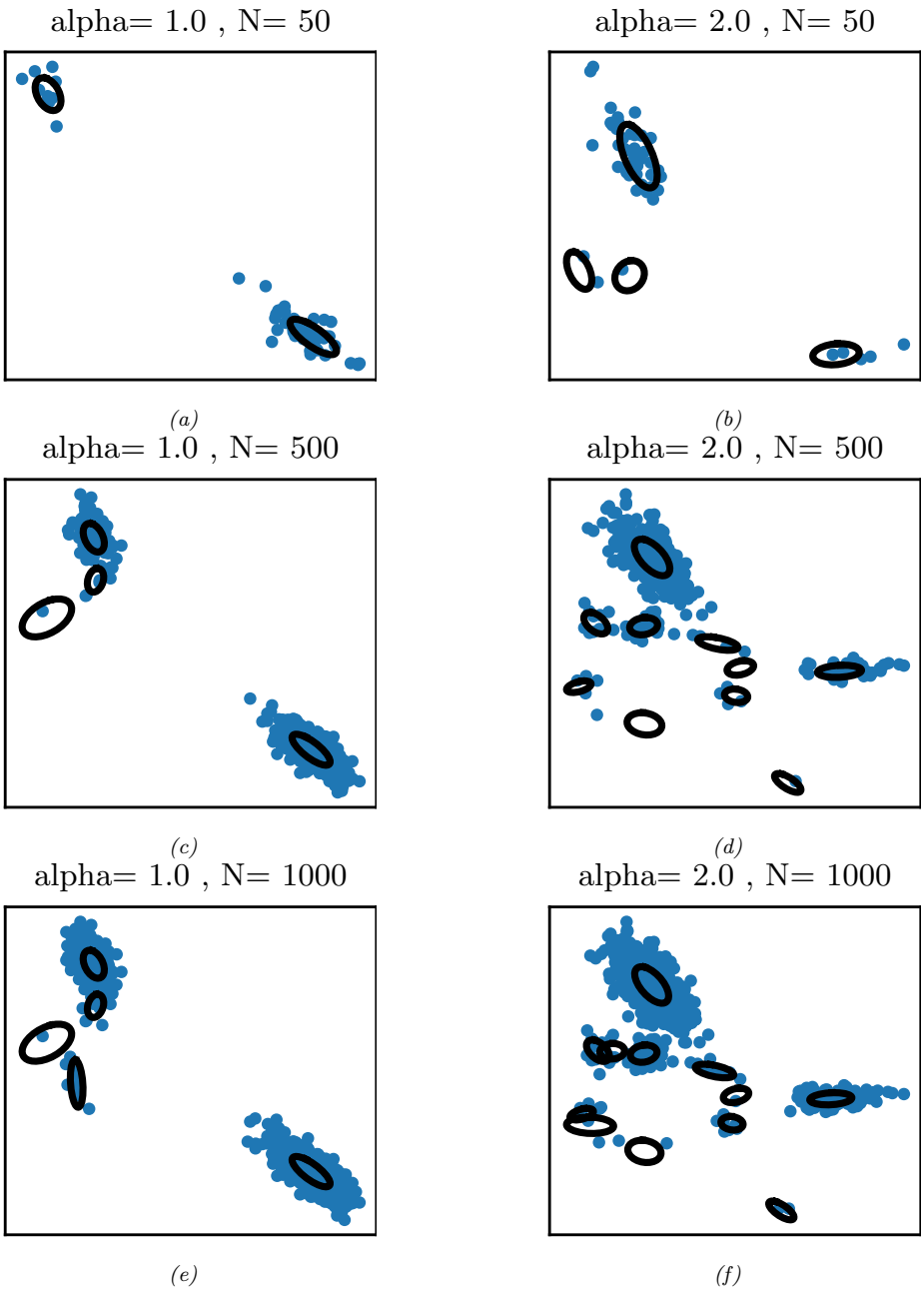


Figure 31.4: Some samples from a Dirichlet process mixture model of 2D Gaussians, with concentration parameter $\alpha = 1$ (left column) and $\alpha = 2$ (right column). From top to bottom, we show $N = 50$, $N = 500$ and $N = 1000$ samples. Generated by dp_mixgauss_sample.ipynb.

the ith datapoint, whose value $\boldsymbol{x}_i$ is a draw from some parametric distribution $F(\boldsymbol{x}|\boldsymbol{\theta})$ indexed by $\boldsymbol{\theta}$, with base measure H. The resulting model follows along the lines of a standard mixture model, but now is an **infinite mixture model**, consisting of an infinite number of components or clusters, one for each atom in G.

A very common setting when $\boldsymbol{x}_i \in \mathbb{R}^d$ is to set F to be the multivariate normal distribution, $\boldsymbol{\theta} = (\boldsymbol{\mu}, \boldsymbol{\Sigma})$, and H to be the Normal-Inverse-Wishart distribution. Then, each of the infinite clusters has an associated mean and covariance matrix, and to generate a new observation, one picks cluster k with probability π_k, and simulates from a normal with mean $\boldsymbol{\mu}_k$ and covariance $\boldsymbol{\Sigma}_k$. See Figure 31.4 for some samples from this model.

We discuss DP mixture models (DPMM) in more detail below.

31.3.1 Model definition

We define the DPMM model as follows:

$$\boldsymbol{\pi} \sim GEM(\alpha), \quad \boldsymbol{\theta}_k \sim H, \qquad k = 1, 2, \ldots \tag{31.16}$$

$$z_i \sim \boldsymbol{\pi}, \qquad \boldsymbol{x}_i \sim F(\boldsymbol{\theta}_{z_i}), \quad i = 1, \ldots, N. \tag{31.17}$$

Equivalently, we can write this as

$$G \sim \mathrm{DP}(\alpha, H) \tag{31.18}$$

$$\overline{\boldsymbol{\theta}}_i \sim G, \quad \boldsymbol{x}_i \sim F(\overline{\boldsymbol{\theta}}_i), \quad i = 1, \ldots, N. \tag{31.19}$$

G and F together define the infinite mixture model: $G_F(\boldsymbol{x}) \sim \sum_{k=1}^{\infty} \pi_k F(\boldsymbol{x}|\boldsymbol{\theta}_k)$. If $F(\boldsymbol{x}|\boldsymbol{\theta})$ is continuous, then so is $G_F(\boldsymbol{x})$, and the Dirichlet process mixture model serves as a nonparametric prior over continuous distributions or probability densities.

Figure 31.5 illustrates two graphical models that summarize this, corresponding to the two sets of equations above. The first generates the set of weights $(\pi_1, \pi_2, \ldots)$ from the GEM distribution, along with an infinite collection of cluster parameters $(\boldsymbol{\theta}_1, \boldsymbol{\theta}_2, \ldots)$. It then generates observations by first sampling a cluster indicator z_i from $\boldsymbol{\pi}$, indexing the associated cluster parameter $\boldsymbol{\theta}_{z_i}$ and then simulating the observation $\boldsymbol{x}_i$ from $F(\boldsymbol{\theta}_{z_i})$. The second graphical model simulates a random measure G from the DP. It generates observations by directly simulating a parameter $\overline{\boldsymbol{\theta}}_i$ from G, and then simulating $\boldsymbol{x}_i$ from $F(\overline{\boldsymbol{\theta}}_i)$. The infinite mixture model can be viewed as the limit of a K-component finite mixture model with a $\mathrm{Dir}(\alpha/K, \ldots, \alpha/K)$ prior on the mixture weights $(\pi_1, \ldots, \pi_K)$ and with mixture parameters $\boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_K$, as $K \to \infty$ [Ras00; Nea00]. Producing exact samples $(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N)$ from this model involves one additional step to the Chinese restaurant process: after selecting a table (cluster) z_i with its associate dish (parameter) $\boldsymbol{\theta}_{z_i}$, the ith customer now samples an observation from the distribution $F(\boldsymbol{\theta}_{z_i})$.

31.3.2 Fitting using collapsed Gibbs sampling

Given a dataset of observations, one is interested in the posterior distribution $p(G, z_1, \ldots, z_N|\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N, \alpha, H)$, or equivalently, $p(\boldsymbol{\pi}, \boldsymbol{\theta}_1, \boldsymbol{\theta}_2, \ldots, z_1, \ldots, z_N|\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N, \alpha, H)$. The most common way to fit a DPMM is via Markov chain Monte Carlo (MCMC), producing samples by constructing a Markov chain that targets this posterior distribution. We describe a **collapsed Gibbs sampler** based on the Chinese restaurant process that marginalizes out the infinite-dimensional random measure G, and

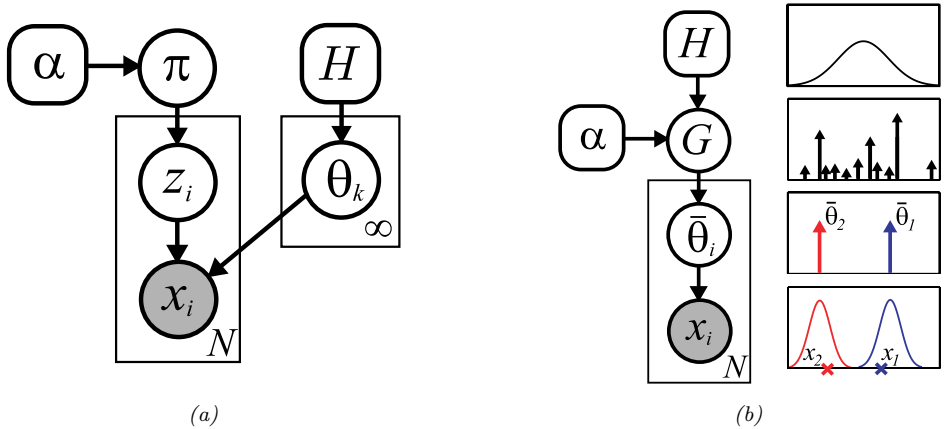


Figure 31.5: Two views of N observations sampled from a DP mixture model. Left: representation where cluster indicators are sampled from the GEM-distributed distribution π. Right: representation where parameters are samples from the DP-distributed random measure G. The rightmost picture illustrates the case where $N = 2$, $\boldsymbol{\theta}$ is real-valued with a Gaussian prior $H(\cdot)$, and $F(x|\theta)$ is a Gaussian with mean θ and variance σ^2. We generate two parameters, $\bar{\theta}_1$ and $\bar{\theta}_2$, from G, one per data point. Finally, we generate two data points, x_1 and x_2, from $\mathcal{N}(\bar{\theta}_1, \sigma^2)$ and $\mathcal{N}(\bar{\theta}_2, \sigma^2)$. From Figure 2.24 of [Sud06]. Used with kind permission of Erik Sudderth.

that targets the distribution $p(z_1, \ldots, z_N|\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N, \alpha, H)$ summarizing all clustering information. It produces samples from this distribution by cycling through each observation $\boldsymbol{x}_i$, and updating its assigned cluster z_i, conditioned on all other variables. Write $\boldsymbol{x}_{-i}$ for all observations other than the ith observation, and $\boldsymbol{z}_{-i}$ for their cluster assignments. Then we have

$$p(z_i = k|\boldsymbol{z}_{-i}, \boldsymbol{x}, \alpha, H) \propto p(z_i = k|\boldsymbol{z}_{-i}, \alpha) p(\boldsymbol{x}_i|\boldsymbol{x}_{-i}, z_i = k, \boldsymbol{z}_{-i}, H) \tag{31.20}$$

By exchangeability, each observation can be treated as the last customer to enter the restaurant. Hence the first term is given by

$$p(z_i|\boldsymbol{z}_{-i}, \alpha) = \frac{1}{\alpha + N - 1} \left(\alpha \mathbb{I}\left(z_i = K_{-i} + 1\right) + \sum_{k=1}^{K_{-i}} N_{k,-i} \mathbb{I}\left(z_i = k\right) \right) \tag{31.21}$$

where $N_{k,-i}$ is the number of observations in cluster k, and K_{-i} is the number of clusters used by $\boldsymbol{x}_{-i}$, both obtained after removing observation i, eliminating empty clusters, and renumbering clusters.

To compute the second term, $p(\boldsymbol{x}_i|\boldsymbol{x}_{-i}, z_i = k, \boldsymbol{z}_{-i}, H)$, let us partition the data $\boldsymbol{x}_{-i}$ into clusters based on $\boldsymbol{z}_{-i}$. Let $\boldsymbol{x}_{-i,c} = \{\boldsymbol{x}_j : z_j = c, j \neq i\}$ be the data points assigned to cluster c. If $z_i = k$, then x_i is conditionally independent of all the data points except those assigned to cluster k. Hence,

$$p(\boldsymbol{x}_i|\boldsymbol{x}_{-i}, \boldsymbol{z}_{-i}, z_i = k) = p(\boldsymbol{x}_i|\boldsymbol{x}_{-i,k}) = \frac{p(\boldsymbol{x}_i, \boldsymbol{x}_{-i,k})}{p(\boldsymbol{x}_{-i,k})}, \tag{31.22}$$

where

$$p(\boldsymbol{x}_i, \boldsymbol{x}_{-i,k}) = \int p(\boldsymbol{x}_i|\boldsymbol{\theta}_k) \left[\prod_{j \neq i: z_j = k} p(\boldsymbol{x}_j|\boldsymbol{\theta}_k) \right] H(\boldsymbol{\theta}_k) d\boldsymbol{\theta}_k \tag{31.23}$$

is the marginal likelihood of all the data assigned to cluster k, including i, and $p(\boldsymbol{x}_{-i,k})$ is an analogous expression excluding i. Thus we see that the term $p(\boldsymbol{x}_i|\boldsymbol{x}_{-i}, \boldsymbol{z}_{-i}, z_i = k)$ is the posterior preditive distribution for cluster k evaluated at $\boldsymbol{x}_i$.

If $z_i = k^*$, corresponding to a new cluster, we have

$$p(\boldsymbol{x}_i|\boldsymbol{x}_{-i}, \boldsymbol{z}_{-i}, z_i = k^*) = p(\boldsymbol{x}_i) = \int p(\boldsymbol{x}_i|\boldsymbol{\theta}) H(\boldsymbol{\theta}) d\boldsymbol{\theta} \tag{31.24}$$

which is just the prior predictive distribution for a new cluster evaluated at $\boldsymbol{x}_i$.

The overall sampler is sometimes called "Algorithm 3" (from [Nea00]). Algorithm 42 provides the pseudocode. The algorithm is very similar to collapsed Gibbs for finite mixtures except that we have to consider the case $z_i = k^*$. Note that in order the evaluate the integrals in Equation (31.23) and Equation (31.24), we require the base measure H to be conjugate to the likelihood F. For example, if we use an NIW prior for the Gaussian likelihood, we can use the results from Section 3.3.3.6 to compute the predictive distributions. Extensions to the case of non-conjugate priors are discussed in [Nea00].

Algorithm 42: Collapsed Gibbs sampler for DP mixture model

foreach $i = 1 : N$ *in random order* **do**
  Remove $\boldsymbol{x}_i$'s sufficient statistics from old cluster z_i
  foreach $k = 1 : K$ **do**
    Compute $p_k(\boldsymbol{x}_i) = p(\boldsymbol{x}_i|\boldsymbol{x}_{-i}(k))$
    Set $N_{k,-i} = \dim(\boldsymbol{x}_{-i}(k))$
    Compute $p(z_i = k|\boldsymbol{z}_{-i}, \mathcal{D}) = \frac{N_{k,-i}}{\alpha + N - 1} p_k(\boldsymbol{x}_i)$
  Compute $p(z_i = *|\boldsymbol{z}_{-i}, \mathcal{D}) = \frac{\alpha}{\alpha + N - 1} p(\boldsymbol{x}_i)$
  Normalize $p(z_i|\cdot)$
  Sample $z_i \sim p(z_i|\cdot)$
  Add $\boldsymbol{x}_i$'s sufficient statistics to new cluster z_i
  If any cluster is empty, remove it and decrease K

31.3.3 Fitting using variational Bayes

This section was written by Xinglong Li.

In this section, we discuss how to fit a DP mixture model using mean field variational Bayes (Section 10.2.3), as described in [BJ+06].

Given samples $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N$ from DP mixture, the mean field variational inference (MFVI) algorithm is based on the stick-breaking representation of the DP mixture. The target of the inference is the joint

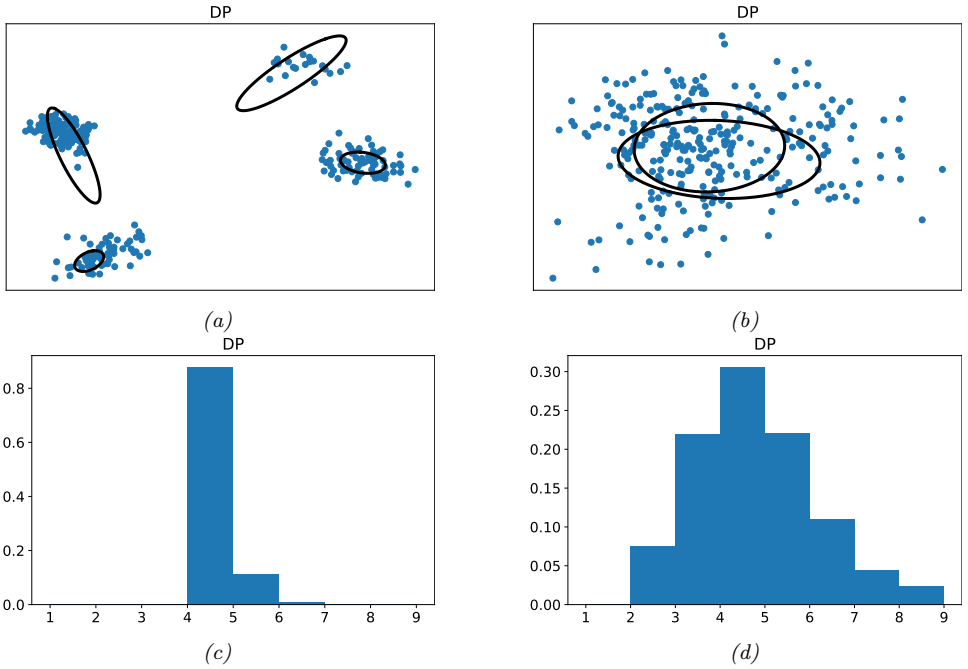


Figure 31.6: Output of the DP mixture model fit using Gibbs sampling to two different datasets. Left column: dataset with 4 clear clusters. Right column: dataset with an unclear number of clusters. Top row: single sample from the Markov chain. Bottom row: empirical fraction of times a given number of cluster is used, computed across all samples from the chain. Generated by dp_mixgauss_sample.ipynb.

posterior distribution of the beta random variables $\boldsymbol{\beta} = \{\beta_1, \beta_2 \ldots\}$ in the stick-breaking construction of DP , the locations $\boldsymbol{\theta} = \{\boldsymbol{\theta}_1, \boldsymbol{\theta}_2, \ldots\}$ of atoms , and the cluster assignments $\boldsymbol{z} = \{z_1, \ldots, z_N\}$:

$$\boldsymbol{w} = \{\boldsymbol{\beta}, \boldsymbol{\theta}, \boldsymbol{z}\} \tag{31.25}$$

The hyperparameters are the concentration parameter of the DP and the parameter of the conjugate base distribution of $\boldsymbol{\theta}$:

$$\boldsymbol{\lambda} = \{\alpha, \boldsymbol{\eta}\} \tag{31.26}$$

The variational inference algorithm minimizes the KL-divergence between $q_\psi(\boldsymbol{w})$ and $p(\boldsymbol{w}|\boldsymbol{x}, \lambda)$:

$$D_{\mathbb{KL}}\left(q_{\boldsymbol{\psi}}(\boldsymbol{w}) \parallel p(\boldsymbol{w}|\boldsymbol{x}, \boldsymbol{\lambda})\right) = \mathbb{E}_q[\log q_{\boldsymbol{\psi}}(\boldsymbol{w})] - \mathbb{E}_q[\log p(\boldsymbol{w}, \boldsymbol{x}|\boldsymbol{\lambda})] + \log p(\boldsymbol{x}|\boldsymbol{\lambda}) \tag{31.27}$$

Minimizing the KL divergence is equivalent to maximizing the evidence lower bound (ELBO):

$$Ł = \mathbb{E}_q[\log p(\boldsymbol{w}, \boldsymbol{x}|\boldsymbol{\lambda})] - \mathbb{E}_q[\log q_{\boldsymbol{\psi}}(\boldsymbol{w})] \tag{31.28}$$

$$= \mathbb{E}_q[\log p(\boldsymbol{\beta}|\alpha)] + \mathbb{E}_q[\log p(\boldsymbol{\theta}|\boldsymbol{\eta})] + \sum_{n=1}^{N} \left(\mathbb{E}_q[\log p(z_n|\boldsymbol{\beta})] + \mathbb{E}_q[\log p(\boldsymbol{x}_n|z_n)]\right) \tag{31.29}$$

$$- \mathbb{E}_q[\log q_{\boldsymbol{\psi}}(\boldsymbol{\beta}, \boldsymbol{\theta}, \boldsymbol{z})] \tag{31.30}$$

To deal with the infinite parameters in $\boldsymbol{\beta}$ and $\boldsymbol{\theta}$, the variational inference algorithm truncates the DP by fixing a value T and setting $q(\beta_T = 1) = 1$, which implies that $\pi_t = 0$ for $t > T$. Therefore, $q_{\boldsymbol{\psi}}(\boldsymbol{w})$ in the MFVI for DP mixture models factorizes into

$$q_{\boldsymbol{\psi}}(\boldsymbol{\beta}, \boldsymbol{\theta}, \boldsymbol{z}) = \prod_{t=1}^{T-1} q_{\boldsymbol{\gamma}_t}(\beta_t) \prod_{t=1}^{T} q_{\boldsymbol{\tau}_t}(\boldsymbol{\theta}_t) \prod_{n=1}^{N} q_{\boldsymbol{\phi}_n}(z_n), \tag{31.31}$$

where $q_{\boldsymbol{\gamma}_t}(\beta_t)$ is the beta distribution with parameters $\{\gamma_{t,1}, \gamma_{t,2}\}$, $q_{\boldsymbol{\tau}_t}(\boldsymbol{\theta}_t)$ is the exponential family distribution with natural parameters $\boldsymbol{\tau}_t$, and $q_{\boldsymbol{\phi}_n}(z_n)$ is the categorical distribution of the cluster assignment of observation $\boldsymbol{x}_n$, with $q(z_n = t) = \phi_{n,t}$. The free variational parameters are

$$\boldsymbol{\psi} = \{\boldsymbol{\gamma}_1, \cdots, \boldsymbol{\gamma}_{T-1}, \boldsymbol{\tau}_1, \cdots, \boldsymbol{\tau}_T, \boldsymbol{\phi}_1, \cdots, \boldsymbol{\phi}_N\}. \tag{31.32}$$

Notice that only $q_{\boldsymbol{\psi}}(\boldsymbol{w})$ is truncated, the true posterior $p(\boldsymbol{w}, \boldsymbol{x}|\boldsymbol{\lambda})$ from the model need not be truncated when minimizing the KL.

The MFVI can be optimized via the coordinate ascent algorithm, and the closed form update in each step exists when the base measure of the DP is conjugate to the likelihood of observations. In particular, suppose that conditional distribution of $\boldsymbol{x}_n$ conditioned on z_n and $\boldsymbol{\theta}$ is an exponential family distribution (Section 2.3):

$$p(\boldsymbol{x}_n|z_n, \boldsymbol{\theta}_1, \boldsymbol{\theta}_2, \ldots) = h(\boldsymbol{x}_n) \exp\{\boldsymbol{\theta}_{z_n}^{\mathsf{T}} \boldsymbol{x}_n - a(\boldsymbol{\theta}_{z_n})\} \tag{31.33}$$

where $\boldsymbol{x}_n$ is the sufficient statistic for the natural parameter $\boldsymbol{\theta}_{z_n}$. Therefore, the conjugate base distribution is

$$p(\boldsymbol{\theta}|\boldsymbol{\eta}) = h(\boldsymbol{\theta}) \exp(\boldsymbol{\eta}_1^{\mathsf{T}} \boldsymbol{\theta} + \eta_2(-a(\boldsymbol{\theta})) - a(\boldsymbol{\eta})), \tag{31.34}$$

where $\boldsymbol{\eta}_1$ contains the first $\dim(\boldsymbol{\theta})$ components and η_2 is a scalar. See Algorithm 43 for the resulting pseudocode.

Extensions of this method to infer the hyperparameters, $\boldsymbol{\lambda} = \{\alpha, \boldsymbol{\eta}\}$, can be found in the appendix of [BJ+06].

31.3.4 Other fitting algorithms

While collapsed Gibbs ssmpling is the simplest approach to posterior inference for DPMMs, a variety of other methods have been proposed as well. One popular class of MCMC samplers works with the stick-breaking representation of the DP instead of CRP, instantiating the random measure G [IJ01]. The sampler then proceeds by sampling the cluster assignments $\boldsymbol{z}$ given G, and then G given $\boldsymbol{z}$. An advantage of this is that the cluster assignments can be updated in parallel, unlike the CRP, where they are updated sequentially. To be feasible however, these methods require truncating G to a finite number of atoms, though the resulting approximation error can be quite small. The posterior approximation error can be eliminated altogether by slice-sampling methods [KGW11], that work with random truncation levels.

Alternatives to MCMC also exist. [Dau07] shows how one can use A* search and beam search to quickly find an approximate MAP estimate. [Man+07] discusses how to fit a DPMM online using particle filtering, which is a like a stochastic version of beam search. This can be more efficient than Gibbs sampling, particularly for large datasets.

Algorithm 43: Variational inference for DP mixture model

Initialize the variational parameters:
ϕ_{nt} is membership probability of $\boldsymbol{x}_n$ in cluster t;
$\boldsymbol{\tau}_t$ are the natural parameters for cluster t;
$\boldsymbol{\gamma}_t$ are the parameters for the stick breaking distribution.
while *not converged* **do**
  foreach $\boldsymbol{\gamma}_t$ **do**
    Update the beta distribution $q_{\boldsymbol{\gamma}_t}(\beta_t)$;
    $\gamma_{t,1} = 1 + \sum_n \phi_{n,t}$
    $\gamma_{t,2} = \alpha + \sum_n \sum_{j=t+1}^{T} \phi_{n,j}$
  foreach $\boldsymbol{\tau}_t$ **do**
    Update the exponential family distribution $q_{\boldsymbol{\tau}_t}(\boldsymbol{\theta}_t)$ given sufficient statistics $\{\boldsymbol{x}_n\}$;
    $\boldsymbol{\tau}_{t,1} = \boldsymbol{\eta}_1 + \sum_n \phi_{n,t} \boldsymbol{x}_n$
    $\tau_{t,2} = \eta_2 + \sum_n \phi_{n,t}$
  foreach $\phi_{n,t}$ **do**
    Update the categorical distribution $q_{\boldsymbol{\phi}_n}(z_n)$ for each observation;
    $\phi_{n,t} \propto \exp(S_t)$
    $S_t = \mathbb{E}_q[\log \beta_t] + \sum_{i=1}^{t-1} \mathbb{E}_q[\log(1-\beta_i)] + \mathbb{E}_q[\boldsymbol{\theta}_t]^{\mathsf{T}} \boldsymbol{x}_n - \mathbb{E}_q[a(\boldsymbol{\theta}_t)]$

In Section 31.3.3 we discussed an approach based on mean field variational inference. A variety of other variational approximation methods have been proposed as well, for example [KWV06; TKW08; Zob09; WB12].

31.3.5 Choosing the hyper-parameters

An important issue is how to set the model hyper-parameters. These include the DP concentration parameter α, as well as any parameters $\boldsymbol{\lambda}$ of the base measure H. For the DP, the value of α does not have much impact on predictive accuracy, but has quite a strong affect the number of clusters. One approach is to put a Gamma prior for α, and then form its posterior, $p(\alpha|K, N, a, b)$ [EW95]. Simulating α given the cluster assignments $\boldsymbol{z}$ is quite straightforward, and can be incorporated into the earlier Gibbs sampler. The same is the case with the hyper-parameters $\boldsymbol{\lambda}$ [Ras00]. Alternatively, one can use empirical Bayes [MBJ06] to fit rather than sample these parameters.

31.4 Generalizations of the Dirichlet process

Dirichlet process mixture models are flexible nonparametric models of continuous probability densities, and if set up with a little care, can possess important frequentist properties like **asymptotic consistency**: with more and more observations, the posterior distribution concentrates around the 'true' data generating distribution, with very little assumed about the this distribution. Nevertheless, DPs still represent very strong prior information, especially in clustering applications. We saw that the number of clusters in a dataset of size N a priori is around $\alpha \log N$. As indicated by Equation (31.15),

not just the number of clusters, but also the distribution of their sizes is controlled by a single parameter α. The resulting clustering model is thus quite inflexible, and in many cases, inappropriate. Two examples from machine learning that highlight its limitations are applications involving text and image data. Here, it has been observed empirically that the number of unique words in a document, the frequency of word usage, the number of objects in an image, or the number of pixels in an object follow power-law distributions. Clusterings sampled from the CRP do not produce this property, and the resulting model mismatch can result in poor predictive performance.

31.4.1 Pitman-Yor process

A popular generalization of the Dirichlet process is the Pitman-Yor process [PY97] (also called the two-parameter Poisson-Dirichlet process). Written at $\text{PYP}(\alpha, d, H)$, the Pitman-Yor process includes an additional **discount parameter** d which must be greater than 0. The concentration parameter can now be negative, but must satisfy $\alpha \geq -d$. As with the DP, a sample G from a PYP is a random probability measure that is discrete almost surely. It has a stick-breaking representation that generalizes that of the DP: again, we start with a stick of length 1, but now at step k, a random $\text{Beta}(1-d, \alpha+kd)$ fraction of the remaining probability mass is broken off, so that G is written as

$$G = \sum_{k=1}^{\infty} \pi_k \delta_{\theta_k}, \tag{31.35}$$

$$\beta_k \sim \text{Beta}(1-d, \alpha+kd), \qquad \theta_k \sim H, \tag{31.36}$$

$$\pi_k = \beta_k \prod_{l=1}^{k-1}(1-\beta_l) = \beta_k(1-\sum_{l=1}^{k-1}\pi_l). \tag{31.37}$$

Because G is discrete, once again observations $\overline{\theta}_1, \ldots, \overline{\theta}_d$ sampled i.i.d. from G will possess a clustering structure. These can directly be sampled following a sequential process that generalizes the CRP. Now, when a new customer enters the restaurant, they join an existing table with N_k customers with probability proportional to $(N_k - d)$, and create a new table with probability proportional to $\alpha + Kd$, where K is the number of clusters.

Observe that the Dirichlet Process is a special instance of the Pitman-Yor process, corresponding to d equal to 0. Non-zero settings of d counteract the rich-get-richer dynamics of the DP to some extent, increasing the probability of creating new clusters. The more clusters there are present in a dataset, the higher the probability of creating a new cluster (relative to the Dirichlet process). This behavior means that a large number of clusters, as well as a few very large clusters are more likely under the PYP than the DP.

An even more general class of probability measures are the so-called Gibbs-type priors [DB+13]. Under such a prior, given N observations, the probability these are clustered into K clusters, the kth having n_k observations, is

$$p(n_1, \ldots, n_k) = V_{N,K} \prod_{k=1}^{K} (\sigma - 1)_{N_k - 1}, \tag{31.38}$$

where $(\sigma)_n = \sigma(\sigma+1)\ldots(\sigma+n-1)$, and for non-negative weights $V_{N,K}$ satisfying $V_{N,K} = (N - \sigma K)V_{N+1,K} + V_{N+1,K+1}$ and $V_{1,1} = 1$. This definition ensures that the probability over

partitions is consistent, exchangeable and tractable, and includes the DP and PYP as special cases. Besides these two, Gibbs-type priors include settings where the number of components (or the number of atoms in the random measures) are random but bounded. Recall that DP and PYP mixture models are infinite mixture models, with the number of components growing with the number of observations. A sometimes undesirable feature of these models is that if a dataset is generated from a finite number of clusters, these models will not xs recover the true number of clusters [MH14]. Instead, the estimated number of clusters will increasexs with the size of the dataset, resulting in redundant clusters that are located very close to each other. Gibbs-type priors with almost surely bounded number of components can learn the true number of clusters while still remaining reasonably tractable: so long as one can calculate the $V_{N,K}$ terms, MCMC for all these models can by carried out by modifications of the CRP-based sampler described earlier.

31.4.2 Dependent random probability measures

Dirichlet processes, and more generally, random probability measures, have also been generalized from settings with a single set of observations to those involving grouped observations, or observations indexed by covariates. Consider T sets of observations $\{\mathbf{X}^1, \ldots, \mathbf{X}^T\}$, each perhaps corresponding to a different country, a different year, or more abstractly, a different set of observed covariates. There are two immediate (though inadequate) ways to model such data: 1) by pooling all datasets into a single dataset, modeled as drawn from a DP or DPMM-distributed random probability measure G, or 2) by treating each group as independent, having its own random probability measure G^t. The first approach fails to capture differences between the groups, while the second ignores similarities between the groups (e.g. shared clusters). Dependent random probability measures seek a compromise between these extremes, allowing statistical information to be shared across different groups.

A seminal paper in the machine learning literature that addresses this problem is the **hierarchical Dirichlet process** (HDP) [Teh+06b]. The basic idea here is simple: each group has its own random measure drawn from a Dirichlet process $\mathrm{DP}(\alpha, H)$. The twist now is that the base measure H itself is random, in fact it is itself drawn from a Dirichlet process. Thus, the overall generative process is

$$H \sim \mathrm{DP}(\alpha_0, H_0), \tag{31.39}$$

$$G^t \sim \mathrm{DP}(\alpha, H), \quad t \in 1, \ldots, T \tag{31.40}$$

$$\overline{\theta}_i^t \sim G^t, \qquad i \in 1, \ldots, N_t, \quad t \in 1, \ldots, T. \tag{31.41}$$

The $\overline{\theta}_i^t$s might be the observations themselves, or the latent parameter underlying each observation, with $\boldsymbol{x}_i^t \sim F(\overline{\theta}_i^t)$.

Recall that if a probability measure G^1 is drawn from $\mathrm{DP}(\alpha, H)$, its atoms are drawn independently from the base measure H. In the HDP, H, which is a draw from a DP, is discrete, so that some atoms of G^1 will sit on top of each other, becoming a single atom. More importantly, all measures G^t will share the same infinite set of locations: each atom of H will eventually be sampled to form a location of an atom of each G^t. This will allow the same clusters to appear in all groups, though they will have different weights. Moreover, a big cluster in one group is *a priori* likely to be a big cluster in another group, as underlying both is a large atom in H. Since the common probability measure H itself is random, its components (both weights as well as locations) will be learned from data. Despite its apparent complexity, it is fairly easy to develop an analogue of the Chinese restaurant

process for the HDP, allowing us to sample observations directly without having to instantiate any of the infinite-dimensional measures. This is called the Chinese restaurant franchise, and essentially amounts to each group having its own Chinese restaurant with the following modification: whenever a customer sits at a new table and orders a dish, that dish itself is sampled from an upper CRP common to all restaurants. It is also possible to develop stick-breaking representations of the HDP.

The HDP has found wide application in the machine learning literature. A common application is document modeling, where the location of each atom is a **topic**, corresponding to some distribution over words. Rather than bounding the number of topics, there are an infinite number of topics, with document d having its own distribution over topics (represented by a measure G^d). By tying all the G^d's together through an HDP, different documents can share the same topics while emphasizing different topics.

Another application involves **infinite-state hidden Markov models**, also called **HDP-HMM**. Recall from Section 29.2 that a Markov chain is parametrized by a transition matrix, with row r giving the distribution over the next state if the current state is r. For an infinite-state HMM, this is an infinite-by-infinite matrix, with row r corresponding to a distribution G^r with an infinite number of atoms. The different G^r's can be tied together by modeling them with an HDP, so that atoms from each correspond to the same states [Fox+08].

Hierarchical nonparametric models of this kind can be constructed with other measures such as the Pitman-Yor process. For certain parameter settings, the PYP possesses convenient marginalization properties that the DP does not. In particular, simulating an RPM in two steps as shown below

$$G_0|H_0 \sim \text{PYP}(\alpha d_0, d_0, H_0), \tag{31.42}$$

$$G_1|G_0 \sim \text{PYP}(\alpha, d_1, G_0) \tag{31.43}$$

is equivalent to directly simulating G_1 without instantiating G_0 as below:

$$G_1|H_0 \sim \text{PYP}(\alpha d_0, d_0 d_1, G_0). \tag{31.44}$$

This marginalization property (also called **coagulation**) allows deep hierarchies of dependent RPMs, with only a smaller, dataset-dependent subset of G's having to be instantiated. In the **sequence memoizer** of [Woo+11], the authors model sequential data (e.g. text) with hierarchies that are *infinitely* deep, but with only a finite number of levels ever having to be instantiated. If needed, intermediate random measures can be instantiated by a dual **fragmentation** operator.

Deeper hierarchies like those described above allow more refined modeling of similarity between different groups. Under the original HDP, the groups themselves are exchangeable, with no subset of groups *a priori* more similar to each other than to others. For instance, suppose each of the measures $G_1, \ldots, G_T$ correspond to distributions over topics in different scientific journals. Modeling the G_t's with an HDP allows statistical sharing across journals (through shared clusters and similar cluster probabilities), but does not regard some journals as *a priori* more similar than others. If one had further information, e.g. that some are physics journals and the rest are biology journals, then one might add another level to the hierarchy. Now rather than each journal having a probability measure G^t drawn from a $\text{DP}(\alpha, H)$, physics and biology journals have their own base measures H_p and H_b that allow statistical sharing among physics and biology journals respectively. Like the HDP, these are draws from a DP with base measure H. To allow sharing *across* disciplines, H is again random, drawn from a DP with base measure H_0. Overall, if there are D disciplines, $1, 2, \ldots, D$, the overall

model is

$$H \sim \text{DP}(\alpha_0, H_0), \tag{31.45}$$

$$H^d \sim \text{DP}(\alpha_1, H), \quad d \in 1, \ldots, D \tag{31.46}$$

$$G^{t,d} \sim \text{DP}(\alpha_2, H^d), \quad t \in 1, \ldots, T_d \tag{31.47}$$

$$\overline{\theta}_i^{t,d} \sim G^{t,d} \qquad d \in 1, \ldots, D, \quad t \in 1, \ldots, T_d, \quad i \in 1, \ldots, N_t \tag{31.48}$$

One might add might further levels to the hierarchy given more information (e.g. if disciplines are grouped into physical sciences, social sciences and humanities).

Dependent random probability measures can also be indexed by covariates in some continuous space, whether time, space, or some Euclidean space or manifold. This space is typically endowed with some distance or similarity function, and one expects that measures with similar covariates are *a priori* more similar to each other. Thus, G^t might represent a distribution over topics in year t, and one might wish to model G^t evolving gradually over time. There is rich history of dependent random probability measures in statistics literature, starting from [Mac99a]. A common requirement in such models is that at any fixed time t, the marginal distribution of G^t follows some well specified distribution, e.g. a Dirichlet process, or a PYP. Approaches exploit the stick-breaking representation, the CRP representation or something else like the Poisson structure underlying such models (see Section 31.7).

As a simple and early example, recall the stick-breaking construction from Section 31.2.2, where a random probability measure is represented by an infinite collection of pairs (β_k, θ_k). To construct a family of RPMs indexed by some covariate t, we need a family of such sets (β_k^t, θ_k^t) for each of the possibly infinite values of t. To ensure each G_t is marginally DP distributed with concentration parameter α and base measure H, we need that for each t, the β_k^ts are marginally i.i.d. draws from a $\text{Beta}(1, \alpha)$, and the θ_k^t's i.i.d. draws from H. Further, we do not want independence across t, rather, for two times t_1 and t_2, we want similarity to increase as $|t_1 - t_2|$ decreases. To achieve this, define an infinite sequence of independent Gaussian processes on T, $f_k(t)$, $k = 1, 2, \ldots$, with mean 0 and some covariance function. At any time t, $f_k(t)$ for all k are i.i.d. draws from a normal distribution. By transforming these Gaussian processes through the cdf of a Gaussian density (to produce a i.i.d. uniform random variables at each t), and then through the inverse cdf of a $\text{Beta}(1, \alpha)$), one has an infinite collection of i.i.d. $\text{Beta}(1, \alpha)$ random variables at any time t. The GP construction means that each β_k^t varies smoothly with t. A similar approach can be used to construct smoothly varying θ_k^ts that are marginally H-distributed, and together, these form a family of gradually varying RPMs G^t, with

$$G^t = \sum_{i=1}^{\infty} \pi_k^t \delta_{\theta_k^t}, \quad \text{where} \quad \pi_k^t = \beta_k^t \prod_{j=1}^{k-1} (1 - \beta_j^t) \tag{31.49}$$

Of course, such a model comes with formidable computational challenges, however the marginal properties allows standard MCMC methods from the DP and other RPMs to be adapted to such settings.

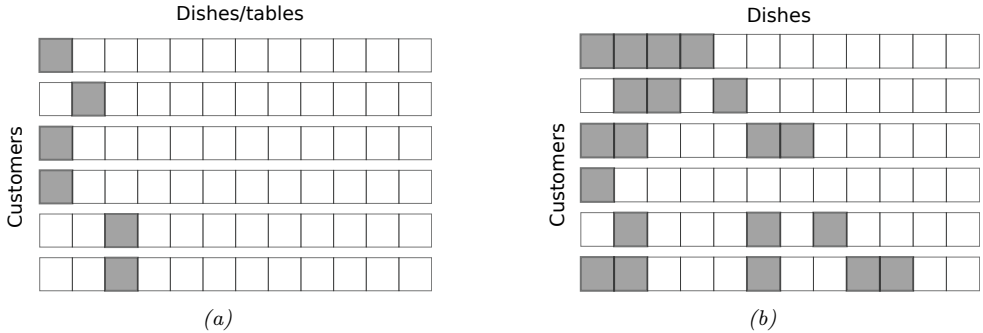


Figure 31.7: (a) A realization of the cluster matrix Z from the Chinese restaurant process (CRP) (b) A realization of the feature matrix Z from the Indian buffet process (IBP). Rows are customers and columns are dishes (for the CRP each table has its own dish). Both produce binary matrices, with the CRP assigning a customer to a single table (cluster), and the IBP assigning a set of dishes (features) to each customer.

31.5 The Indian buffet process and the Beta process

The Dirichlet process is a Bayesian nonparametric model useful for clustering applications, where the number of clusters is allowed to grow with the size of the dataset. Under this generative model, each observation is assigned to a cluster through the variable z_i. Equivalently, z_i can be written as a one-hot binary vector, and the entire clustering structure can be written as a binary matrix Z consisting of N rows, each with exactly one non-zero element (Figure 31.7(a)).

Clustering models that limit each observation to a single cluster can be overly restrictive, failing to capture the complexity of the real datasets. For instance, in computer vision applications, rather than assign an image to a single cluster, it might be more appropriate to assign it a binary vector of features, indicating whether different object types are or are not present in the image. Similarly, in movie recommendation systems, rather than assign a movie to a single genre ('comedy' or 'romance' etc.), it is more realistic to assign to multiple genres ('comedy' AND 'romance'). Now, in contrast to all-or-nothing clustering (which would require a new genre 'romantic comedy'), different movies can have different but overlapping sets of features, allowing a partial sharing of statistical information.

Latent feature models generalize clustering models, allowing each observation to have multiple features. Nonparametric latent feature models allow the number of available features to be infinite (rather than fixed *a priori* to some finite number), with the number of active features in a dataset growing with a dataset size. Such models associate the dataset with an infinite binary matrix consisting of N rows, but now where each row can have multiple elements set to 1, corresponding to the active features. This is shows in Figure 31.7(b). As with the clustering models, each column is associated with a parameter drawn i.i.d. from some base measure H.

The Indian buffet process (IBP) in a Bayesian nonparametric analogue of the CRP for latent feature models. As with the CRP, the IBP is specified by a concentration parameter α, and a base measure H on some space Θ. The former controls the distribution over the binary feature matrix, whereas feature parameters θ_k are drawn i.i.d. from the latter. Under the IBP, individuals enter sequentially into a restaurant, now picking a set of dishes (instead of a single table). The first customer samples a Poisson(α)-distributed random number of dishes, and assigns each of them

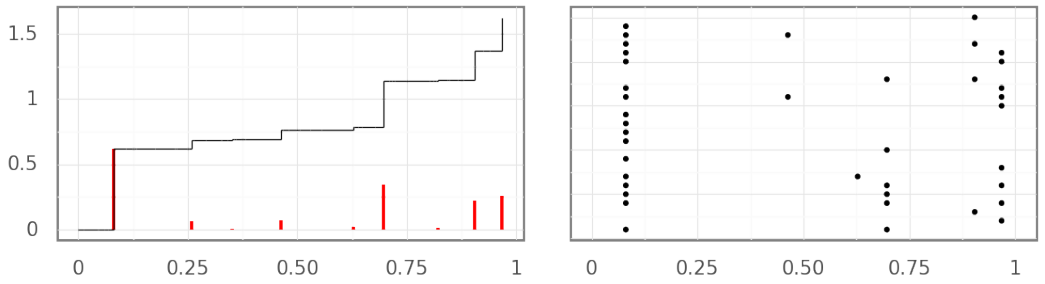


Figure 31.8: (a) A realization of a Beta process on the real line. The the atoms of the random measure are show in red, while the Beta subordinator (whose value at time t sums all atoms up to t) is shown in black. (b) Samples from the Beta process

values drawn from H. When the ith customer enters the restaurant, they first make a pass through all dishes already chosen. Suppose N_d of the earlier customers have chosen dish d: then customer i selects this with probability N_d/i. This results in a rich-get-richer phenomenon, where popular dishes (common features) are more likely to be selected in the future. Additionally, the i customer samples a Poisson(α/i) number of new dishes. This results in a non-zero probability of new dishes, that nonetheless decreases with i.

A key property of the Indian buffet process is that, like the CRP, it is exchangeable. In other words, its statistical properties do not change if its rows are permuted. For instance, one can show that the number of dishes picked by any customer is marginally Poisson(α) distributed. Similarly, the distribution over the number of features shared by the first two customers is the same as the that for the first and last customer. We mention that like the CRP, the ordering of the dishes (or columns) is arbitrary and might appear to violate exchangeability. For instance, the first customer cannot pick the first and third dishes and not the second dish, while this is possible for the second customer. These artefacts disappear if we index columns by their associated parameters. Equivalently, after reordering rows, we can transform the feature matrix to be left-ordered (essentially, all new dishes selected by a customer must by adjacent, see [GG11]), and we can view the IBP as a prior on such left-ordered matrices.

The exchangeability of the rows of the IBP implies via de Finetti's theorem that there exists an underlying random measure G, conditioned on which the rows are i.i.d. draws. Just as the Chinese restaurant process represents observations drawn from a Dirichlet process-distributed random probability measure, the dishes of each customer in the IBP represent observations drawn from a **Beta process**. Like the DP, the Beta process is an atomic measure taking the form

$$G(\boldsymbol{\theta}) = \sum_{k=1}^{\infty} \pi_k \delta_{\boldsymbol{\theta}_k}(\boldsymbol{\theta}). \tag{31.50}$$

Each of the π_k's lie between 0 and 1, but unlike the DP where they add up to 1, the π_k's sum up to a finite random number. The Beta process is thus a **random measure**, rather than a random probability measure. One can imagine the kth atom as a coin located at θ_k, with probability of

success equal to π_k. To simulate a row of the IBP, one flips each of the infinite coins, selecting the feature at θ_k if the k coin comes up heads. One can show that if the π_k's sum up to a finite number, the number of active features will be finite. Of the infinite atoms in G, a few will dominate the rest, these will be revealed through the rich-gets-richer dynamics of the IBP as features common to a large proportion of the observations.

The Beta process has a construction similar to the stick-breaking representation of the Dirichlet process. As with the DP, the locations of the atoms are independent draws from the base measure, while the sequence of weights $\pi_1, \pi_2, \ldots$ are constructed from an infinite sequence of Beta variables: now these are $\text{Beta}(\alpha, 1)$ distributed, rather than $\text{Beta}(1, \alpha)$ distributed. The overall representation of the Beta process is then

$$\beta_k \sim \text{Beta}(\alpha, 1), \qquad \boldsymbol{\theta}_k \sim H, \tag{31.51}$$

$$\pi_k = \beta_k \pi_{k1} = \prod_{l=1}^{k} \beta_l \tag{31.52}$$

It is not hard to see that under the IBP, the total number of dishes underlying a dataset of size N follows a $\text{Poisson}(\alpha H_N)$ distribution, where H_N is $H_N = \sum_{i=1}^{N} 1/i$ is the Nth harmonic number. The Beta process and the IBP have been generalized to three-parameter versions allowing power-law behavior in the total number of dishes (features) in a dataset of size N, as well as in the number of customers trying each dish [TG09]. It has found application in tasks ranging from genetics, collaborative filtering, and in models for graph and graphical model structures. Just as with the DP, posterior inference can proceed via MCMC (exploiting either the IBP or the stick-breaking representation), particle filtering or using variational methods.

31.6 Small-variance asymptotics

Nonparametric Bayesian methods can serve as a basis to develop new and efficient discrete optimization algorithms that have the flavor of Lloyd's algorithm for the k-means objective function. The starting point for this line of work is the view of the k-means algorithm as the *small-variance asymptotic limit* of the EM algorithm for a mixture of Gaussians. Specifically, consider the EM algorithm to estimate the unknown cluster means $\boldsymbol{\mu} \equiv (\mu_1, \ldots, \mu_k)$ of a mixture of k Gaussians, all of which have the same known covariance $\sigma^2 I$. In the limit as $\sigma^2 \to 0$, the E-step, which computes the cluster assignment probabilities of each observation given the current parameters $\boldsymbol{\mu}^{(t)}$, now just assigns each observation to the nearest cluster. The M-step recomputes a new set of parameters $\boldsymbol{\mu}^{(t+1)}$ given the cluster assignment probabilities, and when each observation is hard-assigned to a single cluster, the cluster means are just the means of the assigned observations. The process of repeatedly assigning observations to the nearest cluster, and recomputing cluster locations by averaging the assigned observations is exactly Lloyd's algorithm for k-means clustering.

To avoid having to specify the number of clusters k, [KJ12b] considered a Dirichlet process mixture of Gaussians, with all infinite components again having the same known variance σ^2. The base measure H from which the component means are drawn was set to a zero-mean Gaussian with variance ρ^2, with α the concentration parameter. The authors then considered a Gibbs sampler for this model, very closely related to the sampler in Algorithm 42, except that instead of collapsing or marginalizing out the cluster parameters, these are instantiated. Thus, an observation x_i as assigned

to a cluster c with mean μ_c with probability proportional to $N_{c,-i}\frac{1}{\sqrt{2\pi\sigma^2}}\exp(-\frac{1}{2\sigma^2}\|x_i-\mu_c\|^2)$, while it is assigned to a new cluster with probability proportional to $\alpha\frac{1}{\sqrt{2\pi(\sigma^2+\rho^2)}}\exp(-\frac{1}{2(\sigma^2+\rho^2)}\|x_i\|^2)$. After cycling through all observations, one then resamples the parameters of each cluster. Following the Gaussian base measure and Gaussian likelihood, this too follows a Gaussian distribution, whose mean is a convex combination of the prior mean 0 and data-driven term, specifically the average of the assigned observations. The weight of the prior term is proportional to the inverse of the prior variance ρ^2, while the weight of the likelihood term is proportional to the inverse of the likelihood variance σ^2.

To derive a small-variance limit of the sampler above, we cannot just let σ^2 go to 0, as that would result in each observation being assigned to its own cluster. To prevent the likelihood from dominating the prior in this way, one must also send the concentration parameter α to 0, so that the DP prior enforces an increasingly strong penalty on a large number of clusters (recall that *a priori*, the average number of clusters underlying N observations is $\alpha \log N$). [KJ12b] showed that with α scaled as $\alpha = (1+\rho^2/\sigma^2)^{d/2}\exp(-\frac{\lambda}{2\sigma^2})$ for some parameter λ, and taking the limit $\sigma^2 \to 0$ with ρ fixed, we get the following modification of the k-means algorithm:

1. Assign each observation to the nearest cluster, *unless the distance to the nearest cluster exceeds λ, in which case assign the observation to its own cluster.*

2. Set the cluster means equal to the average of the assigned observations.

Algorithm 44: DP-means hard clustering algorithm for data $\mathcal{D} = \{x_1, \ldots, x_N\}$

Initialize the number of clusters $K = 1$, with cluster parameter μ_1 equal to the global mean: $\mu_1 = \frac{1}{N}\sum_{i=1}^N x_i$
Initialize all cluster assignment indicators $z_i = 1$
while *all z_is have not converged* **do**
  for *each $i = 1 : N$* **do**
    Compute distance $d_{ik} = \|x_i - \mu_k\|^2$ to cluster k for $k = 1, \ldots, K$
    if $\min_k d_{ik} > \lambda$ **then**
      Increase the number of clusters K by 1: $K = K + 1$
      Assign observation i to this cluster: $z_i = K$ and $\mu_K = x_i$
    else
      Set $z_i = \arg\min_c d_{ik}$
    for *each $k = 1 : K$* **do**
      Set $\mathcal{D}_k$ be the observations assigned to cluster k. Compute cluster mean $\mu_k = \frac{1}{|\mathcal{D}_k|}\sum_{x\in\mathcal{D}_k} x$

Like k-means, this is a hard-clustering algorithm, except that instead of having to specify the number of clusters k, one just specifies a penalty λ for introducing a new cluster. The actual number of clusters is determined by the data, [KJ12b] refer to this algorithm as DP-means. One can show that

the iterates above converge monotonically to a local maximum of the following objective function:

$$\sum_{k=1}^{K} \sum_{x \in \mathcal{D}_k} \|x - \mu_k\|^2 + \lambda K, \quad \text{where } \mu_k = \frac{1}{|\mathcal{D}_k|} \sum_{x \in \mathcal{D}_k} x. \tag{31.53}$$

The first term in the expression above is exactly the objective function of the k-means algorithm with K clusters. The second term is an penalty term that introduces a cost λ for each additional cluster. Interestingly, the penalty term above corresponds to the so called Akaike Information Criterion (AIC), a well studied approach to penalizing model complexity.

It is also possible to derive hard-clustering algorithms for simulataneously clustering multiple datasets, while allowing these to share clusters. This is possible through the small-variance limit of a Gibbs sampler for the hierarchical Dirichlet process (HDP) and results in a clustering algorithm that now has two thresholding parameters, a local one λ_l and a global one λ_g. The algorithm proceeds by maintaining a set of global clusters, with the local clusters of each dataset assigned to a subset of the global clusters. It then repeats the following steps until convergence:

Assign observations to local clusters For x_{ij}, the ith datapoint in dataset j, compute the distance to all global clusters. For those global clusters not currently present in dataset j, add λ_l to their distance; this reflects the cost of introducing a new cluster into dataset j. Now, assign x_{ij} to the cluster with the smallest distance, unless the smallest distance exceeds $\lambda_g + \lambda_l$, in which case, create a new global cluster and assign x_{ij} to it. Observe that in the latter case, the distance of x_{ij} to the new cluster is 0, with $\lambda_g + \lambda_l$ reflecting the cost of introducing a new global and then local cluster.

Assign local clusters to global clusters For each local cluster l, compute the sum of the distances of all its assigned observations to the cluster mean. Call this d_l. Also compute the sum of the distances of the assigned observations to each global cluster. For global cluster p, call this $d_{l,p}$. Then assign local cluster l to the global cluster with the smallest $d_{l,p}$, unless $\min d_{l,p} > d_l + \lambda_g$ in which case we create a new global cluster.

Recompute global cluster means Set the global cluster means equal to the average of the assigned observations across all datasets.

In [JKJ12], these ideas were extended from DP mixtures of Gaussians to DP mixtures of more general exponential family distributions. Briefly, the hard-clustering algorithms maintained the same structure, the only difference being that distance from clusters was measured using a **Bregman divergence** specific to that exponential family distribution. For Gaussians, the Bregman divergence reduces to the usual Euclidean distance.

In [BKJ13], the authors showed how such small-variance algorithms could be derived directly from a probabilistic model, and independent of any specific computational algorithm such as EM or Gibbs sampling. Their approach involves computing the MAP solution of the parameters of the model, and then taking the small-variance limit to obtain an objective function. This approach, called MAP-based Asymptotic Derivations from Bayes or MAD-Bayes allowed them to derive, among other things, an analog of the DP-means algorithm from feature-based models. The called this the BP-means algorithm, after the Beta process underlying the Indian Buffet process.

Write X for an $N \times D$ matrix of N D-dimensional observations, Z for an $N \times K$ matrix of binary feature assignments, and A for a $K \times D$ matrix of K D-dimensional features, with one seeking a pair

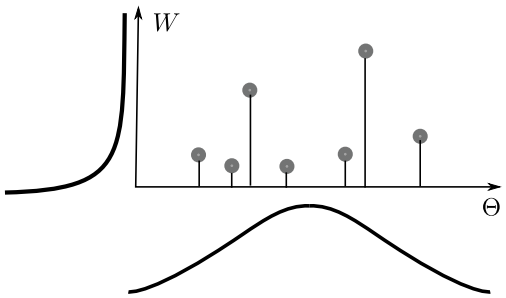


Figure 31.9: Poisson process construction of a completely random measure $G = \sum_i w_i \delta_{\theta_i}$. The set of pairs $\{(w_1, \theta_1), (w_2, \theta_2), \dots\}$ is a realization from a Poisson process with intensity $\lambda(\theta, w) = H(\theta)\gamma(w)$.

(A, Z) such that ZA approximates X as well as possible. [BKJ13] showed in the small-variance limit, finding the MAP solution for the IBP is equivalent to the following problem:

$$\text{argmin}_{K,Z,A} \text{trace}[(X - ZA)^t(X - ZA)] + \lambda K, \tag{31.54}$$

where again λ is a parameter of the algorithm, govering how the concentration parameter scales with the variance, and specifying a penalty on introducing new features. This objective function is very intuitive: the first term corresponds to the approximation error from K features, while the second term penalizes model complexity resulting from a large number of features K. This objective funtion can be optimized greedily by repeating three steps for each observation i until convergence:

1. Given A and K, compute the optimal value of the binary feature assignment vector z_i of observation i and update Z.

2. Given Z and A, introduce an additional feature vector equal to the residual for the ith observation, namely, $x_i - z_i A$. Call A' the updated feature matrix, and Z' the updated feature assignment matrix, where only observation i has been assigned this feature. If the configuration $(K+1, Z', A')$ results in a lower value of the objective function than (K, Z, A), set the former as the new configuration. In other words, if the benefit of introducing a new feature outweights the penalty λ, then do so.

3. Update the feature vectors A given the feature assignment vectors Z.

[BKJ13] showed that this algorithm monotonically decreases the objective in Equation (31.54), converging eventually to a local optimum. Subsequent works have considered the small-variance asymptotics of more structured models, such as topic models, hidden Markov models and even continuous-time stochastic process models.

31.7 Completely random measures

The Dirichlet process is a example of a random probability measure, a class of random measures which always integrate to 1. The Beta process, while not an RPM, belongs to another class of random measures: **completely random measures** (CRM). A completely random measure G satisfies the

following property: for any two disjoint subsets T_1 and T_2 of Θ, the values $G(T_1)$ and $G(T_2)$ are independent random variables:

$$G(T_1) \perp\!\!\!\perp G(T_2) \quad \forall\, T_1, T_2 \subset \Theta \; s.t. \; T_1 \cap T_2 = \varnothing. \tag{31.55}$$

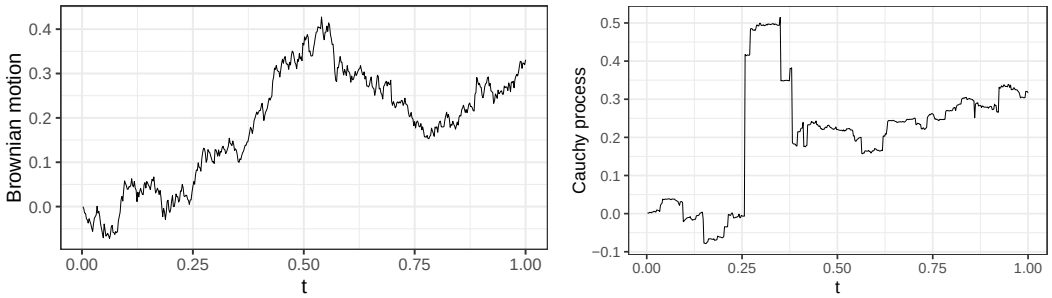
Note that the Dirichlet process is not a CRM, since for disjoint sets T and its complement $T^c = \Theta \setminus T$, we have $G(T) = 1 - G(T^c)$, which is as far from independent as can be. More generally, under the DP, for disjoint sets T_1 and T_2, the measures $G(T_1)$ and $G(T_2)$ are negatively correlated. We will see later though that the Dirichlet process is closely related to another CRM, the Gamma process. As a side point, beyond the sum-to-one constraint, $G(T_1)$ does not tell us anything about the distribution of probability within $G(T_1^c)$, making the DP what is known as a **neutral process**.

The simplest example of a CRM is the **Poisson process** (see Section 31.9). A Poisson process with intensity $\lambda(\theta)$ is a **point process** producing points or events in Θ, with the number of points in any set T following a Poisson$(\int_T \lambda(\theta)d\theta)$-distribution, and with the counts in any two disjoint sets independent of each other. While it is common to think of this as a point process, one can also think of a realization from a Poisson process as an **integer-valued random measure**, where the measure of any set is the number of points falling within that set. It is clear then that the Poisson process is an example of a CRM.

It turns out that the Poisson process underlies all completely random measures in a fundamental way. For some space Θ, and $\mathbb{W}$ the positive real line, simulate a Poisson process on the product space $W \times \Theta$ with intensity $\lambda(\theta, w)$. Figure 31.9 shows a realization from this Poisson process, write it as $M = \{(\theta_1, w_1), \ldots, (\theta_{|M|}, w_{|M|})\}$ where $|M|$ is the number of events. This can be used to construct an atomic measure $G = \sum_{i=1}^{|M|} w_i \delta_{\theta_i}$ as illustrated in Figure 31.9. From its Poisson construction, this is a completely random measure on Θ, with set $T \in \Theta$ having measure $\sum_i w_i \mathbb{I}(\theta_i \in T)$. Different settings of $\lambda(\theta, w)$ give rise to different CRMs, and in fact, other than CRMs with atoms at some fixed locations in Θ, this construction characterizes *all* CRMs.

For a CRM, the Poisson intensity is typically chosen to factor as $\lambda(\theta, w) = H(\theta)\gamma(w)$, with $\int_\Theta H(\theta)d\theta = 1$. Then, $H(\theta)$ is the base measure controlling the locations of the atoms in the CRM, while the measure $\gamma(w)$ controls the number of atoms, and the distribution of their weights. Setting $\gamma(w) = w^{-1}(1-w)^{\alpha-1}$ gives the Beta process with base measure $H(\theta)$. Other choices include the Gamma process ($\gamma(w) = \alpha w^{-1} \exp(-w)$), the stable process ($\gamma(w) = \frac{1}{\Gamma(1+\sigma)} \alpha w^{-1-\sigma}$), and the generalized Gamma process ($\gamma(w) = \frac{1}{\Gamma(1+\sigma)} \alpha w^{-1-\sigma} \exp(-\zeta w)$).

For all three processes described earlier, the $\gamma(w)$ integrates to infinity, so that $\int_\Theta \int_W \lambda(\theta, w) d\theta dw = \infty$. Consequently, the number of Poisson events, and thus the number of atoms in the CRMs are infinite with probability one. At the same time, mass of the $\gamma(w)$ function is mostly concentrated around 0 (see Figure 31.9), and for any $\epsilon > 0$, $\int_\epsilon^\infty \gamma(w) dw$ is finite. It is easy to show that the sum of the w's is finite almost surely. Call this sum W, then the first condition ensures W greater than 0, while the second ensures it is finite. These two conditions make it sensible to divide a realization of a CRM by its sum, resulting in a random measure that integrates to 1: a random probability measure. Such RPMs are called **normalized completely random measures**, or sometimes just **normalized random measures (NRMs)**. The Dirichlet process we saw earlier is an example of an NRM: it is a normalized Gamma process. This result mirrors the situation with the finite Dirichlet distribution: one can simulate from a d-dimensional Dir$(\alpha_1, \ldots, \alpha_d)$ distribution by first simulating d independent Gamma variables $g_i \sim \text{Ga}(\alpha_i, 1), i = 1, \ldots, d$, and then defining the probability vector $\frac{1}{\sum_{i=1}^d g_i}(g_1, \ldots, g_d)$. The Pitman-Yor process is not an NRM except for special settings of its



(a) *(b)*

Figure 31.10: (a) A realization of a Brownian motion (b) A realization of Cauchy process (an α-stable process with $\alpha = 1$).

parameters: like we saw, $d = 0$ is a Dirichlet process or normalized Gamma process. $\alpha = 0$ is a **normalized stable process**. The normalized generalized Gamma process is an NRM that includes the DP and the normalized stable process as special cases.

31.8 Lévy processes

Completely random measures are also closely related to **Lévy processes** and **Lévy subordinators** [Ber96]. A Lévy process is a continuous-time stochastic process $\{L_t\}_{t\geq 0}$ taking values in some space (e.g. $\mathbb{R}^d$) that satisfies two properties[1]:

stationary increments: for $t, \Delta > 0$, the random variable $L_{t+\Delta} - L_t$ does not depend on t

independent increments: for $t, \Delta > 0$, $L_{t+\Delta} - L_t$ is independent of values before t.

A Lévy subordinator is a real-valued, **nondecreasing** Lévy process. If we drop the stationarity condition, it should be clear that the increments of a Lévy subordinator are exactly the atoms of a completely random measure (see Figure 31.8). Common examples of Lévy subordinators are Beta subordinators (or Beta processes), Gamma subordinators, stable subordinators and generalized Gamma subordinators. CRMs generalize Lévy subordinators, by allowing them to be indexed by some general space Θ (rather than by nonnegative reals), and by relaxing the stationarity condition to allow the atoms to follow some general base measure H.

Unlike Lévy subordinators, a general Lévy process can have negative changes as well, with the change $L_{t+\Delta} - L_t$ belonging to an **infinitely divisible distribution**, scaled by Δ. A random variable X follows an infinitely divisible distribution if, for any positive integer N, there exists some probability distribution such that the sum of N i.i.d. samples from that distribution has the same distribution as X. Examples of infinitely divisible distributions include the Poisson distribution, the Gamma distribution, the Cauchy distribution, the α-stable distribution, and the inverse-Gaussian distribution (among many others), though by far the most well-known example is the Gaussian distribution. It is easy to see why the change in a Lévy process $L_{t+\Delta} - L_t$ over some time interval

1. There is also a technical continuity condition that we do not discuss here.

Δ follows an infinitely divisible distribution: just divide the interval into N equal-length segments. From the properties of the Lévy process, the changes over these segments are independent and identically distributed, and their sum equals $L_{t+\Delta} - L_t$. The **Lévy-Kintchine** formula shows that the converse is also true: any infinitely divisible distribution represents the change of an associated Lévy process. The Lévy process corresponding to the Gaussian distribution is the celebrated **Brownian motion** (or **Weiner process**). Brownian motion is a fundamental and widely applied stochastic process, whose mathematics was first studied by Louis Bachelier to model stock markets, and later, famously, by Albert Einstein to argue about the existence of atoms. For a Brownian motion, the increment $L_{t_1+\Delta} - L_{t_1}$ follows a normal $N(\mu\Delta, \sigma\Delta)$ distribution, where μ and σ are the *drift* and *diffusion* coefficients. Setting μ and σ to 0 and 1 gives **standard Brownian motion**. Paths sampled from the Weiner process are continuous with probability one, all other Lévy processes are jump processes. Figure 31.10 shows realizations from some processes. The jump processes have a Poisson construction related to the Lévy subordinators and completely random measures, and the **Lévy-Itô decomposition** shows that every Lévy process can be decomposed into Brownian and Poisson components. Levy processes have been widely applied in mathematical finance to model asset prices, insurance claims, stock prices and other financial assets.

31.9 Point processes with repulsion and reinforcement

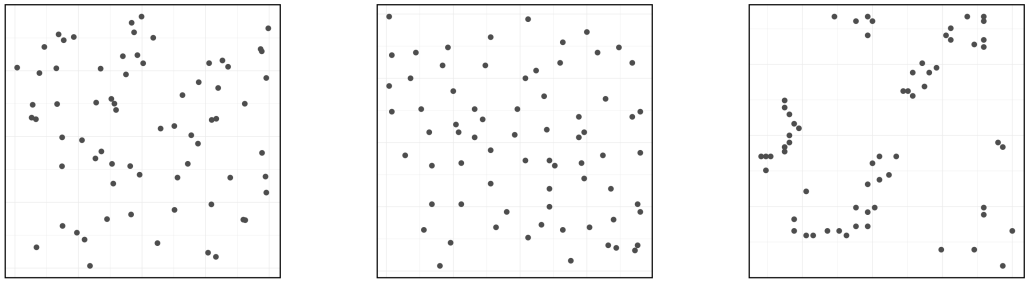
In this section, we look more closely at the Poisson process, as well as other, more general point processes that allow inter-event interactions.

31.9.1 Poisson process

We have already briefly seen the Poisson process: this is a point process on some space Θ that is parametrized by an intensity function $\lambda(\theta) \geq 0$, and that produces a Poisson($\int_T \lambda(\theta)d\theta$)-distributed number of points of events in any set $T \in \Theta$, with the counts in any two disjoint sets independent of each other. Recall that if $N_i \sim \text{Poisson}(\lambda_i)$ are independent, then a well-known property of the Poisson distribution is that $N_1 + N_2 \sim \text{Poisson}(\lambda_1 + \lambda_2)$. This relates to the infinite divisibility of the Poisson distribution. It is clear that the average number of points is large in areas where $\lambda(\theta)$ is high, and small where $\lambda(\theta)$ is small. When the intensity $\lambda(\theta)$ is some constant λ, we have a **homogeneous Poisson process**, otherwise we have an **inhomogeneous Poisson process**.

Depending on whether $\int_\Theta \lambda(\theta)d\theta$ is infinite or finite, a Poisson process will either produce an infinite number of points (recall the Poisson process underlying the CRMs of Section 31.7) or a finite number of points. The latter is more common in applications, such as modeling phone calls or financial shocks (Θ is some finite time-interval), the locations of trees or forest fires (Θ is some subset of the Euclidean plane), the locations of cells or galaxies (Θ is a 3-dimensional space) or events in higher-dimensional spaces (for example, spatio-temporal activity). One way to simulate a finite Poisson process is to first simulate the number of total number of points N, which follows a Poisson($\int_\Theta \lambda(\theta)d\theta$) distribution. One can then simulate the locations of these points by sampling N times from the probability density $\frac{\lambda(\theta)}{\int_\Theta \lambda(\theta)d\theta}$. For a homogeneous Poisson process, the locations are uniformly distributed over Θ. One can also easily simulate a rate-λ homogeneous Poisson on the real line by exploiting the fact that inter-event times follow an exponential distribution with mean $1/\lambda$.

If the integral $\int_\Theta \lambda(\theta)d\theta$ is difficult to evaluate, one can also simulate a Poisson process using the



(a) (b) (c)

Figure 31.11: A realization of (a) a homogeneous Poisson process. (b) an underdispersed point process (Swedish pine sapling locations [Rip05]). (c) A realization of an ovderdispersed point process (California redwood tree locations [Rip77]).

thinning theorem [LS79]. Here, one needs to find a function $\gamma(\theta)$ such that $\gamma(\theta) \geq \lambda(\theta), \forall\theta$, and such that it is easy to simulate from a rate-$\gamma(\theta)$ Poisson process. Suppose the result is $\Psi = \{\psi_1, \ldots, \psi_N\}$. Since $\gamma(\theta) \geq \lambda(\theta)$, Ψ is going to contain more events, and one *thins* Ψ by keeping each element ψ_i in Ψ with probability $\lambda(\psi_i)/\gamma(\psi_i)$ (otherwise one discards it). Once can show that the set of surviving points is then a realization of a Poisson process with rate $\lambda(\theta)$.

The defining feature of the Poisson process is the assumption of independence among events. In many settings, this is inappropriate and unrealistic, and the knowledge of an event at some location θ might suggest a reduced or elevated probability of events in neighboring areas. Point processes satisfying the former property are called underdispersed (Figure 31.11(b)), while point process satisfying the latter property are called overdispersed (Figure 31.11(c)). Examples of underdispersed point processes include the locations of trees or train stations, which tend to be more spread out than Poisson because of limited resources. An example of an overdispersed point process is earthquake locations, where aftershocks tend to occur in the vicinity of the main shock.

A simple approach to modeling overdispersed point processes is through hierarchical extensions of the Poisson process that allow the Poisson intensity function $\lambda(\cdot)$ to be a random variable. Such models are called doubly-stochastic Poisson processes or Cox processes [CI80], and a common approach is to model the intensity $\lambda(\cdot)$ via a Gaussian process. Note though that the Poisson process intensity function must be nonnegative, so that λ is often a transformed Gaussian process:

$$\lambda(\theta) = g(\ell(\theta)), \quad \ell(\theta) \sim \text{GP}. \tag{31.56}$$

Common examples for g include exponentiation, sigmoid transformation or just thresholding. Because of the smoothness of the unknown $\lambda(\cdot)$, observing an event at some location suggests events are likely in the neighborhood. Such models still do not capture direct interactions between point process events, rather, these are mediated through the unknown intensity functions, making them inappropriate for many applications. For instance, in neuroscience a neuron's spiking can be driven directly by past activity, and activity of other neurons in the network, rather than just through some shared stimulus $\lambda(t)$. Similarly, social media activity has a strong reciprocal component, where, for instance, emails sent out by a user might be in response to past activity, or activity of other users. The next few subsections show how one might explicitly model such interactions.

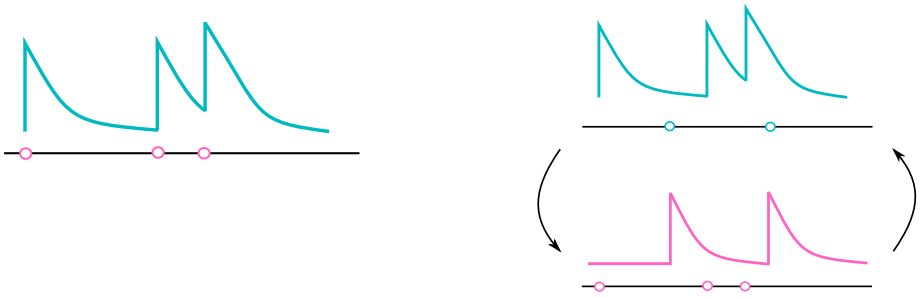


Figure 31.12: (a) A self-exciting Hawkes process. Each event causes a rise in the event rate, which can lead to burstiness. (b) A pair of mutually exciting Hawkes processes, events from each cause a rise in the event rate of the other.

31.9.2 Renewal process

Renewal processes are one class of models of repulsion and reinforcement for point processes defined on the real line (typically regarded as time). Recall that for a homogeneous Poisson process, inter-event times follow an exponential distribution. The exponential distribution has the property of **memorylessness**, where the time until the next event is independent of how far in the past the last event occurred. That is, if τ follows the exponential distribution, then for any δ and $\Delta > 0$, we have $p(\tau > \Delta + \delta | \tau \geq \delta) = p(\tau > \Delta)$). Renewal processes incorporate memory by allowing the interevent times to follow some general distribution on the positive reals. Examples include Gamma renewal processes and Weibull renewal processes, where interevent times follow the Gamma and Weibull distribution respectively. Both these processes include parameter settings that recover the Poisson process, but also allow **burstiness** and **refractoriness**. Burstiness refers to the phenomenon where, after an event has just occured, one is more likely to see more events than a longer time afterwards. This is useful for modeling email activity for instance. Refractoriness refers to the opposite situation, where an event occuring implies new events temporarily less likely to occur. This is useful to model neural spiking activity, for instance, where after spiking, a neuron is depleted of resources, and requires a recovery period before it can fire again.

31.9.3 Hawkes process

Hawkes processes [Haw71] are another class of reinforcing point processes that have attracted much recent attention. Hawkes processes provide an intuitive framework for modeling reinforcement in point processes, through self-excitation when a single point process is involved, and through mutual excitation (sometimes called reciprocity) when collections of point processes are under study. The former, shown in Figure 31.12(a), is relevant when modeling bursty phenomena like visits to a hospital by an individual, or purchases and sales of a particular stock. The latter, displayed in Figure 31.12(b) is useful to characterize activity on social or biological networks, such as email communications or neuronal spiking activity. In both examples, each event serves as a trigger for subsequent bursts of activity. This is achieved by letting $\lambda(t)$, the event rate at time t, be a function

of past activity or *event history*. Write $\mathcal{H}(t) = \{t_k : t_k \leq t\}$ for the set of event times up to time t. Then the rate at time t, called the *conditional intensity function* at that time, is given by

$$\lambda(t|\mathcal{H}(t)) = \gamma + \sum_{t_k \in \mathcal{H}(t)} \phi(t - t_k)\,, \tag{31.57}$$

where γ is called the **base-rate** and the function $\phi(\cdot)$ is called the **triggering kernel**. The latter characterizes the excitatory effect of a past event on the current event rate. Figure 31.12 shows the common situation where $\phi(\cdot)$ is an exponential kernel $\phi(\Delta) = \beta e^{-\Delta/\tau}$, $\Delta \geq 0$. Here, a new event causes a jump of magnitude β in the intensity $\lambda(t)$, with its excitatory influence decaying exponentially back to γ, with τ the time-scale of the decay. For a multivariate Hawkes process, there are m point processes $(N_1(t), N_2(t), \cdots, N_m(t))$ associated with m users or nodes. Write $\mathcal{H}_i(t)$ for the event history of the ith process at time t. Its conditional intensity can depend on all m event histories, taking the form

$$\lambda_i(t|\{\mathcal{H}_j(t)\}_{j=1}^m) = \gamma_i + \sum_{j=1}^{m} \sum_{t_k \in H_j(t)} \phi_{ij}(t - t_k). \tag{31.58}$$

More typically, the conditional intensity of user i can depend only on the event histories of those nodes 'connected' to it, with connectivity specified by a graph structure, and with $\phi_{ij}(\cdot) = 0$ if there is no edge linking i and j. Alternately, events can have marks indicating whom they are sent to (e.g. receipients of an email), with each event only updating the conditional intensities of its recipients.

Simulating from a Hawkes process is a fairly straightforward extension of simulating from an inhomogeneous Poisson process. Consider the univariate Hawkes process, and suppose the last event occured at time t_i. Then, until the next event occurs, at any time $t > t_i$, we have that $\mathcal{H}(t) = \mathcal{H}(t_i)$, so that the conditional intensity $\lambda(t|\mathcal{H}(t)) = \lambda(t|\mathcal{H}(t_i))$. The next event time, t_{i+1}, is then just the time of the first event of a Poisson process with intensity $\lambda(t|\mathcal{H}(t_i))$ on the interval $[t_i, \infty)$. With most choices of the kernel ϕ, t_{i+1} can easily be simulated, either by integrating $\lambda(t|\mathcal{H}(t_i))$, or by Poisson thinning. At time t_{i+1}, the history is updated to incorporate the new event, and the conditional intensity experiences a jump. The updated intensity is used to simulate the next event at some time $t_{i+2} > t_{i+1}$ from a rate-$\lambda(t|\mathcal{H}(t_{i+1}))$ Poisson process, and the process is repeated until the end of the observation interval. For the case of multivariate Hawkes processes, one has a collection of competing intensities $\lambda_i(t|\{\mathcal{H}_j(t)\}_{j=1}^m)$. The next event is the first event among all events produced by these intensities, after which the intensities are updated and the process is repeated. A realization Ψ from a Hawkes process has log-likelihood

$$\ell(\Psi) = \sum_{t^* \in \Psi} \log \lambda(t^*|\mathcal{H}(t^*)) - \int \lambda(t|\mathcal{H}(t))dt. \tag{31.59}$$

This can typically be evaluated quite easily, so that maximum likelihood estimates of parameters like the base-rate as well as parameters of the excitation kernel can be obtained straightforwardly.

The Hawkes process as described is a fairly simple model, and there have been a number of extensions enriching its structure. An early example is [BBH12], where the authors considered the multivariate Hawkes process, now with an underlying clustering structure. Instead of each individual point process having its own conditional intensity function, each cluster has an intensity function shared by all point process assigned to it. The interaction kernels are also defined at the cluster level,

with an event in process i causing a jump $\phi_{c_i c}(\tau)$ in the intensity function of cluster c (where c_i is the cluster point process i belongs to). The cluster structure was modeled through a Dirichlet process, allowing the authors to learn the underlying clustering, as well as the inter-cluster interaction kernels from interaction data. In [Tan+16], the authors considered **marked** point processes, where event i at time t_i has some associated content y_i (for example, each event is a social media post, and y_i is the text associated with the post at time t_i). The authors allowed the jump in the conditional intensity to depend on the associated mark, with the Hawkes kernel taking the form $\phi(\Delta) = f(y_i)\exp(-\Delta/\tau)$. In that work, the authors modeled the function f with a Gaussian process, though other approaches, such as ones based on neural networks, are possible.

The neural Hawkes process [ME17] is a more fleshed out approach to modeling point processes using neural networks. This models event intensities through the state of a continuous-time LSTM, a modification the more standard discrete-time LSTMs from Section 16.3.4. Central to an LSTM is a memory cell $\mathbf{c}_i$ to store long-term memory, summarizing the past until time step i. Continuous-time LSTMs include two long-term memory cells, $\mathbf{c}_i$ and $\tilde{\mathbf{c}}_i$, summarizing the history until the ith Hawkes event. The first cell represents the starting value to which the intensity jumps after the i'th event, and the second represents a baseline rate after the i'th event. These are both updated after each event, with $\mathbf{c}(t)$, the instantaneous rate at any intermediate time determined by $\mathbf{c}_i$ decaying exponentially towards the baseline $\tilde{\mathbf{c}}_i$. This mechanism allows the intensity at any time to be influenced not just by the number of events in the past, but also the waiting times between them. It can be extended to marked point processes, where each event is also associated with a mark $\mathbf{y}$: now both a learned embedding of the mark as well as the time since the last event is used to update state and long-term memory. For more details, see also Du et al. [Du+16].

31.9.4 Gibbs point process

Gibbs point processes [MW07] from the statistical physics and spatial statistics literature provide a general framework for modeling interacting point processes on higher-dimensional spaces. Such spaces are more challenging than the real line, since there is now no ordering of points, and thus no natural notion of history affecting future activity. Instead, Gibbs point processes use an **energy function** E to quantify deviations from a Poisson process with rate 1. Specifically, under a Gibbs process, the probability density of any configuration Ψ with respect to a rate-1 Poisson process takes the form

$$P_\beta(\Psi) = \frac{1}{Z_\beta}\exp(-\beta E(\Psi)) \tag{31.60}$$

where β is the inverse-temperature parameter, and $Z_\beta = \mathbb{E}_\pi[\exp(-\beta E(\Psi))]$ is the normalization constant (the expectation is with respect to π, the unit-rate Poisson process). Under some conditions on the energy function E, the expectation Z_β is finite, and P_β is a well-defined density, whose integral with respect to π is 1. While the above equation resembles a Markov random field, the domain of Ψ is much more complicated, and evaluating Z_β now involves solving an infinite dimensional integral.

Equation (31.60) states that configurations Ψ for which $E(\Psi)$ is small are more likely than under a Poisson process, with β controlling how peaked this is. The most common energy functions are **pairwise potentials**, taking the form

$$E(\Psi) = \sum_{(s,s')\in\Psi} \phi(\|s - s'\|), \tag{31.61}$$

where the summation is over all pairs of events in Ψ, and $\phi : \mathbb{R}^+ \to \mathbb{R} \cup \infty$. The Strauss process is a specific example, with energy function specified by positive parameters a and R as

$$E(\Psi) = \sum_{s,s' \in \Psi} a \cdot \mathbb{I}\left(\|s - s'\| \leq R\right). \tag{31.62}$$

This is a repulsive process that penalizes configurations with events separated by distance less than R, and as $a \to \infty$ becomes a **hardcore repulsive process**, forbidding configurations with two points separated by less than R. More generally, the energy function can be piecewise-constant, parametrized by a collection of pairs $(a_1, R_1), \ldots, (a_n, R_n)$:

$$E(\Psi) = \sum_{s,s' \in \Psi} \sum_{i=1}^{n} a_i \cdot \mathbb{I}\left(\|s - s'\| \leq R_i\right). \tag{31.63}$$

Another natural option is to use smooth functions like a squared exponential kernel. Gibbs point processes can also involve higher-order interactions, examples being Geyer's triplet point process (which penalizes occurrences of 3 events that are all within some distance R), or area-interaction point processes (that center disks of radius R on each event, calculate the area of the union of these disks, and define E as some function of this area).

While Gibbs point processes are flexible and interpretable point process models, the intractable normalization constant Z_β makes estimating parameters like β a formidable challenge. In practice, these models have to be fit using approximate approaches such as maximizing a pseudo-likelihood function (instead of Equation (31.60)).

31.9.5 Determinantal point process

Determinantal point processes or **DPPs** or DPPs are another approach to modeling repulsion, and have seen considerable popularity in the machine learning literature (see e.g., ([KT+12; Mac75; Bor; LMR15])). Like any point process, a DPP is a probability distribution over subsets of a fixed set $\mathcal{S}$, and in the DPP literature this is often called the **ground set**. Point process applications typically have $\mathcal{S}$ with uncountably infinite cardinality (for example, $\mathcal{S}$ could be the real line), although machine learning applications of DPPs often focus on $\mathcal{S}$ with a finite number of elements. For instance, $\mathcal{S}$ could be a database of images, a collection of news articles, or a group of individuals. A sample from a DPP is a random subset of $\mathcal{S}$, produced, for instance, in response to a search query. The repulsive nature of DPPs ensures diversity and parsimony in the returned subset. This could be useful, for example, to ensure responses to a search query are not minor variations of the same image. Another application is clustering, where a DPP serves as a prior over the number of clusters and their locations, with the repulsiveness discouraging redundant clusters that are very similar to each other.

DPPs are parametrized by a similarity kernel K, whose element K_{ij} gives the similarity between elements i and j of the ground set. For finite ground sets, K is just a similarity matrix. DPPs require K to be positive definite, with largest eigenvalue less than 1, that is $0 \preceq K \preceq \mathbf{I}$. For simplicity, K is often assumed symmetric, though this is not necessary. Given a kernel K, the associated DPP is defined as follows: if Y is the random subset of $\mathcal{S}$ drawn from the DPP, then the probability that Y contains any subset A of $\mathcal{S}$ is given by

$$p(A \subseteq Y) = \det(K_A). \tag{31.64}$$

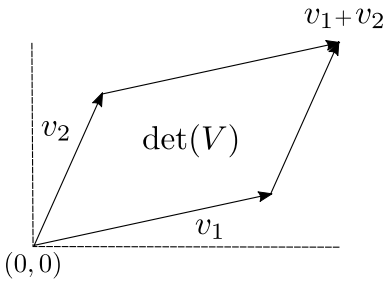


Figure 31.13: The determinant of a matrix V is the volume of the parallelogram spanned by the columns of V and the origin.

Here K_A is the submatrix obtained by restricting K to rows and columns in A, and we define $\det(K_\phi) = 1$ for the empty set ϕ. Observe that this probability is specified exactly, and not just up to a normalization constant. We immediately see that Y contains the empty set with probability one (this is trivially true since the empty set is a subset of every set). We also see that the probability that element i is selected in Y is the ith diagonal element of K:

$$p(i \in Y) = \det(K_{ii}) = K_{ii}, \tag{31.65}$$

so that the diagonal of K gives the inclusion probabilities of the individual elements in $\mathcal{S}$. More interestingly, the probability a pair of elements $\{i, j\}$ are both contained in Y is given by

$$p(\{i, j\} \subseteq Y) = K_{ii}K_{jj} - K_{ij}^2. \tag{31.66}$$

The first term $K_{ii}K_{jj}$ is the probability of including i and j if they were independent, and this probability is adjusted by subtracting K_{ij}^2, a measure of similarity between i and j. This is the source of repulsiveness or diversity in DPPs. More generally, the determinant of a set of vectors is the volume of the parallelogram spanned by them and the origin (see Figure 31.13), and by making the probability of inclusion proportional to the volume, DPPs encourage diversity. We mention for completeness that for uncountable ground sets like a Euclidean space, the determinant $\det(K_A)$ now gives the **product density function** of a realization A. This generalizes the intensity of a Poisson process to account for interactions between point process events, and we refer the interested reader to Lavancier, Møller, and Rubak [LMR15] for more details.

Equation (31.64) characterizes the marginal probabilities of a determinantal point process: what is the probability that a realization Y *contains* some subset of $\mathcal{S}$. More often, one is interested in the probability that the realization Y *equals* some subset of $\mathcal{S}$. The latter is of interest for simulation, inference and parameter learning with DPPs. For this, it is typical to work with a narrower class of DPPs called **L-ensembles** [Bor; KT+12]. Like general DPPs, such a process is characterized by a positive semidefinite matrix L, with the probability that Y equals some configuration A given by:

$$p(Y = A) \propto \det(L_A). \tag{31.67}$$

Note that this probability is specified only upto a normalization constant, and so we do not need to upperbound eigenvalues of $\mathcal{L}$. In fact, it is not hard to show that the normalizer is given by

$(I + \det(L))$, so that

$$p(Y = A) = \frac{\det(L_A)}{I + \det(L)}. \tag{31.68}$$

A similar calculation can be used to show that $p(A \subseteq Y) = \det(K)$, where $K = L(I + L)^{-1} = I - (I + L)^{-1}$, showing that L-ensembles are indeed a special kind of DPP. Equation (31.68) and Equation (31.64) allow parameters of the DPP to be estimated from realizations of a point process, typically by gradient descent. Without additional structure, naively calculating determinants is cubic in the cardinality of $\mathcal{S}$, and this represents a substantial saving when one considers the number of possible subsets of $\mathcal{S}$. When even cubic scaling is too expensive, a number of approximation approaches can be adopted, and these are often closely related to approaches to solve the cubic cost of Gaussian processes.

So far, we have only discussed how to calculate the probability of samples from a DPP. Simulating from a DPP, while straightforward, is a bit less intuitive, and we refer the reader to [LMR15; KT+12]. At a high level, these approaches use the eigenstructure of K to express a DPP as a mixture of **determinantal projection point processes**. The latter are DPPs whose similarity kernel has binary eigenvalues (either 0 or 1) and are easier to sample from. Observe that any eigenvalue λ_i of the similarity kernel K must lie in the interval $[0, 1]$. This allows us to generate a random similarity kernel $\hat{K}$ with the same eigenvectors as K but with binary eigenvalues as follows: for each i, replace eigenvalue λ_i of K with a binary variable $\hat{\lambda}_i$ simulated from a Bernoulli(λ_i) distribution. One can show that the DPP simulated from $\hat{K}$ after simulating $\hat{K}$ from K in this fashion is distributed exactly as a DPP with kernel K. We refer the reader to [LMR15] for details on how to simulate a determinantal projection point process with similarity kernel $\hat{K}$.

32 Representation learning

This chapter was written by Ben Poole and Simon Kornblith.

32.1 Introduction

Representation learning is a paradigm for training machine learning models to transform raw inputs into a form that makes it easier to solve new tasks. Unlike supervised learning, where the task is known at training time, representation learning often assumes that we do not know what task we wish to solve ahead of time. Without this knowledge, are there transformations of the input we can learn that are useful for a variety of tasks we might care about?

One point of evidence that representation learning is possible comes from us. Humans can rapidly form rich representations of new classes [LST15] that can support diverse behaviors: finding more instances of that class, decomposing that instance into parts, and generating new instances from that class. However, it is hard to directly specify what representations we would like our machine learning systems to learn. We may want it make it easy to solve new tasks with small amounts of data, we may want to construct novel inputs or answer questions about similarities between inputs, and we may want the representation to encode certain information while discarding other information.

In building methods for representation learning, the goal is to design a task whose solution requires learning an improved representation of the input instead of directly specifying what the representation should do. These tasks can vary greatly, from building generative models of the dataset to learning to cluster datapoints. Different methods often involve different assumptions on the dataset, different kinds of data, and induce different biases on the learned representation. In this chapter we first discuss methods for evaluating learned representations, then approaches for learning representations based on supervised learning, generative modeling, and self-supervised learning, and finally the theory behind when representation learning is possible.

32.2 Evaluating and comparing learned representations

How can we make sense of representations learned by different neural networks, or of the differences between representations learned in different layers of the same network? Although it is tempting to imagine representations of neural networks as points in a space, this space is high-dimensional. In order to determine the quality of representations and how different representations differ, we need ways to summarize these high-dimensional representations or their relationships with a few

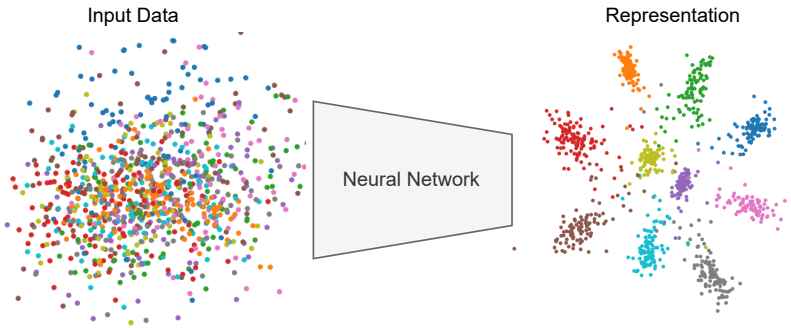


Figure 32.1: Representation learning transforms input data (left) where data from different classes (color) are mixed together to a representation (right) where attributes like class are more easily distinguished. Generated by vib_demo.ipynb.

relevant scalars. Doing so requires making assumptions about what structure in the representations is important.

32.2.1 Downstream performance

The most common way to evaluate the quality of a representation is to adapt it to one or more downstream tasks thought to be representative of real-world scenarios. In principle, one could choose any strategy to adapt the representation, but a small number of adaptation strategies dominate the literature. We discuss these strategies below.

Clearly, downstream performance can only differ from pretraining performance if the downstream task is different from the pretraining task. Downstream tasks can differ from the pretraining task in their input distributions, target distributions, or both. The downstream tasks used to evaluate unsupervised or self-supervised representation learning often involve the same distribution of inputs as the pretraining task, but require predicting targets that were not provided during pretraining. For example, in self-supervised visual representation learning, representations learned on the ImageNet dataset without using the accompanying labels are evaluated on ImageNet using labels, either by performing linear evaluation with all the data or by fine-tuning using subsets of the data. By contrast, in **transfer learning** (Section 19.5.1), the input distribution of the downstream task differs from the distribution of the pretraining task. For example, we might pretrain the representation on a large variety of natural images and then ask how the representation performs at distinguishing different species of birds not seen during pretraining.

32.2.1.1 Linear classifiers and linear evaluation

Linear evaluation treats the trained neural network as a fixed feature extractor and trains a linear classifier on top of fixed features extracted from a chosen network layer. In earlier work, this linear classifier was often a support vector machine [Don+14; SR+14; Cha+14], but in more recent work, it is typically an L^2-regularized multinomial logistic regression classifier [ZIE17; KSL19; KZB19]. The process of training this classifier is equivalent to attaching a new layer to the chosen

representation layer and training only this new layer, with the rest of the network's weights frozen and any normalization/regularization layers set to "inference mode" (see Figure 32.2).

Although linear classifiers are conceptually simple compared to deep neural networks, they are not necessarily simple to train. Unlike deep neural network training, objectives associated with commonly-used linear classifiers are convex and thus it is possible to find global minima, but it can be challenging to do so with stochastic gradient methods. When using SGD, it is important to tune both the learning rate schedule and weight decay. Even with careful tuning, SGD may still require substantially more epochs to converge when training the classifier than when training the original neural network [KZB19]. Nonetheless, linear evaluation with SGD remains a commonly used approach in the representation learning literature.

When it is possible to maintain all features in memory simultaneously, it is possible to use full-batch optimization method with line search such as L-BFGS in place of SGD [KSL19; Rad+21]. These optimization methods ensure that the loss decreases at every iteration of training, and thus do not require manual tuning of learning rates. To obtain maximal accuracy, it is still important to tune the amount of regularization, but this can be done efficiently by sweeping this hyperparameter and using the optimal weights for the previous value of the hyperparameter as a warm start. Using a full-batch optimizer typically implies precomputing the features before performing the optimization, rather than recomputing features on each minibatch. Precomputing features can save a substantial amount of computation, since the forward passes through the frozen model are typically much more expensive than computing the gradient of the linear classifier. However, precomputing features also limits the number of augmentations of each example that can be considered.

It is important to keep in mind that the accuracy obtainable by training a linear classifier on a finite dataset is only a lower bound on the accuracy of the Bayes-optimal linear classifier. The datasets used for linear evaluation are often small relative to the number of parameters to be trained, and the classifier typically needs to be regularized to obtain maximal accuracy. Thus, linear evaluation accuracy depends not only on whether it is possible to linearly separate different classes in the representation, but also on how much data is required to find a good decision boundary with a given training objective and regularizer. In practice, even an invertible linear transformation of a representation can affect linear evaluation accuracy.

32.2.1.2 Fine-tuning

It is also possible to adapt all layers from the pretraining task to the downstream task. This process is typically referred to as **fine-tuning** [HS06b; Gir+14]. In its simplest form, fine-tuning, like linear evaluation, involves attaching a new layer to a chosen representation layer, but unlike linear evaluation, all network parameters, and not simply those of the new layer, are updated according to gradients computed on the downstream task. The new layer may be initialized with zeros or using the solution obtained by training it with all other parameters frozen. Typically, the best results are obtained when the network is fine-tuned at a lower learning rate than was used for pretraining.

Fine-tuning is substantially more expensive than training a linear classifier on top of fixed feature representations, since each training step requires backpropagating through multiple layers. However, fine-tuned networks typically outperform linear classifiers, especially when the pretraining and downstream tasks are very different [KSL19; AGM14; Cha+14; Azi+15]. Linear classifiers perform better only when the number of training examples is very small (~5 per class) [KSL19].

Fine-tuning can also involve adding several new network layers. For detection and segmentation

tasks, which require fine-grained knowledge of spatial position, it is common to add a *feature pyramid network* (FPN) [Lin+17b] that incorporates information from different feature maps in the pretrained network. Alternatively, new layers can be interspersed between old layers and initialized to preserve the network's output. *Net2Net* [CGS15] follows this approach to construct a higher-capacity network that makes use of representations contained in the pretrained weights of a smaller network, whereas *adapter modules* [Hou+19] incorporate new, parameter-efficient modules into a pretrained network and freeze the old ones to reduce the number of parameters that need to be stored when adapting models to different tasks.

32.2.1.3 Disentanglement

Given knowledge about how a dataset was generated, for example that there are certain factors of variation such as position, shape, and color that generated the data, we often wish to estimate how well we can recover those factors in our learned representation. This requires using disentangled representation learning methods (see Section 21.3.1.1). While there are a variety of metrics for disentanglement, most measure to what extent there is a one-to-one correspondence between latent factors and dimensions of the learned representation.

32.2.2 Representational similarity

Rather than measure compatibility between a representation and a downstream task, we might seek to directly examine relationships between two fixed representations without reference to a task. In this section, we assume that we have two sets of fixed representations corresponding to the same n examples. These representations could be extracted from different layers of the same network or layers of different neural networks, and need not have the same dimensionality. For notational convenience, we assume that each set of representations has been stacked row-wise to form matrices $\boldsymbol{X} \in \mathbb{R}^{n \times p_1}$ and $\boldsymbol{Y} \in \mathbb{R}^{n \times p_2}$ such that $\boldsymbol{X}_{i,:}$ and $\boldsymbol{Y}_{i,:}$ are two different representations of the same example.

32.2.2.1 Representational similarity analysis and centered kernel alignment

Representational similarity analysis (RSA) is the dominant technique for measuring similarity of representations in neuroscience [KMB08], but has also been applied in machine learning. RSA reduces the problem of measuring similarity between representation matrices to measuring the similarities between representations of individual examples. RSA begins by forming representational similarity matrices (RSMs) from each representation. Given functions $k : \mathcal{X} \times \mathcal{X} \mapsto \mathbb{R}$ and $k' : \mathcal{Y}' \times \mathcal{Y}' \mapsto \mathbb{R}$ that measure the similarity between pairs of representations of individual examples $\boldsymbol{x}, \boldsymbol{x}' \in \mathcal{X}$, and $\boldsymbol{y}, \boldsymbol{y}' \in \mathcal{Y}$, the corresponding representational similarity matrices $\boldsymbol{K}, \boldsymbol{K}' \in \mathbb{R}^{n \times n}$ contain the similarities between the representations of all pairs of examples $\boldsymbol{K}_{ij} = k(\boldsymbol{X}_{i,:}, \boldsymbol{X}_{j,:})$ and $\boldsymbol{K}'_{ij} = k'(\boldsymbol{Y}_{i,:}, \boldsymbol{Y}_{j,:})$. These representational similarity matrices are transformed into a scalar similarity value by applying a matrix similarity function $s(\boldsymbol{K}, \boldsymbol{K}')$.

The RSA framework can encompass many different forms of similarity through the selection of the similarity functions $k(\cdot, \cdot)$, $k'(\cdot, \cdot)$, and $s(\cdot, \cdot)$. How these functions should be selected is a contentious topic [BS+20; Kri19]. In practice, it is common to choose $k(\boldsymbol{x}, \boldsymbol{x}') = k'(\boldsymbol{x}, \boldsymbol{x}') = \text{corr}[\boldsymbol{x}, \boldsymbol{x}']$, the Pearson correlation coefficient between examples. $s(\cdot, \cdot)$ is often chosen to be the Spearman rank correlation between the representational similarity matrices, which is computed by reshaping $\boldsymbol{K}$ and

$\boldsymbol{K}'$ to vectors, computing the rankings of their elements, and measuring the Pearson correlation between these rankings.

Centered kernel alignment (CKA) is a technique that was first proposed in machine learning literature [Cri+02; CMR12] but that can be interpreted as a form of RSA. In centered kernel alignment, the per-example similarity functions k and k' are chosen to be positive semi-definite kernels so that $\boldsymbol{K}$ and $\boldsymbol{K}'$ are kernel matrices. The matrix similarity function s is the cosine similarity between centered kernel matrices

$$s(\boldsymbol{K}, \boldsymbol{K}') = \frac{\langle \boldsymbol{HKH}, \boldsymbol{HK'H} \rangle_{\mathrm{F}}}{\|\boldsymbol{HKH}\|_{\mathrm{F}} \|\boldsymbol{HK'H}\|_{\mathrm{F}}}, \tag{32.1}$$

where $\langle \boldsymbol{A}, \boldsymbol{B} \rangle_{\mathrm{F}} = \mathrm{vec}(\boldsymbol{A})^\top \mathrm{vec}(\boldsymbol{B}) = \mathrm{tr}(\boldsymbol{A}^\top \boldsymbol{B})$ is the Frobenius product, and $\boldsymbol{H} = \boldsymbol{I} - \frac{1}{n}\boldsymbol{1}\boldsymbol{1}^\top$ is the centering matrix. As it is applied above, the centering matrix subtracts the means from the rows and columns of the similarity index.

A special case of centered kernel alignment arises when k and k' are chosen to be the linear kernel $k(\boldsymbol{x}, \boldsymbol{x}') = k'(\boldsymbol{x}, \boldsymbol{x}') = \boldsymbol{x}^\top \boldsymbol{x}'$. In this case, $\boldsymbol{K} = \boldsymbol{X}\boldsymbol{X}^\top$ and $\boldsymbol{K}' = \boldsymbol{Y}\boldsymbol{Y}^\top$, allowing for an alternative expression for CKA in terms of the similarities between pairs of *features* rather than pairs of examples. The representations themselves must first be centered by subtracting the means from their columns, yielding $\tilde{\boldsymbol{X}} = \boldsymbol{HX}$ and $\tilde{\boldsymbol{Y}} = \boldsymbol{HY}$. Then, so-called linear centered kernel alignment is given by

$$s(\boldsymbol{K}, \boldsymbol{K}') = \frac{\langle \tilde{\boldsymbol{X}}\tilde{\boldsymbol{X}}^\top, \tilde{\boldsymbol{Y}}\tilde{\boldsymbol{Y}}^\top \rangle_{\mathrm{F}}}{\|\tilde{\boldsymbol{X}}\tilde{\boldsymbol{X}}^\top\|_{\mathrm{F}} \|\tilde{\boldsymbol{Y}}\tilde{\boldsymbol{Y}}^\top\|_{\mathrm{F}}} = \frac{\|\tilde{\boldsymbol{X}}^\top \tilde{\boldsymbol{Y}}\|_{\mathrm{F}}^2}{\|\tilde{\boldsymbol{X}}^\top \tilde{\boldsymbol{X}}\|_{\mathrm{F}} \|\tilde{\boldsymbol{Y}}^\top \tilde{\boldsymbol{Y}}\|_{\mathrm{F}}}. \tag{32.2}$$

Linear centered kernel alignment is equivalent to the RV coefficient [RE76] between centered features, as shown in [Kor+19].

32.2.2.2 Canonical correlation analysis and related methods

Given two datasets (in this case, the representation matrices $\boldsymbol{X}$ and $\boldsymbol{Y}$), canonical correlation analysis or CCA (Section 28.3.4.3) seeks to map both datasets to a shared latent space such that they are maximally correlated in this space. The i^{th} pair of canonical weights $(\boldsymbol{w}_i, \boldsymbol{w}_i')$ maximize the correlation between the corresponding canonical vectors $\rho_i = \mathrm{corr}[\boldsymbol{X}\boldsymbol{w}_i, \boldsymbol{Y}\boldsymbol{w}_i']$ subject to the constraint that the new canonical vectors are orthogonal to previous canonical vectors,

$$\begin{aligned} &\text{maximize } \mathrm{corr}[\boldsymbol{X}\boldsymbol{w}_i, \boldsymbol{Y}\boldsymbol{w}_i'] \\ &\text{subject to } \quad \forall_{j<i} \;\; \boldsymbol{X}\boldsymbol{w}_i \perp \boldsymbol{X}\boldsymbol{w}_j \\ &\qquad\qquad\;\; \forall_{j<i} \;\; \boldsymbol{Y}\boldsymbol{w}_i' \perp \boldsymbol{Y}\boldsymbol{w}_j' \\ &\qquad\qquad\;\; \|\boldsymbol{X}\boldsymbol{w}_i\| = \|\boldsymbol{Y}\boldsymbol{w}_i'\| = 1. \end{aligned} \tag{32.3}$$

The maximum number of non-zero canonical correlations is the minimum of the ranks, $p = \min(\mathrm{rk}(\boldsymbol{X}), \mathrm{rk}(\boldsymbol{Y}))$.

The standard algorithm for computing the canonical weights and correlations [BG73] first decomposes the individual representations as the product of an orthogonal matrix and a second matrix, $\tilde{\boldsymbol{X}} = \boldsymbol{QR} : \boldsymbol{Q}^\top \boldsymbol{Q} = \boldsymbol{I}$ and $\tilde{\boldsymbol{Y}} = \boldsymbol{Q}'\boldsymbol{R}' : \boldsymbol{Q}'^\top \boldsymbol{Q}' = \boldsymbol{I}$. These decompositions can be obtained either by QR factorization or singular value decomposition. A second singular value decomposition of $\boldsymbol{Q}^\top \boldsymbol{Q}' = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^\top$ provides the quantities needed to determine the canonical weights and correlations.

Specifically, the canonical correlations are the singular values $\boldsymbol{\rho} = \text{diag}(\boldsymbol{\Sigma})$; the canonical vectors are the columns of $\boldsymbol{XW} = \boldsymbol{QU}$ and $\boldsymbol{YW'} = \boldsymbol{Q'V}$; and the canonical weights are $\boldsymbol{W} = \boldsymbol{R}^{-1}\boldsymbol{U}$ and $\boldsymbol{W'} = \boldsymbol{R'}^{-1}\boldsymbol{V}$.

Two common strategies to turn the resulting vector of canonical correlations into a scalar are to take the **mean squared canonical correlation**,

$$R^2_{\text{CCA}}(\boldsymbol{X}, \boldsymbol{Y}) = \|\boldsymbol{\rho}\|_2^2/p = \|\boldsymbol{Q}^\top \boldsymbol{Q'}\|_{\text{F}}^2/p, \tag{32.4}$$

or the **mean canonical correlation**,

$$\bar{\rho} = \sum_{i=1}^{p} \boldsymbol{\rho}_i/p = \|\boldsymbol{Q}^\top \boldsymbol{Q'}\|_*/p, \tag{32.5}$$

where $\|\cdot\|_*$ denotes the nuclear norm. The mean squared canonical correlation has several alternative names, including Yanai's GCD [Yan74; RBS84], Pillai's trace, or the eigenspace overlap score [May+19].

Although CCA is a powerful tool, it suffers from the curse of dimensionality. If at least one representation has more neurons than examples and each neuron is linearly independent of the others, then all canonical correlations are 1. In practice, because neural network representations are high-dimensional, we can find ourselves in the regime where there are not enough data to accurately estimate the canonical correlations. Moreover, even when we can accurately estimate the canonical correlations, it may be desirable for a similarity measure to place less emphasis on the similarity of low-variance directions.

Singular vector CCA (SVCCA) mitigates these problems by retaining only the largest principal components of $\tilde{\boldsymbol{X}}$ and $\tilde{\boldsymbol{Y}}$ when performing CCA. Given the singular value decomposition of the representations $\tilde{\boldsymbol{X}} = \boldsymbol{U\Sigma V}^\top$ and $\tilde{\boldsymbol{Y}} = \boldsymbol{U'\Sigma' V'}^\top$, SVCCA retains only the first k columns of $\boldsymbol{U}$ corresponding to the largest k singular values of $\boldsymbol{\sigma} = \text{diag}(\boldsymbol{\Sigma})$ (i.e., the k largest principal components) and the first k' columns of $\boldsymbol{U'}$ corresponding to the largest k' singular values $\boldsymbol{\sigma'} = \text{diag}(\boldsymbol{\Sigma'})$. With these singular value decompositions, the canonical correlations, vectors, and weights can then be computed using the algorithm of Björck and Golub [BG73] described above, by setting

$$\boldsymbol{Q} = \boldsymbol{U}_{:,1:k}, \qquad \boldsymbol{R} = \boldsymbol{\Sigma}_{1:k,1:k}\boldsymbol{V}^\top_{:,1:k}, \qquad \boldsymbol{Q'} = \boldsymbol{U'}_{:,1:k}, \qquad \boldsymbol{R'} = \boldsymbol{\Sigma'}_{1:k,1:k}\boldsymbol{V'}^\top_{:,1:k}. \tag{32.6}$$

Raghu et al. [Rag+17] suggest retaining enough components to explain 99% of the variance, i.e., setting k to the minimum value such that $\|\boldsymbol{\sigma}_{1:k}\|^2/\|\boldsymbol{\sigma}\|^2 = 0.99$ and k' to the minimum value such that $\|\boldsymbol{\sigma'}_{1:k'}\|^2/\|\boldsymbol{\sigma'}\|^2 = 0.99$. However, for a fixed value of $\min(k, k')$, CCA-based similarity measures increase with the value of $\max(k, k')$. Representations that require more components to explain 99% of the variance of representations may thus appear "more similar" to all other representations than representations with more rapidly decaying singular value spectra. In practice, it is often better to set k and k' to the same fixed value.

Projection-weighted CCA (PWCCA) [MRB18] instead weights the correlations by a measure of the variability in the original representation that they explain. The resulting similarity measure is

$$\text{PWCCA}(\boldsymbol{X}, \boldsymbol{Y}) = \frac{\sum_{i=1}^{p} \alpha_i \rho_i}{\sum_{i=1}^{p} \alpha_i}, \qquad \alpha_i = \|(\boldsymbol{Y}\boldsymbol{w}_i')^\top \boldsymbol{Y}\|_1. \tag{32.7}$$

PWCCA enjoys somewhat widespread use in representational similarity literature, but it has potentially undesirable properties. First, it is asymmetric; its value depends on which of the two representations is used for the weighting. Second, it is unclear why weightings should be determined using the L^1 norm, which is not invariant to rotations of the representation. It is arguably more intuitive to weight the correlations directly by the amount of variance in the representation that the canonical component explains, which corresponds to replacing the L^1 norm with the squared L^2 norm. The resulting similarity measure is

$$R^2_{\text{LR}}(\boldsymbol{X}, \boldsymbol{Y}) = \frac{\sum_{i=1}^{p} \alpha'_i \rho_i^2}{\sum_{i=1}^{p} \alpha'_i}, \qquad \alpha'_i = \sum_{j=1}^{p_2} \|(\boldsymbol{Y}\boldsymbol{w}'_i)^\top \boldsymbol{Y}\|_2^2. \tag{32.8}$$

As shown by Kornblith et al. [Kor+19], when $p_2 \geq p_1$, this alternative similarity measure is equivalent to the overall variance explained by using linear regression to fit every neuron in $\tilde{\boldsymbol{Y}}$ using the representation $\tilde{\boldsymbol{X}}$,

$$R^2_{\text{LR}}(\boldsymbol{X}, \boldsymbol{Y}) = 1 - \sum_{i=1}^{p_2} \min_{\boldsymbol{\beta}} \|\tilde{\boldsymbol{Y}}_{:,i} - \tilde{\boldsymbol{X}}\boldsymbol{\beta}\|^2 / \|\tilde{\boldsymbol{Y}}\|_{\text{F}}^2. \tag{32.9}$$

Finally, there is also a close relationship between CCA and linear CKA. This relationship can be clarified by writing similarity indexes directly in terms of the singular value decompositions $\tilde{\boldsymbol{X}} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^\top$ and $\tilde{\boldsymbol{Y}} = \boldsymbol{U}'\boldsymbol{\Sigma}'\boldsymbol{V}'^\top$. The left-singular vectors $\boldsymbol{u}_i = \boldsymbol{U}_{:,i}$ correspond to the principal components of $\boldsymbol{X}$ normalized to unit length, and the squared singular values $\lambda_i = \Sigma_{ii}^2$ are the amount of variance that those principal components explain (up to a factor of $1/n$). Given these singular value decompositions, R^2_{CCA}, R^2_{LR}, and linear CKA become:

$$R^2_{\text{CCA}}(\boldsymbol{X}, \boldsymbol{Y}) = \sum_{i=1}^{p_1} \sum_{j=1}^{p_2} \left(\boldsymbol{u}_i^\top \boldsymbol{u}'_j\right)^2 / p_1 \tag{32.10}$$

$$R^2_{\text{LR}}(\boldsymbol{X}, \boldsymbol{Y}) = \sum_{i=1}^{p_2} \sum_{j=1}^{p_2} \lambda'_j \left(\boldsymbol{u}_i^\top \boldsymbol{u}'_j\right)^2 / \sum_{j=1}^{p_2} \lambda'_j \tag{32.11}$$

$$\text{CKA}_{\text{linear}}(\boldsymbol{X}, \boldsymbol{Y}) = \frac{\sum_{i=1}^{p_1} \sum_{j=1}^{p_2} \lambda_i \lambda'_j \left(\boldsymbol{u}_i^\top \boldsymbol{u}'_j\right)^2}{\sqrt{\sum_{i=1}^{p_1} \lambda_i^2} \sqrt{\sum_{j=1}^{p_2} {\lambda'_j}^2}}. \tag{32.12}$$

Thus, these similarity indexes all involve the similarities between all pairs of principal components from $\boldsymbol{X}$ and $\boldsymbol{Y}$, but place different weightings on these similarities according to the fraction of variance that these principal components explain.

32.2.2.3 Comparing representational similarity measures

What properties are desirable in a representational similarity measure is an open question, and this question may not have a unique answer. Whereas evaluations of downstream accuracy approximate real-world use cases for neural network representations, the goal of representational similarity is instead to develop understanding of how representations evolve across neural networks, or how they differ between neural networks with different architectures or training settings.

One way to taxonomize different similarity measures is through the transformations of a representation that they are invariant to. The minimum form of invariance is invariance to permutation of a representation's constituent neurons, which is needed because neurons in neural networks generally have no canonical ordering: For commonly-used initialization strategies, any permutation of a given initialization is equiprobable, and nearly all architectures and optimizers produce training trajectories that are equivariant under permutation. On the other hand, invariance to arbitrary invertible transformations, as provided by mutual information, is clearly undesirable, since many realistic neural networks are injective functions of the input [Gol+19] and thus there always exists an invertible transformation between any pair of representations. In practice, most similarity measures in common use are invariant to rotations (orthogonal transformations) of representations, which implies invariance to permutation. Similarity measures based solely on CCA correlations, such as R^2_{CCA} and $\bar{\rho}$, are invariant to all invertible linear transformations of representations. However, SVCCA and PWCCA are not.

A different way to distinguish similarity measures is to investigate situations where we know the relationships among representations and to empirically evaluate their ability to recover these relationships. Kornblith et al. [Kor+19] propose a simple "sanity check": Given two architecturally identical networks A and B trained from different random initializations, any layer in network A should be more similar to the architecturally corresponding layer in network B than to any other layer. They show that, when considering flattened representations of CNNs, similarity measures based on centered kernel alignment satisfy this sanity check whereas other similarity measures do not. By contrast, when considering representations of individual tokens in Transformers, all similarity measures perform reasonably well. However, Maheswaranathan et al. [Mah+19] show that both CCA and linear CKA are highly sensitive to seemingly innocuous RNN design choices such as the activation function, even though analysis of the fixed points of the dynamics of different networks suggests they all operate similarly.

32.3 Approaches for learning representations

The goal of representation learning is to learn a transformation of the inputs that makes it easier to solve future tasks. Typically the tasks we want the representaiton to be useful for are not known when learning the representation, so we cannot directly train to improve performance on the task. Learning such generic representations requires collecting large-scale unlabeled or weakly-labeled datasets, and identifying tasks or priors for the representations that one can solve without direct access to the downstream tasks. Most methods focus on learning a parametric mapping $z = f_\theta(x)$ that takes an input x and transforms it into a representation z using a neural network with parameters θ.

The main challenge in representation learning is coming up with a task that requires learning a good representation to solve. If the task is too easy, then it can be solved without learning an interesting transformation of the inputs, or by learning a *shortcut*. If a task is too different from the downstream task that the representation will be evaluated on, then the representation may also not be useful. For example, if the downstream task is object detection, then the representation needs to encode both the identity and location of objects in the image. However, if we only care about classification, then the representation can discard information about position. This leads to approaches for learning representations that are often not generic: different training tasks may perform better for different downstream tasks.

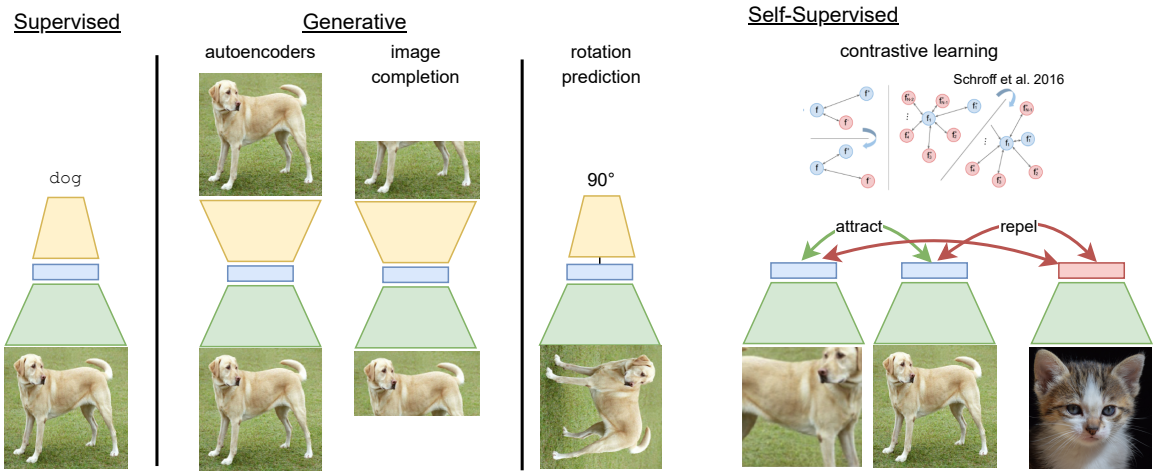


Figure 32.2: Approaches for representation learning from images. An input image is encoded through a deep neural network (green) to produce a representation (blue). An additional shallow or deep neural network (yellow) is often used to train the representation, but is thrown out after the representation is learned when solving downstream tasks. In the supervised case, the mapping from the representation to logits is typically linear, while for autoencoders the mapping from representation to images can be highly complex and stochastic. Unlike supervised or generative approaches, contrastive methods rely on other data points in the form of positive pairs (often created through data augmentation) and negative pairs (typically other datapoints) to learn a representation.

In Figue 32.2, we outline three approaches we will discuss for representation learning. **Supervised** approaches train on large-scale supervised or weakly-supervised data using standard supervised losses. **Generative** approaches aim to learn generative models of the dataset or parts of a dataset. **Self-supervised** approaches are based on transformation prediction or multi-view learning, where we design a task that where labels can be easily synthesized without needing human input.

32.3.1 Supervised Representation Learning and Transfer

The first major successes in visual representation learning with deep learning came from networks trained on large labeled datasets. Following the discovery that supervised deep neural networks could outperform classical computer vision models for natural image classification [KSH12b; CMS12], it became clear that the representations learned by these networks could outperform handcrafted features used across a wide variety of tasks [Don+14; SR+14; Oqu+14; Gir+14]. Although unsupervised visual representation learning has recently achieved competitive results on many domains, supervised representation learning remains the dominant approach.

Larger networks trained on larger datasets generally achieve better performance on both pretraining and downstream tasks. When other design choices are held fixed, architectures that achieve higher accuracy during pretraining on natural image datasets such as ImageNet also learn better representations for downstream natural image tasks, as measured by both linear evaluation and

fine-tuned accuracy [KSL19; TL19; Zha+19a; Zha+21a; Abn+21]. However, when the domain shift from the pretraining task to the downstream task becomes larger (e.g., from ImageNet to medical imaging), the correlation between pretraining and downstream accuracy can be much lower [Rag+19; Ke+21; Abn+21]. Studies that vary pretraining dataset size generally find that larger pretraining datasets yield better representations for downstream tasks [HAE16; Mah+18; Kol+20; Zha+21a; Abn+21], although there is an interaction between model size and dataset. When training small models with the intention of transferring to a specific downstream task, it is sometimes preferable to pretrain on a smaller dataset that is more closely related to that task rather than a larger dataset that is less closely related [Cui+18; Mah+18; Ngi+18; Kol+20], but larger models seem to derive greater benefit from larger, more diverse datasets [Mah+18; Kol+20].

Whereas scaling the architecture and dataset size generally improves both pretraining and downstream accuracy, other design choices can improve pretraining accuracy at the expense of transfer, or vice versa. Regularizers such as penultimate layer dropout and label smoothing improve accuracy on pretraining tasks but produce worse representations for downstream tasks [KSL19; Kor+21]. Although most convolutional neural networks are trained with batch normalization, Kolesnikov et al. [Kol+20] find that the combination of group normalization and weight standardization leads to networks that perform similarly on pretraining tasks but substantially better on transfer tasks. Adversarial training produces networks that perform worse on pretraining tasks as compared to standard training, but these representations transfer better to other tasks [Sal+20]. For certain combinations of pretraining and downstream datasets, increasing the amount of weight decay on the network's final layer can improve transferability at the cost of pretraining accuracy [Zha+21a; Abn+21].

The challenge of collecting ever-larger pretraining datasets has led to the emergence of **weakly-supervised representation learning**, which eschews the expensive human annotations of datasets such as ImageNet and instead relies on data that can be readily collected from the Internet, but which may have greater label noise. Supervision sources include hashtags accompanying images on websites such as Instagram and Flickr [CG15; Iza+15; Jou+16; Mah+18], image labels obtained automatically using proprietary algorithms involving user feedback signals [Sun+17; Kol+20], or image captions/alt text [Li+17a; SPL20; DJ21; Rad+21; Jia+21]. Hashtags and automatic labeling give rise to image classification problems that closely resemble their more strongly supervised counterparts. The primary difference versus standard supervised representation learning is that the data are noisier, but in practice, the benefits of more data often outweigh the detrimental effects of the noise.

Image-text supervision has provided more fertile ground for innovation, as there are many different ways of jointly processing text and images. The simplest approach is again to convert the data into an image classification problem, where the network is trained to predict which words or n-grams appear in the text accompanying a given image [Li+17a]. More sophisticated approaches train image-conditional language models [DJ21] or masked language models [SPL20], which can make better use of the structure of the text. Recently, there has been a surge in interest in *contrastive* image/text pretraining models such as CLIP [Rad+21] and ALIGN [Jia+21], details of which we discuss in Section 32.3.4. These models process images and text independently using two separate "towers," and learn an embedding space where embeddings of images lie close to the embeddings of the corresponding text. As shown by Radford et al. [Rad+21], contrastive image/text pretraining learns high-quality representations faster than alternative approaches.

Beyond simply learning good visual representations, pretrained models that embed image and text in a common space enable **zero-shot transfer** of learned representations. In zero-shot transfer, an

image classifier is constructed using only textual descriptions of the classes of interest, without any images from the downstream task. Early co-embedding models relied on pretrained image models and word embeddings that were then adapted to a common space [Fro+13], but contrastive image/text pretraining provides a means to learn co-embedding models end-to-end. Compared to linear classifiers trained using image embeddings, zero-shot classifiers typically perform worse, but zero-shot classifiers are far more robust to distribution shift [Rad+21].

32.3.2 Generative Representation Learning

Supervised representation learning often fails to learn representations for tasks that differ significantly from the task the representation was trained on. How can we learn representations when the task we wish to solve differs a lot from tasks where we have large labeled datasets?

Generative representation learning aims to model the entire distribution of a dataset $q(x)$ with a parametric model $p_\theta(x)$. The hope of generative representation learning is that. if we can build models that can create all the data that we have seen, then we implicitly may learn a representation that can be used to answer any question about the data, not just the questions that are related to a supervised task for which we have labels. For example, in the case of digit classification, it is hard to collect labels for the style of a handwritten digit, but if the model has to product all possible handwritten digits in our dataset it needs to learn to produce digits with different styles. On the other hand, supervised learning to classify digits aims to learn a representation that is invariant to style.

There are two main approaches for learning representations with generative models: (1) latent-variable models that aim to capture the underlying factors of variation in data with latent variables z that act as the representation (see the chapter on VAEs, Chapter 21), and (2) fully-observed models where a neural architecture is trained with a tractable generative objective (see the chapters on AR models, Chapter 22, and flow models, Chapter 23), and then a representation is extracted from the learned architecture.

32.3.2.1 Latent-variable models

One criterion for learning a good representation of the world is that it is useful for synthesizing observed data. If we can build a model that can create new observations, and has a simple set of latent variables, then hopefully this model will learn variables that are related to the underlying physical process that created the observations. For example, if we are trying to model a dataset of 2d images of shapes, knowing the position, size, and type of the shape would enable easy synthesis of the image. This approach to learning is known as **analysis-by-synthesis**, and is a theory of perception that aims at identifying a set of underlying latent factors (analysis) that could be used to synthesize observations [Rob63; Bau74; LM03]. Our goal is to learn a generative model $p_\theta(x, z)$ over the observations x and latents z, with parameters θ. Given an observation x, performing the analysis step to extract a representation requires running inference to sample or compute the posterior mean of $p_\theta(z|x)$. Different choices for the model $p_\theta(x, z)$ and inference procedure for $p_\theta(z|x)$ represent different ways of learning representations from a dataset.

Early work on deep latent-variable generative models aimed to learn stacks of features often based on training simple energy-based models or directed sparse coding models, each of which could explain the previous set of latent factors, and which learned increasingly abstract representation [HOT06b;

Lee+09; Ran+06]. Bengio, Courville, and Vincent [BCV13] provides an overview of several methods based on stacking latent-variable generative modeling approaches to learn increasingly abstract representation. However greedy approaches to generative representation learning have failed to scale to larger natural datasets.

If the generative process that created the data is simple and can be described, then encoding that structure into a generative model is a tremendously powerful way of learning useful and robust representations. Lake, Salakhutdinov, and Tenenbaum [LST15] and George et al. [Geo+17] use knowledge of how characters are composed of strokes to build hierarchical generative models with representations that excel at several downstream tasks. However, for many real-world datasets the generative structure is not known, and the generative model must also be learned. There is often a tradeoff between imposing structure in the generative process (such as sparsity) vs. learning that structure from data.

Directed latent-variable generative models have proven easier to train and scale to natural datasets. Variational autoencoders (Chapter 21) train a directed latent-variable generative model with variational inference, and learn a prior $p_\theta(z)$, decoder $p_\theta(x|z)$, and an amortized inference network $q_\phi(z|x)$ that can be used to extract a representation on new datapoints. Higgins et al. [Hig+17b] show β-VAEs (Section 21.3.1) are capable of learning latent variables that correspond to factors of variation on simple synthetic datasets. Kingma et al. [Kin+14b] and Rasmus et al. [Ras+15] demonstrate improved performing on semi-supervised learning with VAEs. While there have been several recent advances to scale up VAEs to natural datasets [VK20b; Chi21b], none of these methods have yet led to representations that are competitive for downstream tasks such as classification or segmentation.

Adversarial methods for training directed latent-variable models have also proven useful for representation learning. In particular, GANs (Chapter 26) trained with encoders such as BiGAN [DKD17], ALI [Dum+17], and [Che+16] were able to learn representations on small scale datasets that performed well at object classification. The discriminators from GANs have also proven useful for learning representations [RMC16b]. More recently, these methods were scaled up to ImageNet in BigBiGAN [DS19], with learned representations that performed strongly on classification and segmentation tasks.

32.3.2.2 Fully observed models

The neural network architectures used in fully observed generative models can also learn useful representations without the presence of latent-variables. ImageGPT [Che+20a] demonstrate that an autoregressive model trained on pixels can learn internal representations that excel at image classification. Unlike with latent-variable models where the representation is often thought of as the latent variables, ImageGPT extracted representations from the deterministic layers of the transformer architecture used to compute future tokens. Similar approaches have shown progress for learning features in language modeling [Raf+20b], however alternative objectives, based on masked training (as in BERT, [Dev+19]), often leads to better performance.

32.3.2.3 Autoencoders

A related set of methods for representation learning are based on learning a representation from which the original data can be reconstructed. These methods are often called **autoencoders** (see Section 16.3.3), as the data is encoded in a way such that the input data itself can be recreated.

However, unlike generative models, they cannot typically be used to synthesize observations from scratch or assign likelihoods to observations. Autoencoders learn an encoder that outputs a representation $z = f_\theta(x)$, and a decoder $g_\phi(z)$ that takes the representation z and tries to recreate the input data, x. The quality of the approximate reconstruction , $\hat{x} = g_\phi(z)$ is often measured using a domain-specific loss, for example mean-squared error for images:

$$\mathcal{L}(\theta, \phi) = \frac{1}{|\mathcal{D}|} \sum_{x \in \mathcal{D}} \|x - g_\phi(f_\theta(x))\|_2^2. \tag{32.13}$$

If there are no constraints on the encoder or decoder, and the dimensionality of the representation z matches the dimensionality of the input x, then there exists a trivial solution to minimize the autoencoding objective: set both f_θ and g_ϕ to identity functions. In this case the representation has not learned anything interesting, and thus in practice an additional regularizer is often placed on the learned representation.

Reducing the dimensionality of the representation z is one effective mechanism to avoid trivial solutions to the autoencoding objective. If both the encoder and decoder networks are linear, and the loss is mean-squared-error, then the resulting linear autoencoder model can learn the principal components of a dataset [Pla18].

Other methods maintain higher-dimensional representations by adding sparsity (for example, penalties on $\|z\|_1$ in Ng et al. [Ng+11]) or smoothness regularizers [Rif+11], or adding noise to the input [Vin+08] or intermediate layers of the network [Sri+14b; PSDG14]. These added regularizers aim to bias the encoder and decoder to learn representations that are not just the identity function, but instead are nonlinear transformations of the input that may be useful for downstream tasks. See Bengio, Courville, and Vincent [BCV13] for a more detailed discussion of regularized autoencoders and their applications. A recent re-evaluation of several algorithms based on iteratively learning features by stacked regularized autoencoders have been shown to degrade performance versus training end-to-end from scratch [Pai+14]. However, we will see in Section 32.3.3.1 that denoising autoencoders have shown promise for representaiton learning in discrete domains and when applied with more complex noise and masking patterns.

32.3.2.4 Challenges in generative representation learning

Despite several success in generative representation learning, they have empirically fallen behind. Generative methods for representation learning have to learn to match complex high-dimensional and diverse training datasets, which requires modeling all axis of variation of the inputs, regardless of whether they are semantically relevant for downstream tasks. For example, the exact pattern of blades of grass in an image matter for generation quality, but are unlikely to be useful for many of the semantic evaluations that are typically used. Ways to bias generative models to focus on the semantic features and ignore "noise" in the input is an open area of research.

32.3.3 Self-Supervised Representation Learning

When given large amounts of labeled data, standard supervised learning is a powerful mechanism for training deep neural networks. When only presented with unlabeled data, building generative models requires modeling all variations in a dataset, and is often not explicit about what is the signal and noise that we aim to capture in a representation. The methods and architectures for

building these generative models also differs substantially from those of supervised learning, where largely feedforward architectures are used to predict low-dimensional representations. Instead of trying to model all aspects of variation, self-supervised learning aims to design tasks where labels can be generated cheaply, and help to encode the structure of what we may care about for other downstream tasks. Self-supervised learning methods allow us to apply the tools and techniques of supervised learning to unlabeled data by designing a task for which we can cheaply produce labels.

In the image domain, several self-supervised tasks, also known as **pretext tasks**, have been proven effective for learning representations. Models are trained to perform these tasks in a supervised fashion using data generated by the pretext task, and then the learned representation is transferred to a target task of interest (such as object recognition), by training a linear classifier or fine-tuning the model in a supervised fashion.

32.3.3.1 Denoising and masked prediction

Generative representation learning is challenging because generative models must learn to produce the entire data distribution. A simpler option is *denoising*, in which some variety of noise is added to the input and the model is trained to reconstruct the noiseless input. A particularly successful variant of denoising is *masked prediction*, in which input patches or tokens are replaced with uninformative masks and the network is trained to predict only these missing patches or tokens.

The **denoising autoencoder** [Vin+08; Vin+10a] was the first deep model to exploit denoising for representation learning. A denoising autoencoder resembles a standard autoencoder architecturally, but it is trained to perform a different task. Whereas a standard autoencoder attempts to reconstruct its input exactly, a denoising autoencoder attempts to produce a noiseless output from a noisy input. Vincent et al. [Vin+08] argue that the network must learn the structure of the data manifold in order to solve the denoising task.

Newer approaches retain the conceptual approach of the denoising autoencoder, but adjust the masking strategy and objective. **BERT** [Dev+18] introduced the **masked language modeling** task, where 15% of the input tokens are selected for masking and the network is trained to predict them. 80% of the time, these tokens are replaced with an uninformative `[MASK]` token. However, the `[MASK]` token does not appear at fine-tuning time, producing some domain shift between pretraining and fine-tuning. Thus, 10% of the time, tokens are replaced with random tokens, and 10% of the time, they are left intact. BERT and the masked language modeling task have been extremely influential for representation learning in natural language processing, inspiring substantial follow-up work [Liu+19c; Jos+20].

Although denoising-based approaches to representation learning were first employed for computer vision, they received little attention for the decade that followed. Vincent et al. [Vin+08] greedily trained stacks of up to three denoising autoencoders that were then fine-tuned end-to-end to perform digit classification, but greedy unsupervised pretraining was abandoned as it was shown that it was possible to attain good performance using CNNs and other architectures trained end-to-end. Context encoders [Pat+16] mask contiguous image regions and train models to perform inpainting, achieving transfer learning performance competitive with other contemporary unsupervised visual representation learning methods. The use of image colorization as a pretext task [ZIE16; ZIE17] is also related to denoising in that colorization involves reconstructing the original image from a corrupted input, although generally color is dropped in a deterministic fashion rather than stochastically.

Recently, the success of BERT in NLP has inspired new approaches to visual representation learning

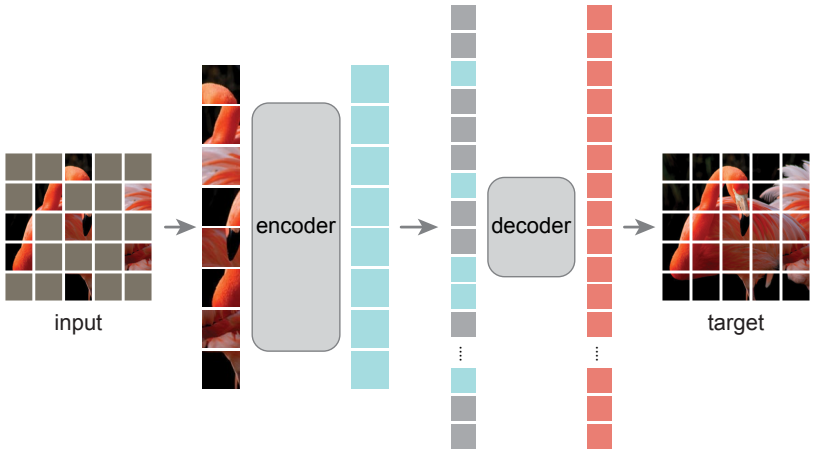


Figure 32.3: Masked autoencoders learn a representation of images by randomly masking out input patches and trying to predict them (from He et al. [He+21]).

based on masked prediction. Image GPT [Che+20a] trained a Transformer directly upon pixels to perform a BERT-style masked image modeling task. While the resulting model achieves very high accuracy when fine-tuned CIFAR-10, the cost of self-attention is quadratic in the number of pixels, limiting applicability to larger image sizes. BEiT [Bao+22] addresses this challenge by combining the idea of masked image modeling with the patch-based architecture of Vision Transformers [Dos+21]. BEiT splits images into 16×16 pixel image patches and then discretizes these patches using a discrete VAE [Ram+21b]. At training time, 40% of tokens are masked. The network receives continuous patches as input and is trained to predict the discretized missing tokens using a softmax over all possible tokens.

The **masked autoencoder** or **MAE** [He+22] further simplifies the masked image modeling task (see Figure 32.3). The MAE eliminates the need to discretize patches and instead predicts the constituent pixels of each patch directly using a shallow decoder trained with L_2 loss. Because the MAE encoder operates only on the unmasked tokens, it can be trained efficiently even while masking most (75%) of the tokens. Models pretrained using masked prediction and then fine-tuned with labels currently hold the top positions on the ImageNet leaderboard among models trained without additional data [He+22; Don+21].

32.3.3.2 Transformation prediction

An even simpler approach to representation learning involves applying a transformation to the input image and then predicting the transformation that was applied (see Figure 32.4). This prediction task is usually formulated as a classification problem. For visual representation learning, transformation prediction is appealing because it allows reusing exactly the same training pipelines as standard supervised image classification. However, it is not clear that networks trained to perform transformation prediction tasks learn rich visual representations. Transformation prediction tasks are potentially susceptible to "shortcut" solutions, where networks learn trivial features that are

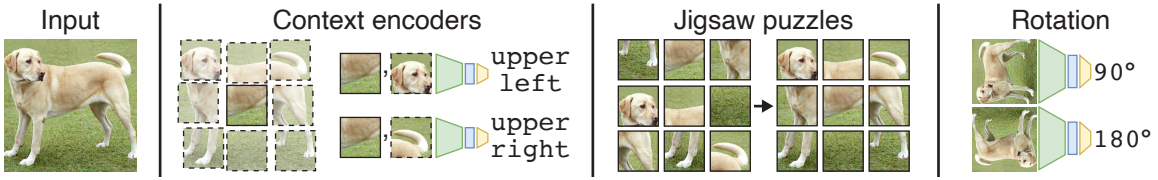


Figure 32.4: Transformation prediction involves training neural networks to predict a transformation applied to the input. Context encoders predict the position of a second crop relative to the first. The jigsaw puzzle task involves predicting the way in which patches have been permuted. Rotation prediction involves predicting the rotation that was applied to the input.

nonetheless sufficient to solve the task with high accuracy. For many years, self-supervised learning methods based on transformation prediction were among the top-performing methods, but they have since been displaced by newer methods based on contrastive learning and masked prediction.

Some pretext tasks operate by cutting images into patches and training networks to recover the spatial arrangement of the patches. In context prediction [DGE15], a network receives two adjacent image patches as input and is trained to recover their spatial relationship by performing an eight-class classification problem. To prevent the network from directly matching the pixels at the patch borders, the two patches must be separated by a small variable gap. In addition, to prevent networks from using chromatic aberration to localize the patches relative to the lens, color channels must be distorted or stochastically dropped. Other work has trained networks to solve jigsaw puzzles by splitting images into a 3×3 grid of patches [NF16]. The network receives shuffled patches as input and learns to predict how they were permuted. By limiting the permutations to a subset of all possibilities, the jigsaw puzzle task can be formulated as a standard classification task [NF16].

Another widely used pretext task is rotation prediction [GSK18], where input images are rotated 0, 90, 180, or 270 degrees and networks are trained to classify which rotation was applied. Although this task is extremely simple, the learned representations often perform better than those learned using patch-based methods [GSK18; KZB19]. However, all approaches based on transformation prediction currently underperform masked prediction and multiview approaches on standard benchmark datasets such as CIFAR-10 and ImageNet.

32.3.4 Multiview representation learning

The field of **multiview representation learning** aims to learn a representation where "similar" inputs or *views* of an input are mapped nearby in the representation space, and "dissimilar" inputs are mapped further apart. This representation space is often high-dimensional, and relies on collecting data or designing a task where one can generative "positive" pairs of examples that are similar, and "negative" pairs of examples that are dissimilar. There are many motivations and objectives for multiview representation learning, but all rely on coming up with sets of positive pairs, and a mechanism to prevent all representations from collapsing to the same point. Here we use the term multiview representation learning to encompass **contrastive learning** which combines positive and negative pairs, metric learning, and "non-contrastive" learning which eliminates the need for negative pairs.

Unlike generative methods for representation learning, multiview representation learning makes

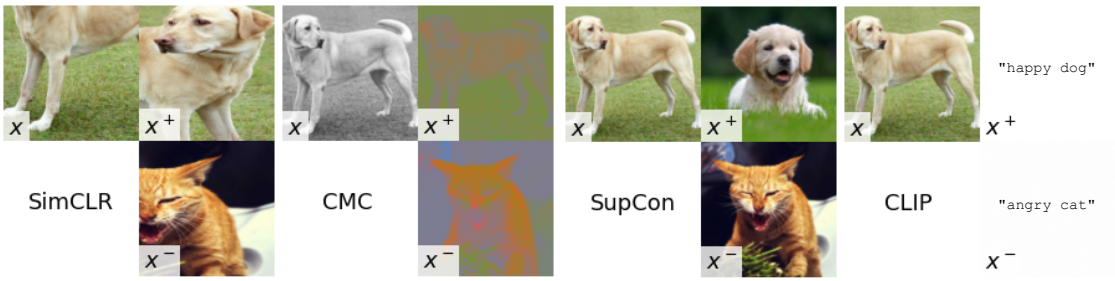


Figure 32.5: Positive and negative pairs used by different multiview representation learning methods.

it easy to incorporate prior knowledge about what inputs should be closer in the embedding space. Furthermore, these inputs need not be from the same modality, and thus multiview representaiton learning can be applied with rich multimodal datasets. The simplicity of the way in which prior knowledge can be incorporated into a model through data has made multiview representation learning one of the most powerful and performant methods for learning representations.

While there are a variety of methods for multiview representation learning, they all involve a repulsion component that pulls positive pairs closer together in embedding space, and a mechanism to prevent collapse of the representation to a single point in embedding space. We begin by describing loss functions for multiview representation learning and how they combine attractive and repulsive terms to shape the representation, then discuss the role of view generation, and finally practical considerations in deploying multiview representation learning.

32.3.4.1 View selection

Multiview representation learning depends on a datapoint or "anchor" x, a positive example x^+ that x will be attracted to, and zero or more negative examples x^- that x is repelled from. We assume access to a data-generating process for the positive pair: $p^+(x, x^+)$, and a process that generates the negative examples given the datapoint x: $p^-(x^-|x)$. Typically $p^+(x, x^+)$ generate (x, x^+) that are different augmentations of an underlying image from the dataset, and x^- represents an augmented view of a different random image from the dataset. The generative process for x^- is then independent of x, i.e. $p^-(x^-|x) = p^-(x^-)$.

The choice of views used to generate positive and negative pairs is critical to the success of representation learning. Figure 32.5 shows the positive pair (x, x^+) and negative x^- for several methods which we discuss below: SimCLR, CMC, SupCon, and CLIP.

SimCLR [Che+20c] creates positive pairs by applying two different data augmentations defined by transformations t and t' to an initial image x_0 twice: $x = t(x_0), x^+ = t(x_0)$. The data augmentations used are random crops (with horizontal flips and resize), color distortion, and Gaussian blur. The strengths of these augmentations (e.g. the amount of blur) impact performance and are typically treated as a hyperparameter.

If we access to additional information, such as a categorical label, we can use this to select positive pairs with the same label, and negative pairs with different labels. The resulting objective, when used with a contrastive loss, is called **SupCon** [Kho+20], and resembles Neighborhood Component

Analysis [Gol+04]. It was shown to imrove robustness when compared to standard supervised learning.

Contrastive Multiview Coding (CMC) [TKI20] generates views by splitting an initial image into orthogonal dimensions, such as the luma and chroma dimensions. These views are now no longer in the same space (or same dimensionality), and thus we must learn different encoders for the different inputs. However, the output of these encoders all live in the same-dimensional embedding space, and can be used in contrastive losses. At test-time, we can then combine embeddings from these different views through averaging or concatenation.

Views do not need to be from the same modality. **CLIP** [Rad+21] uses contrastive learning on image-text pairs, where x is an image, and x^+ and x^- are text descriptions. When applied to massive datasets of image-text pairs scraped from the Internet, CLIP is able to learn robust representations without any of the additional data augmentation needed by SimCLR or other image-only contrastive methods.

In most contrastive methods, negative examples are selected by randomly choosing x^+ from other elements in a minibatch. However, if the batch size is small it may be the case that none of the negative examples are close in embedding space to the positive example, and so learning may be slow. Instead of randomly choosing negatives, they may be chosen more intelligently through **hard negative mining** that selects negative examples that are close to the positive example in embedding space [Fag+18]. This typically requires maintaining and updating a database of negative examples over the course of training; this incurs enough computational overhead that the technique is infrequently used. However, reweighting examples within a minibatch can also lead to improved performance [Rob+21].

The choice of positive and negative views directly impacts what features are learned and what invariances are encouraged. Tian et al. [Tia+20] discusses the role of view selection on the learned representations, showing how choosing positives based on shared attributes (as in SupCon) can lead to learning those attributes or ignoring them. They also present a method for learning views (whereas all prior approaches fix views) based on targeting a "sweet spot" in the level of mutual information between the views that is neither too high or too low. However, understanding what views will work well for what downstream tasks remains an open area of study.

32.3.4.2 Contrastive losses

Given p^+ and p^-, we seek loss functions that learn an embedding $f_\theta(x)$ where x and x^+ are close in the embedding space, while x and x^- are far apart. This is called **metric learning**.

Chopra, Hadsell, LeCun, et al. [CHL+05] present a family of objectives that implements this intuition by enforcing the distance between negative pairs to always be at least ϵ bigger than the distance between positive pairs. The contrastive loss as instantiated in [HCL06] is:

$$\mathcal{L}_{\text{contrastive}} = \mathbb{E}_{x,x^+,x^-}\left[\|f_\theta(x) - f_\theta(x^+)\|^2 + \max(0, \epsilon - \|f_\theta(x) - f_\theta(x^-)\|^2\right]. \tag{32.14}$$

This loss pulls together the positive pairs by making the squared ℓ_2 distance between them small, and tries to ensure that negative pairs are at least a distance of ϵ apart. One challenge with using the contrastive loss in practice is tuning the hyperparameter ϵ.

Similarly, the **triplet loss** [SKP15] tries to ensure that the positive pair (x, x^+) is always at least some distance ϵ closer to each other than the negative pair (x, x^-):

$$\mathcal{L}_{\text{triplet}} = \mathbb{E}_{x,x^+,x^-}\left[\max(0, \|f_\theta(x) - f_\theta(x^+)\|^2 - \|f_\theta(x) - f_\theta(x^-)\|^2 + \epsilon)\right]. \tag{32.15}$$

A downside to the triplet loss approach is that one has to be careful about choosing hard negatives: if the negative pair is already sufficiently far away then the objective function is zero and no learning occurs.

An alternative contrastive loss which has gained popularity due to its lack of hyperparameters and empirical effectiveness is known as the **InfoNCE loss** [OLV18b] or the **Multi-Class N-pair loss** [Soh16]:

$$\mathcal{L}_{\text{InfoNCE}} = -\mathbb{E}_{x,x^+,x^-_{1:M}}\left[\log \frac{\exp f_\theta(x)^T g_\phi(x^+)}{\exp f_\theta(x)^T g_\phi(x^+) + \sum_{i=1}^M \exp f_\theta(x)^T g_\phi(x_i^-)}\right], \tag{32.16}$$

where M are the number of negative examples. Typically the embeddings $f(x)$ and $g(x')$ are ℓ_2-normalized, and an additional hyperparameter τ can be introduced to rescale the inner products [Che+20c]. Unlike the triplet loss, which uses a hard threshold of ϵ, $\mathcal{L}_{\text{InfoNCE}}$ can always be improved by pushing negative examples further away. Intuitively, the InfoNCE loss ensures that the positive pair is closer together than any of the M negative pairs in the minibatch. The InfoNCE loss can be related to a lower bound on the mutual information between the input x and the learned representation z [OLV18b; Poo+19a]:

$$I(X;Z) \geq \log M - \mathcal{L}_{\text{InfoNCE}}, \tag{32.17}$$

and has also been motivated as a way of learning representations through the InfoMax principle [OLV18b; Hje+18; BHB19]. When applying the InfoNCE loss to parallel views that are the same modality and dimension, the encoder f_θ for the anchor x and the positive and negative examples g_ϕ can be shared.

32.3.4.3 Negative-free losses

Negative-free representation learning (sometimes called **non-contrastive representation learning**) learns representations using only positive pairs, without explicitly constructing negative pairs. Whereas contrastive methods prevent collapse by enforcing that positive pairs are closer together than negative pairs, negative-free methods make use of other mechanisms. One class of negative-free objectives includes both attractive terms and terms that prevent collapse. Another class of methods uses objectives that include only attractive terms, and instead relies on the learning dynamics to prevent collapse.

The **Barlow Twins** loss [Zbo+21] is

$$\mathcal{L}_{\text{BT}} = \sum_{i=1}^p (1 - \boldsymbol{C}_{ii})^2 + \lambda \sum_{i=1}^p \sum_{j \neq i} \boldsymbol{C}_{ij}^2 \tag{32.18}$$

where $\boldsymbol{C}$ is the cross-correlation matrix between two batches of features that arise from the two views. The first term is an attractive term that encourages high similarity between the representations of the two views, whereas the second term prevents collapse to a low-rank representation. The loss is minimized when $\boldsymbol{C}$ is the identity matrix. Similar losses based on ensuring the variance of features being non-zero have also been useful for preventing collapse [BPL21b]. The Barlow Twins loss can be related to kernel-based independence criterion such as HSIC which have also been useful as losses for representation learning [Li+21; Tsa+21].

BYOL (Bootstrap Your Own Latent) [Gri+20] and SimSiam [Che+20c] simply minimizes the mean squared error between two representations:

$$\mathcal{L}_{\text{BYOL}} = \mathbb{E}_{\boldsymbol{x},\boldsymbol{x}^+}\left[\|g_\phi(f_\theta(\boldsymbol{x})) - f_{\theta'}(\boldsymbol{x}^+)\|^2\right]. \tag{32.19}$$

Following Grill et al. [Gri+20], g_ϕ is known as the *predictor*, f_θ is the *online network*, and $f_{\theta'}$ is the *target network*. When optimizing this loss function, weights are backpropagated to update ϕ and θ, but optimizing θ' directly leads the representation to collapse [Che+20c]. Instead, BYOL sets θ' as an exponential moving average of θ, and SimSiam sets $\theta' \leftarrow \theta$ at each iteration of training. The reasons why BYOL and SimSiam avoid collapse are not entirely clear, but Tian, Chen, and Ganguli [TCG21] analyze the gradient flow dynamics of a simplified linear BYOL model and show that collapse can indeed be avoided given properly set hyperparameters.

DINO (self-distillation with no labels) [Car+21] is another non-contrastive loss that relies on the dynamics of learning to avoid collapse. Like BYOL, DINO uses a loss that consists only of an attractive term between an online network and a target network formed by an exponential moving average of the online network weights. Unlike BYOL, DINO uses a cross-entropy loss where the target network produces the targets for the online network, and avoids the need for a predictor network. The DINO loss is:

$$\mathcal{L}_{\text{DINO}} = \mathbb{E}_{\boldsymbol{x},\boldsymbol{x}^+}\left[H(f_{\theta'}(\boldsymbol{x})/\tau, \text{center}(f_\theta(\boldsymbol{x}^+))/\tau')\right]. \tag{32.20}$$

where, with some abuse of notation, center is a mean-centering operation applied across the minibatch that contains x^+. Centering the output of the target network is necessary to prevent collapse to a single "class", whereas using a lower temperature $\tau' < \tau$ for the target network is necessary to prevent collapse to a uniform distribution. The DINO loss provides marginal gains over the BYOL loss when performing self-supervised representation learning with Vision Transformers on ImageNet [Car+21].

32.3.4.4 Tricks of the trade

Beyond view selection and losses, there are a number of useful architectures and modifications that enable more effective multiview representation learning.

Normalizing the output of the encoders and computing cosine similarity instead of predicting unconstrained representations has shown to improve performance [Che+20c]. This normalization bounds the similarity between points between -1 and 1, so an additional temperature parameter τ is typically introduced and fixed or annealed over the course of learning.

While the learned representation with multiview learning are often useful for downstream tasks, the losses when combined with data augmentation typically lead to too much invariance for some tasks. Instead, one can extract an earlier layer in the encoder as the representation, or alternatively, add an additional layer known as a projection head to the encoder before computing the loss [Che+20c]. When training we compute the loss on the output of the projection head, but when evaluating the quality of the representation we discard this additional layer.

Given the summation over negative examples in the denominator of the InfoNCE loss, it is often sensitive to the batch size used for training. In practice, large batch sizes of 4096 or more are needed to achieve good performance with this loss, which can be computationally burdensome. **MoCo** (Momentum Contrast) [He+20] introduced a memory queue to store negative examples from previous minibatches to expand the size of negatives at each iteration. Additionally, they use a momentum

encoder, where the encoder for the positive and negative examples uses an exponential moving average of the anchor encoder parameters. This momentum encoder approach was also found useful in BYOL to prevent collapse. As in BYOL, adding an extra predictor network that maps from the online network to the target network has shown to improve the performance of MoCo, and removes the requirement of a memory queue [CXH21].

The backbone architectures of the encoder networks play a large role in the quality of representations. For representation learning in vision, recent work has switched from ConvNet-based backbones to Vision Transformers, resulting in larger-scale models with improved performance on several downstream tasks [CXH21].

32.4 Theory of Representation Learning

While deep representation learning has replaced hand-designed features for most applications, the theory behind what features are learned and what guarantees these methods provide are limited. Here we review several theoretical directions in understanding representation learning: identifiability, information maximization, and transfer bounds.

32.4.1 Identifiability

In this section, we assume a latent-variable generative model that generated the data, where $z \sim p(z)$ are the latent variables, and $x = g(z)$ is a determinstic generator that maps from the latent variables to observations. Our goal is to learn a representation $h = f_\theta(x)$ that inverts the generative model and recovers $h = z$. If we can do this, we say the model is **identifiable**. Often times we are not able to recover the true latent variables exactly, for example the dimensions of the latent variables may be permuted, or individual dimensions may be transformed version of an underlying latent variable: $h_i = f_i(z_i)$. Thus most theoretical work on identifiability focuses on the case of learning a representation that can be permuted and elementwise transformed to match the true latent variables. Such representations are referred to as **disentangled** as the dimensions of the learned representaiton do not mix together multiple dimensions of the true latent variables.

Methods for recovering are typically based around latent-variable models such as VAEs combined with various regularizers (see Section 21.3.1.1). While several publications showed promising empirical progress, a large-scale study by Locatello et al. [Loc+20a] on disentangled representation learning methods showed that several existing approaches cannot work without additional assumptions on the data or model. Their argument relies on the observation that we can form a bijection f that takes samples from a factorial prior $p(z) = \prod_i p_i(z_i)$ and maps to $z' = f(z)$ that (a) preservers the marginal distribution, and (b) has entirely entangled latents (each dimension of z influences every dimension of z'). Transforming the marginal in this way changes the representation, but preserves the marginal likelihood of the data, and thus one cannot use marginal likelihood alone to identify or distinguish between the entangled and disentangled model. Empirically, they show that past methods largely succeeded due to careful hyperparameter selection on the target disentanglement metrics that require supervised labels. While further work has developed unsupervised methods for hyperparameter that address several of these issues [Dua+20], at this point there are no known robust methods for learning disentangled representations without further assumptions.

To address the empirical and theoretical gap in learning disentangled representations, several papers have proposed using additional sources of information in the form of weakly-labeled data to provide

guarantees. In theoretical work on nonlinear ICA [RMK21; Khe+20; Häl+21], this information comes in the form of additional observations for each datapoint that are related to the underlying latent variable through an exponential family. Work on **causal representation learning** has expanded the applicability of these methods and highlighted the settings where such strong assumptions on weakly-labeled data may be attainable [Sch+21c; WJ21; Rei+22]

Alternatively, one can assume access to pairs of observations where the relationship between latent variables is known. In Shu et al. [Shu+19b], they show that one can provably learn a disentangled representation of data when given access to pairs of data where only one of the latent variables is changed at a time. In real world datasets, having access to pairs of data like this is challenging, as not all the latent-variables of the model may be under the control of the data collector, and covering the full space of settings of the latent variable may be probihitively expensive. Locatello et al. [Loc+20b] develops this method further but leverages a heuristic to detect which latent variable has changed, and shows this performs empirically well, and under some restricted settings may lead to learning disentangled representations. It remains an open question what realistic assumptions can be made about the nature of a dataset that will easily enable learning a disentangled representaiton, and practical methods on arbitrary datasets remain out of reach.

32.4.2 Information maximization

When learning representations of an input x, one desiderata is to preserve as much information about x as possible. Any information we discard cannot be recovered, and if that information is useful for a downstream task then performance will decrease. Early work on understanding biological learning by Linsker [Lin88c] and Bell and Sejnowski [BS95b] argued that information maximization or **InfoMax** is a good learning principle for biological systems as it enables the downstream processing systems access to as much sensory input as possible. However, these biological systems aim to communicate information subject to strong constraints, and these constraints can likely be tuned over time by evolution to sculpt the kinds of representations that are learned.

When applying information maximization to neural networks, we are often able to realize trivial solutions which biological systems may not face: being able to losslessly copy the input. Information theory does not "color" the bits, it does not tell us which bits of an input are more important than others. Simply sending the image losslessly maximizes information, but does not provide a transformation of the input that can improve performance according to the metrics in Section 32.2. Architectural and optimization constraints can guide the bits we learn and the bits we dispose of, but we can also leverage additional sources of information, for example labels, to identify which bits to extract.

The **information bottleneck** method (Section 5.6) aims to learn representations Z of an input X that are predictive of another observed variable Y, while being as compressed as possible. The observed variable Y guides the bits learned in the representation Z towards those that are predictive, and penalizes content that does not predict Y. We can formalize the information bottleneck as an optimization problem [TPB00]:

$$\text{maximize}_\theta I(Z;Y) - \beta I(X;Z). \tag{32.21}$$

Estimating mutual information in high dimensions is challenging, but we can form variational bounds on mutual information that are amenable to optimization with modern neural networks, such

as Variational Information bottleneck (VIB, see Section 5.6.2). Approaches built on VIB have shown improved robustness to adversarial examples and natural variations [FA20].

Unlike information bottleneck methods, many recent approaches motivated by InfoMax have no explicit compression objective [Hje+18; BHB19; OLV18b]. They aim to maximize information subject to constraints, but without any explicit penalty on the information contained in the representation.

In spite of the appeal of explaining representation learning with information theory, there are a number of challenges. One of the greatest challenges in applying information theory to understand the content in learned representations is that most learned representations have determinstic encoders, $z = f_\theta(x)$ that map from a continuous input x to a continuous representation z. These mappings can typically preserve infinite information about the input. As mutual information estimators scale poorly with the true mutual information, estimating MI in this setting is difficult and typically results in weak lower bounds.

In the absence of constraints, maximizing information between an input and a learned representation has trivial solutions that do not result in any interesting transformation of the input. For example, the identity mapping $z = x$ maximizes information but does not alter the input. Tschannen et al. [Tsc+19] show that for invertible networks where the true mutual information between the input and representation is infinite, maximizing estimators of mutual information can result in meaningful learned represenations. This highlights that the geometric dependence and bias of these estimators may have more to do with their success for representation learning than the information itself (as it is infinite throughout training).

There have been several proposed methods for learning stochastic representations that constrain the amount of information in learned representations [Ale+17]. However, these approaches have not yet resulted in improved performance on most downstream tasks. Fischer and Alemi [FA20] shows that constraining information can improve robustness on some benchmarks, but scaling up models and datasets with determinstic representations currently presents the best results [Rad+21]. More work is needed to identify whether constraining information can improve learned representations.

33 Interpretability

This chapter was written by Been Kim and Finale Doshi-Velez.

33.1 Introduction

As machine learning models become increasingly commonplace, there exists increasing pressure to ensure that these models' behaviors align with our values and expectations. It is essential that models that automate even mundane tasks (e.g. processing paperwork, flagging potential fraud) do not harm their users or society at large. Models with large impacts on health and welfare (e.g. recommending treatment, driving autonomously) must not only be safe but often also function collaboratively with their users.

However, determining whether a model is harmful is not easy. Specific performance metrics may be too narrowly focused—e.g. just because an autonomous car stays in lane does not mean it is safe. Indeed, the narrow objectives used in common decision formalisms such as Bayesian decision theory (Section 34.1), multi-step decision problems (Chapter 34), and reinforcement learning (Chapter 35) can often be easily exploited (e.g., reward hacking). Incomplete sets of metrics also result in models that learn shortcuts that do not generalize to new situations (e.g., [Gei+20b]). Even when one knows the desired metrics, those metrics can be hard to estimate with limited data or a distribution shift (Chapter 19). Finally, normative concepts, such as fairness, may be impossible to fully formalize. As a result, not only unexpected and irreversible harms may occur (e.g., an adverse drug reaction) but more subtle harms may go unnoticed until sufficient reporting data accrues [Amo+16].

Interpretability allows human experts to inspect a model. Alongside traditional statistical measures of performance, this human inspection can help expose issues and thus mitigate potential harms. Exposing the workings of a model can also help people identify ways to incorporate information they have into a final decision. More broadly, even when we are satisfied with a model's performance, we may be interested in understanding *why* they work to gain scientific and operational insights. For example, one might gain insights in language structure by asking why a language model performs so well; understanding why patient data cluster along particular axes may result in a better understanding of disease and the common treatment pathways. Ultimately, interpretation helps humans to communicate better with machines to accomplish our tasks better.

In this chapter, we lay out the role and terminologies in interpretable ML before introducing methods, properties and evaluation of interpretability methods.

33.1.1 The Role of Interpretability: Unknowns and Under-Specifications

As noted above, ensuring that models behave as desired is challenging. In some cases, the desired behavior can be guaranteed by design, such as certain notions of privacy via differentially-private learning algorithms or some chosen mathematical metric of fairness. In other cases, tracking various metrics, such as adverse events or subgroup error rates, may be the appropriate and sufficient way to identify concerns. Much of this textbook deals with uncertainty quantification: basic models in Chapter 3, Bayesian neural networks in Chapter 17, Gaussian processes in Chapter 18). When well-calibrated uncertainties can be computed, they may provide sufficient warning that a model's output may be suspect.

However, in many cases, the ultimate goal may be fundamentally impossible to fully specify and thus formalize. For example, Section 20.4.8 discusses the challenge of evaluating the quality of samples from a generative model. In such cases, human inspection of the machine learning model may be necessary. Below we describe several examples.

Blindspot Discovery. Inspection may reveal **blindspots** in our modeling, objective, or data [Bad+18; Zec+18b; Gur+18]. For example, suppose a company has trained a machine learning system for credit scoring. The model was trained on a relatively affluent, middle-aged population, and now the company is considering using it on a different, college-aged population. Suppose that inspection of the model reveals that it relies heavily on the applicant's mortgage payments. Not only might this suggest that the model might not transfer well to the college population, but it might encourage us to check for bias in the existing application because we know historical biases have prevented certain populations from achieving home ownership (something that a purely quantitative definition of fairness may not be able to recognize). Indeed, the most common application of interpretability in industry settings is for engineers to debug models and make deployment decisions [Pai].

Novel Insights. Inspection may catalyze the discovery of **novel insights.** For example, suppose an algorithm determines that surgical procedures fall into three clusters. The surgeries in one of the clusters of patients seem to consistently take longer than expected. A human inspecting these clusters may determine that a common factor in the cluster with the delays is that those surgeries occur in a different part of the hospital, a feature not in the original dataset. This insight may result in ideas to improve on-time surgery performance.

Human+ML Teaming. Inspection may empower effective **human+ML interaction and teaming**. For example, suppose an anxiety treatment recommendation algorithm reveals the patient's comorbid insomnia constrained its recommendations. If the patient reports that they no longer have trouble sleeping, the algorithm could be re-run with that constraint removed to get additional treatment options. More broadly, inspection can reveal places where people may wish to adjust the model, such as correcting an incorrect input or assumption. It can also help people use only part of a model in their own decision-making, such as using a model's computation of which treatments unsafe vs. which treatments are best. In these ways, the human+ML team may be able to produce better combined performance than either alone (e.g. [Ame+19; Kam16]).

Individual-Level Recourse. Inspection can help determine whether a specific harm or error happened in a specific context. For example, if a loan applicant knows what features were used to

deny them a loan, they have a starting point to argue that an error might have been made, or that the algorithm denied them unjustly. For this reason, inspectability is sometimes a legal requirement [Zer+19; GF17; Cou16].

As we look at the examples above, we see that one common element is that *interpretability is needed when we need to combine human insights with the ML algorithm to achieve the ultimate goal.* [1] However, looking at the list above also emphasizes that beyond this very basic commonality, *each application and task represents very different needs.* A scientist seeking to glean insights from a clustering on molecules may be interested in global patterns—such as all molecules with certain loop structures are more stable—and be willing to spend hours puzzling over a model's outputs. In contrast, a clinician seeking to make a specific treatment decision may only care about aspects of the model relevant to the specific patient; they must also reach their decision within the time-pressure of an office visit. This brings us to our most important point: The best form of explanation depends on the *context*; interpretability is a means to an end.

33.1.2 Terminology and Framework

In broad strokes, "to interpret means to explain or present in understandable terms,"[Mer] [to a human]. Understanding, in turn, involves an alignment of mental models. In interpretable machine learning, that alignment is between what (perhaps part of) the machine learning model is doing and what the user thinks the model is doing.

As a result, interpretable machine learning ecosystem includes not only standard machine learning (e.g., a prediction task) but also what information is provided to the human user, in what context, and the user's ultimate goal. The broader *socio-technical system*—the collection of interactions between human, social, organizational, and technical (hardware and software) factors—cannot be ignored [Sel+19]. The goal of interpretable machine learning is to help a user do *their* task, with *their* cognitive strengths and weaknesses, with *their* focus and distractions [Mil19]. Below we define the key terms of this expanded ecosystem and describe how they relate to each other. Before continuing, however, we note that the field of interpretable machine learning is relatively new, and a consensus around terminology is still evolving. Thus, it is always important to define terms.

Two core **social** or **human-factors** elements in interpretable machine learning are the *context* and the *end-task*.

Context. We use the term *context* to describe the setting in which an interpretable machine learning system will be used. Who is the user? What information do they have? What constraints are present on their time, cognition, or attention? We will use the terms context and application interchangeably [Sta].

End-task. We use the term *end-task* to refer to the user's ultimate goal. What are they ultimately trying to achieve? We will use the terms end-task and downstream tasks interchangeably.

Three core **technical** elements in interpretable machine learning are the *method*, the *metrics*, and the *properties* of the methods.

1. We emphasize that interpretability is different from manipulation or persuasion, where the goal is to intentionally deceive or convince users of a predetermined choice.

Method. How do we does the interpretability happen? We use the term *explanation* to mean the output provided by the method to the user: interpretable machine learning *methods* provide *explanations* to the users. If the explanation is the model itself, we call the method *inherently interpretable* or *interpretable by design*. In other cases, the model may be too complex for a human to inspect it in its entirety: perhaps it is a large neural network that no human could expect to comprehend; perhaps it is a medium-sized decision tree that could be inspected if one had twenty minutes but not if one needs to make a decision in two minutes. In such cases, the explanation may be a *partial view* of the model, one that is ideally suited for performing the end-task in the given context. Finally, we note that even inherently interpretable models do not reveal everything: one might be able to fully inspect the function (e.g. a two-node decision tree) but not know what data it was trained on or which data points were most influential.

Metrics. How is the interpretability method evaluated? Evaluation is one of the most essential and challenging aspects of interpretable machine learning, because we are interested in the **end-task** performance of the *human*, when explanation is provided. We call this the *downstream performance*. Just as different goals in ML require different metrics (e.g., positive predictive value, log likelihood, AUC), different **contexts** and **end-tasks** will have different metrics. For example, the model with the best predictive performance (e.g., log likelihood loss) may not be the model that results in the best downstream performance.

Properties. What characteristics does the explanation have in relation to the model, the context and the end-tasks? Different **contexts** and different **end-tasks** might require different properties. For example, suppose that an explanation is being used to identify ways in which a denied loan applicant could improve their application. Then, it may be important that the explanation only include factors that, if changed, would change the outcome. In contrast, suppose the explanation is being used to determine if the denial was fair. Then, it may be important that the explanation does not leave out any relevant factors. In this way, properties serve as a glue between interpretability methods, contexts and end-tasks: properties allow us to specify and quantify aspects of the explanation relevant to our ultimate end-task goals. Then we can make sure that our interpretability method has those properties.

How they all relate. Formulating an interpretable machine learning problem generally starts by specifying the context and the end-task. Together the context and the end-task imply what metrics are appropriate to evaluate the downstream performance on the end-task and suggest what properties will be important in the explanation. Meanwhile, the context also determines the data and training metric for the ML model. The appropriate choice of explanation methods will depend on the model and properties desired, and it will be evaluated with respect to the end-task metric to determine the downstream performance. Figure 33.1 shows these relationships.

Interpretable machine learning involves many challenges, from computing explanations and optimizing interpretable models and creating explanations with certain properties to understanding the associated human factors. That said, the grand challenge is to (1) understand what properties are needed for different contexts and end-tasks and (2) identify and create interpretable machxine learning methods that have those properties.

A Simple Example In the following sections, we will expand upon methods for interpretability,

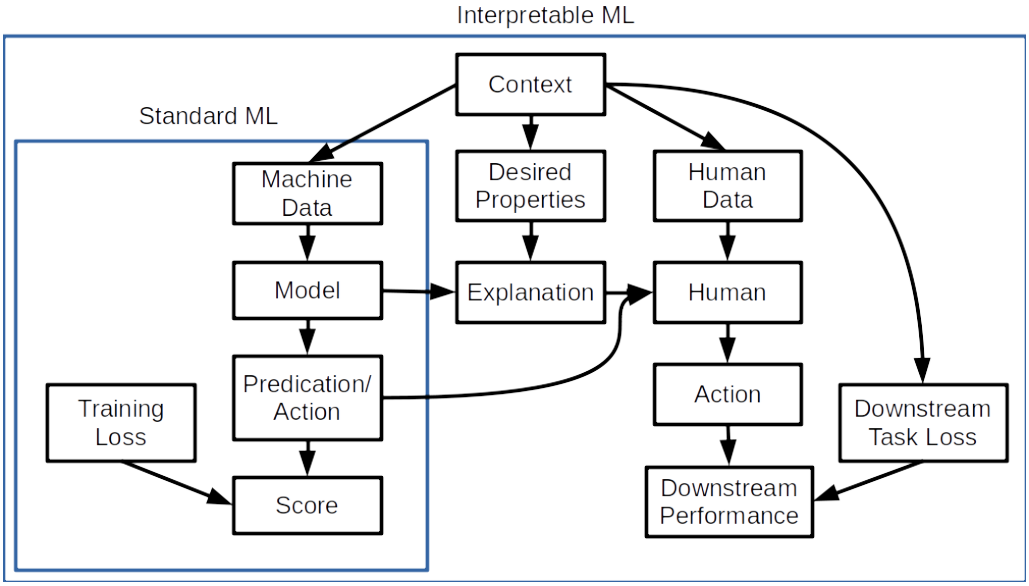


Figure 33.1: The Interpretable Machine Learning ecosystem. While standard machine learning can often abstract away elements of the context and consider only the process of learning models given a data distribution and a loss, interpretable machine is inextricably tied to a socio-technical context.

metrics for evaluation, and types of properties. First, however, we provide a simple example connecting all of the concepts we discussed above.

Suppose our *context* is that we have a lemonade stand, and our *end-task* is to understand when the stand is most successful in order to prioritize which days it is worth setting it up. (We have heard that sometimes machine learning models latch on to incorrect mechanisms and want to check the model before using it to inform our business strategy.) Our *metric* for the downstream performance is whether we correctly determine if the model can be trusted; this could be quantified as the amount of profit that we make by opening on busy days and being closed on quiet days.

To train our model, we collect data on two input features—the average temperature for the day (measured in degrees Fahrenheit) and the cleanliness of the sidewalk near our stand (measured as a proportion of the sidewalk that is free of litter, between 0 and 1)—and the output feature of whether the day was profitable. Two models seem to fit the data approximately equally well:

Model 1:

$$p(\texttt{profit}) = .9 * (\texttt{temperature} > 75) + .1(\texttt{howCleanSidewalk}) \tag{33.1}$$

Model 2:

$$p(\texttt{profit}) = \sigma(.9(\texttt{temperature} - 75)/\texttt{maxTemperature} + .1(\texttt{howCleanSidewalk} - .5)) \tag{33.2}$$

These models are illustrated in Figure 33.2. Both of these models are inherently interpretable in

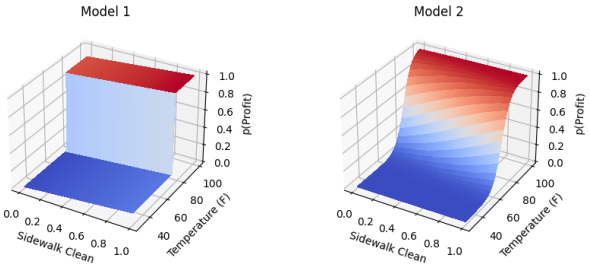


Figure 33.2: Models described in the simple example. Both of these models have the same qualitative characteristics, but different explanation methods will describe these models quite differently, potentially causing confusion.

the sense that they are easy to inspect and understand. While we were not explicitly seeking causal models (for that, see Chapter 36), both rely mostly on the temperature, which seems reasonable.

For the sake of this example, suppose that the models above were black boxes, and we could only request partial views of it. We decide to ask the model for the most important features. Let us see what happens when we consider two different ways of computing important features.[2]

Our first (feature-based) explanation method computes, for each training point, whether individually changing each feature to its max or min value changes the prediction. Important features are those that change the prediction for many training points. One can think of this explanation method as a variant of computing feature importance based on how important a feature is to the coalition that produces the prediction. In this case, both models will report temperature to be the dominating feature. If we used this explanation to vet our models, we would correctly conclude that both models use the features in a sensible way (and thus may be worth considering for deciding when to open our lemonade stand).

Our second (feature-based) explanation method computes the magnitude of the derivatives of the output with respect to the inputs for each training point. Important features are those that have a large sum of absolute derivatives across the training set. One can think of this explanation method as a variant of computing feature importances based on local geometry. In this case, Model 2 will still report that temperature has higher derivatives. However, Model 1, which has very similar behavior to Model 2, will report that sidewalk cleanliness is the dominating feature because the derivative with respect to temperature is zero nearly everywhere. If we used this explanation to vet our models, we would incorrectly conclude that Model 1 relies on an unimportant feature (and that Model 1 and 2 rely on different features).

What happened? The different explanations had different properties. The first explanation had the property of fidelity with respect to identifying features that, if changed, will affect the prediction, whereas the second explanation had the property of correctly identifying features that have the most local curvature. In this example, the first property is more important for the task of determining whether our model can be used to determine our business strategy. [3]

2. In the remainder of the chapter we will describe many other ways of creating and computing explanations.
3. Other properties may be important for this end-task. This example is just the simplest one.

33.2 Methods for Interpretable Machine Learning

There exist many methods for interpretable machine learning. Each method has different properties and the right choice will depend on context and end-tasks. As we noted in Section 33.1.2, the grand challenge in interpretable machine learning is determining what kinds of properties are needed for what contexts, and what explanation methods satisfy those properties. Thus, one should consider this section a high-level snapshot of the rapidly changing options of methods that one may want to choose for interpretable machine learning.

33.2.1 Inherently Interpretable Models: The Model is its Explanation

We consider certain classes of models inherently interpretable: a person can inspect the full model and, with reasonable effort, understand how inputs become outputs.[4] Specifically, we define inherently interpretable models as those that require no additional process or proxies in order for them to be used as explanation for the end-task. For example, suppose a model consists of a relatively small set of rules. Then, those rules might suffice as the explanation for end-tasks that do not involve extreme time pressure. (Note: in this way, a model might be inherently interpretable for one end-task and not another.)

Inherently interpretable models fall into two main categories: sparse (or otherwise compact) models and logic-based models.

Compact or **sparse** feature-based models include various kinds of sparse regressions. Earlier in this textbook, we discussed simple models such as HMMs (Section 29.2), generalized linear models (Chapter 15), and various latent variable models (Chapter 28). When small enough, these are generally inherently interpretable. More advanced models in this category include super-sparse linear integer models and other checklist models [DMV15; UTR14].

While simple functionally, sparsity has its drawbacks when it comes to inspection and interpretation. For example, if a model picks only one of several correlated features, it may be harder to identify what signal is actually driving the prediction. A model might also assign correlated features different signs that ultimately cancel, rendering an interpretion of weights meaningless.

To handle these issues, as well as to express more complex functions, some models in this category impose hierarchical or modular structures in which each component is still relatively compact and can be inspected. Examples include topic models (e.g. [BNJ03b], (small) discrete time series models (e.g. [FHDV20]), generalized additive models (e.g. [HT17]) and monotonicity-enforced models (e.g., [Gup+16]).

Logic-based models use logical statements as basis. Models in this category include decision-trees [Bre+17], decision lists [Riv87; WR15; Let+15a; Ang+18; DMV15] , decision tables, decision sets [Hau+10; Wan+17a; LBL16; Mal+17; Bén+21] and logic programming [MDR94]. A broader discussion, as well as a survey of user studies on these methods, can be found in [Fre14]. Logic-based models easily model non-linear relationships but can have trouble modeling continuous relationships between the input and output (e.g. expressing a linear function vs. a step-wise constant function). Like the compact models, hierarchies and other forms of modularity can be used to extend the expressivity of the model while keeping it human-inspectable. For example, one can define a new

4. There may be other questions, such as how training data influenced the model, which may still require additional computation or information.

concept as a formula based on some literals, and then use the new concept to build more complex rules.

When using inherently interpretable models, three key decisions need to be made: the choice of the model class, how to manage uninterpretable input features, and the choice of optimization method.

Decision: Model Class. Since the model is its own explanation, the decision on the model class becomes the decision on the form of explanation. Thus, we need to consider both whether the model class is a good choice for modeling the data as well as providing the necessary information to the user. For example, if one chooses to use a linear model to describe one's data, then it is important that the intended users can understand or manipulate the linear model. Moreover, if the fitting process produces a model that is too large to be human-inspectable, then it is no longer inherently interpretable, even if it belongs to one of the model classes described above.

Decision: Optimization Methods for Training. The kinds of model classes that are typically inherently interpretable often require more advanced methods for optimization: compact, sparse, and logic-based models all involve learning discrete parameters. Fortunately, there is a long and continuing history of research for optimizing such models, including directly via optimization programs, relaxation and rounding techniques, and search-based approaches. Another popular optimization approach is via distillation or mimics: one first trains a complex model (e.g., a neural network) and then uses the complex model's output to train a simpler model to mimic the more complex model. The more complex model is then discarded. These optimization techniques are beyond the scope of this chapter but covered in optimization textbooks.

Decision: How to Manage Uninterpretable Input Features. Sometimes the input features themselves are not directly interpretable (e.g. pixels of an image or individual amplitudes spectrogram); only collections of inputs have semantic meaning for human users. This situation challenges our ability not only to create inherently interpretable models but also explanations in general.

To address this issue, more advanced methods attempt to add a "concept" layer that first converts the uninterpretable raw input to a set of human-interpretable concept features. Next, these concepts are mapped to the model's output [Kim+18a; Bau+17]. This second stage can still be inherently interpretable. For example, one could first map a pattern of spectrogram to a semantically meaningful sound (e.g., people chatting, cups clinking) and then from those sounds to a scene classification (e.g., in a cafe). While promising, one must ensure that the initial data-to-concept mapping truly maps the raw data to concepts as the user understands them, no more and no less. Creating and validating that machine-derived concepts correspond to a semantically meaningful human concepts remains an open research challenge.

When might we want to consider inherently interpretable models? When not? Inherently interpretable models have several advantages over other approaches. When the model is its explanation, one need not worry about whether the explanation is faithful to the model or whether it provides the right partial view of the model for the intended task. Relatedly, if a person vets the model and finds nothing amiss, they might feel more confident about avoiding surprises. For all these reasons, inherently interpretable models have been advocated for in high-stakes scenarios, as well as generally being the first go-to to try[Rud19].

That said, these models do have their drawbacks. They typically require more specialized

optimization approaches. With appropriate optimization, inherently interpretable models can often match the performance of more complex models, but there are domains—in particular, images, waveforms, and language—in which deep models or other more complex models typically give significantly higher performance. Trying to fit complex behavior with a simple function may result not only in high bias in the trained model but also invite people to (incorrectly) rationalize why that highly biased model is sensible [Lun+20]. In an industry setting, seeking a migration away from a legacy, uninterpretable, business-critical model that has been tuned over decades would run into resistance.

Lastly, we note that just because a model is inherently interpretable, it does not guard against all kinds of surprises: as noted in Section 33.1, interpretability is just one form of validation mechanism. For example, if the data distribution shifts, then one may observe unexpected model behavior.

33.2.2 Semi-Inherently Interpretable Models: Example-Based Methods

Example-based models use examples as their basis for their predictions. For example, an example-based classifier might predict the class of a new input by first identifying the outputs for similar instances in the training set and next taking a vote. K-nearest neighbors is one of the best known models in this class. Extensions include methods to identify exemplars for predicted classes and clusters (e.g. [KRS14; KL17b; JL15a; FD07b][RT16; Arn+10]), to generate exemplars (e.g. [Li+17d]), to define similarities between instances via sophisticated embeddings (e.g.[PM18a]), and to first decompose an instance into parts and then find neighbors or exemplars between the parts (e.g. [Che+18b]). Like logic-based models, example-based models can describe highly non-linear boundaries.

On one hand, individual decisions made by example-based methods seem fully inspectable: one can provide the user with exactly the training instances (including their output labels) that were used to classify a particular input in a particular way. However, it may be difficult to convey a potentially complex distance metric used to define "similarity." As a result, the user may incorrectly infer what features or patterns made examples similar. It is also often difficult to convey the intuition behind the global decision boundary using examples.

33.2.3 Post-hoc or Joint training: The Explanation gives a Partial View of the Model

Inherently interpretable models are a subset of all machine learning models, and circumstances may require working with a model that is not inherently interpretable. As noted above, large neural models (Chapter 16) have demonstrated large performance benefits for certain kinds of data (e.g. images, waveform, and text); one might have to work with a legacy, business critical model that has been tuned for decades; one might be trying to understand a system of interconnected models.

In these cases, the view that the explanation gives into the model will necessarily be *partial*: the explanation may only be an approximation of the model or be otherwise incomplete. Thus, more decisions have to be made. Below, we split these decisions into two broad categories—what the explanation should consist of to best serve the context and how the explanation should be computed from the trained model. More detail on the abilities and limitations of these partial explanation methods can be found in [Sla+20; Yeh+19a; Kin+19; Ade+20a].

33.2.3.1 What does the explanation consist of?

One set of decisions center around the content of the explanation and what properties it should have. One choice is the form: Should the explanation be a list of important features? The top interactions? One must also choose the scope of the explanation: Is it trying to explain the whole model (global)? The model's behavior near a specific input (local)? Something else? Determining what properties the explanation must have will help answer these and other questions. We expand on each of these points below; the right choice, as always, will depend on the user—whom the explanation is for—and their end-task.

Decision: Form of the Explanation. In the case of inherently interpretable models, the model class used to fit the data was also the explanation. Now, the model class and the explanation are two different entities. For example, the model could be a deep network and the explanation a decision tree.

Works in interpretable machine learning have used a large variety of forms of explanations. The form could be a list of "important" input features [RSG16b; Lun+20; STY17; Smi+17; FV17] or "important" concepts [Kim+18a; Bau+20; Bau+18]. Or it could be a simpler model that approximates the complex model (e.g. a local linear approximation, an approximating rule set)[FH17; BKB17; Aga+21b; Yin+19c]. Another choice could be a set of similar or prototypical examples [KRS14; AA18; Li+17d; JL15a; JL15b; Arn+10]. Finally, one can choose whether the explanation should include a contrast against an alternative (also sometimes described as a counterfactual explanation) [Goy+19; WMR18; Kar+20a] or include or influential examples [KL17b].

Different forms of explanations will facilitate different tasks in different contexts. For example, a contrastive explanation of why treatment A is better than treatment B may help a clinician decide between treatments A and B. However, a contrast between treatments A and B may not be useful when comparing treatments A and C. Given the large number of choices, literature on how people communicate in the desired context can often provide some guidance. For example, if the domain involves making quick, high-stakes decisions, one might turn to how trauma nurses and firefighters explain their decisions (known as recognition-primed decision making, [Kle17]).

Decision: Scope of the Explanation: Global or Local. Another major decision regarding the parameters of the explanation is its scope.

Local explanation: In some cases, we may only need to interrogate an existing model about a specific decision. For example, why was this image predicted as a bird? Why was this patient predicted to have diabetes? Local explanations can help see if a consequential decision was made incorrectly or determine what could have been done differently to produce a different outcome (i.e., provide a recourse).

Local explanations can take many forms. A family of methods called saliency maps or attribution maps [STY17; Smi+17; ZF14; Sel+17; Erh+09; Spr+14; Shr+16] estimate the importances of each input dimension (e.g. via first-order derivatives with respect to the input). More generally, one might locally-fit simpler model in the neighborhood of the input of interest (e.g. LIME [RSG16b]). A local explanation may also consist of representative examples, including identifying which training points were most influential for a particular decision [KL17b] or identifying nearby data points with different predictions [MRW19; LHR20; Kar+20a].

All local explanation methods are partial views because they only attempt to explain the model around an input of interest. A key risk is that the user may overgeneralize the explanation to a wider

region than it applies. They may also interpolate an incorrect mental model of the model based on a few local explanations.

Global explanation: In other cases, we may desire insight into the model as a whole or for a collection of data points (e.g., all inputs predicted to one class). For example, suppose that our end-task is to decide whether to deploy a model. Then, we care about understanding the *entire* model.

Global explanations can take many forms. One choice is to fit a simpler model (e.g. an inherently interpretable model) that approximates the original model (e.g. [HVD14]). One can also identify concepts or features that affect decisions across many inputs (e.g. [Kim+18b]). Another approach is to provide a carefully chosen set of representative examples[Yeh+18]. These examples might be chosen to be somehow characteristic of, or providing coverage of, a class (e.g. [AA18]), to draw attention to decision boundaries (e.g. [Zhu+18]), or to identify inputs particularly influential in training the model.

Unless a model is inherently interpretable, it is still important to remember that a global explanation is still a partial view. To make a complex model accessible to the user, the global explanation will need to leave some things out.

Decision: Determining what Properties the Context Needs. Different forms of explanations have have different levels of expressivity. For example, an explanation listing important features, or fitting a local linear model around a particular input, does not expose interactions—but fitting a local decision tree would. For each form, there will also be many ways to compute an explanation of that form (more on this in Section 33.2.3.2). How do we choose amongst all of these different ways to compute the explanation? We suggest that the first step in determining the form and computation of an explanation should be to determine what properties are needed from it.

Specifying properties is especially important because not only may different forms of explanations have different intrinsic properties—e.g. can it model interactions?—but the properties may depend on the model being explained. For example, if the model is relatively smooth, then a feature-based explanation relying on local gradients may be fairly faithful to the original model. However if the model has spiky contours, the same explanation may not adequately capture the model's behavior. Once the desired properties are determined, one can determine what kind of computation is necessary to achieve them. We will list commonly desirable properties in Section 33.3.

33.2.3.2 How the explanation is computed

Another set of decisions has to do with how the explanation is computed.

Decision: Computation of Explanation. Once we make the decisions above, we must decide how the explanation will actually be computed. This choice will have a large impact on the explanation's properties. Thus, it is crucial to carefully choose a computational approach that provides the properties needed for the context and end-task.

For example, suppose one is seeking to identify the most "important" input features that change a prediction. Different computations correspond to different definitions of importance. One definition of importance might be the smallest region in an image that, when changed, changes the prediction—a perturbation-based analysis. Even within this definition, we would need to specify how that

perturbation will be computed: Do we keep the pixel values within the training distribution? Do we preserve correlations between pixels? Different works take different approaches [SVZ13; DG17; FV17; DSZ16; Adl+18; Bac+15a].

A related approach is to define importance in terms of sensitivity (e.g., largest gradients of the output with respect to the input feature). Even then, there are many computational decisions to be made [STY17; Smi+17; Sel+17; Erh+09; Shr+16]. Yet another common definition of importance is how often the input feature is part of a "winning coalition" that drives the prediction, e.g. a Shapley or Banzaf score[LL17]. Each of these definitions have different properties, as well as require different amounts of computation.

Similar issues come up with other forms of explanations. For example, for an example-based explanation, one has to define what it means to be similar or otherwise representative: Is it the cosine similarity between activations? A uniform L2 ball of a certain size between inputs? Likewise, there are many different ways to obtain counterfactuals. One can rely on distance functions to identify nearby inputs that with different outputs [WMR17; LHR20], causal frameworks [Kus+18], or SAT formulations [Kar+20a], among other choices.

Decision: Joint Training vs. Post-hoc Application. So far, we have described our partial explanation techniques as extracting some information from an already-trained model. This approach is called deriving a post-hoc explanation. As noted above, post-hoc, partial explanations may have some limitations: for example, an explanation based on a local linear approximation may be great if the model is generally smooth, but provide little insight if the model has high curvature. Note that this limitation is not because the partial explanation is wrong, but because the view that local gradients provide isn't sufficient if the true decision boundary is curvy.

One approach to getting explanations to have desired properties we is to train the model and the explanation jointly. For example, a regularizer that penalizes violations of desired properties can help steer the overall optimization process towards learning models that both perform well and are amenable to the desired explanation[Plu+20]. It is often possible to find such a model because most complex model classes have multiple high-performing optima [Bre01].

The choice of regularization will depend on the desired properties, the form of the explanation, and its computation. For example, in some settings, we may desire the explanation use the same features that people do for the task (e.g., lower frequency vs. higher frequency features in image classifiers [Wan+20b])—and still be faithful to the model. In other settings, we may want to control the input dimensions used or not used in the explanation, or for the explanation to be somehow compact (e.g. a small decision tree) while still being faithful to the underlying model, [RHDV17; Shu+19a; Vel+17; Nei+18; Wu+19b; Plu+20]. (Certain attention models fall into this category [JW19; WP19].) We may also have constraints on the properties of concepts or other intermediate features [AMJ18b; Koh+20a; Hen+16; BH20; CBR20; Don+17b]. In all of these cases, these desired properties could be included as a regularizer when training the model.

When choosing between a post-hoc explanation or joint training, one key consideration is that joint training assumes that one can re-train the model or the system of interest. In many cases in practice, this may not be possible. Replacing a complex and well-validated system in deployment for a decade may not be possible or take a prohibitively long time. In that case, one can still extract approximated explanations using post-hoc methods. Finally, a joint optimization, even when it can be performed, is not a panacea: optimization for some properties may result in unexpected violations of other (unspecified but desired) properties. For this reason, explanations from jointly trained model are still partial.

When might we want to consider post-hoc methods, and when not?. The advantage of post-hoc interpretability methods is that they can be applied to any model. This family of methods is especially useful in real-world scenarios where one needs to work with a system that contains many models as its parts, where one cannot expect to replace the whole system with one model. These approaches can also provide at least some broader knowledge about the model to identify unexpected concerns.

That said, post-hoc explanations, as approximations of the true model, may not be fully faithful to the model nor cover the model completely. As such, an explanation method tailored for one context may not be transferable in another; even in the intended context, there may be blindspots about the model that the explanation misses completely. For these reasons, in high stakes situations, one should attempt to use an inherently interpretable model first if possible [Rud19]. In all situations when post-hoc explanations are used, one must keep in mind that they are only one tool in a broader accountability toolkit and warn users appropriately.

33.2.4 Transparency and Visualization

The scope of interpretable machine learning is around methods that expose the process by which a trained model makes a decision. However, the behavior of a model also depends on the objective function, the training data, how the training data were collected and processed, and how the model was trained and tested. Conveying to a human these other aspects of what goes into the creation of a model can be as important as explaining the trained model itself. While a full discussion of transparency and visualization is outside the scope of this chapter, we provide a brief discussion here to describe these important adjacent concepts.

Transparency is an umbrella term for the many things that one could expose about the modeling process and its context. Interpreting models is one aspect. However, one could also be transparent about other aspects, such as the data collection process or the training process (e.g. [Geb+21; Mit+19; Dnp]). There are also situations in which a trained model is released (whether or not it is inherently interpretable), and thus the software can be inspected and run directly.

Visualization is one way to create transparency. One can visualize the data directly or various aspects of the model's process (e.g. [Str+17]). Interactive visualizations can convey more than text or code descriptions [ZF14; OMS17; MOT15; Ngu+16; Hoh+20]. Finally, in the specific context of interpretable machine learning, how the explanation is presented—the visualization—can make a large difference in how easily users can consume it. Even something as simple as a rule list has many choices of layout, highlighting, and other organization.

When might we want to consider transparency and visualization? When not? In many cases, the trouble with a model comes not from the model itself, but parts of its training pipeline. The problem might be the training data. For example, since policing data contain historical bias, predictions of crime hot spots based on that data will be biased. Similarly, if clinicians only order tests when they are concerned about a patient's condition, then a model trained to predict risk based on tests ordered will only recapitulate what the clinicians already know. Transparency about the properties of the data, and how training and testing were performed, can help identify these issues.

Of course, inspecting the data and the model generation process is something that takes time and attention. Thus, visualizations of this kind and other descriptions to increase transparency are best-suited to situations in which a human inspector is not under time pressure to sift through potentially complex patterns for sources of trouble. These methods are not well-suited for situations in which a specific decision must be made in a relatively short amount of time, e.g. providing decision-support to a clinician at the bedside.

Finally, transparency in the form of making code available can potentially assist in understanding how a model works, identifying bugs, and allowing independent testing by a third party (e.g., testing with a new set of inputs, evaluating counterfactuals in different testing distributions). However, if a model is sufficiently complex, as many modern models are, then simply having access to the code is likely not be enough for a human to gain sufficient understanding for their task.

33.3 Properties: The Abstraction Between Context and Method

Recall from the Terminology and Framework in Section 33.1.2 that the context and end-task determine what properties are needed for the explanation. For example, in a high-stakes setting—such as advising on interventions for an unstable patient—it may be important that the explanation completely and accurately reflects the model (fidelity). In contrast, in a discovery-oriented setting, it might be more important for any explanation to allow for efficient iterative refinement, revealing different aspects of the model in turn (interactivity). Not all contexts and end-tasks need all properties, and the lack of a key property may result in poor downstream performance.

While the research is still evolving, there exists a growing informal understanding about how properties may work as an abstraction between methods and contexts. Many interpretability methods from Section 33.2 share the same properties, and methods with the same properties may have similar downstream performance in a specific end-task and context. If two contexts and end-tasks require the same properties, then a method that works well for one may work well for the other. A method with properties well-matched for one context could miserably fail in another context.

How to find desired properties? Of course, identifying what properties are important for a particular context and end-task is not trivial. Indeed, identifying what properties are important for what contexts, end-tasks, and downstream performance metrics is one facet of the grand challenge of interpretable machine learning. For the present, the process of identifying the correct properties will likely require iteration via user studies. However, iterating over properties is still a much smaller space than iterating over methods. For example, if one wants to test whether the sparsity of the explanation is key to good downstream performance, one could intentionally create explanations of varying levels of sparsity to test that hypothesis. This is a much more precise knob to test than exhaustively trying out different explanation methods with different hyperparameters.

Below, we first describe examples of properties that have been discussed in the interpretable machine learning literature. Many of these properties are purely computational—that is, they can be determined purely from the model and the explanation. A few have some user-centric elements. Next we list examples of properties of explanation from cognitive science (on human to human explanations) and human-computer interaction (on machine to human explanations). Some of these properties have analogs in the machine learning list, while others may serve as inspiration for areas to formalize.

33.3.1 Properties of Explanations from Interpretable Machine Learning

Many lists of potentially-important properties of interpretable machine learning models have been compiled, sometimes using different terms for similar concepts and sometimes using the similar terms for different concepts. Below we list some commonly-described properties of explanations, knowing that this list will evolve over time as the field advances.

Faithfulness, Fidelity (e.g. as described in [JG20; JG21]). When the explanation is only a partial view of the model, how well does it match the model? There are many ways to make this notion precise. For example, suppose a mimic (simple model) is used to provide a global explanation of a more complex model. One possible measure of faithfulness could be how often the mimic gives the same outputs as the original. Another could be how often the mimic has the same first derivatives (local slope) as the original. In the context of a local explanation consisting of the 'key' features for a prediction, one could measure faithfulness by whether the prediction changes if the supposedly important features are flipped. Another measure could check to make sure the prediction does not change if a supposedly unimportant feature is flipped. The appropriate formalization will depend on the context.

Compactness, Sparsity (e.g. as described in [Lip18; Mur+19]). In general, an explanation must be small enough such that the user can process it within the constraints of the task (e.g. how quickly a decision must be made). Sparsity generally corresponds to some notion of smallness (a few features, a few parameters, $L1$ norm etc.). Compactness generally carries an additional notion of not including anything irrelevant (that is, even if the explanation is small enough, it could be made smaller). Each must be formalized for the context.

Completeness (e.g. as described in [Yeh+19b]). If the explanation is not the model, does it still include all of the relevant elements? For example, if an explanation consists of important features for a prediction, does it include all of them, or leave some out? Moreover, if the explanation uses derived quantities that are not the raw input features—for example, some notion of higher-level concepts—are they expressive enough to explain all possible directions of variation that could change the prediction? Note that one can have a faithful explanation in certain ways but not complete in others: Fore example, an explanation may be faithful in the sense that flipping features considered important flips the prediction and flipping features considered unimportant does not. However, the explanation may fail to include that flipping a set of unimportant features does change the prediction.

Stability (e.g. as described in [AMJ18a]) To what extent are the explanations similar for similar inputs? Note that the underlying model will naturally affect whether the explanation can be stable. For example, if the underlying model has high curvature and the explanation has limited expressiveness, then it may not be possible to have a stable explanation.

Actionability (e.g. as described in [Kar+20b; Poy+20]). Actionability implies filtering the content of the explanation to focus on only aspects of the model that the user might be able to intervene on. For example, if a patient is predicted to be at high risk of heart disease, an actionable explanation might only include mutable factors such as exercise and not immutable factors such as age or genetics. The notion of recourse corresponds to actionability in a justice context.

Modularity (e.g. as described in [Lip18; Mur+19]). Modularity implies that the explanation can be broken down into understandable parts. While modularity does not guarantee that the user can explain the system as a whole, for more complex models, modular explanations—where the user can inspect each part—can be an effective way to provide a reasonable level of insight into the model's workings.

Interactivity. (e.g., [Ten+20]) Does the explanation allow the user to ask questions, such as how the explanation changes for a related input, or how an output changes given a change in input? In some contexts, providing everything that a user might want or need to know from the start might be overwhelming, but it might be possible to provide a way for the user to navigate the information about the model in their own way.

Translucence (e.g. as described in [SF20; Lia+19]). Is the explanation clear about its limitations? For example, if a linear model is locally fit to a deep model at a particular input, is there a mechanism that reports that this explanation may be limited if there are strong feature interactions around that input? We emphasize that translucence is about exposing limitations in the explanation, rather than the model. As with all accountability methods, the goal of the explanation is to expose the limitations of the model.

Simulability (e.g. as described in [Lip18; Mur+19]). A model is simulable if a user can take the model and an input and compute the output (within any constraints of time and cognition). A simulable explanation is an explanation that is a simulable model. For example, a list of features is not simulable, because a list of features alone does not tell us how to compute the output. In contrast, an explanation in the form of a decision tree does include a computation process: the user can follow the logic of the tree, as long as it is not too deep. This example also points out an important difference between compactness and simulability: if an explanation is too large, it may not be simulable. However, just because an explanation is compact—such as a short list of features—does not mean that a person can compute the model's output with it.

It may seem that simulability is different from the other properties because its definition involves human input. However, in practice, we often know what kinds of explanations are easy for people to simulate (e.g. decision trees with short path lengths, rule lists with small formulas, etc.). This knowledge can be turned into a purely computational training constraint where we seek simulatable explanations.

Alignment to the User's Vocabulary and Mental Model. (e.g., as described in [Kim+18a]) Is the content of the explanation designed for the user's vocabulary? For example, the explanation could be given in the semantics a user knows, such as medical conditions vs. raw sensor readings. Doing so can help the user more easily connect the explanation to their knowledge and existing decision-making guidelines [Clo+19]. Of course, the right vocabulary will depend on the user: an explanation in terms of parameter variances and influential points may be comprehensible to an engineer debugging a lending model but not to a loan applicant.

Like simulability, mental-model alignment is more human-centric. However, just as before, we can imagine an abstraction between eliciting vocabulary and mental models from users (i.e., determining how they define their terms and how to think), and ensuring that an explanation is provided in alignment with whatever that elicited user vocabulary and mental model is.

Once desired properties are identified, we need to operationalize them. For example, if sparsity is a desired property, would using the L1 norm be enough? Or does a more sophisticated loss term need to be designed? This decision will necessarily be human-centric: how small an explanation needs to be, or in what ways it needs to be small, is a decision that needs to consider how people will be using the explanation. Once operationalized, most properties can be optimized computationally. That said, the properties should be evaluated with the context, end-task, model and chosen explanation methods. Once evaluated, one may revisit the choice of the explanation and model.

Finally, we emphasize that the ability to achieve a particular property will depend on the ***intrinsic*** characteristics of the model. For example, the behavior of a highly nonlinear model with interactions between the inputs will, in general, be harder to understand than a linear model. No matter how we try to explain it, if we are trying to explain something complicated, then users will have a harder time understanding it.

33.3.2 Properties of Explanations from Cognitive Science

Above we focused on computational properties between models and explanations. The fields of cognitive science and human-computer interaction have long examined what people consider good properties of an explanation. These more human-centered properties may be ones that researchers in machine learning may be less aware of, yet essential for communicating information to people.

Unsurprisingly, the literature on human explanation concurs that the explanation must fit the context [VF+80]; different contexts require different properties and different explanations. That said, human explanations are also social constructs, often including post-hoc rationalizations and other biases. We should focus on properties that help users achieve their goals, not ones simply "because people sometimes do it."

Below we list several of these properties.

Soundness (e.g., as described in [Kul+13]). Explanations should contain nothing but the truth with respect to whatever they are describing. Soundness corresponds to notions of *compactness* and *faithfulness* above.

Completeness (e.g., as described in [Kul+13]). Explanations should contain the whole truth with respect to whatever they are describing. Completeness corresponds to notions of *completeness* and *faithfulness* above.

Generality (e.g., as described in [Mil19]). Overall, people understand that an explanation for one context may not apply in another. That said, there is an expectation that an explanation should reflect some underlying mechanism or principle and will thus apply to similar cases—for whatever notion of similarity is in the person's mental model. Explanations that do not generalize to similar cases may be misinterpreted. Generality corresponds to notions of *stability* above.

Simplicity (e.g., as described in [Mil19]). All of the above being equal, simpler explanations are generally preferred. Simplicity relates to notions of *sparsity* and *complexity* above.

Contrastiveness (e.g., as described in [Mil19]). Contrastive explanations provide information of

how something differs from an alternate decision or prediction. For example, instead of providing a list of features for why a particular drug is recommended, it might provide a list of features that explain why one drug is recommended over another. Contrastiveness relates to notions of *actionability* above, and more generally explanation types that include *counterfactuals*.

Finally, the cognitive science literature also notes that explanations are often goal directed. This matches the notion of explanation in ML as information that helps a person improve performance on their end-task. Different information may help with different goals, and thus human explanations take many forms. Examples include deductive-nomological forms (i.e. a logical proofs) [HO48], forms that provide a sense of an underlying mechanism [BA05; Gle02; CO06], and forms that conveying understanding [Kei06]. Knowing these forms can help us consider what options might be best among different sets of interpretable machine learning methods.

33.4 Evaluation of Interpretable Machine Learning Models

One cannot formalize the notion of interpretability without specifying the context, the end-task, and the downstream performance metric [VF+80]. If one explanation empowers the human to get better performance on their end-task over another explanation, then it is more useful. While the grand challenge of interpretable machine learning is to develop a general understanding of what properties are needed for good downstream performance on different end-tasks in different contexts, in this section, we will focus on rigorous evaluation within one context [DVK17].

Specifically, we describe two major categories for evaluating interpretable machine learning methods:

Computational evaluations of properties (without people). Computational evaluations of whether explanations have desired properties do not user studies. For example, one can computationally measure whether a particular explanation satisfies a definition of faithfulness under different training and test data distributions or whether the outputs of one explanation are more sparse than another. Such measures are valuable when one already knows that certain properties may be important for certain contexts. Computational evaluations also serve as intermediate evaluations and sanity checks to identify undesirable explanation behavior prior to a more expensive user study-based evaluation.

User studies (with people). Ultimately, user studies are needed to measure how well an interpretable machine learning method enables the user to complete their end-task in a given context. Performing a rigorous, well-designed user study in a real context is significant work—much more so than computing a test likelihood on benchmark datasets. It requires significant asks of not only the researchers but also the target users. Methods for different contexts will also have different evaluation challenges: while a system designed to assist with optimizing music recommendations might be testable on a wide population, a system designed to help a particle physicist identify new kinds of interactions might only be tested with one or two physicists because people with that expertise are hard to find. In all cases, the evaluation can be done rigorously given careful attention to experimental design.

33.4.1 Computational Evaluation: Does the Method have Desired Properties?

While the ultimate measure of interpretability is whether the method successfully empowers the user to perform their task, properties can serve as a valuable abstraction. Checking whether an explanation has the right computational and desired properties can ensure that the method works as expected (e.g., no implementation errors, no obviously odd behaviors). One can iterate on novel, computationally-efficient methods to optimize the quantitative formalization of a property before conducting expensive human experiments. Computational checks can also ensure whether properties that held for one model continue to hold when applied to another model. Finally, checking for specific properties can also help pinpoint in what way an explanation is falling short, which may be less clear from a user study due to confounding.

In some cases, one might be able to prove mathematically that an explanation has certain properties, while in others the test must be empirical. For empirical testing, one umbrella strategy is to use a hypothesis-based sanity check; if we think a phenomenon X should never occur (hypothesis), we can test whether we can create situations where X may occur. If it does, then the method fails this sanity check. Another umbrella strategy is to create datasets with known characteristics or ground truth explanations. These could be purely synthetic constructions (e.g. generated tables with intentionally correlated features), semi-synthetic approaches (e.g. intentionally changing the labels on an image dataset), or taking slices of a real dataset (e.g. introduce intentional bias by only selecting real image, label pairs that are of outdoor environments). Note that these tests can only tell us if something is wrong; if a method passes a check, there may still be missing blindspots.

Examples of Sanity Checks. One strategy is to come up statements of the form "if this explanation is working, then phenomenon X should not be occurring" and then try to create a situation in which phenomenon X occurs. If we succeed, then the sanity check fails. By asking about out-of-the-box phenomena, this strategy can reveal some surprising failure modes of explanation methods.

For example, [Ade+20a] operates under the assumption that a faithful explanation should be a function of a model's prediction. The hypothesis is that the explanation should significantly change when comparing a trained model to an untrained model (where prediction is random). They show that many existing methods fail to pass this sanity check (Figure 33.4).

In another example, [Kin+19] hypothesize that a faithful explanation should be invariant to input transformations that do not affect model predictions or weights, such as constant shift of inputs (e.g., all inputs are added by 10). This hypothesis can be seen as testing both faithfulness and stability properties. Their work shows that some methods fail this sanity check.

Adversarial attacks on explanations also fall into this category. For example, [GAZ19] shows that two perceptively indistinguishable inputs with the same predicted label can be assigned very different explanations.

Examples using (semi-)Synthetic Datasets. Constructed datasets can also help score properties of various methods. We use the work of [YK19] as an example. Here, the authors were interested in explanations with the properties of compactness and faithfulness: it should not identify features as important if they are not. To test for these properties, the authors generate images with known correlations. Specifically, they generate multiple datasets, each with a different rate of how often each particular foreground object co-occurs with each particular background (see Figure 33.3). Each

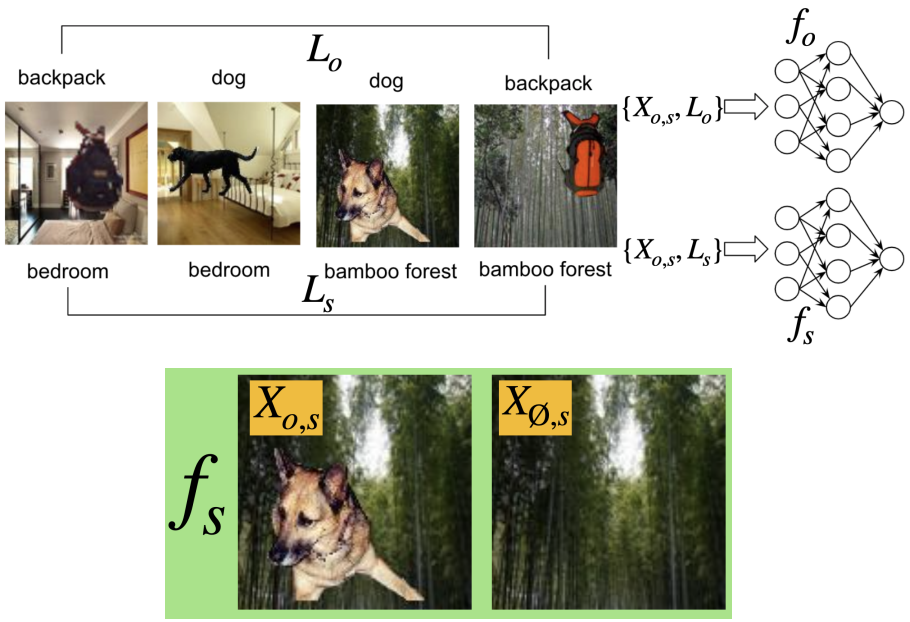


Figure 33.3: An example of computational evaluation using (semi-)synthetic datasets from [YK19]: foreground images (e.g. dogs, backpacks) are placed on different backgrounds (e.g. indoors, outdoors) to test what an explanation is looking for.

dataset comes with two labels per image: for the object and the background.

Now, the authors compare two models: one trained to classify objects and one trained to classify backgrounds (left, Figure 33.3). If a model is trained to classify objects and they all happen to have the same background, the background should be less important than in a model trained to classify backgrounds ([YK19] call this 'Model Contrast Score'). They also checked that the model trained to predict backgrounds was not providing attributions to the foreground objects (see right Figure 33.3). Other works using similar strategies include [Wil+20b; Gha+21; PMT18; KPT21; Yeh+19b; Kim+18b].

Examples with Real Datasets. While more difficult, it is possible to at least partially check for certain kinds of properties on real datasets that have no ground-truth.

For example, suppose an explanation ranks features from most to least important. We want to determine if this ranking is faithful. Further, suppose we can assume that the features do not interact. Then, one can attempt to make the prediction just with the top-1 most important feature, just the top-2 ranked features, etc. and observe if the change in prediction accuracy exhibits diminishing returns as more features are added. (If the features do interact, this test will not work. For example, if features A, B, C are the top-3 features, but C is only important if feature B is present, the test above would over-estimate the importance of the feature C.)

Figure 33.5 shows an example of this kind of test [Gho+19]. Their method outputs a set of

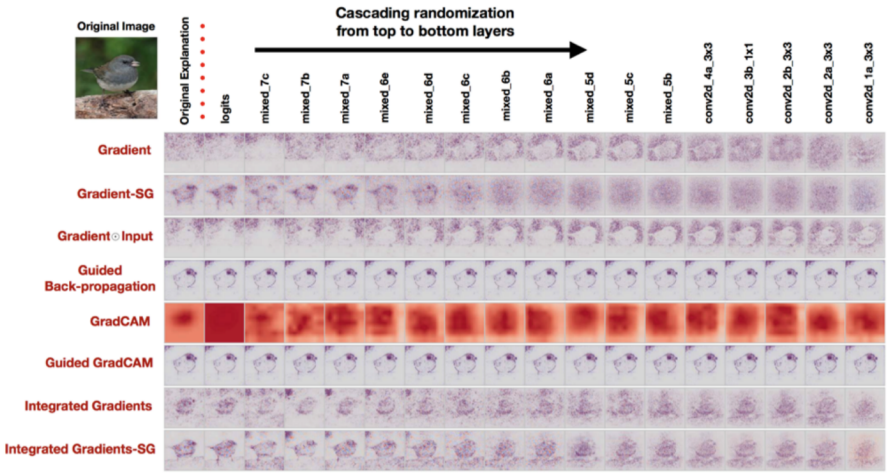


Figure 33.4: Interpretability methods (each row) and their explanations as we randomize layers starting from the logits, and cumulatively to the bottom layer (each column), in the context of image classification task. The rightmost column is showing a completely randomized network. Most methods output similar explanations for the first two columns; one predicts the bird, and the other predicts randomly. This sanity check tests the hypothesis that the explanation should significantly change (quantitatively and qualitatively) when comparing a trained model and an untrained model [Ade+20a].

image patches (e.g., a set of connected pixels) that correlates with the prediction. They add top-n image patches provided by the explanation one by one and observe the desired trend in accuracy. A similar experiment in reverse direction (i.e., deleting top-n most important image patches one by one) provides additional evidence. Similar experiments are also conducted in [FV17; RSG16a].

For example, in [DG17], authors define properties in plain English first: Smallest sufficient region (smallest region of the image that alone allows a confident classification) and Smallest destroying region (smallest region of the image that when removed, prevents a confident classification), followed by careful operationalization of these properties such that they become the objective for optimization. Then, separately, an evaluation metric of saliency is defined to be " the tightest rectangular crop that contains the entire salient region and to feed that rectangular region to the classifier to directly verify whether it is able to recognise the requested class". While the "rectangular" constraint may introduce artifacts, it is a neat trick to make evaluation possible. By checking expected behavior as described above, authors confirm that methods' behavior on the real data is aligned with the defined property compared to baselines.

Evaluating the Evaluations. As we have seen so far, there are many ways to formalize a given property and many empirical tests to determine whether a property is present. Each empirical test will have different qualities. As an illustration, in [Tom+20], the authors ask whether popular saliency metrics give consistent results across literature. They tested whether different metrics for assessing the quality of saliency-based explanations (explanations that identify important pixels or regions in images) is evaluating similar properties. In other words, this work tests consistency and stability

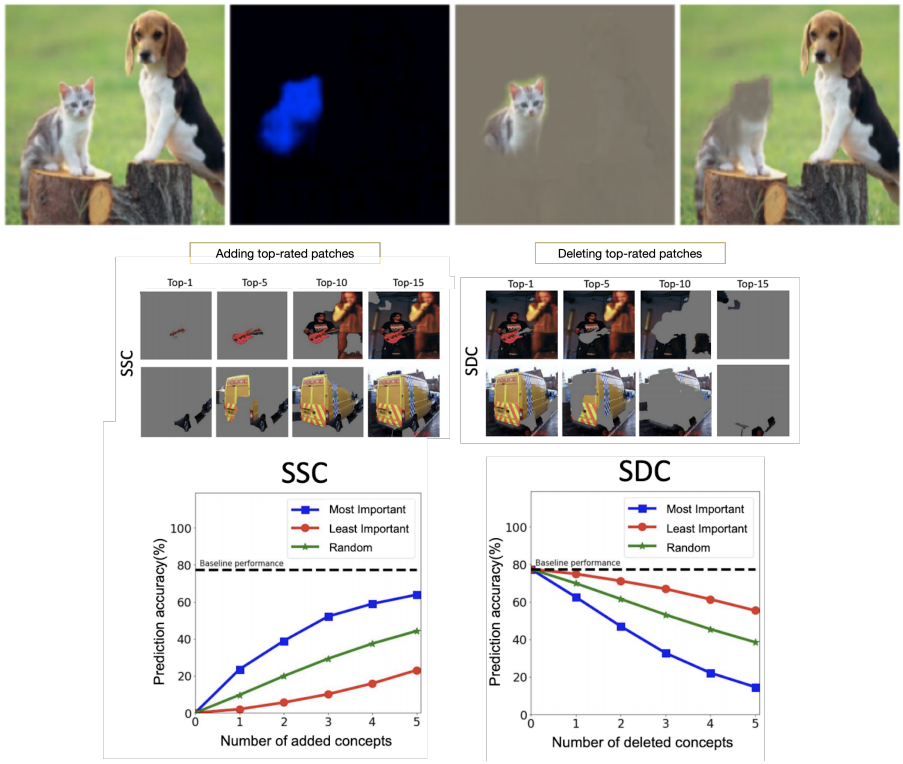


Figure 33.5: Examples of computational evaluation with real datasets. Top row is from Figure 1 of [DG17], used with kind permission of Yarin Gal. Bottom row is from Figure 4 of [Gho+19]. One would expect that adding or deleting patches rated as most 'relevant' for an image classification would have a large effect on the classification compared to patches not rated as important.

properties of *metrics*. They show many metrics are statistically unreliable and inconsistent. While each metric may still have a particular use [Say+19], knowing this inconsistency exists helps us better understand the landscape and limitations of evaluation approaches. Developing good evaluations for computational properties is an ongoing area of research.

33.4.2 User Study-based Evaluation: Does the Method Help a User Perform a Target Task?

User study-based evaluations measure whether an interpretable machine learning method helps a human perform some task. This task could be the ultimate end-task of interest (e.g. does a method help a doctor make better treatment decisions) or a synthetic task that mirrors contexts of interest (e.g. a simplified situation with artificial diseases and symptoms). In both cases, rigorous experimental design is critical to ensuring that the experiment measures what we want it to measure. Understanding experimental design for user studies is essential for research in interpretable machine

learning.

33.4.2.1 User Studies in Real Contexts.

The gold standard for testing whether an explanation is useful is to test it in the intended context: Do clinicians make better decisions with a certain kind of decision support? Do programmers debug code faster with a certain kind of explanation about model errors? Do product teams create more fair models for their businesses? A complete guide on how to design and conduct user studies is out of scope for this chapter; below we point out some basic considerations.

33.4.2.2 Basic elements of user studies

Performing a high-quality user study is a nuanced and non-trivial endeavor. There are many sources of bias, some obvious (e.g. learning and fatigue effects during a study) and some less obvious (e.g. participants willing to work with us are more optimistic about AI technology than those we could not recruit, or different participants may have different needs for cognition).

Interface Design. The explanation must be presented to the user. Unlike the *intrinsic* difficulty of explaining a model (i.e., complex models are harder to explain than simple ones), the design of the interface is an *extrinsic* source of difficulty that can confound the experimental results. For example, it may be easier, in general, to scan a list of features ordered by importance rather than alphabetically.

When we perform an evaluation with respect to an end-task, intrinsic and extrinsic difficulties can get conflated. Does one explanation type work better because it does a better job of explaining the complex system? Or does it work better simply because it was presented in a way that was easier for people to use? Especially if the goal is to test the difference between one explanation and another in the experiment, it is important that the interface for each is designed to tease out the effect from the explanations and their presentations. (Note that good presentations and visualization are an important but different object of study.) Moreover, if using the interface requires training, it is important to deliver the training to users in a way that is neutral in each testing condition. For example, how the end-task and goals of the study are described during training (e.g. with practice questions) will have a large impact on how users approach the task.

Baselines. Simply the presence of an explanation may change the way in which people interact with an ML system. Thus, it is often important to consider how a human performs with no ML system, with an ML system and no explanation, with an ML system and a placebo explanation (an explanation that provides no information), and with an ML system and hand-crafted explanations (manually generated by humans who are presumably good communicators).

Experimental Design and Hypothesis Testing. Independent and dependent variables, hypotheses, and inclusion and exclusion criteria must be clearly defined prior to the start of the study. For example, suppose that one hypothesizes that a particular explanation will help a developer debug an image classifier. In this case, the independent variable would be a form of assistance: the particular explanation, competing explanation methods, and the baselines above. The dependent variable would be whether the developer can identify bugs. Inclusion and exclusion criteria might

include a requirement that the developer has sufficient experience training image classifiers (as determined by an initial survey, or a pre-test), demonstrates engagement (as measured by a base level of performance on practice rounds), and does not have prior experience with the particular explanation types (as determined by an initial survey). Other exclusion criteria could be removing outliers. For example, one could decide, in advance, to exclude data from any participant that takes an unusually long or short time to perform task as a proxy for engagement.

As noted in Section 33.2, there are many decisions that go into any interpretable machine learning method, and each context is nuanced. Studies of the form "Does explanation X (computed via some pipeline Y) help users in context Z compared to explanation X'?" may not provide much insight as to *why* that particular explanation is better or worse—making it harder not only to iterate on a particular explanation but also to generalize to other explanations or contexts. There are many factors of potential variation in the results, ranging from the properties of the explanation and its presentation to the difficulty of the task.

To reduce this variance, and to get more useful and generalizable insights, we can manipulate some factors of variation directly. For example, suppose the research question is whether complete explanations are better than incomplete explanations in a particular context. One might write out hand-crafted explanations that are complete in what features they implicate, explanations in which one important feature is missing, and explanations in which several important features are missing. Doing so ensures even coverage of the different experimental regimes of interest, which may not occur if the explanations were simply output from a pipeline. As another example, one might intentionally create an image classifier with known bugs, or simply pretend to have an image classifier that makes certain predictions (as done in [Ade+20b]). These kinds of studies are called *wizard-of-oz* studies, and they can help us more precisely uncover the science of why an explanation is useful (e.g. as done in [Jac+21]).

Once the independent and dependent variables, hypotheses, and participant criteria (including how the independent and dependent variables may be manipulated) are determined, the next step is setting up the study design itself. Broadly speaking, *randomization* marginalizes over potential confounds. For example, randomization in assigning subjects to tasks marginalizes the subject's prior knowledge; randomization in the order of tasks marginalizes out learning effects. *Matching* and *repeated measures* reduce variance. An example of matching would be asking the same subject to perform the same end-task with two different explanations. An example of repeated measures would be asking the subject to perform the end-task for several different inputs.

Other techniques for designing user studies include block randomized designs/Latin square designs that randomize the order of explanation types while keeping tasks associated with each explanation type grouped together. This can be used to marginalize the effects of learning and fatigue without too much context switching. Careful consideration should be given to what will be compared within subjects and across subjects. Comparisons of task performance within subjects will have lower variance but a potential bias from learning effects from the first task to the second. Comparisons across subjects will have higher variance and also potential bias from population shift during experimental recruitment. Finally, each of these study designs, as well as the choice of independent and dependent variables, will imply an appropriate significance test. It is essential to choose the right test and multiple hypothesis correction to avoid inflated significance values while retaining power.

Qualitative Studies. So far, we have described the standard approach for the design of a quantitative user study–one in which the dependent variable is numerically measured (e.g. time

taken to correctly identify a bug, % bugs detected). While quantitative studies provide value by demonstrating that there is a consistent, quantifiable effect across many users, they usually do not tell us *why* a certain explanation worked. In contrast, qualitative studies, often performed with a "think-aloud" or other discussion-based protocol in which users expose their thought process as they perform the experiment, can help identify why a particular form of explanation seems to be useful or not. The experimenter can gain insights by hearing how the user was using the information, and depending on the protocol, can ask for clarifications.

For example, suppose one is interested in how people use an example-based explanation to understand a video-game agent's policy. The idea is to show a few video clips of an automated agent in the video game, and then ask the user what the agent might do in novel situations. In a think-aloud study, the user would perform this task while talking through how they are connecting the videos they have seen to the new situation. By hearing these thoughts, a researcher might not only gain deeper insight into how users make these connections—e.g. users might see the agent collect coins in one video and presume that the agent will always go after coins—but they might also identify surprising bugs: for example, a user might see the agent fall into a pit and attribute it to a one-off sloppy fingers, not internalizing that an automated agent might make that mistake every time.

While a participant in a think-aloud study is typically more engaged in the study than they might be otherwise (because they are describing their thinking), knowing their thoughts can provide insight into the causal process between what information is being provided by the explanation and the action that the human user takes, ultimately helping advance the science of how people interact with machine-provided information.

Pilot Studies: The above descriptions are just a very high-level overview of the many factors that must be designed properly for a high-quality evaluation. In practice, one does not typically get all of these right the first time. Small scale pilot studies are essential to checking factors such as whether participants attend to the provided information in unexpected ways or whether instructions are clear and well-designed. Modifying the experiments after iterative small scale pilot studies can save a lot of time and energy down the road. In these pilots, one should collect not only the usual information about users and the dependent variables, but also discuss with the participants how they approached the study tasks and whether any aspects of the study were confusing. These discussions will lead to insights and confidence that the study is testing what it is intended to test. The results from pilot studies should not be included in the final results.

Finally, as the number of factors to test increases (e.g., baselines, independent variables), the study design becomes more complex and may require more participants and longer participation times to determine if the results are significant—which can in turn increase costs and effects of fatigue. Pilots, think-aloud studies, and careful thinking about what aspects of the evaluation require user studies and what can be completed computationally can all help distill down a user-based evaluation to the most important factors.

33.4.2.3 User Studies in Synthetic Contexts

It is not always appropriate or possible to test an interpretable machine learning method in the real context: for example, it would be unethical to test a prototype explanation system on patients each time one has a new way to convey information about a treatment recommendation. In such cases, we

might want to run an experiment in which clinicians perform a task on made-up patients, or in some analogous non-medical context where the participant pool is bigger and more affordable. Similarly, one might create a relatively accessible image classification debugging context where one can control the incorrect labels, distribution shifts, etc. (e.g. [Ade+20b]) and see what explanations help users detect problems in this simpler setting. The convenience and scalability of using a simpler setting could shed light on what properties of explanations are important generally (e.g., for debugging image classification). For example, we can test how different forms of explanation have different cognitive loads or how a particular property affects performance with a relatively large pool of subjects (e.g., [Lag+19]). The same principles we outlined above for user studies in real contexts continue to apply, but there are some important cautions.

Cautions regarding synthetic contexts: While user studies with synthetic contexts can be valuable for identifying scientific principles, one must be cautious. For example, experimental subjects in a synthetic high-stakes context may not treat the stakes of the problem as seriously, may be relatively unburdened with respect to distractions or other demands on their time and attention (e.g. a quiet study environment vs. a chaotic hospital floor), and ignore important factors of the task (e.g., clicking through to complete the task as quickly as possible). Moreover, small differences in task definition can have big effects: even the difference between asking users to simply perform a task with an explanation available vs. asking users to answer some questions about the explanation first, may create very different results as the latter forces the user to pay attention to the explanation and the former does not. Priming users by giving them a specific scenario where they can put themselves into a mindset could help. For example: "Imagine now you are an engineer at a company selling a risk calculator. A deadline is approaching and your boss wants to make sure the product will work for a new client. Describe how you would use the following explanation."

33.5 Discussion: How to Think about Interpretable Machine Learning

Interpretable machine learning is a young, interdisciplinary field of study. As a result, consensus on definitions, evaluation methods, and appropriate abstractions is still forming. The goal of this section is to lay out a core set of principles about interpretable machine learning. While specifics in the previous sections may change, the principles below will be durable.

There is no universal, mathematical definition of interpretability, and there never will be. Defining a downstream performance metric (and justifying it) for each context is a must. The information that best communicates to the human what is needed to perform a task will necessarily vary: for example, what a clinical expert needs to determine whether to try a new treatment policy is very different than what a person to determine how to get a denied loan approved. Similarly, methods to communicate characteristics of models built on pixel data may not be appropriate for communicating characteristics of models built on language data. We may hope to identify desired properties in explanations to maximize downstream task performance for different classes of end tasks—that is the grand challenge of interpretable machine learning—but there will never be one metric for all contexts.

While this lack of a universal metric may feel disappointing, other areas of machine learning also lack universal metrics. For example, not only is it impossible to satisfy the many metrics on fairness at the same time [KMR16], but also in a particular situation, none may exactly match the

desires of the stakeholders. Even in a standard classification setting, there are many metrics that correspond to making the predicted and true labels as close as possible. Does one care about overall accuracy? Precision? Recall? It is unlikely that one objective captures everything that is needed in one situation, much less across different contexts. Evaluation can still be rigorous as long as assumptions and requirements are made precise.

What sets interpretable machine learning apart from other areas of machine learning, however, is that a large class of evaluations require human input. As a necessarily interdisciplinary area, rigorous work in interpretable machine learning requires not only knowledge of computation and statistics but also experimental design and user studies.

Interpretability is only a part of the solution for fairness, calibrated trust, accountability, causality and other important problems. Learning models that are fair, safe, causal, or engender calibrated trust are all goals, whereas interpretability is one *means* toward that goal.

In some cases, we don't need interpretability. For example, if the goal can be fully formalized in mathematical terms (e.g., a regulatory requirement may mandate a model satisfy certain fairness metrics), we do not need any human input. If a model behaves as expected across an exhaustive set of pre-defined inputs, then it may be less important to understand how it produced its outputs. Similarly, if a model performs well across a variety of regimes, that might (appropriately) increase one's trust in it; if it makes errors, that might (appropriately) decrease trust without an inspection of any of the system's internals.

In other cases, human input is needed to achieve the end-task. For example, while there is much work in the identification of causal models (see Chapter Chapter 36), under many circumstances, it is not possible to learn a model that is *guaranteed* to be causal from a dataset alone. Here, interpretability could assist the end-task of "Is the model causal?" by allowing a human to inspect the model's prediction process.

As another example, one could measure the safety of a clinical decision support system by tracking how often its recommendations causes harm to patients—and stop using the system if it causes too much harm. However, if we use this approach to safety, we will only discover that the system is unsafe *after* a significant number of patients have been harmed. Here, interpretability could support the end-task of safety by allowing clinical experts to inspect the model's decision process for red flags *prior* to deployment.

In general, complex contexts and end-tasks will require a constellation of methods (and people) to achieve them. For example, formalizing a complex notion such as accountability will require a broad collection of people—from policy makers and ethicists to corporations, engineers, and users—unifying vocabularies, exchanging domain knowledge, and identifying goals. Evaluating or monitoring it will involve various empirical measures of quality and insights from interpretability.

Interpretability is not about understanding everything about the model; it is about understanding enough to do the end-task. The ultimate measure of an interpretable machine learning method is whether it helps the user perform their end-task. Suppose the end-task is to fix an overheating laptop. An explanation that lists the likely sources of heat is probably sufficient to address the issue, even if one does not know the chemical properties of its components. On the other hand, if the laptop keeps freezing up, knowing about the sources of heat may not be the right information. Importantly, both end-tasks have clear downstream performance metrics: we can observe whether the information helped the user perform actions that make the laptop overheat or freeze up less.

As another example, consider AlphaGo, Google DeepMind's AI Go player that beat the human world champion, Lee SeDol. The model is so complex that one cannot fully understand its decision process, including surprising moves like its famous move 37[Met16]. That said, partial probes (e.g., does AlphaGo believe the same move would have made a different impact if it was made earlier but similar position in the game) might still help a Go expert gain insights on the rationale for the move in the context of what they already know about the game.

Relatedly, interpretability is distinct from full transparency into the model or knowing the model's code. Staring at the weights of every neuron in a large network is likely to be as effective as taking one's laptop apart to understand a bug in your code. There are many good reasons for open source projects and models, but open source code itself may or may not be sufficient for a user to accomplish their end-task. For example, a typical user will not be able to reason through $100K$ lines of parameters despite having all the pieces available.

That said, any partial view of a model is, necessarily, only a partial view; it does not tell the full story. While we just argued that many end-tasks do not require knowing everything about a model, we also must acknowledge that a partial view does not convey the full model. For example, the set of features needed to change a loan decision may be the right partial view for a denied applicant, but convey nothing about whether the model is discriminatory. Any probe will only return what it is designed to compute (e.g., an approximation of a complex function with a simpler one). Different probes may be able to reveal different properties at different levels of quality. Incorrectly believing the partial view is the full story could result in incorrect insights.

Partial views can lack stability and enable attacks. Relatedly, any explanation that reveals only certain parts of a model can lack stability (e.g. see [AMJ18a]) and can be more easily attacked (e.g. see [Yeh+19a; GAZ19; Dom+19; Sla+20]). Especially when models are overparameterized such as neural networks, it is possible to learn models whose explanations say one thing (e.g. a feature is not important, according to some formalization of feature importance) while the model does another (e.g. uses the prohibited feature). Joint training can also exacerbate the issue, as it allows the model to learn boundaries that pass some partial-view test while in reality violating the underlying constraint. Other adversarial approaches can work on the input, minimally perturbing it to change the explanation's partial view while keeping the prediction constant or to change the prediction while keeping the explanation constant.

These concerns highlight an important open area: We need to improve ways to endow explanations with the property of translucence, that is, explanations that communicate what they can and cannot say about the model. Translucence is important because misinterpreted explanations that happen to favor a user's views create false basis for trust.

Trade-offs between inherently interpretable models and performance often do not exist; partial views can help when they do.

While some have claimed that there exists an inherent trade-off between using an inherently-interpretable model and performance (defined as a model's performance on some test data), this trade-off does not always exist in practice for several reasons[Rud19].

First, in many cases, the data can be surprisingly well-fit by a fairly simple model (due to high noise, for example) or a model that can be decomposed into interpretable parts. One can often find a

combination of architecture, regularizer, and optimizer that produces inherently interpretable models with performance comparable to, or sometimes even better than, blackbox approaches [Wan+17a; LCG12; Car+15; Let+15b; UR16; FHDV20; KRS14]. In fact, interpretability and performance can be synergistic: methods for encoding a preference for simpler models (e.g., L1 regularizer for sparsity property) were initially developed to increase performance and avoid overfitting, and interpretable models are often more robust [RDV18].

Second, a narrow focus on the trade-off between using inherently interpretable models and a predefined metric of performance, as usually measured on a validation set, overlooks a broader issue: that predefined metric of performance may not tell the full story about the quality of the model. For example, using an inherently interpretable model may enable a person to realize that a prediction is based on confounding, not causation—or other ways it might fail in deployment. In this way, one might get better performance with an inherently interpretable model in practice even if a blackbox appears to have better performance numbers in validation. An inherently interpretable model may also enable better human+model teaming by allowing the human user to step in and override the system appropriately.

Human factors are essential. All machine learning systems ultimately connect to broader socio-technical contexts. However, in many cases, many aspects of model construction and optimization can be performed in a purely computational setting: there are techniques to check for appropriate model capacity, techniques for tuning a gradient descent or convex optimization. In contrast, interpretable machine learning must consider human factors *from the beginning*: there is no point optimizing an explanation to have various properties if it still fails to improve the user's performance on the end-task.

Over-reliance. Just because an explanation is present, does not mean that the user will analytically and reasonably incorporate the information provided into their ultimate decision-making task. The presence of *any* explanation can increase a user's trust in the model, exacerbating the general issue of over-trust in human+ML teams. Recent studies have found that even data scientists over-trust explanations in unintended ways [Kau+20]; their excitement about the tool led them to take it at face-value rather than dig deeper. [LM20] reports a similar finding, noting that inaccurate but evocative presentations can create a feeling of comprehension.

Over-reliance can be combated with explicit measures to force the user to engage analytically and skeptically with the information in the explanation. For example, one could ask the user to submit their decision first and only then show the recommendation and accompanying explanation to pique their interest in why their choice and the recommendation might disagree (and prompting whether they want to change their choice). Another option is to ask the user some basic questions about the explanation prior to submitting their decision to force them to look at the explanation carefully. Yet another option is to provide only the relevant information (the explanation) without the recommendation, forcing the user to synthesize the additional information on their own. However, in all these cases, there is a delicate balance: users will often be amenable to expending additional cognitive effort if they can see it achieves better results, but if they feel the effort is too much, they may start ignoring the information entirely.

Potential for Misuse. A malicious version of over-reliance is when explanations are used to manipulate a user rather than facilitating the user's end-task. Further, users may report that they *like* explanations that are simple, require little cognitive effort, etc. even when those explanations do not

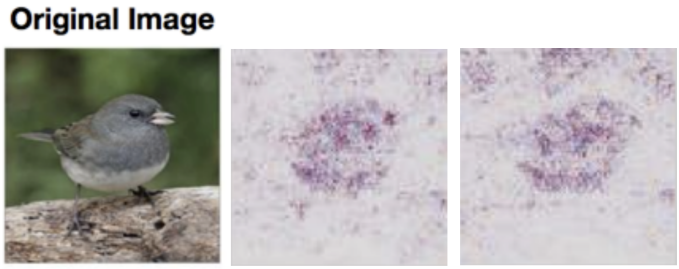


Figure 33.6: (Potential) perception issues: an explanation from a trained network (left) is visually indistinguishable to humans from one from an untrained network (right)—even if they are not exactly identical.

help them perform their end-task. As creators of interpretable machine learning methods, one must be on alert to ensure that the explanations help the user achieve what they want to (ideally in a way that they also like).

Misunderstandings from a lack of understanding of machine learning. Even when correctly engaged, users in different contexts will have different levels of knowledge about machine learning. For example, not everyone may understand concepts such as additive factors or Shapley values [Sha16]. Users may also attribute more understanding to a model than it actually has. For example, if they see a set of pixels highlighted around a beak, or a set of topic model terms about a disease, they may mistakenly believe that the machine learning model has some notion of concepts that matches theirs, when the truth might be quite different.

Related: Perception issues in image explanations. The nature of our visual processing system adds another layer of nuance when it comes to interpreting and misinterpreting explanations. In Figure 33.6, two explanations (in terms of important pixels in a bird image) seem to communicate a similar message; for most people, both explanations seem to suggest that the belly and cheek of the bird are the important parts for this prediction. However, one of them is generated from a trained network (left), but the other one is from a network that returns random predictions (right). While the two saliency maps aren't identical to machines, they look similar because humans don't parse an image as pixel values, but as whole: they see a bird in both pictures.

Another common issue with pixel-based explanations is that explanation creators often multiply the original image with an importance "mask" (black and clear saliency mask, where black pixel represents no importance and a clear pixel represents maximum importance), introducing the arbitrary artifact that black objects never appear important [Smi+17]. In addition, this binary mask is produced by clipping important pixels in a certain percentile (e.g., only taking 99−th percentile), which can also introduce another artifact [Sun+19c]. The balancing act between artifacts introduced by visualization for the ease of understanding and faithfully representing the explanation remains a challenge.

Together, all of these points on human factors emphasize what we said from the start: we cannot divorce the study and practice of interpretable machine learning from its intended socio-technical context.

Part VI

Action

34 Decision making under uncertainty

34.1 Bayesian decision theory

Bayesian inference provides the optimal way to update our beliefs about hidden quantities H given observed data $\mathbf{X} = \boldsymbol{x}$ by computing the posterior $p(H|\boldsymbol{x})$. However, at the end of the day, we need to turn our beliefs into **actions** that we can perform in the world. How can we decide which action is best? This is where **Bayesian decision theory** comes in. In this section, we give a brief introduction. For more details, see e.g., [DeG70; KWW22].

34.1.1 Basics

In decision theory, we assume the decision maker, or **agent**, has a set of possible actions, $\mathcal{A}$, to choose from. Each of these actions has costs and benefits, which will depend on the underlying **state of nature** $h \in \mathcal{H}$. We can encode this information into a **loss function** $\ell(h, a)$, that specifies the loss we incur if we take action $a \in \mathcal{A}$ when the state of nature is $h \in \mathcal{H}$.

Once we have specified the loss function, we can compute the **posterior expected loss** or **risk** for each possible action:

$$R(a|\boldsymbol{x}) \triangleq \mathbb{E}_{p(h|\boldsymbol{x})}\left[\ell(h, a)\right] = \sum_{h \in \mathcal{H}} \ell(h, a) p(h|\boldsymbol{x}) \tag{34.1}$$

The **optimal policy** (also called the **Bayes estimator**) specifies what action to take for each possible observation so as to minimize the risk:

$$\pi^*(\boldsymbol{x}) = \operatorname*{argmin}_{a \in \mathcal{A}} \mathbb{E}_{p(h|\boldsymbol{x})}\left[\ell(h, a)\right] \tag{34.2}$$

An alternative, but equivalent, way of stating this result is as follows. Let us define a **utility function** $U(h, a)$ to be the desirability of each possible action in each possible state. If we set $U(h, a) = -\ell(h, a)$, then the optimal policy is as follows:

$$\pi^*(\boldsymbol{x}) = \operatorname*{argmax}_{a \in \mathcal{A}} \mathbb{E}_h\left[U(h, a)\right] \tag{34.3}$$

This is called the **maximum expected utility principle**.

34.1.2 Classification

Suppose the states of nature correspond to class labels, so $\mathcal{H} = \mathcal{Y} = \{1, \ldots, C\}$. Furthermore, suppose the actions also correspond to class labels, so $\mathcal{A} = \mathcal{Y}$. In this setting, a very commonly used loss function is the **zero-one loss** $\ell_{01}(y^*, \hat{y})$, defined as follows:

	$\hat{y} = 0$	$\hat{y} = 1$
$y^* = 0$	0	1
$y^* = 1$	1	0

(34.4)

We can write this more concisely as follows:

$$\ell_{01}(y^*, \hat{y}) = \mathbb{I}(y^* \neq \hat{y}) \tag{34.5}$$

In this case, the posterior expected loss is

$$R(\hat{y}|\boldsymbol{x}) = p(\hat{y} \neq y^*|\boldsymbol{x}) = 1 - p(y^* = \hat{y}|\boldsymbol{x}) \tag{34.6}$$

Hence the action that minimizes the expected loss is to choose the most probable label:

$$\pi(\boldsymbol{x}) = \operatorname*{argmax}_{y \in \mathcal{Y}} p(y|\boldsymbol{x}) \tag{34.7}$$

This corresponds to the **mode** of the posterior distribution, also known as the **maximum a posteriori** or **MAP estimate**.

We can generalize the loss function to associate different costs for false positives and false negatives. We can also allow for a "**reject action**", in which the decision maker abstains from classifying when it is not sufficiently confident. This is called **selective prediction**; see Section 19.3.3 for details.

34.1.3 Regression

Now suppose the hidden state of nature is a scalar $h \in \mathbb{R}$, and the corresponding action is also a scalar, $y \in \mathbb{R}$. The most common loss for continuous states and actions is the ℓ_2 **loss**, also called **squared error** or **quadratic loss**, which is defined as follows:

$$\ell_2(h, y) = (h - y)^2 \tag{34.8}$$

In this case, the risk is given by

$$R(y|\boldsymbol{x}) = \mathbb{E}\left[(h - y)^2|\boldsymbol{x}\right] = \mathbb{E}\left[h^2|\boldsymbol{x}\right] - 2y\mathbb{E}\left[h|\boldsymbol{x}\right] + y^2 \tag{34.9}$$

The optimal action must satisfy the condition that the derivative of the risk (at that point) is zero (as explained in Chapter 6). Hence the optimal action is to pick the posterior mean:

$$\frac{\partial}{\partial y} R(y|\boldsymbol{x}) = -2\mathbb{E}\left[h|\boldsymbol{x}\right] + 2y = 0 \;\Rightarrow\; \pi(\boldsymbol{x}) = \mathbb{E}\left[h|\boldsymbol{x}\right] = \int h\, p(h|\boldsymbol{x}) dh \tag{34.10}$$

This is often called the **minimum mean squared error** estimate or **MMSE** estimate.

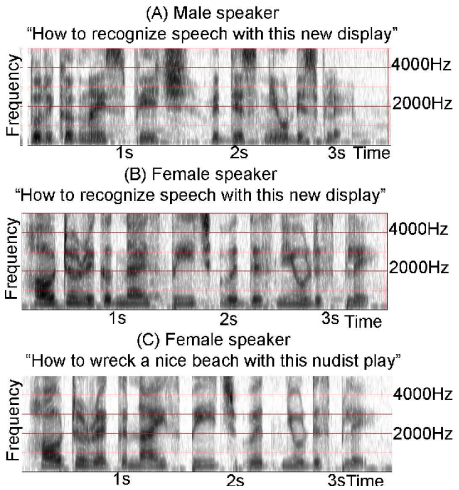


Figure 34.1: Spectograms for three different spoken sentences. The x-axis shows progression of time and the y-axis shows different frequency bands. The energy of the signal in different bands is shown as intensity in grayscale values with progression of time. (A) and (B) show spectrograms of the same sentence "How to recognize speech with this new display" spoken by two different speakers, male and female. Although the frequency characterization is similar, the formant frequencies are much more clearly defined in the speech of the female speaker. (C) shows the spectrogram of the utterance "How to wreck a nice beach with this nudist play" spoken by the same female speaker as in (B). (A) and (B) are not identical even though they are composed of the same words. (B) and (C) are similar to each other even though they are not the same sentences. From Figure 1.2 of [Gan07]. Used with kind permission of Madhavi Ganapathiraju.

34.1.4 Structured prediction

In some problems, such as natural language processing or computer vision, the desired action is to return an output object $\boldsymbol{y} \in \mathcal{Y}$, such as a set of labels or body poses, that not only is probable given the input $\boldsymbol{x}$, but is also internally consistent. For example, suppose $\boldsymbol{x}$ is a sequence of phonemes and $\boldsymbol{y}$ is a sequence of words. Although $\boldsymbol{x}$ might sound more like $\boldsymbol{y}$ = "How to wreck a nice beach" on a word-by-word basis, if we take the sequence of words into account then we may find (under a language model prior) that $\boldsymbol{y}$ = "How to recognize speech" is more likely overall. (See Figure 34.1.) We can capture this kind of dependency amongst outputs, given inputs, using a **structured prediction model**, such as a conditional random field (see Section 4.4).

In addition to modeling dependencies in $p(\boldsymbol{y}|\boldsymbol{x})$, we may prefer certain action choices $\hat{\boldsymbol{y}}$, which we capture in the loss function $\ell(\boldsymbol{y}, \hat{\boldsymbol{y}})$. For example, referring to Figure 34.1, we may be reluctant to assume the user said $\hat{y}_t$="nudist" at step t unless we are very confident of this prediction, since the cost of mis-categorizing this word may be higher than for other words.

Given a loss function, we can pick the optimal action using **minimum Bayes risk** decoding:

$$\hat{\boldsymbol{y}} = \min_{\hat{\boldsymbol{y}} \in \mathcal{Y}} \sum_{\boldsymbol{y} \in \mathcal{Y}} p(\boldsymbol{y}|\boldsymbol{x}) \ell(\boldsymbol{y}, \hat{\boldsymbol{y}}) \tag{34.11}$$

We can approximate the expectation empirically by sampling M solutions $\boldsymbol{y}^m \sim p(\boldsymbol{y}|\boldsymbol{x})$ from the

posterior predictive distribution. (Ideally these are diverse from each other.) We use the same set of M samples to approximate the minimization to get

$$\hat{\boldsymbol{y}} \approx \min_{\boldsymbol{y}^i, i \in \{1,\ldots,M\}} \sum_{j \in \{1,\ldots,M\}} p(\boldsymbol{y}^j|\boldsymbol{x}) \ell(\boldsymbol{y}^j, \boldsymbol{y}^i) \tag{34.12}$$

This is called **empirical MBR** [Pre+17a], who applied it to computer vision problems. A similar approach was adopted in [Fre+22], who applied it to neural machine translation.

34.1.5 Fairness

Models trained with ML are increasingly being used to high-stakes applications, such as deciding whether someone should be released from prison or not, etc. In such applications, it is important that we focus not only on accuracy, but also on **fairness**. A variety of definitions for what is meant by fairness have been proposed (see e.g., [VR18]), many of which entail conflicting goals [Kle18]. Below we mention a few common definitions, which can all be interpreted decision theoretically.

We consider a binary classification problem with true label Y, predicted label $\hat{Y}$ and **sensitive attribute** S (such as gender or race). The concept of **equal opportunity** requires equal true positive rates across subgroups, i.e., $p(\hat{Y} = 1|Y = 1, S = 0) = p(\hat{Y} = 1|Y = 1, S = 1)$. The concept of **equal odds** requires equal true positive rates across subgroups, and also equal false positive rates across subgroups, i.e., $p(\hat{Y} = 1|Y = 0, S = 0) = p(\hat{Y} = 1|Y = 0, S = 1)$. The concept of **statistical parity** requires positive predictions to be unaffected by the value of the protected attribute, regardless of the true label, i.e., $p(\hat{Y} = 1|S = 0) = p(\hat{Y}|S = 1)$.

A simple and generic way to achieve these fairness goals is to use constrained Bayesian optimization (Section 6.8), in which we maximize for accuracy subject to achieving one or more of the above goals. See [Per+21] for details.

34.2 Decision (influence) diagrams

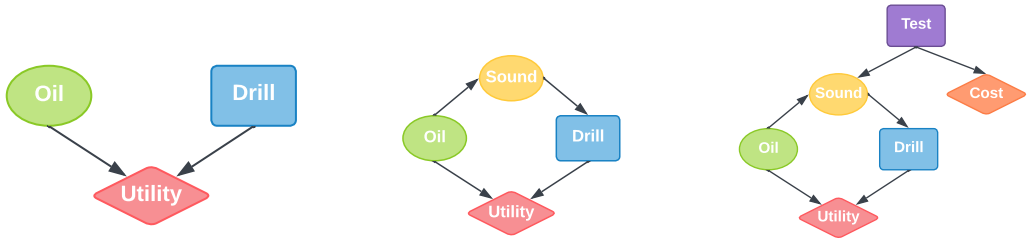
When dealing with structured multi-stage decision problems, it is useful to use a graphical notation called an **influence diagram** [HM81; KM08], also called a **decision diagram**. This extends directed probabilistic graphical models (Chapter 4) by adding **decision nodes** (also called **action nodes**), represented by rectangles, and **utility nodes** (also called **value nodes**), represented by diamonds. The original random variables are called **chance nodes**, and are represented by ovals, as usual.

34.2.1 Example: oil wildcatter

As an example (from [Rai68]), consider creating a model for the decision problem faced by an oil "**wildcatter**", which is a person who drills wildcat wells, which are exploration wells drilled in areas not known to be oil fields.

Suppose you have to decide whether to drill an oil well or not at a given location. You have two possible actions: $d = 1$ means drill, $d = 0$ means don't drill. You assume there are 3 states of nature: $o = 0$ means the well is dry, $o = 1$ means it is wet (has some oil), and $o = 2$ means it is soaking (has a lot of oil). We can represent this as a decision diagram as shown in Figure 34.2(a).

Suppose your prior beliefs are $p(o) = [0.5, 0.3, 0.2]$, and your utility function $U(d, o)$ is specified by the following table:



(a) (b) (c)

Figure 34.2: Influence diagrams for the oil wild catter problem. Ovals are random variables (chance nodes), squares are decision (action) nodes, diamonds are utility (value) nodes. (a) Basic model. (b) An extension in which we have an information arc from the Sound chance node to the Drill decision node. (c) An extension in which we get to decide whether to perform a test or not, as well as whether to drill or not.

	$o=0$	$o=1$	$o=2$
$d=0$	0	0	0
$d=1$	-70	50	200

We see that if you don't drill, you incur no costs, but also make no money. If you drill a dry well, you lose \$70; if you drill a wet well, you gain \$50; and if you drill a soaking well, you gain \$200.

What action should you take if you have no information beyond your prior knowledge? Your prior expected utility for taking action d is

$$\text{EU}(d) = \sum_{o=0}^{2} p(o)U(d,o) \tag{34.13}$$

We find $\text{EU}(d=0) = 0$ and $\text{EU}(d=1) = 20$ and hence the maximum expected utility is

$$\text{MEU} = \max\{\text{EU}(d=0), \text{EU}(d=1)\} = \max\{0, 20\} = 20 \tag{34.14}$$

Thus the optimal action is to drill, $d^* = 1$.

34.2.2 Information arcs

Now let us consider a slight extension to the model, in which you have access to a measurement (called a "sounding"), which is a noisy indicator about the state of the oil well. Hence we add an $O \to S$ arc to the model. In addition, we assume that the outcome of the sounding test will be available before we decide whether to drill or not; hence we add an **information arc** from S to D. This is illustrated in Figure 34.2(b). Note that the utility depends on the action and the true state of the world, but not the measurement.

We assume the sounding variable can be in one of 3 states: $s = 0$ is a diffuse reflection pattern, suggesting no oil; $s = 1$ is an open reflection pattern, suggesting some oil; and $s = 2$ is a closed reflection pattern, indicating lots of oil. Since S is caused by O, we add an $O \to S$ arc to our model. Let us model the reliability of our sensor using the following conditional distribution for $p(S|O)$:

	$s=0$	$s=1$	$s=2$
$o=0$	0.6	0.3	0.1
$o=1$	0.3	0.4	0.3
$o=2$	0.1	0.4	0.5

Suppose the sounding observation is s. The posterior expected utility of performing action d is

$$\text{EU}(d|s) = \sum_{o=0}^{2} p(o|s)U(o,d) \tag{34.15}$$

We need to compute this for each possible observation, $s \in \{0,1,2\}$, and each possible action, $d \in \{0,1\}$. If $s=0$, we find the posterior over the oil state is $p(o|s=0) = [0.732, 0.219, 0.049]$, and hence $\text{EU}(d=0|s=0) = 0$ and $\text{EU}(d=1|s=0) = -30.5$. If $s=1$, we similarly find $\text{EU}(d=0|s=1) = 0$ and $\text{EU}(d=1|s=1) = 32.9$. If $s=2$, we find $\text{EU}(d=0|s=2) = 0$ and $\text{EU}(d=1|s=2) = 87.5$. Hence the optimal policy $d^*(s)$ is as follows: if $s=0$, choose $d=0$ and get \$0; if $s=1$, choose $d=1$ and get \$32.9; and if $s=2$, choose $d=1$ and get \$87.5.

The maximum expected utility of the wildcatter, before seeing the experimental sounding, can be computed using

$$\text{MEU} = \sum_s p(s)\text{EU}(d^*(s)|s) \tag{34.16}$$

where prior marginal on the outcome of the test is $p(s) = \sum_o p(o)p(s|o) = [0.41, 0.35, 0.24]$. Hence the MEU is

$$\text{MEU} = 0.41 \times 0 + 0.35 \times 32.9 + 0.24 \times 87.5 = 32.2 \tag{34.17}$$

These numbers can be summarized in the **decision tree** shown in Figure 34.3.

34.2.3 Value of information

Now suppose you can choose whether to do the test or not. This can be modelled as shown in Figure 34.2(c), where we add a new test node T. If $T=1$, we do the test, and S can enter states $\{0,1,2\}$, determined by O, exactly as above. If $T=0$, we don't do the test, and S enters a special unknown state. There is also some cost associated with performing the test.

Is it worth doing the test? This depends on how much our MEU changes if we know the outcome of the test (namely the state of S). If you don't do the test, we have MEU = 20 from Equation (34.14). If you do the test, you have MEU = 32.2 from Equation (34.17). So the improvement in utility if you do the test (and act optimally on its outcome) is \$12.2. This is called the **value of perfect information** (VPI). So we should do the test as long as it costs less than \$12.2.

In terms of graphical models, the VPI of a variable S can be determined by computing the MEU for the base influence diagram, $\mathcal{G}$, in Figure 34.2(b), and then computing the MEU for the same influence diagram where we add information arcs from S to the action node, and then computing the difference. In other words,

$$\text{VPI} = \text{MEU}(\mathcal{G} + S \rightarrow D) - \text{MEU}(\mathcal{G}) \tag{34.18}$$

where D is the decision node and S is the variable we are measuring. This will tell us whether it is worth adding obtaining measurement S.

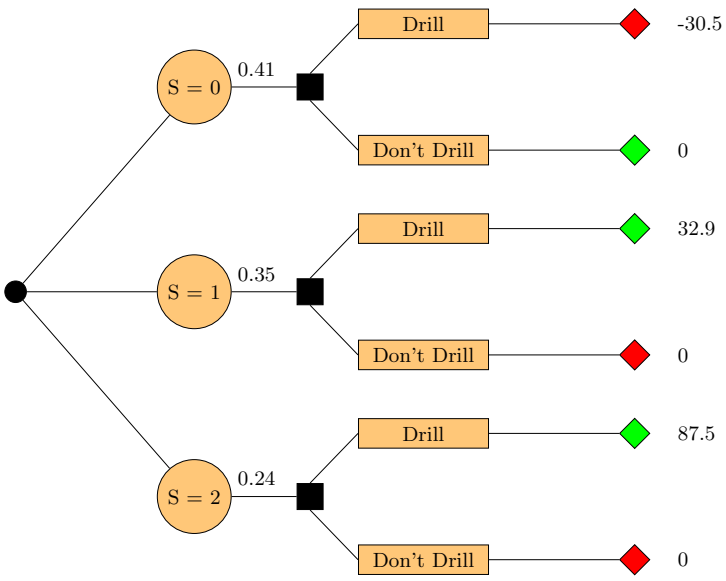


Figure 34.3: Decision tree for the oil wildcatter problem. Black circles are chance variables, black squares are decision nodes, diamonds are the resulting utilities. Green leaf nodes have higher utility than red leaf nodes.

34.2.4 Computing the optimal policy

In general, given an influence diagram, we can compute the optimal policy automatically by modifiying the variable elimination algorithm (Section 9.4), as explained in [LN01; KM08]. The basic idea is to work backwards from the final action, computing the optimal decision at each step, assuming all following actions are chosen optimally. When the influence diagram has a simple chain structure, as in a Markov decision process (Section 34.5), the result is equivalent to Bellman's equation (Section 34.5.5).

34.3 A/B testing

Suppose you are trying to decide which version of a product is likely to sell more, or which version of a drug is likely to work better. Let us call the versions you are choosing between A and B; sometimes version A is called the **control**, and version B is called the **treatment**. (Sometimes the different actions are called **"arms"**.)

A very common approach to such problems is to use an **A/B test**, in which you try both actions out for a while, by randomly assigning a different action to different subsets of the population, and then you measure the resulting accumulated **reward** from each action, and you pick the winner. (This is sometimes called a "**test and roll**" approach, since you test which method is best, and then roll it out for the rest of the population.)

A key problem in A/B testing is to come up with a decision rule, or policy, for deciding which action is best, after obtaining potentially noisy results during the test phase. Another problem is

to choose how many people to assign to the treatment, n_1, and how many to the control, n_0. The fundamental tradeoff is that using larger values of n_1 and n_0 will help you collect more data and hence be more confident in picking the best action, but this incurs an **opportunity cost**, because the testing phase involves performing actions that may not result in the highest reward. (This is an example of the exploration-exploitation tradeoff, which we discuss more in Section 34.4.3.) In this section, we give a simple Bayesian decision theoretic analysis of this problem, following the presentation of [FB19].[1] More details on A/B testing can be found in [KTX20].

34.3.1 A Bayesian approach

We assume the i'th reward for action j is given by $Y_{ij} \sim \mathcal{N}(\mu_j, \sigma_j^2)$ for $i = 1 : n_j$ and $j = 0 : 1$, where $j = 0$ corresponds to the control (action A), $j = 1$ corresponds to the treatment (action B), and n_j is the number of samples you collect from group j. The parameters μ_j are the expected reward for action j; our goal is to estimate these parameters. (For simplicity, we assume the σ_j^2 are known.)

We will adopt a Bayesian approach, which is well suited to sequential decision problems. For simplicity, we will use Gaussian priors for the unknowns, $\mu_j \sim \mathcal{N}(m_j, \tau_j^2)$, where m_j is the prior mean reward for action j, and τ_j is our confidence in this prior. We assume the prior parameters are known. (In practice we can use an empirical Bayes approach, as we discuss in Section 34.3.2.)

34.3.1.1 Optimal policy

Initially we assume the sample size of the experiment (i.e. the values n_1 for the treatment and n_0 for the control) are known. Our goal is to compute the optimal policy or decision rule $\pi(\boldsymbol{y}_1, \boldsymbol{y}_0)$, which specifies which action to deploy, where $\boldsymbol{y}_j = (y_{1j}, \ldots, y_{n_j,j})$ is the data from action j.

The optimal policy is simple: choose the action with the greater expected posterior expected reward:

$$\pi^*(\boldsymbol{y}_1, \boldsymbol{y}_0) = \begin{cases} 1 & \text{if } \mathbb{E}\left[\mu_1 | \boldsymbol{y}_1\right] \geq \mathbb{E}\left[\mu_0 | \boldsymbol{y}_0\right] \\ 0 & \text{if } \mathbb{E}\left[\mu_1 | \boldsymbol{y}_1\right] < \mathbb{E}\left[\mu_0 | \boldsymbol{y}_0\right] \end{cases} \tag{34.19}$$

All that remains is to compute the posterior. over the unknown parameters, μ_j. Applying Bayes' rule for Gaussians (Equation (2.82)), we find that the corresponding posterior is given by

$$p(\mu_j | \boldsymbol{y}_j, n_j) = \mathcal{N}(\mu_j | \, \widehat{m}_j, \widehat{\tau}_j^2) \tag{34.20}$$

$$1/\, \widehat{\tau}_j = n_j/\sigma_j^2 + 1/\tau_j^2 \tag{34.21}$$

$$\widehat{m}_j \,/\, \widehat{\tau}_j = n_j \overline{y}_j/\sigma_j^2 + m_j/\tau_j^2 \tag{34.22}$$

We see that the posterior precision (inverse variance) is a weighted sum of the prior precision plus n_j units of measurement precision. We also see that the posterior precision weighted mean is a sum of the prior precision weighted mean and the measurement precision weighted mean.

Given the posterior, we can plug $\widehat{m}_j$ into Equation (34.19). In the fully symmetric case, where $n_1 = n_0$, $m_1 = m_0 = m$, $\tau_1 = \tau_0 = \tau$, and $\sigma_1 = \sigma_0 = \sigma$, we find that the optimal policy is to simply

1. For a similar set of results in the time-discounted setting, see https://chris-said.io/2020/01/10/optimizing-sample-sizes-in-ab-testing-part-I.

"pick the winner", which is the arm with higher empirical performance:

$$\pi^*(\boldsymbol{y}_1, \boldsymbol{y}_0) = \mathbb{I}\left(\frac{m}{\tau^2} + \frac{\overline{y}_1}{\sigma^2} > \frac{m}{\tau^2} + \frac{\overline{y}_0}{\sigma^2}\right) = \mathbb{I}\left(\overline{y}_1 > \overline{y}_0\right) \tag{34.23}$$

However, when the problem is asymmetric, we need to take into account the different sample sizes and/or different prior beliefs.

34.3.1.2 Optimal sample size

We now discuss how to compute the optimal sample size for each arm of the experiment, i.e, the values n_0 and n_1. We assume the total population size is N, and we cannot reuse people from the testing phase,

The prior expected reward in the testing phase is given by

$$\mathbb{E}\left[R_{\text{test}}\right] = n_0 m_0 + n_1 m_1 \tag{34.24}$$

The expected reward in the roll phase depends on the decision rule $\pi(\boldsymbol{y}_1, \boldsymbol{y}_0)$ that we use:

$$\mathbb{E}_\pi\left[R_{\text{roll}}\right] = \int_{\mu_1}\int_{\mu_0}\int_{\boldsymbol{y}_1}\int_{\boldsymbol{y}_0} (N - n_1 - n_0)\left(\pi(\boldsymbol{y}_1, \boldsymbol{y}_0)\mu_1 + (1 - \pi(\boldsymbol{y}_1, \boldsymbol{y}_0))\mu_0\right) \tag{34.25}$$

$$\times\, p(\boldsymbol{y}_0|\mu_0)p(\boldsymbol{y}_1|\mu_1)p(\mu_0)p(\mu_1)d\boldsymbol{y}_0 d\boldsymbol{y}_1 d\mu_0 d\mu_1 \tag{34.26}$$

For $\pi = \pi^*$ one can show that this equals

$$\mathbb{E}\left[R_{\text{roll}}\right] \triangleq \mathbb{E}_{\pi*}\left[R_{\text{roll}}\right] = (N - n_1 - n_0)\left(m_1 + e\Phi(\frac{e}{v}) + v\phi(\frac{e}{v})\right) \tag{34.27}$$

where ϕ is the Gaussian pdf, Φ is the Gaussian cdf, $e = m_0 - m_1$ and

$$v = \sqrt{\frac{\tau_1^4}{\tau_1^2 + \sigma_1^2/n_1} + \frac{\tau_0^4}{\tau_0^2 + \sigma_0^2/n_0}} \tag{34.28}$$

In the fully symmetric case, Equation (34.27) simplifies to

$$\mathbb{E}\left[R_{\text{roll}}\right] = \underbrace{(N - 2n)m}_{R_a} + \underbrace{(N - 2n)\frac{\sqrt{2}\tau^2}{\sqrt{\pi}\sqrt{2\tau^2 + \frac{2}{n}\sigma^2}}}_{R_b} \tag{34.29}$$

This has an intuitive interpretation. The first term, R_a, is the prior reward we expect to get before we learn anything about the arms. The second term, R_b, is the reward we expect to see by virtue of picking the optimal action to deploy.

Let us we write $R_b = (N - 2n)R_i$, where R_i is the incremental gain. We see that the incremental gain increases with n, because we are more likely to pick the correct action with a larger sample size; however, this gain can only be accrued for a smaller number of people, as shown by the $N - 2n$ prefactor. (This is a consequence of the explore-exploit tradeoff.)

The total expected reward is given by adding Equation (34.24) and Equation (34.29):

$$\mathbb{E}[R] = \mathbb{E}[R_{\text{test}}] + \mathbb{E}[R_{\text{roll}}] = Nm + (N-2n)\left(\frac{\sqrt{2}\tau^2}{\sqrt{\pi}\sqrt{2\tau^2 + \frac{2}{n}\sigma^2}}\right) \tag{34.30}$$

(The equation for the non-symmetric case is given in [FB19].)

We can maximize the expected reward in Equation (34.30) to find the optimal sample size for the testing phase, which (from symmetry) satisfies $n_1^* = n_2^* = n^*$, and from $\frac{d}{dn^*}\mathbb{E}[R] = 0$ satisfies

$$n^* = \sqrt{\frac{N}{4}u^2 + \left(\frac{3}{4}u^2\right)^2} - \frac{3}{4}u^2 \leq \sqrt{N}\frac{\sigma}{2\tau} \tag{34.31}$$

where $u^2 = \frac{\sigma^2}{\tau^2}$. Thus we see that the optimal sample size n^* increases as the observation noise σ increases, since we need to collect more data to be confident of the right decision. However, the optimal sample size decreases with τ, since a prior belief that the effect size $\delta = \mu_1 - \mu_0$ will be large implies we expect to need less data to reach a confident conclusion.

34.3.1.3 Regret

Given a policy, it is natural to wonder how good it is. We define the **regret** of a policy to be the difference between the expected reward given **perfect information** (PI) about the true best action and the expected reward due to our policy. Minimizing regret is equivalent to making the expected reward of our policy equal to the best possible reward (which may be high or low, depending on the problem).

An oracle with perfect information about which μ_j is bigger would pick the highest scoring action, and hence get an expected reward of $N\mathbb{E}[\max(\mu_1, \mu_2)]$. Since we assume $\mu_j \sim \mathcal{N}(m, \tau^2)$, we have

$$\mathbb{E}[R|PI] = N\left(m + \frac{\tau}{\sqrt{\pi}}\right) \tag{34.32}$$

Therefore the regret from the optimal policy is given by

$$\mathbb{E}[R|PI] - (\mathbb{E}[R_{\text{test}}|\pi^*] + \mathbb{E}[R_{\text{roll}}|\pi^*]) = N\frac{\tau}{\sqrt{\pi}}\left(1 - \frac{\tau}{\sqrt{\tau^2 + \frac{\sigma^2}{n^*}}}\right) + \frac{2n^*\tau^2}{\sqrt{\pi}\sqrt{\tau^2 + \frac{\sigma^2}{n^*}}} \tag{34.33}$$

One can show that the regret is $O(\sqrt{N})$, which is optimal for this problem when using a time horizon (population size) of N [AG13].

34.3.1.4 Expected error rate

Sometimes the goal is posed as **best arm identification**, which means identifying whether $\mu_1 > \mu_0$ or not. That is, if we define $\delta = \mu_1 - \mu_0$, we want to know if $\delta > 0$ or $\delta < 0$. This is naturally phrased as a **hypothesis test**. However, this is arguably the wrong objective, since it is usually not worth spending money on collecting a large sample size to be confideny that $\delta > 0$ (say) if the

magnitude of δ is small. Instead, it makes more sense to optimize total expected reward, using the method in Section 34.3.1.1.

Nevertheless, we may want to know the probability that we have picked the wrong arm if we use the policy from Section 34.3.1.1. In the symmetric case, this is given by the following:

$$\Pr(\pi(\boldsymbol{y}_1, \boldsymbol{y}_0) = 1 | \mu_1 < \mu_0) = \Pr(Y_1 - Y_0 > 0 | \mu_1 < \mu_0) = 1 - \Phi\left(\frac{\mu_1 - \mu_0}{\sigma\sqrt{\frac{1}{n_1} + \frac{1}{n_0}}}\right) \tag{34.34}$$

The above expression assumed that μ_j are known. Since they are not known, we can compute the expected error rate using $\mathbb{E}\left[\Pr(\pi(\boldsymbol{y}_1, \boldsymbol{y}_0) = 1 | \mu_1 < \mu_0)\right]$. By symmetry, the quantity $\mathbb{E}\left[\Pr(\pi(\boldsymbol{y}_1, \boldsymbol{y}_0) = 0 | \mu_1 > \mu_0)\right]$ is the same. One can show that both quantities are given by

$$\text{Prob. error} = \frac{1}{4} - \frac{1}{2\pi} \arctan\left(\frac{\sqrt{2}\tau}{\sigma}\sqrt{\frac{n_1 n_0}{n_1 + n_0}}\right) \tag{34.35}$$

As expected, the error rate decreases with the sample size n_1 and n_0, increases with observation noise σ, and decreases with variance of the effect size τ. Thus a policy that minimizes the classification error will also maximize expected reward, but it may pick an overly large sample size, since it does not take into account the magnitude of δ.

34.3.2 Example

In this section, we give a simple example of the above framework. Suppose our goal is to do **website testing**, where have two different versions of a webpage that we want to compare in terms of their **click through rate**. The observed data is now binary, $y_{ij} \sim \text{Ber}(\mu_j)$, so it is natural to use a Beta prior, $\mu_j \sim \text{Beta}(\alpha, \beta)$ (see Section 3.2.1). However, in this case the optimal sample size and decision rule is harder to compute (see [FB19; Sta+17] for details). As a simple approximation, we can assume $\overline{y}_{ij} \sim \mathcal{N}(\mu_j, \sigma^2)$, where $\mu_j \sim \mathcal{N}(m, \tau^2)$, $m = \frac{\alpha}{\alpha+\beta}$, $\tau^2 = \frac{\alpha\beta}{(\alpha+\beta)^2(\alpha+\beta+1)}$, and $\sigma^2 = m(1-m)$.

To set the Gaussian prior, [FB19] used empirical data from about 2000 prior A/B tests. For each test, they observed the number of times the page was served with each of the two variations, as well as the total number of times a user clicked on each version. Given this data, they used a hierarchical Bayesian model to infer $\mu_j \sim \mathcal{N}(m = 0.68, \tau = 0.03)$. This prior implies that the expected effect size is quite small, $\mathbb{E}\left[|\mu_1 - \mu_0|\right] = 0.023$. (This is consistent with the results in [Aze+20], who found that most changes made to the Microsoft Bing EXP platform had negligible effect, although there were occasionally some "big hits".)

With this prior, and assuming a population of $N = 100,000$, Equation (34.31) says that the optimal number of trials to run is $n_1^* = n_0^* = 2284$. The expected reward (number of clicks or **conversions**) in the testing phase is $\mathbb{E}\left[R_{\text{test}}\right] = 3106$, and in the deployment phase $\mathbb{E}\left[R_{\text{roll}}\right] = 66430$, for a total reward of 69536. The expected error rate is 10%.

In Figure 34.4a, we plot the expected reward vs the size of the test phase n. We see that the reward increases sharply with n to the global maximum at $n^* = 2284$, and then drops off more slowly. This indicates that it is better to have a slightly larger test than one that is too small by the same amount. (However, when using a heavy tailed model, [Aze+20] finds that it is better to do lots of smaller tests.)

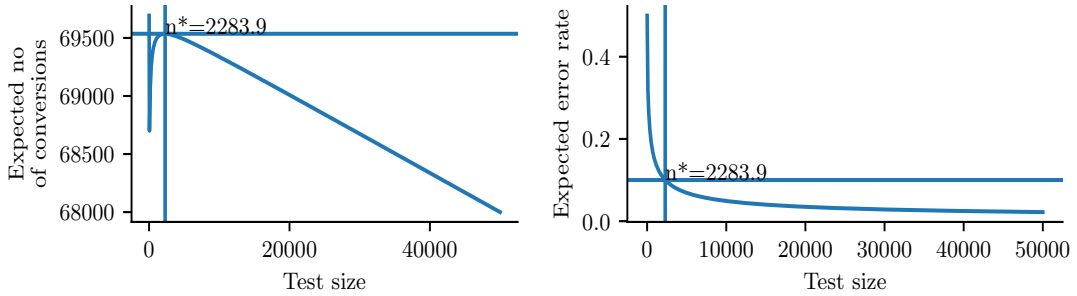


Figure 34.4: Total expected profit (a) and error rate (b) as a function of the sample size used fo website testing. Generated by ab_test_demo.ipynb.

In Figure 34.4b, we plot the probability of picking the wrong action vs n. We see that tests that are larger than optimal only reduce this error rate marginally. Consequently, if you want to make the misclassification rate low, you may need a large sample size, particularly if $\mu_1 - \mu_0$ is small, since then it will be hard to detect the true best action. However, it is also less important to identify the best action in this case, since both actions have very similar expected reward. This explains why classical methods for A/B testing based on frequentist statistics, which use hypothesis testing methods to determine if A is better than B, may often recommend sample sizes that are much larger than necessary. (See [FB19] and references therein for further discussion.)

34.4 Contextual bandits

This section was co-authored with Lihong Li.

In Section 34.3, we discussed A/B testing, in which the decision maker tries two different actions, a_0 and a_1, a fixed number of times, n_1 and n_0, measures the resulting sequence of rewards, $\boldsymbol{y}_1$ and $\boldsymbol{y}_0$, and then picks the best action to use for the rest of time (or the rest of the population) so as to maximize expected reward.

We can obviously generalize this beyond two actions. More importantly, we can generalize this beyond a one-stage decision problem. In particular, suppose we allow the decision maker to try an action a_t, observe the reward r_t, and then decide what to do at time step $t+1$, rather than waiting until $n_1 + n_0$ experiments are finished. This immediate feedback allows for **adaptive policies** that can result in much higher expected reward (lower regret). We have converted a one-stage decision problem into a **sequential decision problem**. There are many kinds of sequential decision problems, but in this section, we consider the simplest kind, known as a **bandit problem** (see e.g., [LS19; Sli19]).

34.4.1 Types of bandit

In a **multi-armed bandit** problem (MAB) there is an agent (decision maker) that can choose an **action** from some **policy** $a_t \sim \pi_t$ at each step, after which it receives a **reward** sampled from the

environment, $r_t \sim p_R(a_t)$, with expected value $R(s,a) = \mathbb{E}[R|a]$.[2]

We can think of this in terms of an agent at a casino who is faced with multiple slot machines, each of which pays out rewards at a different rate. A slot machine is sometimes called a **one-armed bandit**, so a set of K such machines is called a **multi-armed bandit**; each different action corresponds to pulling the arm of a different slot machine, $a_t \in \{1, \ldots, K\}$. The goal is to quickly figure out which machine pays out the most money, and then to keep playing that one until you become as rich as possible.

We can extend this model by defining a **contextual bandit**, in which the input to the policy at each step is a randomly chosen state or context $s_t \in \mathcal{S}$. The states evolve over time according to some arbitrary process, $s_t \sim p(s_t|\boldsymbol{s}_{1:t-1})$, independent of the actions of the agent. The policy now has the form $a_t \sim \pi_t(a_t|s_t)$, and the reward function now has the form $r_t \sim p_R(r_t|s_t, a_t)$, with expected value $R(s,a) = \mathbb{E}[R|s,a]$. At each step, the agent can use the observed data, $\mathcal{D}_{1:t}$ where $\mathcal{D}_t = (s_t, a_t, r_t)$, to update its policy, to maximize expected reward.

In the **finite horizon** formulation of (contextual) bandits, the goal is to maximize the expected **cumulative reward**:

$$J \triangleq \sum_{t=1}^{T} \mathbb{E}_{p_R(r_t|s_t,a_t)\pi_t(a_t|s_t)p(s_t|\boldsymbol{s}_{1:t-1})}[r_t] = \sum_{t=1}^{T} \mathbb{E}[r_t] \tag{34.36}$$

(Note that the reward is accrued at each step, even while the agent updates its policy; this is sometimes called "**earning while learning**".) In the **infinite horizon** formulation, where $T = \infty$, the cumulative reward may be infinite. To prevent J from being unbounded, we introduce a **discount factor** $0 < \gamma < 1$, so that

$$J \triangleq \sum_{t=1}^{\infty} \gamma^{t-1} \mathbb{E}[r_t] \tag{34.37}$$

The quantity γ can be interpreted as the probability that the agent is terminated at any moment in time (in which case it will cease to accumulate reward).

Another way to write this is as follows:

$$J = \sum_{t=1}^{\infty} \gamma^{t-1} \mathbb{E}[r_t] = \sum_{t=1}^{\infty} \gamma^{t-1} \mathbb{E}\left[\sum_{a=1}^{K} R_a(s_t, a_t)\right] \tag{34.38}$$

where we define

$$R_a(s_t, a_t) = \begin{cases} R(s_t, a) & \text{if } a_t = a \\ 0 & \text{otherwise} \end{cases} \tag{34.39}$$

Thus we conceptually evaluate the reward for all arms, but only the one that was actually chosen (namely a_t) gives a non-zero value to the agent, namely r_t.

2. This is known as a **stochastic bandit**. It is also possible to allow the reward, and possibly the state, to be chosen in an adversarial manner, where nature tries to minimize the reward of the agent. This is known as an **adversarial bandit**.

There are many extensions of the basic bandit problem. A natural one is to allow the agent to perform **multiple plays**, choosing $M \leq K$ distinct arms at once. Let $\boldsymbol{a}_t$ be the corresponding action vector which specifies the identity of the chosen arms. Then we define the reward to be

$$r_t = \sum_{a=1}^{K} R_a(s_t, \boldsymbol{a}_t) \tag{34.40}$$

where

$$R_a(s_t, \boldsymbol{a}_t) = \begin{cases} R(s_t, a) & \text{if } a \in \boldsymbol{a}_t \\ 0 & \text{otherwise} \end{cases} \tag{34.41}$$

This is useful for modeling **resource allocation** problems.

Another variant is known as a **restless bandit** [Whi88]. This is the same as the multiple play formulation, except we additionally assume that each arm has its own state vector s_t^a associated with it, which evolves according to some stochastic process, regardless of whether arm a was chosen or not. We then define

$$r_t = \sum_{a=1}^{K} R_a(s_t^a, \boldsymbol{a}_t) \tag{34.42}$$

where $s_t^a \sim p(s_t^a | \boldsymbol{s}_{1:t-1}^a)$ is some arbitrary distribution, often assumed to be Markovian. (The fact that the states associated with each arm evolve even if the arm is not picked is what gives rise to the term "restless".) This can be used to model serial dependence between the rewards given by each arm.

34.4.2 Applications

Contextual bandits have many applications. For example, consider an **online advertising system**. In this case, the state s_t represents features of the web page that the user is currently looking at, and the action a_t represents the identity of the ad which the system chooses to show. Since the relevance of the ad depends on the page, the reward function has the form $R(s_t, a_t)$, and hence the problem is contextual. The goal is to maximize the expected reward, which is equivalent to the expected number of times people click on ads; this is known as the **click through rate** or **CTR**. (See e.g., [Gra+10; Li+10; McM+13; Aga+14; Du+21; YZ22] for more information about this application.)

Another application of contextual bandits arises in **clinical trials** [VBW15]. In this case, the state s_t are features of the current patient we are treating, and the action a_t is the treatment the doctor chooses to give them (e.g., a new drug or a **placebo**). Our goal is to maximize expected reward, i.e., the expected number of people who get cured. (An alternative goal is to determine which treatment is best as quickly as possible, rather than maximizing expected reward; this variant is known as **best-arm identification** [ABM10].)

34.4.3 Exploration-exploitation tradeoff

The fundamental difficulty in solving bandit problems is known as the **exploration-exploitation tradeoff**. This refers to the fact that the agent needs to try multiple state/action combinations (this

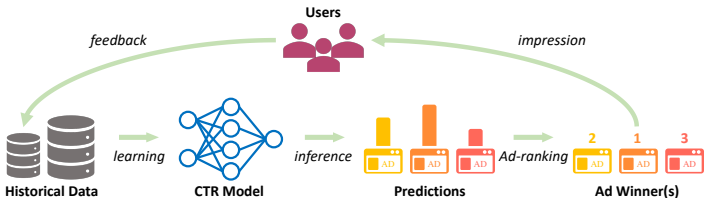


Figure 34.5: Illustration of the feedback problem in online advertising and recommendation systems. The click through rate (CTR) model is used to decide what ads to show, which affects what data is collected, which affects how the model learns. From Figure 1–2 of [Du+21]. Used with kind permission of Chao Du.

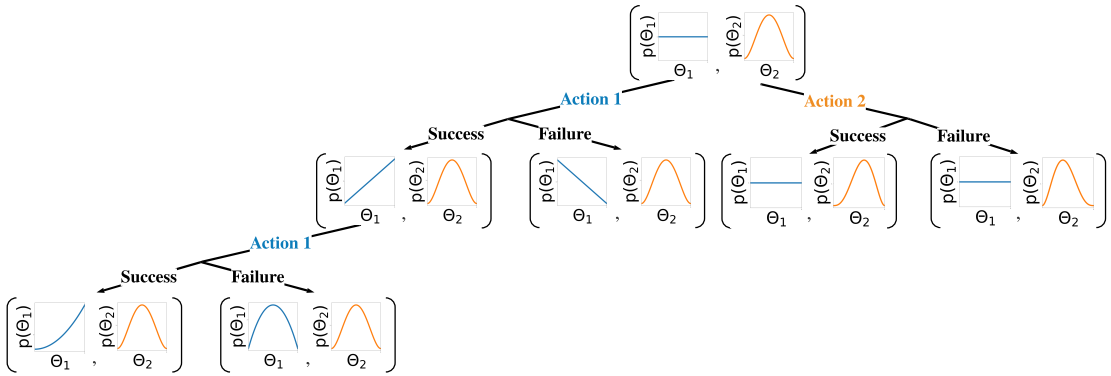


Figure 34.6: Illustration of sequential belief updating for a two-armed beta-Bernoulli bandit. The prior for the reward for action 1 is the (blue) uniform distribution $\text{Beta}(1,1)$*; the prior for the reward for action 2 is the (orange) unimodal distribution* $\text{Beta}(2,2)$*. We update the parameters of the belief state based on the chosen action, and based on whether the observed reward is success (1) or failure (0).*

is known as exploration) in order to collect enough data so it can reliably learn the reward function $R(s,a)$; it can then exploit its knowledge by picking the predicted best action for each state. If the agent starts exploiting an incorrect model too early, it will collect suboptimal data, and will get stuck in a negative **feedback loop**, as illustrated in Figure 34.5. This is different from supervised learning, where the data is drawn iid from a fixed distribution (see e.g.m [Jeu+19] for details).

We discuss some solutions to the exploration-exploitation problem below.

34.4.4 The optimal solution

In this section, we discuss the optimal solution to the exploration-exploitation tradeoff. Let us denote the posterior over the parameters of the reward function by $\boldsymbol{b}_t = p(\boldsymbol{\theta}|\boldsymbol{h}_t)$, where $\boldsymbol{h}_t = \{s_{1:t-1}, a_{1:t-1}, r_{1:t-1}\}$ is the history of observations; this is known as the **belief state** or **information state**. It is a finite sufficient statistic for the history $\boldsymbol{h}_t$. The belief state can be updated deterministically using Bayes rule:

$$\boldsymbol{b}_t = \text{BayesRule}(\boldsymbol{b}_{t-1}, a_t, r_t) \tag{34.43}$$

For example, consider a context-free **Bernoulli bandit**, where $p_R(r|a) = \text{Ber}(r|\mu_a)$, and $\mu_a = p_R(r=1|a) = R(a)$ is the expected reward for taking action a. Suppose we use a factored beta prior

$$p_0(\boldsymbol{\theta}) = \prod_a \text{Beta}(\mu_a|\alpha_0^a, \beta_0^a) \tag{34.44}$$

where $\boldsymbol{\theta} = (\mu_1, \ldots, \mu_K)$. We can compute the posterior in closed form, as we discuss in Section 3.2.1. In particular, we find

$$p(\boldsymbol{\theta}|\mathcal{D}_t) = \prod_a \text{Beta}(\mu_a|\underbrace{\alpha_0^a + N_t^0(a)}_{\alpha_t^a}, \underbrace{\beta_0^a + N_t^1(a)}_{\beta_t^a}) \tag{34.45}$$

where

$$N_t^r(a) = \sum_{s=1}^{t-1} \mathbb{I}\left(a_t = a, r_t = r\right) \tag{34.46}$$

This is illustrated in Figure 34.6 for a two-armed Bernoulli bandit.

We can use a similar method for a **Gaussian bandit**, where $p_R(r|a) = \mathcal{N}(r|\mu_a, \sigma_a^2)$, using results from Section 3.2.3. In the case of contextual bandits, the problem becomes more complicated. If we assume a **linear regression bandit**, $p_R(r|s, a; \boldsymbol{\theta}) = \mathcal{N}(r|\boldsymbol{\phi}(s,a)^\mathsf{T}\boldsymbol{\theta}, \sigma^2)$, we can use Bayesian linear regression to compute $p(\boldsymbol{\theta}|\mathcal{D}_t)$ in closed form, as we discuss in Section 15.2. If we assume a **logistic regression bandit**, $p_R(r|s, a; \boldsymbol{\theta}) = \text{Ber}(r|\sigma(\boldsymbol{\phi}(s,a)^\mathsf{T}\boldsymbol{\theta}))$, we can use Bayesian logistic regression to compute $p(\boldsymbol{\theta}|\mathcal{D}_t)$, as we discuss in Section 15.3.4. If we have a **neural bandit** of the form $p_R(r|s, a; \boldsymbol{\theta}) = \text{GLM}(r|f(s, a; \boldsymbol{\theta}))$ for some nonlinear function f, then posterior inference becomes more challenging, as we discuss in Chapter 17. However, standard techniques, such as the extended Kalman filter (Section 17.5.2) can be applied. (For a way to scale this approach to large DNNs, see the "**subspace neural bandit**" approach of [DMKM22].)

Regardless of the algorithmic details, we can represent the belief state update as follows:

$$p(\boldsymbol{b}_t|\boldsymbol{b}_{t-1}, a_t, r_t) = \mathbb{I}\left(\boldsymbol{b}_t = \text{BayesRule}(\boldsymbol{b}_{t-1}, a_t, r_t)\right) \tag{34.47}$$

The observed reward at each step is then predicted to be

$$p(r_t|\boldsymbol{b}_t) = \int p_R(r_t|s_t, a_t; \boldsymbol{\theta}) p(\boldsymbol{\theta}|\boldsymbol{b}_t) d\boldsymbol{\theta} \tag{34.48}$$

We see that this is a special form of a (controlled) Markov decision process (Section 34.5) known as a **belief-state MDP**.

In the special case of context-free bandits with a finite number of arms, the optimal policy of this belief state MDP can be computed using dynamic programming (c.f., Section 34.6); the result can be represented as a table of action probabilities, $\pi_t(a_1, \ldots, a_K)$, for each step; this is known as the **Gittins index** [Git89]. However, computing the optimal policy for general contextual bandits is intractable [PT87], so we have to resort to approximations, as we discuss below.

34.4.5 Upper confidence bounds (UCB)

The optimal solution to explore-exploit is intractable. However, an intuitively sensible approach is based on the principle known as "**optimism in the face of uncertainty**". The principle selects

actions greedily, but based on optimistic estimates of their rewards. The most important class of strategies with this principle are collectively called **upper confidence bound** or **UCB** methods.

To use a UCB strategy, the agent maintains an optimistic reward function estimate $\tilde{R}_t$, so that $\tilde{R}_t(s_t, a) \geq R(s_t, a)$ for all a with high probability, and then chooses the greedy action accordingly:

$$a_t = \operatorname*{argmax}_a \tilde{R}_t(s_t, a) \tag{34.49}$$

UCB can be viewed a form of **exploration bonus**, where the optimistic estimate encourages exploration. Typically, the amount of optimism, $\tilde{R}_t - R$, decreases over time so that the agent gradually reduces exploration. With properly constructed optimistic reward estimates, the UCB strategy has been shown to achieve near-optimal regret in many variants of bandits [LS19]. (We discuss regret in Section 34.4.7.)

The optimistic function $\tilde{R}$ can be obtained in different ways, sometimes in closed forms, as we discuss below.

34.4.5.1 Frequentist approach

One approach is to use a **concentration inequality** [BLM16] to derive a high-probability upper bound of the estimation error: $|\hat{R}_t(s, a) - R_t(s, a)| \leq \delta_t(s, a)$, where $\hat{R}_t$ is a usual estimate of R (often the MLE), and δ_t is a properly selected function. An optimistic reward is then obtained by setting $\tilde{R}_t(s, a) = \hat{R}_t(s, a) + \delta_t(s, a)$.

As an example, consider again the context-free Bernoulli bandit, $R(a) \sim \text{Ber}(\mu(a))$. The MLE $\hat{R}_t(a) = \hat{\mu}_t(a)$ is given by the empirical average of observed rewards whenever action a was taken:

$$\hat{\mu}_t(a) = \frac{N_t^1(a)}{N_t(a)} = \frac{N_t^1(a)}{N_t^0(a) + N_t^1(a)} \tag{34.50}$$

where $N_t^r(a)$ is the number of times (up to step $t-1$) that action a has been tried and the observed reward was r, and $N_t(a)$ is the total number of times action a has been tried:

$$N_t(a) = \sum_{s=1}^{t-1} \mathbb{I}(a_t = a) \tag{34.51}$$

Then the **Chernoff-Hoeffding inequality** [BLM16] leads to $\delta_t(a) = c/\sqrt{N_t(a)}$ for some proper constant c, so

$$\tilde{R}_t(a) = \hat{\mu}_t(a) + \frac{c}{\sqrt{N_t(a)}} \tag{34.52}$$

34.4.5.2 Bayesian approach

We may also derive $\tilde{R}$ from Bayesian inference. If we use a beta prior, we can compute the posterior in closed form, as shown in Equation (34.45). The posterior mean is $\hat{\mu}_t(a) = \mathbb{E}[\mu(a)|\boldsymbol{h}_t] = \frac{\alpha_t^a}{\alpha_t^a + \beta_t^a}$. From Equation (3.20), the posterior standard deviation is approximately

$$\hat{\sigma}_t(a) = \sqrt{\mathbb{V}[\mu(a)|\boldsymbol{h}_t]} \approx \sqrt{\frac{\hat{\mu}_t(a)(1 - \hat{\mu}_t(a))}{N_t(a)}} \tag{34.53}$$

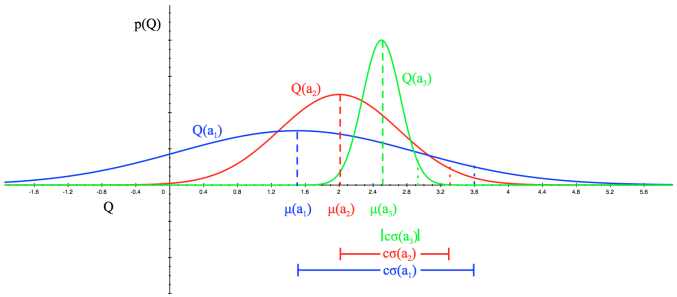


Figure 34.7: Illustration of the reward distribution $Q(a)$ for 3 different actions, and the corresponding lower and upper confidence bounds. From [Sil18]. Used with kind permission of David Silver.

We can use similar techniques for a Gaussian bandit, where $p_R(R|a, \boldsymbol{\theta}) = \mathcal{N}(R|\mu_a, \sigma_a^2)$, μ_a is the expected reward, and σ_a^2 the variance. If we use a conjugate prior, we can compute $p(\mu_a, \sigma_a|\mathcal{D}_t)$ in closed form (see Section 3.2.3). Using an uninformative version of the conjugate prior, we find $\mathbb{E}[\mu_a|\boldsymbol{h}_t] = \hat{\mu}_t(a)$, which is just the empirical mean of rewards for action a. The uncertainty in this estimate is the standard error of the mean, given by Equation (3.66), i.e., $\sqrt{\mathbb{V}[\mu_a|\boldsymbol{h}_t]} = \hat{\sigma}_t(a)/\sqrt{N_t(a)}$, where $\hat{\sigma}_t(a)$ is the empirical standard deviation of the rewards for action a.

This approach can also be extended to contextual bandits, modulo the difficulty of computing the belief state.

Once we have computed the mean and posterior standard deviation, we define the optimistic reward estimate as

$$\tilde{R}_t(a) = \hat{\mu}_t(a) + c\hat{\sigma}_t(a) \tag{34.54}$$

for some constant c that controls how greedy the policy is. We see that this is similar to the frequentist method based on concentration inequalities, but is more general.

34.4.5.3 Example

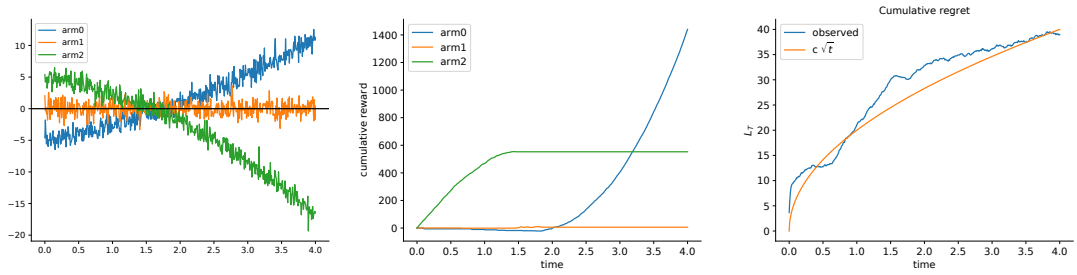
Figure 34.7 illustrates the UCB principle for a Gaussian bandit. We assume there are 3 actions, and we represent $p(R(a)|\mathcal{D}_t)$ using a Gaussian. We show the posterior means $Q(a) = \mu(a)$ with a vertical dotted line, and the scaled posterior standard deviations $c\sigma(a)$ as a horizontal solid line.

34.4.6 Thompson sampling

A common alternative to UCB is to use **Thompson sampling** [Tho33], also called **probability matching** [Sco10]. In this approach, we define the policy at step t to be $\pi_t(a|s_t, \boldsymbol{h}_t) = p_a$, where p_a is the probability that a is the optimal action. This can be computed using

$$p_a = \Pr(a = a_*|s_t, \boldsymbol{h}_t) = \int \mathbb{I}\left(a = \operatorname*{argmax}_{a'} R(s_t, a'; \boldsymbol{\theta})\right) p(\boldsymbol{\theta}|\boldsymbol{h}_t) d\boldsymbol{\theta} \tag{34.55}$$

If the posterior is uncertain, the agent will sample many different actions, automatically resulting in exploration. As the uncertainty decreases, it will start to exploit its knowledge.



(a) *(b)* *(c)*

Figure 34.8: Illustration of Thompson sampling applied to a linear-Gaussian contextual bandit. The context has the form $\boldsymbol{s}_t = (1, t, t^2)$. (a) True reward for each arm vs time. (b) Cumulative reward per arm vs time. (c) Cumulative regret vs time. Generated by thompson_sampling_linear_gaussian.ipynb.

To see how we can implement this method, note that we can compute the expression in Equation (34.55) by using a single Monte Carlo sample $\tilde{\boldsymbol{\theta}}_t \sim p(\boldsymbol{\theta}|\boldsymbol{h}_t)$. We then plug in this parameter into our reward model, and greedily pick the best action:

$$a_t = \operatorname*{argmax}_{a'} R(s_t, a'; \tilde{\boldsymbol{\theta}}_t) \tag{34.56}$$

This sample-then-exploit approach will choose actions with exactly the desired probability, since

$$p_a = \int \mathbb{I}\left(a = \operatorname*{argmax}_{a'} R(s_t, a'; \tilde{\boldsymbol{\theta}}_t)\right) p(\tilde{\boldsymbol{\theta}}_t|\boldsymbol{h}_t) = \Pr_{\tilde{\boldsymbol{\theta}}_t \sim p(\boldsymbol{\theta}|\boldsymbol{h}_t)}\left(a = \operatorname*{argmax}_{a'} R(s_t, a'; \tilde{\boldsymbol{\theta}}_t)\right) \tag{34.57}$$

Despite its simplicity, this approach can be shown to achieve optimal (logarithmic) regret (see e.g., [Rus+18] for a survey). In addition, it is very easy to implement, and hence is widely used in practice [Gra+10; Sco10; CL11].

In Figure 34.8, we give a simple example of Thompson sampling applied to a linear regression bandit. The context has the form $\boldsymbol{s}_t = (1, t, t^2)$. The true reward function has the form $R(\boldsymbol{s}_t, a) = \boldsymbol{w}_a^\mathsf{T} \boldsymbol{s}_t$. The weights per arm are chosen as follows: $\boldsymbol{w}_0 = (-5, 2, 0.5)$, $\boldsymbol{w}_1 = (0, 0, 0)$, $\boldsymbol{w}_2 = (5, -1.5, -1)$. Thus we see that arm 0 is initially worse (large negative bias) but gets better over time (positive slope), arm 1 is useless, and arm 2 is initially better (large positive bias) but gets worse over time. The observation noise is the same for all arms, $\sigma^2 = 1$. See Figure 34.8(a) for a plot of the reward function.

We use a conjugate Gaussian-Gamma prior and perform exact Bayesian updating. Thompson sampling quickly discovers that arm 1 is useless. Initially it pulls arm 2 more, but it adapts to the non-stationary nature of the problem and switches over to arm 0, as shown in Figure 34.8(b).

34.4.7 Regret

We have discussed several methods for solving the exploration-exploitation tradeoff. It is useful to quantify the degree of suboptimality of these methods. A common approach is to compute the **regret**, which is defined as the difference between the expected reward under the agent's policy and

the oracle policy π_*, which knows the true reward function. (Note that the oracle policy will in general be better than the Bayes optimal policy, which we disucssed in Section 34.4.4.)

Specifically, let π_t be the agent's policy at time t. Then the **per-step regret** at t is defined as

$$l_t \triangleq \mathbb{E}_{p(s_t)}\left[R(s_t, \pi_*(s_t))\right] - \mathbb{E}_{\pi_t(a_t|s_t)p(s_t)}\left[R(s_t, a_t)\right] \tag{34.58}$$

If we only care about the final performance of the best discovered arm, as in most optimization problems, it is enough to look at the **simple regret** at the last step, namely l_T. Optimizing simple regret results in a problem known as **pure exploration** [BMS11], since there is no need to exploit the information during the learning process. However, it is more common to focus on the the **cumulative regret**, also called the **total regret** or just the **regret**, which is defined as

$$L_T \triangleq \mathbb{E}\left[\sum_{t=1}^{T} l_t\right] \tag{34.59}$$

Here the expectation is with respect to randomness in determining π_t, which depends on earlier states, actions and rewards, as well as other potential sources of randomness.

Under the typical assumption that rewards are bounded, L_T is at most linear in T. If the agent's policy converges to the optimal policy as T increases, then the regret is sublinear: $L_T = o(T)$. In general, the slower L_T grows, the more efficient the agent is in trading off exploration and exploitation.

To understand its growth rate, it is helpful to consider again a simple context-free bandit, where $R_* = \text{argmax}_a R(a)$ is the optimal reward. The total regret in the first T steps can be written as

$$L_T = \mathbb{E}\left[\sum_{t=1}^{T} R_* - R(a_t)\right] = \sum_{a \in \mathcal{A}} \mathbb{E}\left[N_{T+1}(a)\right](R_* - R(a)) = \sum_{a \in \mathcal{A}} \mathbb{E}\left[N_{T+1}(a)\right]\Delta_a \tag{34.60}$$

where $N_{T+1}(a)$ is the total number of times the agent picks action a up to step T, and $\Delta_a = R_* - R(a)$ is the reward **gap**. If the agent under-explores and converges to choosing a suboptimal action (say, $\hat{a}$), then a linear regret is suffered with a per-step regret of $\Delta_{\hat{a}}$. On the other hand, if the agent over-explores, then $N_t(a)$ will be too large for suboptimal actions, and the agent also suffers a linear regret.

Fortunately, it is possible to achieve sublinear regrets, using some of the methods discussed above, such as UCB and Thompson sampling. For example, one can show that Thompson sampling has $O(\sqrt{KT \log T})$ regret [RR14]. This is shown empirically in Figure 34.8(c).

In fact, both UCB and Thompson sampling are optimal, in the sense that their regrets are essentially not improvable; that is, they match regret lower bounds. To establish such a lower bound, note that the agent needs to collect enough data to distinguish different reward distributions, before identifying the optimal action. Typically, the deviation of the reward estimate from the true reward decays at the rate of $1/\sqrt{N}$, where N is the sample size (see e.g., Equation (3.66)). Therefore, if two reward distributions are similar, distinguishing them becomes harder and requires more samples. (For example, consider the case of a bandit with Gaussian rewards with slightly different means and large variance, as shown in Figure 34.7.)

The following fundamental result is proved by [LR85] for the asymptotic regret (under certain mild assumptions not given here):

$$\liminf_{T \to \infty} L_T \geq \log T \sum_{a: \Delta_a > 0} \frac{\Delta_a}{D_{\mathbb{KL}}\left(p_R(a) \parallel p_R(a_*)\right)} \tag{34.61}$$

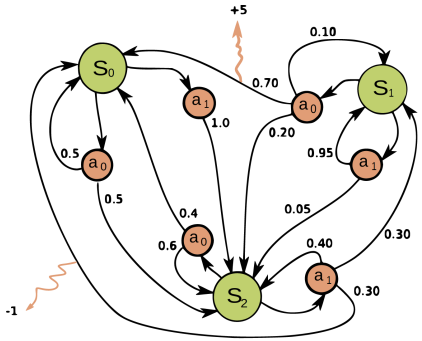


Figure 34.9: Illustration of an MDP as a finite state machine (FSM). The MDP has three discrete states (green cirlces), two discrete actions (orange circles), and two non-zero rewards (orange arrows). The numbers on the black edges represent state transition probabilities, e.g., $p(s' = s_0|a = a_0, s' = s_0) = 0.7$; most state transitions are impossible (probability 0), so the graph is sparse. The numbers on the yellow wiggly edges represent expected rewards, e.g., $R(s = s_1, a = a_0, s' = s_0) = +5$; state transitions with zero reward are not annotated. From https://en.wikipedia.org/wiki/Markov_decision_process. Used with kind permission of Wikipedia author waldoalvarez.

Thus, we see that the best we can achieve is logarithmic growth in the total regret. Similar lower bounds have also been obtained for various bandits variants.

34.5 Markov decision problems

In this section, we generalize the discussion of contextual bandits by allowing the state of nature to change depending on the actions chosen by the agent. The resulting model is called a **Markov decision process** or **MDP**, as we explain in detail below. This model forms the foundation of reinforcement learning, which we discuss in Chapter 35.

34.5.1 Basics

A **Markov decision process** [Put94] can be used to model the interaction of an **agent** and an **environment**. It is often described by a tuple $\langle \mathcal{S}, \mathcal{A}, p_T, p_R, p_0 \rangle$, where $\mathcal{S}$ is a set of environment states, $\mathcal{A}$ a set of actions the agent can take, p_T a **transition model**, p_R a **reward model**, and p_0 the initial state distribution. The interaction starts at time $t = 0$, where the initial state $s_0 \sim p_0$. Then, at time $t \geq 0$, the agent observes the environment state $s_t \in \mathcal{S}$, and follows a **policy** π to take an action $a_t \in \mathcal{A}$. In response, the environment emits a real-valued reward signal $r_t \in \mathcal{R}$ and enters a new state $s_{t+1} \in \mathcal{S}$. The policy is in general stochastic, with $\pi(a|s)$ being the probability of choosing action a in state s. We use $\pi(s)$ to denote the conditional probability over $\mathcal{A}$ if the policy is stochastic, or the action it chooses if it is deterministic. The process at every step is called a **transition**; at time t, it consists of the tuple (s_t, a_t, r_t, s_{t+1}), where $a_t \sim \pi(s_t)$, $s_{t+1} \sim p_T(s_t, a_t)$, and $r_t \sim p_R(s_t, a_t, s_{t+1})$. Hence, under policy π, the probability of generating a trajectory $\boldsymbol{\tau}$ of

length T can be written explicitly as

$$p(\boldsymbol{\tau}) = p_0(s_0) \prod_{t=0}^{T-1} \pi(a_t|s_t) p_T(s_{t+1}|s_t, a_t) p_R(r_t|s_t, a_t, s_{t+1}) \tag{34.62}$$

It is useful to define the **reward function** from the reward model p_R, as the average immediate reward of taking action a in state s, with the next state marginalized:

$$R(s,a) \triangleq \mathbb{E}_{p_T(s'|s,a)} \left[\mathbb{E}_{p(r|s,a,s')} [r] \right] \tag{34.63}$$

Eliminating the dependence on next states does not lead to loss of generality in the following discussions, as our subject of interest is the total (additive) expected reward along the trajectory. For this reason, we often use the tuple $\langle \mathcal{S}, \mathcal{A}, p_T, R, p_0 \rangle$ to describe an MDP.

In general, the state and action sets of an MDP can be discrete or continuous. When both sets are finite, we can represent these functions as lookup tables; this is known as a **tabular representation**. In this case, we can represent the MDP as a **finite state machine**, which is a graph where nodes correspond to states, and edges correspond to actions and the resulting rewards and next states. Figure 34.9 gives a simple example of an MDP with 3 states and 2 actions.

The field of **control theory**, which is very closely related to RL, uses slightly different terminology. In particular, the environment is called the **plant**, and the agent is called the **controller**. States are denoted by $\boldsymbol{x}_t \in \mathcal{X} \subseteq \mathbb{R}^D$, actions are denoted by $\boldsymbol{u}_t \in \mathcal{U} \subseteq \mathbb{R}^K$, and rewards are denoted by costs $c_t \in \mathbb{R}$. Apart from this notational difference, the fields of RL and control theory are very similar (see e.g., [Son98; Rec19]), although control theory tends to focus on provably optimal methods (by making strong modeling assumptions), whereas RL tends to tackle harder problems with heuristic methods, for which optimality guarantees are often hard to obtain.

34.5.2 Partially observed MDPs

An important generalization of the MDP framework relaxes the assumption that the agent sees the hidden world state s_t directly; instead we assume it only sees a potentially noisy observation generated from the hidden state, $x_t \sim p(\cdot|s_t, a_t)$. The resulting model is called a **partially observable Markov decision process** or **POMDP** (pronounced "pom-dee-pee"). Now the agent's policy is a mapping from all the available data to actions, $a_t \sim \pi(\mathcal{D}_{1:t-1}, x_t)$, $\mathcal{D}_t = (x_t, a_t, r_t)$. See Figure 34.10 for an illustration. MDPs are a special case where $x_t = s_t$.

In general, POMDPs are much harder to solve than MDPs. A common approximation is to use the last several observed inputs, say $\boldsymbol{x}_{t-h:t}$ for history of size h, as a proxy for the hidden state, and then to treat this as a fully observed MDP.

34.5.3 Episodes and returns

The Markov decision process describes how a trajectory $\boldsymbol{\tau} = (s_0, a_0, r_0, s_1, a_1, r_1, \ldots)$ is stochastically generated. If the agent can potentially interact with the environment forever, we call it a **continuing task**. Alternatively, the agent is in an **episodic task**, if its interaction terminates once the system enters a **terminal state** or **absorbing state**; s is absorbing if the next state from s is always s with 0 reward. After entering a terminal state, we may start a new **epsiode** from a new initial state $s_0 \sim p_0$. The episode length is in general random. For example, the amount of time a robot takes to

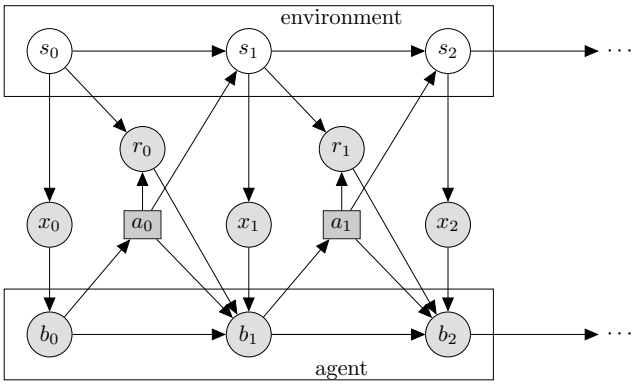


Figure 34.10: Illustration of a partially observable Markov decision process (POMDP) with hidden environment state s_t which generates the observation x_t, controlled by an agent with internal belief state b_t which generates the action a_t. The reward r_t depends on s_t and a_t. Nodes in this graph represent random variables (circles) and decision variables (squares).

reach its goal may be quite variable, depending on the decisions it makes, and the randomness in the environment. Note that we can convert an episodic MDP to a continuing MDP by redefining the transition model in absorbing states to be the initial-state distribution p_0. Finally, if the trajectory length T in an episodic task is fixed and known, it is called a **finite horizon problem**.

Let $\boldsymbol{\tau}$ be a trajectory of length T, where T may be ∞ if the task is continuing. We define the **return** for the state at time t to be the sum of expected rewards obtained going forward, where each reward is multiplied by a **discount factor** $\gamma \in [0, 1]$:

$$G_t \triangleq r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots + \gamma^{T-t-1} r_{T-1} \tag{34.64}$$

$$= \sum_{k=0}^{T-t-1} \gamma^k r_{t+k} = \sum_{j=t}^{T-1} \gamma^{j-t} r_j \tag{34.65}$$

G_t is sometimes called the **reward-to-go**. For episodic tasks that terminate at time T, we define $G_t = 0$ for $t \geq T$. Clearly, the return satisfies the following recursive relationship:

$$G_t = r_t + \gamma(r_{t+1} + \gamma r_{t+2} + \cdots) = r_t + \gamma G_{t+1} \tag{34.66}$$

The discount factor γ plays two roles. First, it ensures the return is finite even if $T = \infty$ (i.e., infinite horizon), provided we use $\gamma < 1$ and the rewards r_t are bounded. Second, it puts more weight on short-term rewards, which generally has the effect of encouraging the agent to achieve its goals more quickly (see Section 34.5.5.1 for an example). However, if γ is too small, the agent will become too greedy. In the extreme case where $\gamma = 0$, the agent is completely **myopic**, and only tries to maximize its immediate reward. In general, the discount factor reflects the assumption that there is a probability of $1 - \gamma$ that the interaction will end at the next step. For finite horizon problems, where T is known, we can set $\gamma = 1$, since we know the life time of the agent a priori.[3]

3. We may also use $\gamma = 1$ for continuing tasks, targeting the (undiscounted) average reward criterion [Put94].

34.5.4 Value functions

Let π be a given policy. We define the **state-value function**, or **value function** for short, as follows (with $\mathbb{E}_\pi[\cdot]$ indicating that actions are selected by π):

$$V_\pi(s) \triangleq \mathbb{E}_\pi[G_0|s_0 = s] = \mathbb{E}_\pi\left[\sum_{t=0}^{\infty} \gamma^t r_t | s_0 = s\right] \tag{34.67}$$

This is the expected return obtained if we start in state s and follow π to choose actions in a continuing task (i.e., $T = \infty$).

Similarly, we define the **action-value function**, also known as the Q**-function**, as follows:

$$Q_\pi(s,a) \triangleq \mathbb{E}_\pi[G_0|s_0 = s, a_0 = a] = \mathbb{E}_\pi\left[\sum_{t=0}^{\infty} \gamma^t r_t | s_0 = s, a_0 = a\right] \tag{34.68}$$

This quantity represents the expected return obtained if we start by taking action a in state s, and then follow π to choose actions thereafter.

Finally, we define the **advantage function** as follows:

$$A_\pi(s,a) \triangleq Q_\pi(s,a) - V_\pi(s) \tag{34.69}$$

This tells us the benefit of picking action a in state s then switching to policy π, relative to the baseline return of always following π. Note that $A_\pi(s,a)$ can be both positive and negative, and $\mathbb{E}_{\pi(a|s)}[A_\pi(s,a)] = 0$ due to a useful equality: $V_\pi(s) = \mathbb{E}_{\pi(a|s)}[Q_\pi(s,a)]$.

34.5.5 Optimal value functions and policies

Suppose π_* is a policy such that $V_{\pi_*} \geq V_\pi$ for all $s \in \mathcal{S}$ and all policy π, then it is an **optimal policy**. There can be multiple optimal policies for the same MDP, but by definition their value functions must be the same, and are denoted by V_* and Q_*, respectively. We call V_* the **optimal state-value function**, and Q_* the **optimal action-value function**. Furthermore, any finite MDP must have at least one deterministic optimal policy [Put94].

A fundamental result about the optimal value function is **Bellman's optimality equations**:

$$V_*(s) = \max_a R(s,a) + \gamma \mathbb{E}_{p_T(s'|s,a)}[V_*(s')] \tag{34.70}$$

$$Q_*(s,a) = R(s,a) + \gamma \max_{a'} \mathbb{E}_{p_T(s'|s,a)}[Q_*(s',a')] \tag{34.71}$$

Conversely, the optimal value functions are the only solutions that satisfy the equations. In other words, although the value function is defined as the expectation of a sum of infinitely many rewards, it can be characterized by a recursive equation that involves only one-step transition and reward models of the MDP. Such a recursion play a central role in many RL algorithms we will see later in this chapter. Given a value function (V or Q), the discrepancy between the right- and left-hand sides of Equations (34.70) and (34.71) are called **Bellman error** or **Bellman residual**.

Furthermore, given the optimal value function, we can derive an optimal policy using

$$\pi_*(s) = \operatorname*{argmax}_a Q_*(s,a) \tag{34.72}$$

$$= \operatorname*{argmax}_a \left[R(s,a) + \gamma \mathbb{E}_{p_T(s'|s,a)}[V_*(s')]\right] \tag{34.73}$$

Following such an optimal policy ensures the agent achieves maximum expected return starting from any state. The problem of solving for V_*, Q_* or π_* is called **policy optimization**. In contrast, solving for V_π or Q_π for a given policy π is called **policy evaluation**, which constitutes an important subclass of RL problems as will be discussed in later sections. For policy evaluation, we have similar Bellman equations, which simply replace $\max_a\{\cdot\}$ in Equations (34.70) and (34.71) with $\mathbb{E}_{\pi(a|s)}[\cdot]$.

In Equations (34.72) and (34.73), as in the Bellman optimality equations, we must take a maximum over all actions in $\mathcal{A}$, and the maximizing action is called the **greedy action** with respect to the value functions, Q_* or V_*. Finding greedy actions is computationally easy if $\mathcal{A}$ is a small finite set. For high dimensional continuous spaces, we can treat a as a sequence of actions, and optimize one dimension at a time [Met+17], or use gradient-free optimizers such as cross-entropy method (Section 6.9.5), as used in the **QT-Opt** method [Kal+18a]. Recently, **CAQL** (continuous action Q-learning, [Ryu+20]) proposed to use mixed integer programming to solve the argmax problem, leveraging the ReLU structure of the Q-network. We can also amortize the cost of this optimization by training a policy $a_* = \pi_*(s)$ after learning the optimal Q-function.

34.5.5.1 Example

In this section, we show a simple example, to make concepts like value functions more concrete. Consider the 1d **grid world** shown in Figure 34.11(a). There are 5 possible states, among them S_{T1} and S_{T2} are absorbing states, since the interaction ends once the agent enters them. There are 2 actions, $\uparrow$ and $\downarrow$. The reward function is zero everywhere except at the goal state, S_{T2}, which gives a reward of 1 upon entering. Thus the optimal action in every state is to move down.

Figure 34.11(b) shows the Q_* function for $\gamma = 0$. Note that we only show the function for non-absorbing states, as the optimal Q-values are 0 in absorbing states by definition. We see that $Q_*(s_3, \downarrow) = 1.0$, since the agent will get a reward of 1.0 on the next step if it moves down from s_3; however, $Q_*(s, a) = 0$ for all other state-action pairs, since they do not provide nonzero immediate reward. This optimal Q-function reflects the fact that using $\gamma = 0$ is completely myopic, and ignores the future.

Figure 34.11(c) shows Q_* when $\gamma = 1$. In this case, we care about all future rewards equally. Thus $Q_*(s, a) = 1$ for all state-action pairs, since the agent can always reach the goal eventually. This is infinitely far-sighted. However, it does not give the agent any short-term guidance on how to behave. For example, in s_2, it is not clear if it is should go up or down, since both actions will eventually reach the goal with identical Q_*-values.

Figure 34.11(d) shows Q_* when $\gamma = 0.9$. This reflects a preference for near-term rewards, while also taking future reward into account. This encourages the agent to seek the shortest path to the goal, which is usually what we desire. A proper choice of γ is up to the agent designer, just like the design of the reward function, and has to reflect the desired behavior of the agent.

34.6 Planning in an MDP

In this section, we discuss how to compute an optimal policy when the MDP model is known. This problem is called **planning**, in contrast to the learning problem where the models are unknown, which is tackled using reinforcement learning Chapter 35. The planning algorithms we discuss are based on **dynamic programming** (DP) and **linear programming** (LP).

For simplicity, in this section, we assume discrete state and action sets with $\gamma < 1$. However, exact

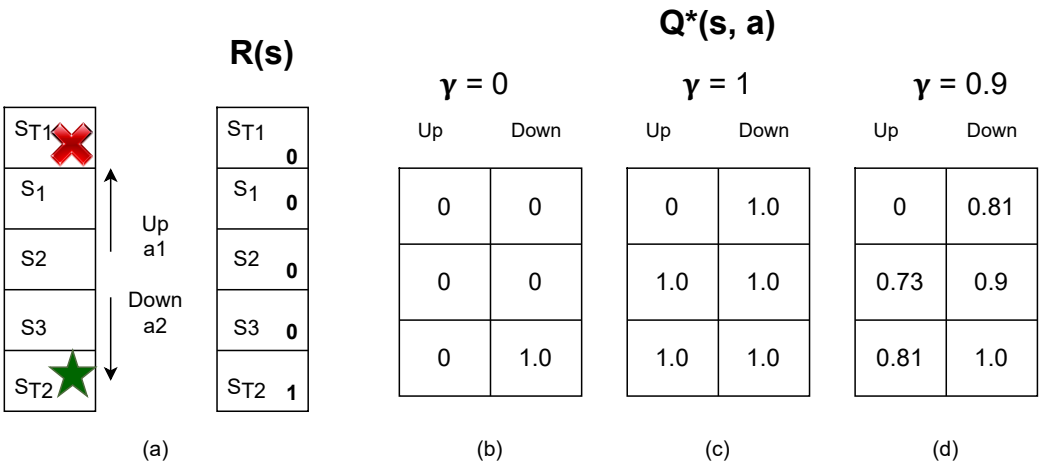


Figure 34.11: Left: illustration of a simple MDP corresponding to a 1d grid world of 3 non-absorbing states and 2 actions. Right: optimal Q-functions for different values of γ. Adapted from Figures 3.1, 3.2, 3.4 of [GK19].

calculation of optimal policies often depends polynomially on the sizes of $\mathcal{S}$ and $\mathcal{A}$, and is intractable, for example, when the state space is a Cartesian product of several finite sets. This challenge is known as the **curse of dimensionality**. Therefore, approximations are typically needed, such as using parametric or nonparametric representations of the value function or policy, both for computational tractability and for extending the methods to handle MDPs with general state and action sets. In this case, we have **approximate dynamic programming** (ADP) and **approximate linear programming** (ALP) algorithms (see e.g., [Ber19]).

34.6.1 Value iteration

A popular and effective DP method for solving an MDP is **value iteration** (VI). Starting from an initial value function estimate V_0, the algorithm iteratively updates the estimate by

$$V_{k+1}(s) = \max_a \left[R(s,a) + \gamma \sum_{s'} p(s'|s,a) V_k(s') \right] \tag{34.74}$$

Note that the update rule, sometimes called a **Bellman backup**, is exactly the right-hand side of the Bellman optimality equation Equation (34.70), with the unknown V_* replaced by the current estimate V_k. A fundamental property of Equation (34.74) is that the update is a **contraction**: it can be verified that

$$\max_s |V_{k+1}(s) - V_*(s)| \leq \gamma \max_s |V_k(s) - V_*(s)| \tag{34.75}$$

In other words, every iteration will reduce the maximum value function error by a constant factor. It follows immediately that V_k will converge to V_*, after which an optimal policy can be extracted

using Equation (34.73). In practice, we can often terminate VI when V_k is close enough to V_*, since the resulting greedy policy wrt V_k will be near optimal. Value iteration can be adapted to learn the optimal action-value function Q_*.

In value iteration, we compute $V_*(s)$ and $\pi_*(s)$ for all possible states s, averaging over all possible next states s' at each iteration, as illustrated in Figure 34.12(right). However, for some problems, we may only be interested in the value (and policy) for certain special starting states. This is the case, for example, in **shortest path problems** on graphs, where we are trying to find the shortest route from the current state to a goal state. This can be modeled as an episodic MDP by defining a transition matrix $p_T(s'|s,a)$ where taking edge a from node s leads to the neighboring node s' with probability 1. The reward function is defined as $R(s,a) = -1$ for all states s except the goal states, which are modeled as absorbing states.

In problems such as this, we can use a method known as **real-time dynamic programming** (RTDP) [BBS95], to efficiently compute an **optimal partial policy**, which only specifies what to do for the reachable states. RTDP maintains a value function estimate V. At each step, it performs a Bellman backup for the current state s by $V(s) \leftarrow \max_a \mathbb{E}_{p_T(s'|s,a)}\left[R(s,a) + \gamma V(s')\right]$. It can picks an action a (often with some exploration), reaches a next state s', and repeats the process. This can be seen as a form of the more general **asynchronous value iteration**, that focuses its computational effort on parts of the state space that are more likely to be reachable from the current state, rather than synchronously updating all states at each iteration.

34.6.2 Policy iteration

Another effective DP method for computing π_* is **policy iteration**. It is an iterative algorithm that searches in the space of deterministic policies until converging to an optimal policy. Each iteration consists of two steps, **policy evaluation** and **policy improvement**.

The policy evaluation step, as mentioned earlier, computes the value function for the current policy. Let π represent the current policy, $\boldsymbol{v}(s) = V_\pi(s)$ represent the value function encoded as a vector indexed by states, $\boldsymbol{r}(s) = \sum_a \pi(a|s)R(s,a)$ represent the reward vector, and $\mathbf{T}(s'|s) = \sum_a \pi(a|s)p(s'|s,a)$ represent the state transition matrix. Bellman's equation for policy evaluation can be written in the matrix-vector form as

$$\boldsymbol{v} = \boldsymbol{r} + \gamma \mathbf{T}\boldsymbol{v} \tag{34.76}$$

This is a linear system of equations in $|\mathcal{S}|$ unknowns, We can solve it using matrix inversion: $\boldsymbol{v} = (\mathbf{I} - \gamma\mathbf{T})^{-1}\boldsymbol{r}$. Alternatively, we can use value iteration by computing $\boldsymbol{v}_{t+1} = \boldsymbol{r} + \gamma\mathbf{T}\boldsymbol{v}_t$ until near convergence, or some form of asynchronous variant that is computationally more efficient.

Once we have evaluated V_π for the current policy π, we can use it to derive a better policy π', thus the name policy improvement. To do this, we simply compute a deterministic policy π' that acts greedily with respect to V_π in every state; that is, $\pi'(s) = \text{argmax}_a\{R(s,a) + \gamma\mathbb{E}\left[V_\pi(s')\right]\}$. We can guarantee that $V_{\pi'} \geq V_\pi$. To see this, define $\boldsymbol{r}'$, $\mathbf{T}'$ and $\boldsymbol{v}'$ as before, but for the new policy π'. The definition of π' implies $\boldsymbol{r}' + \gamma\mathbf{T}'\boldsymbol{v} \geq \boldsymbol{r} + \gamma\mathbf{T}\boldsymbol{v} = \boldsymbol{v}$, where the equality is due to Bellman's equation. Repeating the same equality, we have

$$\boldsymbol{v} \leq \boldsymbol{r}' + \gamma\mathbf{T}'\boldsymbol{v} \leq \boldsymbol{r}' + \gamma\mathbf{T}'(\boldsymbol{r}' + \gamma\mathbf{T}'\boldsymbol{v}) \leq \boldsymbol{r}' + \gamma\mathbf{T}'(\boldsymbol{r}' + \gamma\mathbf{T}'(\boldsymbol{r}' + \gamma\mathbf{T}'\boldsymbol{v})) \leq \cdots \tag{34.77}$$

$$= (\mathbf{I} + \gamma\mathbf{T}' + \gamma^2\mathbf{T}'^2 + \cdots)\boldsymbol{r} = (\mathbf{I} - \gamma\mathbf{T}')^{-1}\boldsymbol{r} = \boldsymbol{v}' \tag{34.78}$$

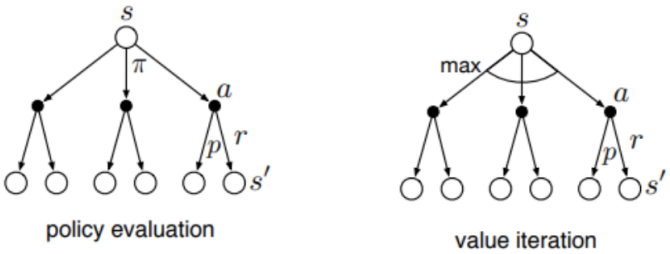


Figure 34.12: Policy iteration vs value iteration represented as backup diagrams. Empty circles represent states, solid (filled) circles represent states and actions. Adapted from Figure 8.6 of [SB18].

Starting from an initial policy π_0, policy iteration alternates between policy evaluation (E) and improvement (I) steps, as illustrated below:

$$\pi_0 \xrightarrow{E} V_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} V_{\pi_1} \cdots \xrightarrow{I} \pi_* \xrightarrow{E} V_* \tag{34.79}$$

The algorithm stops at iteration k, if the policy π_k is greedy with respect to its own value function V_{π_k}. In this case, the policy is optimal. Since there are at most $|\mathcal{A}|^{|\mathcal{S}|}$ deterministic policies, and every iteration strictly improves the policy, the algorithm must converge after finite iterations.

In PI, we alternate between policy evaluation (which involves multiple iterations, until convergence of V_π), and policy improvement. In VI, we alternate between one iteration of policy evaluation followed by one iteration of policy improvement (the "max" operator in the update rule). In **generalized policy improvement**, we are free to intermix any number of these these steps in any order. The process will converge once the policy is greedy wrt its own value function.

Note that policy evaluation computes V_π whereas value iteration computes V_*. This difference is illustrated in Figure 34.12, using a **backup diagram**. Here the root node represents any state s, nodes at the next level represent state-action combinations (solid circles), and nodes at the leaves representing the set of possible resulting next state s' for each possible action. In the former case, we average over all actions according to the policy, whereas in the latter, we take the maximum over all actions.

34.6.3 Linear programming

While dynamic programming is effective and popular, linear programming (LP) provides an alternative that finds important uses, such as in off-policy RL (Section 35.5). The primal form of LP is given by

$$\min_V \sum_s p_0(s)V(s) \quad \text{s.t.} \quad V(s) \geq R(s,a) + \gamma \sum_{s'} p_T(s'|s,a)V(s), \quad \forall (s,a) \in \mathcal{S} \times \mathcal{A} \tag{34.80}$$

where $p_0(s) > 0$ for all $s \in \mathcal{S}$, and can be interpreted as the initial state distribution. It can be verified that any V satisfying the constraint in Equation (34.80) is optimistic [Put94], that is, $V \geq V_*$. When the objective is minimized, the solution V will be "pushed" to the smallest possible, which is V_*. Once V_* is found, any action a that makes the constraint tight in state s is optimal in that state.

The dual LP form is sometimes more intuitive:

$$\max_{d \geq 0} \sum_{s,a} d(s,a) R(s,a) \quad \text{s.t.} \quad \sum_{a} d(s,a) = (1-\gamma) p_0(s) + \gamma \sum_{\bar{s},\bar{a}} p_T(s|\bar{s},\bar{a}) d(\bar{s},\bar{a}) \;\; \forall s \in \mathcal{S} \tag{34.81}$$

Any nonnegative d satisfying the constraint above is the **normalized occupancy distribution** of some corresponding policy $\pi_d(a|s) \triangleq d(s,a)/\sum_{a'} d(s,a')$: [4]

$$d(s,a) = (1-\gamma) \sum_{t=0}^{\infty} \gamma^t p\big(s_t = s, a_t = a | s_0 \sim p_0, a_t \sim \pi_d(s_t)\big) \tag{34.82}$$

The constant $(1-\gamma)$ normalizes d to be a valid distribution, so that it sums to unity. With this interpretation of d, the objective in Equation (34.81) is just the average per-step reward under the normalized occupancy distribution. Once an optimal solution d_* is found, an optimal policy can be immediately obtained by $\pi_*(a|s) = d_*(s,a)/\sum_{a'} d_*(s,a')$.

A challenge in solving the primal or dual LPs for MDPs is the large number of constraints and variables. Approximations are needed, where the variables are parameterized (either linearly or nonlinearly), and the the constraints are sampled or approximated (see e.g., [dV04; LBS17; CLW18]).

34.7 Active learning

This section is coauthored with Zeel B Patel.

In this section, we discuss **active learning** (AL), in which the agent gets to choose which data it wants to use so as to learn the underlying predictive function as quickly as possible, i.e., using the smallest amount of labeled data. This can be much more efficient than using randomly collected data, as illustrated in Figure 34.13. This is useful when labels are expensive to collect, e.g., for medical image classification [GIG17; Wal+20].

There are many approaches to AL, as reviewed in [Set12]. In this section, we just consider a few methods.

34.7.1 Active learning scenarios

One of the earliest AL methods is known as **membership query synthesis** [Ang88]. In this scenario the agent can generate an arbitrary query $\boldsymbol{x} \sim p(\boldsymbol{x})$ and then ask the oracle for its label, $y = f(\boldsymbol{x})$. (An "oracle" is the term given to a system that knows the true answer to every possible question.) This scenario is mostly of theoretical interest, since it is hard to learn good generative models, and it is rarely possible to have access to an oracle on demand (although human-power **crowd computing** platforms can be considered as oracles with high latency).

Another scenario is **stream-based selective sampling** [ACL89], where the agent receives a stream of inputs, $\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots$, and at each step must decide whether to request the label or not. Again, this scenario is mostly of theoretical interest.

The last and widely used setting for machine learning is **pool-based-sampling** [LG94], where the pool of unlabeled samples $\mathcal{X}$ is available from the beginning. At each step we apply an **acquisition**

4. If $\sum_{a'} d(s,a') = 0$ for some state s, then $\pi_d(s)$ may be defined arbitrarily, since s is not visited under the policy.

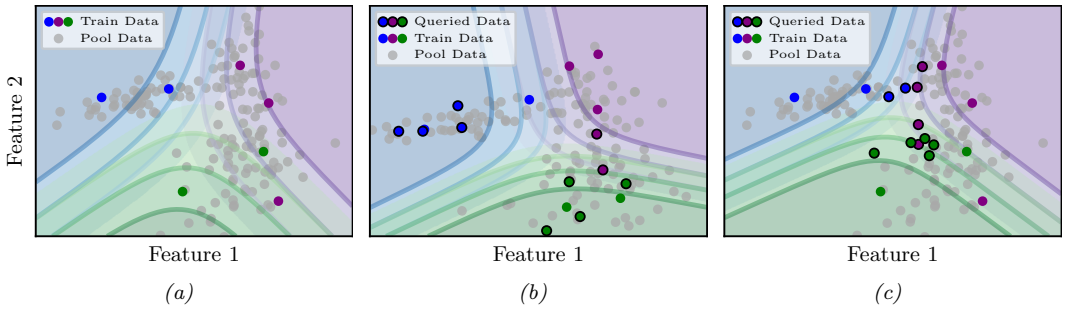


Figure 34.13: Decision boundaries for a logistic regression model applied to a 2-dimensional, 3-class dataset. (a) Results after fitting the model on the initial training data; the test accuracy is 0.818. (b) results after further training on 11 randomly sampled points; accuracy is 0.848. (c) Results after further training on 11 points chosen with least confidence sampling (see Section 34.7.3); accuracy is 0.939. Generated by active_learning_visualization_class.ipynb.

Problem	Goal	Action space
Active Learning	$\text{argmin}_f \mathbb{E}_{p(\boldsymbol{x})} [\ell(f^*(\boldsymbol{x}), f(\boldsymbol{x}))]$	choose $\boldsymbol{x}$ at which to get $\boldsymbol{y} = f^*(\boldsymbol{x})$
Bayesian optimization	$\text{argmax}_{\boldsymbol{x} \in \mathcal{X}} f^*(\boldsymbol{x})$	choose $\boldsymbol{x}$ at which to evaluate $f^*(\boldsymbol{x})$
Contextual bandits	$\text{argmax}_\pi \mathbb{E}_{p(\boldsymbol{x})\pi(a\|\boldsymbol{x})} [R^*(\boldsymbol{x}, a)]$	choose a at which to evaluate $R^*(\boldsymbol{x}, a)$

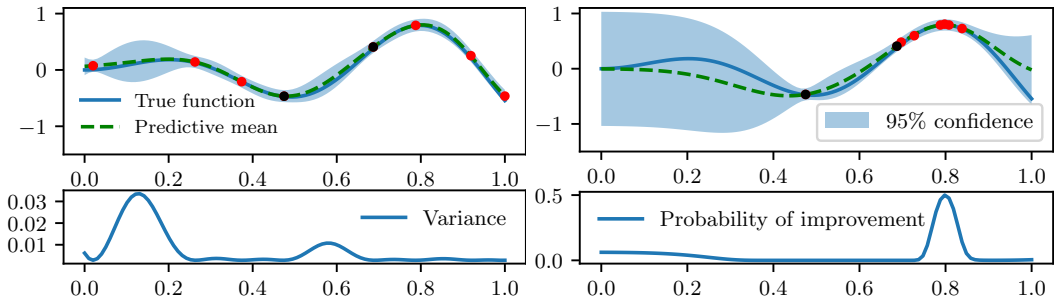
Table 34.1: Comparison among active learning, Bayesian optimization and contextual bandits in terms of goal and action space

function to each candidate in the batch, to decide which one to collect the label for. We then collect the label, update the model with the new data, and repeat the process until we exhaust the pool, run out of time, or reach some desired performance. In the subsequent sections, we will focus only on pool-based sampling.

34.7.2 Relationship to other forms of sequential decision making

(Pool-based) active learning is closely related to Bayesian optimization (BO, Section 6.8) and contexual bandit problems (Section 34.4). The connections are discussed at length [Tou14], but in brief, the methods differ because they solve slightly different objective functions, as summarized in Table 34.1. In particular, in active learning, our goal is to identify a function $f : \mathcal{X} \to \mathcal{Y}$ that will incur minimum expected loss when applied to random inputs $\boldsymbol{x}$; in BO, our goal is to identify an input point $\boldsymbol{x}$ where the function output $f(\boldsymbol{x})$ is maximal; and in bandits, our goal is to identify a policy $\pi : \mathcal{X} \to \mathcal{A}$ that will give maximum expected reward when applied to random inputs (contexts) $\boldsymbol{x}$. (We see that the goal in AL and bandits is similar, but in bandits the agent only gets to choose the action, not the state, so only has partial control over where the (reward) function is evaluated.)

In all three problems, we want to find the optimum with as few actions as possible, so we have to solve the exploration-exploitation problem (Section 34.4.3). One approach is to represent our uncertainty about the function using a method such as a Gaussian process (Chapter 18), which lets



(a) Active learning *(b) Bayesian optimization*

Figure 34.14: Active learning vs Bayesian optimization. Active learning tries to approximate the true function well. Bayesian optimization tries to find maximum value of the true function. Initial and queried points are denoted as black and red dots respectively. Generated by bayes_opt_vs_active_learning.ipynb.

us compute $p(f|\mathcal{D}_{1:t})$. We then define some acquisition function $\alpha(\boldsymbol{x})$ that evaluates how useful it would be to query the function at input location $\boldsymbol{x}$, given the belief state $p(f|\mathcal{D}_{1:t})$ and we pick as our next query $\boldsymbol{x}_{t+1} = \text{argmax}_{\boldsymbol{x}}\, \alpha(\boldsymbol{x})$. (In the bandit setting, the agent does not get to choose the state $\boldsymbol{x}$, but does get to choose action a.) For example, in BO, it is common to use probability of improvement (Section 6.8.3.1), and for AL of a regression task, we can use the posterior predictive variance. The objective for AL will cause the agent to query "all over the place", whereas for BO, the agent will "zoom in" on the most promising regions, as shown in Figure 34.14. We discuss other acquisition functions for AL in Section 34.7.3.

34.7.3 Acquisition strategies

In this section, we discuss some common AL heuristics for choosing which points to query.

34.7.3.1 Uncertainty sampling

An intuitive heuristic for choosing which example to label next is to pick the one for which the model is currently most uncertain. This is called **uncertainty sampling**. We already illustrated this in the case of regression in Figure 34.14, where we represented uncertainty in terms of the posterior variance.

For classification problems, we can measure uncertainty in various ways. Let $\boldsymbol{p}_n = [p(y = c|\boldsymbol{x}_n)]_{c=1}^C$ be the vector of class probabilities for each unlabeled input $\boldsymbol{x}_n$. Let $U_n = \alpha(\boldsymbol{p}_n)$ be the uncertainty for example n, where α is an acquisition function. Some common choices for α are: **entropy sampling** [SW87a], which uses $\alpha(\boldsymbol{p}) = -\sum_{c=1}^C p_c \log p_c$; **margin sampling**, which uses $\alpha(\boldsymbol{p}) = p_1 - p_2$, where p_1 is the probability of the most probable class, and p_2 is the probability of the second most probable class; and **least confident sampling**, which uses $\alpha(\boldsymbol{p}) = 1 - p_{c^*}$, where $c^* = \text{argmax}_c\, p_c$. The difference between these strategies is shown in Figure 34.15. In practice it is often found that margin sampling works the best [Chu+19].

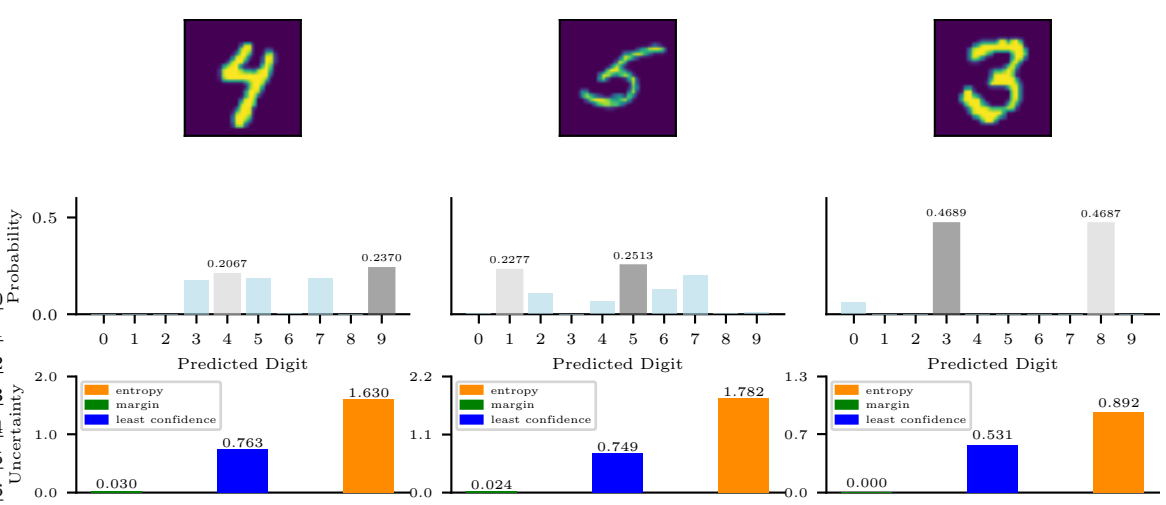


Figure 34.15: Outputs of a logistic regression model fit on some training points, and then applied to 3 candidate query inputs. We show the predicted probabilites for each class label. The highlighted dark gray is the max probability, the light gray bar is the 2nd highest probability. The least confident scores for the 3 inputs are: 1-0.23=0.76, 1-0.25=0.75, and 1-0.47=0.53, so we pick the first query. The entropy scores are: 1.63, 1.78 and 0.89, so we pick the second query. The margin scores are: 0.237-0.2067=0.0303, 0.2513-0.2277=0.0236, and 0.4689-0.4687=0.0002, so we pick the first query. Generated by active_learning_comparison_mnist.ipynb.

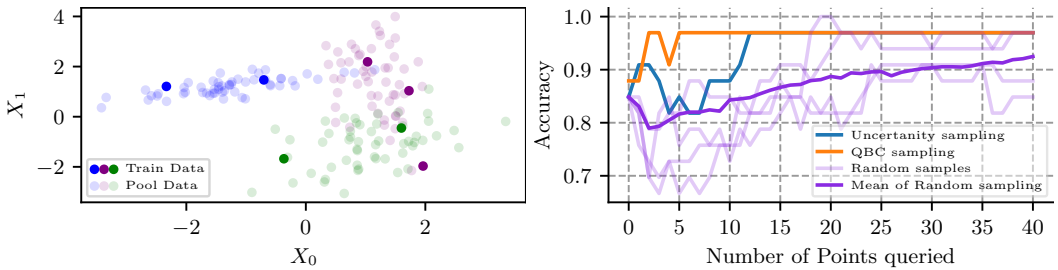
34.7.3.2 Query by committee

In this section, we discuss how to apply uncertainty sampling to models, such as support vector machines (SVMs), that only return a point prediction rather than a probability distribution. The basic approach is to create an ensemble of diverse models, and to use disagreement between the model predictions as a form of uncertainty. (This can be useful even for probabilistic models, such as DNNs, since model uncertainty can often be larger than parametric uncertainty, as we discuss in the section on deep ensembles, Section 17.3.9.)

In more detail, suppose we have K ensemble members, and let c_n^k be the predicted class from member k on input $\boldsymbol{x}_n$. Let $v_{nc} = \sum_{k=1}^{K} \mathbb{I}\left(c_n^k = c\right)$ be the number of votes cast for class c, and $q_{nc} = v_{nc}/C$ be the induced distribution. (A similar method can be used for regression models, where we use the standard deviation of the prediction across the members.) We can then use margin sampling or entropy sampling with distribution $\boldsymbol{q}_n$. This approach is called **query by committee (QBC)** [SOS92], and can often out-perform vanilla uncertainty sampling with a single model, as we show in Figure 34.16.

34.7.3.3 Information theoretic methods

A natural acquisition strategy is to pick points whose labels will maximimally reduce our uncertainty about the model parameters $\boldsymbol{w}$. This is known as the **information gain** criterion, and was first



(a) (b)

Figure 34.16: (a) Random Forest (RF) classifier applied to a 2-dimensional, 3-class data set. (b) Accuracy vs number of query points for margin sampling vs random sampling. We represent uncertainty either using a single RF (based on the predicted distribution over labels induced by the trees in the forest), or a committee containing an RF and a logistic regression model. Generated by active_learning_compare_class.ipynb.

proposed in [Lin56]. It is defined as follows:

$$\alpha(\boldsymbol{x}) \triangleq \mathbb{H}\left(p(\boldsymbol{w}|\mathcal{D})\right) - \mathbb{E}_{p(y|\boldsymbol{x},\mathcal{D})}\left[\mathbb{H}\left(p(\boldsymbol{w}|\mathcal{D},\boldsymbol{x},y)\right)\right] \tag{34.83}$$

(Note that the first term is a constant wrt $\boldsymbol{x}$, but we include it for later convenience.) This is equivalent to the expected change in the posterior over the parameters which is given by

$$\alpha'(\boldsymbol{x}) \triangleq \mathbb{E}_{p(y|\boldsymbol{x},\mathcal{D})}\left[D_{\mathbb{KL}}\left(p(\boldsymbol{w}|\mathcal{D},\boldsymbol{x},y) \parallel p(\boldsymbol{w}|\mathcal{D})\right)\right] \tag{34.84}$$

Using symmetry of the mutual information, we can rewrite Equation (34.83) as follows:

$$\alpha(\boldsymbol{x}) = \mathbb{H}\left(\boldsymbol{w}|\mathcal{D}\right) - \mathbb{E}_{p(y|\boldsymbol{x},\mathcal{D})}\left[\mathbb{H}\left(\boldsymbol{w}|\mathcal{D},\boldsymbol{x},y\right)\right] \tag{34.85}$$

$$= \mathbb{I}(\boldsymbol{w}, y|\mathcal{D},\boldsymbol{x}) \tag{34.86}$$

$$= \mathbb{H}\left(y|\boldsymbol{x},\mathcal{D}\right) - \mathbb{E}_{p(\boldsymbol{w}|\mathcal{D})}\left[\mathbb{H}\left(y|\boldsymbol{x},\boldsymbol{w},\mathcal{D}\right)\right] \tag{34.87}$$

The advantage of this approach is that we now only have to reason about the uncertainty of the predictive distribution over outputs y, not over the parameters $\boldsymbol{w}$. This approach is called **Bayesian active learning by disagreement** or **BALD** [Hou+12].

Equation (34.87) has an interesting interpretation. The first term prefers examples $\boldsymbol{x}$ for which there is uncertainty in the predicted label. Just using this as a selection criterion is equivalent to uncertainty sampling, which we discussed above. However, this can have problems with examples which are inherently ambiguous or mislabeled. By adding the second term, we penalize such behavior, since we add a large negative weight to points whose predictive distribution is entropic even when we know the parameters. Thus we ignore aleatoric (intrinsic) uncertainty and focus on epistemic uncertainty.

34.7.4 Batch active learning

In many applications, we need to select a batch of unlabeled examples at once, since training a model on single examples is too slow. This is called **batch active learning**. The key challenge is that we

need to ensure the different queries that we request are diverse, so we maximize the information gain. Various methods for this problem have been devised; here we focus on the **BatchBALD** method of [KAG19], which extends the BALD method of Section 34.7.3.3.

34.7.4.1 BatchBALD

The naive way to extend the BALD score to a batch of b candidate query points is to define

$$\alpha_{\text{BALD}}(\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_B\}, p(\boldsymbol{w}|\mathcal{D})) = \alpha_{\text{BALD}}(\boldsymbol{x}_{1:B}, p(\boldsymbol{w}|\mathcal{D})) = \sum_{i=1}^{B} \mathbb{I}(y_i; \boldsymbol{w}|\boldsymbol{x}_i, \mathcal{D}) \tag{34.88}$$

However this may pick points that are quite similar in terms of their information content. In BatchBALD, we use joint conditional mutual information between the set of labels and the parameters:

$$\alpha_{\text{BBALD}}(\boldsymbol{x}_{1:B}, p(\boldsymbol{w}|\mathcal{D})) = \mathbb{I}(\boldsymbol{y}_{1:B}; \boldsymbol{w}|\boldsymbol{x}_{1:B}, \mathcal{D}) = \mathbb{H}(\boldsymbol{y}_{1:b}|\boldsymbol{x}_{1:B}, \mathcal{D}) - \mathbb{E}_{p(\boldsymbol{w}|\mathcal{D})}\left[\mathbb{H}(\boldsymbol{y}_{1:B}|\boldsymbol{x}_{1:B}, \boldsymbol{w}, \mathcal{D})\right] \tag{34.89}$$

To understand how this differs from BALD, we will use information diagrams for representing MI in terms of Venn diagrams, as explained in Section 5.3.2. In particular, [Yeu91a] showed that we can define a signed measure, μ^*, for discrete random variables x and y such that $\mathbb{I}(x; y) = \mu^*(x \cap y)$, $\mathbb{H}(x, y) = \mu^*(x \cup y)$, $\mathbb{E}_{p(y)}\left[\mathbb{H}(x|y)\right] = \mu^*(x \setminus y)$, etc. Using this, we can interpret standard BALD as the sum of the individual intersections, $\sum_i \mu^*(y_i \cap \boldsymbol{w})$, which double counts overlaps between the y_i, as shown in Figure 34.17(a). By contrast, BatchBALD takes overlap into account by computing

$$\mathbb{I}(\boldsymbol{y}_{1:B}; \boldsymbol{w}|\boldsymbol{x}_{1:B}, \mathcal{D}) = \mu^*(\cup_i y_i \cap \boldsymbol{w}) = \mu^*(\cup_i y_i) - \mu^*(\cup_i y_i \setminus \boldsymbol{w}) \tag{34.90}$$

This is illustrated in Figure 34.17(b). From this, we can see that $\alpha_{\text{BBALD}} \leq \alpha_{\text{BALD}}$. Indeed, one can show[5]

$$\mathbb{I}(\boldsymbol{y}_{1:B}, \boldsymbol{w}|\boldsymbol{x}_{1:B}, \mathcal{D}) = \sum_{i=1}^{B} \mathbb{I}(y_i, \boldsymbol{w}|\boldsymbol{x}_{1:B}, \mathcal{D}) - \mathbb{TC}(\boldsymbol{y}_{1:B}|\boldsymbol{x}_{1:B}, \mathcal{D}) \tag{34.91}$$

where TC is the total correlation (see Section 5.3.5.1).

34.7.4.2 Optimizing BatchBALD

To avoid the combinatorial explosion that arises from jointly scoring subsets of points, we can use a a greedy approximation for computing BatchBALD one point at a time. In particular, suppose at step $n-1$ we already have a partial batch $\mathcal{A}_{n-1}$. The next point is chosen using

$$\boldsymbol{x}_n = \underset{\boldsymbol{x} \in \mathcal{D}_{\text{pool}} \setminus \mathcal{A}_{n-1}}{\text{argmax}} \; \alpha_{\text{BBALD}}(\mathcal{A}_{n-1} \cup \{\boldsymbol{x}\}, p(\boldsymbol{w}|\mathcal{D})) \tag{34.92}$$

We then add $\boldsymbol{x}_n$ to $\mathcal{A}_{n-1}$ to get $\mathcal{A}_n$. Fortunately the BatchBALD acquisition function is submodular, as shown in [KAG19]. Hence this greedy algorithm is within $1 - 1/e \approx 0.63$ of optimal (see Section 6.11.4.1).

5. See http://blog.blackhc.net/2022/07/kbald/

(a) BALD *(b) BatchBALD*

Figure 34.17: Intuition behind BALD and BatchBALD. D_{pool} is an unlabelled dataset (from which $\boldsymbol{x}_{1:b}$ are taken) , D_{train} is the current training set, $\boldsymbol{w}$ is set of model parameters, $p(y|\boldsymbol{x},\boldsymbol{w},D_{train})$ are output predictions for data point $\boldsymbol{x}$. BALD overestimates the joint mutual information whereas BatchBALD takes the overlap between variables into account. Areas contributing to the respective score are shown in grey, and areas that are double-counted in dark grey. From Figure 3 of [KAG19]. Used with kind permission of Andreas Kirsch.

34.7.4.3 Computing BatchBALD

Computing the joint (conditional) mutual information is intractable, so in this section, we discuss how to approximate it. For brevity we drop the conditioning on $\boldsymbol{x}$ and $\mathcal{D}$. With this new notation, the objective becomes

$$\alpha_{\text{BBALD}}(\boldsymbol{x}_{1:B}, p(\boldsymbol{w}|\mathcal{D})) = \mathbb{H}(y_1,\dots,y_B) - \mathbb{E}_{p(\boldsymbol{w})}\left[\mathbb{H}(y_1,\dots,y_B|\boldsymbol{w})\right] \tag{34.93}$$

Note that the y_i are conditionally independent given $\boldsymbol{w}$, so $\mathbb{H}(y_1,\dots,y_B|\boldsymbol{w}) = \sum_{i=1}^{B}\mathbb{H}(y_i|\boldsymbol{w})$. Hence we can approximate the second term with Monte Carlo:

$$\mathbb{E}_{p(\boldsymbol{w})}\left[\mathbb{H}(y_1,\dots,y_B|\boldsymbol{w})\right] \approx \frac{1}{S}\sum_{i=1}^{n}\sum_{s}\mathbb{H}(y_i|\hat{\boldsymbol{w}}_s) \tag{34.94}$$

where $\hat{\boldsymbol{w}}_s \sim p(\boldsymbol{w}|\mathcal{D})$.

The first term, $\mathbb{H}(y_1,\dots,y_B)$, is a joint entropy, so is harder to compute. [KAG19] propose the following approximation, summing over all possible label sequences in the batch, and leveraging the fact that $p(\boldsymbol{y}) = \mathbb{E}_{p(\boldsymbol{w})}\left[p(\boldsymbol{y}|\boldsymbol{w})\right]$:

$$\mathbb{H}(\boldsymbol{y}_{1:B}) = \mathbb{E}_{p(\boldsymbol{w})p(\boldsymbol{y}_{1:B}|vw)}\left[-\log p(\boldsymbol{y}_{1:B}|\boldsymbol{w})\right] \tag{34.95}$$

$$\approx \sum_{\hat{\boldsymbol{y}}_{1:B}}\left(\frac{1}{S}\sum_{s=1}^{S}p(\hat{\boldsymbol{y}}_{1:B}|\hat{\boldsymbol{w}}_s)\right)\log\left(\frac{1}{S}\sum_{s=1}^{S}p(\hat{\boldsymbol{y}}_{1:B}|\hat{\boldsymbol{w}}_s)\right) \tag{34.96}$$

The sum over all possible labels sequences can be made more efficient by noting that $p(\boldsymbol{y}_{1:n}|\boldsymbol{w}) = p(y_n|\boldsymbol{w})p(\boldsymbol{y}_{1:n-1}|\boldsymbol{w})$, so when we implement the greedy algorithm, we can incrementally update the probabilities, reusing previous computations. See [KAG19] for the details.

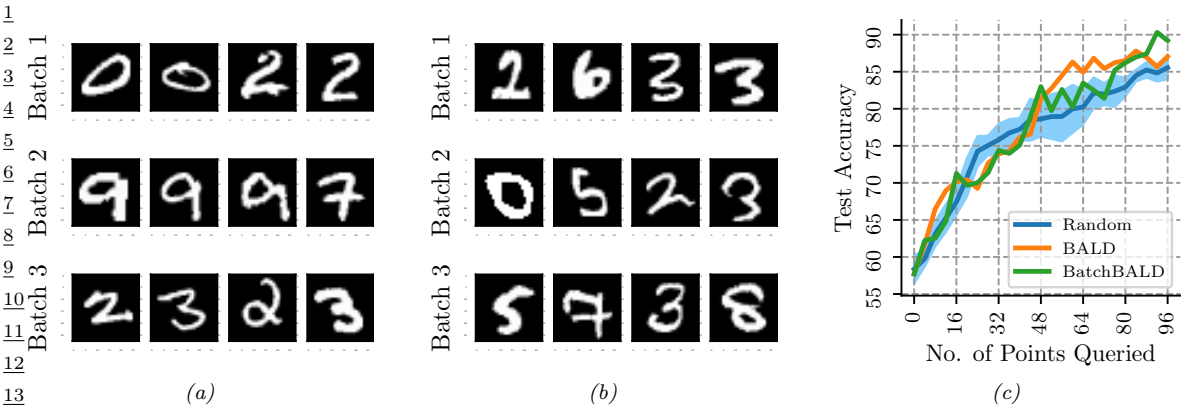


Figure 34.18: Three batches (each of size 4) queried from the MNIST pool by (a) BALD and (b) BatchBALD. (c) Plot of accuracy vs number of points queried. BALD may select replicas of single informative datapoint while BatchBALD selects diverse points, thus increasing data efficiency. Generated by batch_bald_mnist.ipynb.

34.7.4.4 Experimental comparison of BALD vs BatchBALD on MNIST

In this section, we show some experimental results applying BALD and BatchBALD to train a CNN on the standard MNIST dataset. We use a batch size of 4, and approximate the posterior over parameters $p(\boldsymbol{w}|\mathcal{D})$ using MC dropout (Section 17.3.1). In Figure 34.18(a), we see that BALD selects examples that are very similar to each other, whereas in Figure 34.18(b), we see that BatchBALD selects a greater diversity of points. In Figure 34.18(c), we see that BatchBALD results in more efficient learning than BALD, which in turn is more efficient than randomly sampling data.

35 Reinforcement learning

This chapter was co-authored with Lihong Li.

35.1 Introduction

Reinforcement learning or **RL** is a paradigm of learning where an agent sequentially interacts with an initially unknown environment. The interaction typically results in a **trajectory**, or multiple trajectories. Let $\boldsymbol{\tau} = (s_0, a_0, r_0, s_1, a_1, r_1, s_2, \ldots, s_T)$ be a trajectory of length T, consisting of a sequence of states s_t, actions a_t, and rewards r_t.[1] The goal of the agent is to optimize her action-selection policy, so that the discounted cumulative reward, $G_0 \triangleq \sum_{t=0}^{T-1} \gamma^t r_t$, is maximized for some given **discount factor** $\gamma \in [0, 1]$.

In general, G_0 is a random variable. We will focus on maximizing its expectation, inspired by the maximum expected utility principle (Section 34.1.1), but note other possibilities such as **conditional value at risk**[2] that can be more appropriate in risk-sensitive applications.

We will focus on the Markov decision process, where the generative model for the trajectory $\boldsymbol{\tau}$ can be factored into single-step models. When these model parameters are known, solving for an optimal policy is called **planning** (see Section 34.6); otherwise, RL algorithms may be used to obtain an optimal policy from trajectories, a process called **learning**.

In **model-free RL**, we try to learn the policy without explicitly representing and learning the models, but directly from the trajectories. In **model-based RL**, we first learn a model from the trajectories, and then use a planning algorithm on the learned model to solve for the policy. See Figure 35.1 for an overview. This chapter will introduce some of the key concepts and techniques, and will mostly follow the notation from [SB18]. More details can be found in textbooks such as [Sze10; SB18; Ber19; Aga+21a; Mey22; Rei22], and reviews such as [WO12; Aru+17; FL+18; Li18].

35.1.1 Overview of methods

In this section, we give a brief overview of how to compute optimal policies when the MDP model is not known. Instead, the agent interacts with the environment and learns from the observed

1. Note that the time starts at 0 here, while it starts at 1 when we discuss bandits (Section 34.4). Our choices of notation is to be consistent with conventions in respective literature.
2. The conditional value at risk, or CVaR, is the expected reward conditioned on being in the worst 5% (say) of samples. See [Cho+15] for an example application in RL.

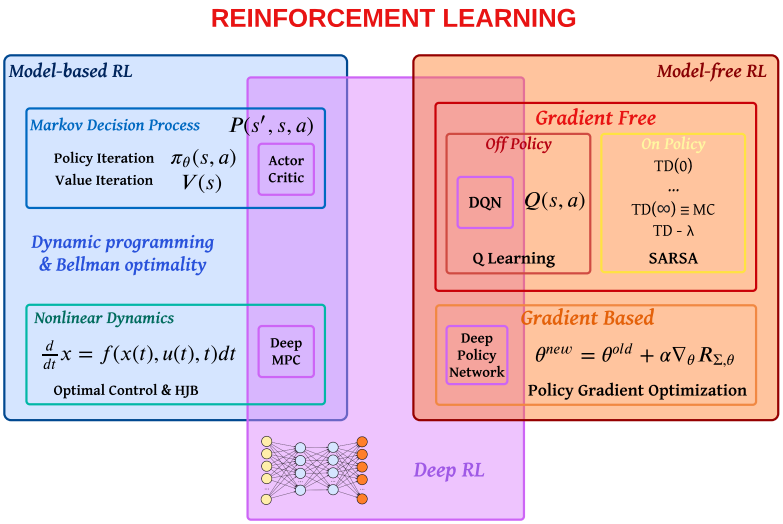


Figure 35.1: Overview of RL methods. Abbreviations: DQN = Deep Q network (Section 35.2.6); MPC = Model Predictive Control (Section 35.4); HJB = Hamilton Jacobi Bellman equation; TD = temporal difference learning (Section 35.2.2). Adapted from a slide by Steve Brunton.

Method	Functions learned	On/Off	Section
SARSA	$Q(s, a)$	On	Section 35.2.4
Q-learning	$Q(s, a)$	Off	Section 35.2.5
REINFORCE	$\pi(a\|s)$	On	Section 35.3.2
A2C	$\pi(a\|s)$, $V(s)$	On	Section 35.3.3.1
TRPO / PPO	$\pi(a\|s)$, $A(s, a)$	On	Section 35.3.4
DDPG	$a = \pi(s)$, $Q(s, a)$	Off	Section 35.3.5
Soft actor-critic	$\pi(a\|s)$, $Q(s, a)$	Off	Section 35.6.1
Model-based RL	$p(s'\|s, a)$	Off	Section 35.4

Table 35.1: Summary of some popular methods for RL. On/off refers to on-policy vs off-policy methods.

trajectories. This is the core focus of RL. We will go into more details into later sections, but first provide this roadmap.

We may categorize RL methods by the quantity the agent represents and learns: value function, policy, and model; or by how actions are selected: on-policy (actions must be selected by the agent's current policy), and off-policy. Table 35.1 lists a few representative examples. More details are given in the subsequent sections. We will also discuss at greater depth two important topics of off-policy learning and inference-based control in Sections 35.5 and 35.6.

35.1.2 Value based methods

In a value based method, we often try to learn the optimal Q-function from experience, and then derive a policy from it using Equation (34.72). Typically, a function approximator (e.g., a neural network), $Q_{\boldsymbol{w}}$, is used to represent the Q-function, which is trained iteratively. Given a transition (s, a, r, s'), we define the **temporal difference** (also called the **TD error**) as

$$r + \gamma \max_{a'} Q_{\boldsymbol{w}}(s', a') - Q_{\boldsymbol{w}}(s, a)$$

Clearly, the expected TD error is the Bellman error evaluated at (s, a). Therefore, if $Q_{\boldsymbol{w}} = Q_*$, the TD error is 0 on average by Bellman's optimality equation. Otherwise, the error provides a signal for the agent to change $\boldsymbol{w}$ to make $Q_{\boldsymbol{w}}(s, a)$ closer to $R(s, a) + \gamma \max_{a'} Q_{\boldsymbol{w}}(s', a')$. The update on $Q_{\boldsymbol{w}}$ is based on a target that is computed using $Q_{\boldsymbol{w}}$. This kind of update is known as **bootstrapping** in RL, and should not be confused with the statistical bootstrap (Section 13.2.3.1). Value based methods such as **Q-learning** and **SARSA** are discussed in Section 35.2.

35.1.3 Policy search methods

In **policy search**, we try to directly maximize $J(\pi_{\boldsymbol{\theta}})$ wrt the policy parameter $\boldsymbol{\theta}$. If $J(\pi_{\boldsymbol{\theta}})$ is differentiable wrt $\boldsymbol{\theta}$, we can use stochastic gradient ascent to optimize $\boldsymbol{\theta}$, which is known as **policy gradient**, as described in Section 35.3.1. The basic idea is to perform **Monte Carlo rollouts**, in which we sample trajectories by interacting with the environment, and then use the score function estimator (Section 6.5.3) to estimate $\nabla_{\boldsymbol{\theta}} J(\pi_{\boldsymbol{\theta}})$. Here, $J(\pi_{\boldsymbol{\theta}})$ is defined as an expectation whose distribution depends on $\boldsymbol{\theta}$, so it is invalid to swap ∇ and $\mathbb{E}$ in computing the gradient, and the score function estimator can be used instead. An example of policy gradient is **REINFORCE**.

Policy gradient methods have the advantage that they provably converge to a local optimum for many common policy classes, whereas Q-learning may diverge when approximation is used (Section 35.5.3). In addition, policy gradient methods can easily be applied to continuous action spaces, since they do not need to compute $\text{argmax}_a Q(s, a)$. Unfortunately, the score function estimator for $\nabla_{\boldsymbol{\theta}} J(\pi_{\boldsymbol{\theta}})$ can have a very high variance, so the resulting method can converge slowly.

One way to reduce the variance is to learn an approximate value function, $V_{\boldsymbol{w}}(s)$. and to use it as a baseline in the score function estimator. We can learn $V_{\boldsymbol{w}}(s)$ using one of the value function methods similar to Q-learning. Alternatively, we can learn an advantage function, $A_{\boldsymbol{w}}(s, a)$, and use it to estimate the gradient. These policy gradient variants are called **actor critic** methods, where the actor refers to the policy $\pi_{\boldsymbol{\theta}}$ and the critic refers to $V_{\boldsymbol{w}}$ or $A_{\boldsymbol{w}}$. See Section 35.3.3 for details.

35.1.4 Model-based RL

Value-based methods, such as Q-learning, and policy search methods, such as policy gradient, can be very **sample inefficient**, which means they may need to interact with the environment many times before finding a good policy. If an agent has prior knowledge of the MDP model, it can be more sample efficient to first learn the model, and then compute an optimal (or near-optimal) policy of the model without having to interact with the environment any more.

This approach is called **model-based RL**. The first step is to learn the MDP model including the $p_T(s'|s, a)$ and $R(s, a)$ functions, e.g., using DNNs. Given a collection of (s, a, r, s') tuples, such a model can be learned using standard supervised learning methods. The second step can be done

by running an RL algorithm on synthetic experiences generated from the model, or by running a planning algorithm on the model directly (Section 34.6). In practice, we often interleave the model learning and planning phases, so we can use the partially learned policy to decide what data to collect. We discuss model-based RL in more detail in Section 35.4.

35.1.5 Exploration-exploitation tradeoff

A fundamental problem in RL with unknown transition and reward models is to decide between choosing actions that the agent knows will yield high reward, or choosing actions whose reward is uncertain, but which may yield information that helps the agent get to parts of state-action space with even higher reward. This is called the **exploration-exploitation tradeoff**, which has been discussed in the simpler contextual bandit setting in Section 34.4. The literature on efficient exploration is huge. In this section, we briefly describe several representative techniques.

35.1.5.1 ϵ-greedy

A common heuristic is to use an **ϵ-greedy** policy π_ϵ, parameterized by $\epsilon \in [0, 1]$. In this case, we pick the greedy action wrt the current model, $a_t = \text{argmax}_a \hat{R}_t(s_t, a)$ with probability $1 - \epsilon$, and a random action with probability ϵ. This rule ensures the agent's continual exploration of all state-action combinations. Unfortunately, this heuristic can be shown to be suboptimal, since it explores every action with at least a constant probability $\epsilon/|\mathcal{A}|$.

35.1.5.2 Boltzmann exploration

A source of inefficiency in the ϵ-greedy rule is that exploration occurs uniformly over all actions. The **Boltzmann policy** can be more efficient, by assigning higher probabilities to explore more promising actions:

$$\pi_\tau(a|s) = \frac{\exp(\hat{R}_t(s_t, a)/\tau)}{\sum_{a'} \exp(\hat{R}_t(s_t, a')/\tau)} \tag{35.1}$$

where $\tau > 0$ is a temperature parameter that controls how entropic the distribution is. As τ gets close to 0, π_τ becomes close to a greedy policy. On the other hand, higher values of τ will make $\pi(a|s)$ more uniform, and encourage more exploration. Its action selection probabilities can be much "smoother" with respect to changes in the reward estimates than ϵ-greedy, as illustrated in Table 35.2.

35.1.5.3 Upper confidence bounds and Thompson sampling

The upper confidence bound (UCB) (Section 34.4.5) and Thompson sampling (Section 34.4.6) approaches may also be extended to MDPs. In contrast to the contextual bandit case, where the only uncertainty is in the reward function, here we must also take into account uncertainty in the transition probabilities.

As in the bandit case, the UCB approach requires to estimate an upper confidence bound for all actions' Q-values in the current state, and then take the action with the highest UCB score. One way to obtain UCBs of the Q-values is to use **count-based exploration**, where we learn the optimal

$\hat{R}(s,a_1)$	$\hat{R}(s,a_2)$	$\pi_\epsilon(a\|s_1)$	$\pi_\epsilon(a\|s_2)$	$\pi_\tau(a\|s_1)$	$\pi_\tau(a\|s_2)$
1.00	9.00	0.05	0.95	0.00	1.00
4.00	6.00	0.05	0.95	0.12	0.88
4.90	5.10	0.05	0.95	0.45	0.55
5.05	4.95	0.95	0.05	0.53	0.48
7.00	3.00	0.95	0.05	0.98	0.02
8.00	2.00	0.95	0.05	1.00	0.00

Table 35.2: Comparison of ϵ-greedy policy (with $\epsilon = 0.1$) and Boltzmann policy (with $\tau = 1$) for a simple MDP with 6 states and 2 actions. Adapted from Table 4.1 of [GK19].

Q-function with an **exploration bonus** added to the reward in a transition (s, a, r, s'):

$$\tilde{r} = r + \alpha/\sqrt{N_{s,a}} \tag{35.2}$$

where $N_{s,a}$ is the number of times action a has been taken in state s, and $\alpha \geq 0$ is a weighting term that controls the degree of exploration. This is the approach taken by the **MBIE-EB** method [SL08] for finite-state MDPs, and in the generalization to continuous-state MDPs through the use of hashing [Bel+16]. Other approaches also explicitly maintain uncertainty in state transition probabilities, and use that information to obtain UCBs. Examples are **MBIE** [SL08], **UCRL2** [JOA10], **UCBVI** [AOM17], among many others.

Thompson sampling can be similarly adapted, by maintaining the posterior distribution of the reward and transition model parameters. In finite-state MDPs, for example, the transition model is a categorical distribution conditioned on the state. We may use the conjugate prior of Dirichlet distributions (Section 3.2) for the transition model, so that the posterior distribution can be conveniently computed and sampled from. More details on this approach are found in [Rus+18].

Both UCB and Thompson sampling methods have been shown to yield efficient exploration with provably strong regret bounds (Section 34.4.7) [JOA10], or related PAC bounds [SLL09; DLB17], often under necessary assumptions such as finiteness of the MDPs. In practice, these methods may be combined with function approximation like neural networks and implemented approximately.

35.1.5.4 Optimal solution using Bayes-adaptive MDPs

The Bayes optimal solution to the exploration-exploitation tradeoff can be computed by formulating the problem as a special kind of POMDP known as a **Bayes-adaptive MDP** or **BAMDP** [Duf02]. This extends the Gittins index approach in Section 34.4.4 to the MDP setting.

In particular, a BAMDP has a **belief state** space, $\mathcal{B}$, representing uncertainty about the reward model $p_R(r|s, a, s')$ and transition model $p_T(s'|s, a)$. The transition model on this augmented MDP can be written as follows:

$$T^+(s_{t+1}, b_{t+1}|s_t, b_t, a_t, r_t) = T^+(s_{t+1}|s_t, a_t, b_t)T^+(b_{t+1}|s_t, a_t, r_t, s_{t+1}) \tag{35.3}$$

$$= \mathbb{E}_{b_t}\left[T(s_{t+1}|s_t, a_t)\right] \times \mathbb{I}\left(b_{t+1} = p(R, T|\boldsymbol{h}_{t+1})\right) \tag{35.4}$$

where $\mathbb{E}_{b_t}\left[T(s_{t+1}|s_t, a_t)\right]$ is the posterior predictive distribution over next states, and $p(R, T|\boldsymbol{h}_{t+1})$ is the new belief state given $\boldsymbol{h}_{t+1} = (s_{1:t+1}, a_{1:t+1}, r_{1:t+1})$, which can be computed using Bayes rule.

Similarly, the reward function for the augmented MDP is given by

$$R^+(r|s_t, b_t, a_t, s_{t+1}, b_{t+1}) = \mathbb{E}_{b_{t+1}}\left[R(s_t, a_t, s_{t+1})\right] \tag{35.5}$$

For small problems, we can solve the resulting augmented MDP optimally. However, in general this is computationally intractable. [Gha+15] surveys many methods to solve it more efficiently. For example, [KN09] develop an algorithm that behaves similarly to Bayes optimal policies, except in a provably small number of steps; [GSD13] propose an approximate method based on Monte Carlo rollouts. More recently, [Zin+20] propose an approximate method based on meta-learning (Section 19.6.4), in which they train a (model-free) policy for multiple related tasks. Each task is represented by a task embedding vector m, which is inferred from $\boldsymbol{h}_t$ using a VAE (Section 21.2). The posterior $p(m|\boldsymbol{h}_t)$ is used as a proxy for the belief state b_t, and the policy is trained to perform well given s_t and b_t. At test time, the policy is applied to the incrementally computed belief state; this allows the method to infer what kind of task this is, and then to use a pre-trained policy to quickly solve it.

35.2 Value-based RL

In this section, we assume the agent has access to samples from p_T and p_R by interacting with the environment. We will show how to use these samples to learn optimal Q-functions from which we can derive optimal policies.

35.2.1 Monte Carlo RL

Recall that $Q_\pi(s, a) = \mathbb{E}\left[G_t|s_t = s, a_t = a\right]$ for any t. A simple way to estimate this is to take action a, and then sample the rest of the trajectory according to π, and then compute the average sum of discounted rewards. The trajectory ends when we reach a terminal state, if the task is episodic, or when the discount factor γ^t becomes negligibly small, whichever occurs first. This is the **Monte Carlo estimation** of the value function.

We can use this technique together with policy iteration (Section 34.6.2) to learn an optimal policy. Specifically, at iteration k, we compute a new, improved policy using $\pi_{k+1}(s) = \text{argmax}_a\, Q_k(s, a)$, where Q_k is approximated using MC estimation. This update can be applied to all the states visited on the sampled trajectory. This overall technique is called **Monte Carlo control**.

To ensure this method converges to the optimal policy, we need to collect data for every (state, action) pair, at least in the tabular case, since there is no generalization across different values of $Q(s, a)$. One way to achieve this is to use an ϵ-greedy policy. Since this is an on-policy algorithm, the resulting method will converge to the optimal ϵ-soft policy, as opposed to the optimal policy. It is possible to use importance sampling to estimate the value function for the optimal policy, even if actions are chosen according to the ϵ-greedy policy. However, it is simpler to just gradually reduce ϵ.

35.2.2 Temporal difference (TD) learning

The Monte Carlo (MC) method in Section 35.2.1 results in an estimator for $Q_\pi(s, a)$ with very high variance, since it has to unroll many trajectories, whose returns are a sum of many random rewards generated by stochastic state transitions. In addition, it is limited to episodic tasks (or finite horizon

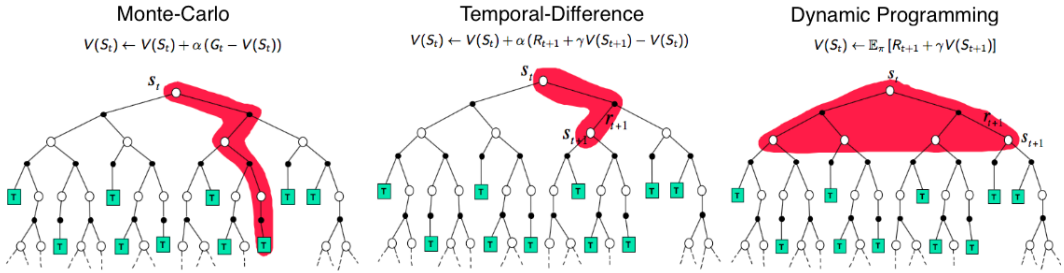


Figure 35.2: Backup diagrams of $V(s_t)$ for Monte Carlo, temporal difference, and dynamic programming updates of the state-value function. Used with kind permission of Andy Barto.

truncation of continuing tasks), since it must unroll to the end of the episode before each update step, to ensure it reliably estimates the long term return.

In this section, we discuss a more efficient technique called **temporal difference** or **TD** learning [Sut88]. The basic idea is to incrementally reduce the Bellman error for sampled states or state-actions, based on transitions instead of a long trajectory. More precisely, suppose we are to learn the value function V_π for a fixed policy π. Given a state transition (s, a, r, s') where $a \sim \pi(s)$, we change the estimate $V(s)$ so that it moves toward the bootstrapping target (Section 35.1.2)

$$V(s_t) \leftarrow V(s_t) + \eta\left[r_t + \gamma V(s_{t+1}) - V(s_t)\right] \tag{35.6}$$

where η is the learning rate. The term multiplied by η above is known as the **TD error**. A more general form of TD update for parametric value function representations is

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + \eta\left[r_t + \gamma V_{\boldsymbol{w}}(s_{t+1}) - V_{\boldsymbol{w}}(s_t)\right] \nabla_{\boldsymbol{w}} V_{\boldsymbol{w}}(s_t) \tag{35.7}$$

of which Equation (35.6) is a special case. The TD update rule for learning Q_π is similar.

It can be shown that TD learning in the tabular case, Equation (35.6), converges to the correct value function, under proper conditions [Ber19]. However, it may diverge when approximation is used (Equation (35.7)), an issue we will discuss further in Section 35.5.3.

The potential divergence of TD is also consistent with the fact that Equation (35.7) is not SGD (Section 6.3) on any objective function, despite a very similar form. Instead, it is an example of **bootstrapping**, in which the estimate, $V_{\boldsymbol{w}}(s_t)$, is updated to approach a target, $r_t + \gamma V_{\boldsymbol{w}}(s_{t+1})$, which is defined by the value function estimate itself. This idea is shared by DP methods like value iteration, although they rely on the complete MDP model to compute an exact Bellman backup. In contrast, TD learning can be viewed as using sampled transitions to approximate such backups. An example of non-bootstrapping approach is the Monte Carlo estimation in the previous section. It samples a complete trajectory, rather than individual transitions, to perform an update, and is often much less efficient. Figure 35.2 illustrates the difference between MC, TD, and DP.

35.2.3 TD learning with eligibility traces

A key difference between TD and MC is the way they estimate returns. Given a trajectory $\boldsymbol{\tau} = (s_0, a_0, r_0, s_1, \ldots, s_T)$, TD estimates the return from state s_t by one-step lookahead, $G_{t:t+1} = r_t +$

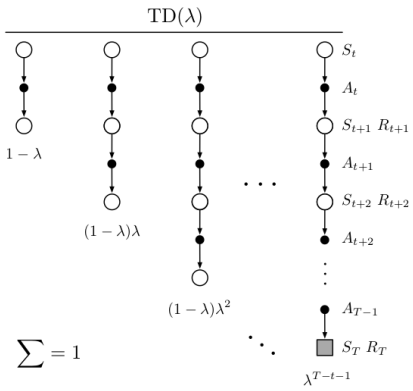


Figure 35.3: The backup diagram for TD(λ). Standard TD learning corresponds to $\lambda = 0$, and standard MC learning corresponds to $\lambda = 1$. From Figure 12.1 of [SB18]. Used with kind permission of Richard Sutton.

$\gamma V(s_{t+1})$, where the return from time $t+1$ is replaced by its value function estimate. In contrast, MC waits until the end of the episode or until T is large enough, then uses the estimate $G_{t:T} = r_t + \gamma r_{t+1} + \cdots + \gamma^{T-t-1} r_{T-1}$. It is possible to interpolate between these by performing an n-step rollout, and then using the value function to approximate the return for the rest of the trajectory, similar to heuristic search (Section 35.4.1.1). That is, we can use the n-step estimate

$$G_{t:t+n} = r_t + \gamma r_{t+1} + \cdots + \gamma^{n-1} r_{t+n-1} + \gamma^n V(s_{t+n}) \tag{35.8}$$

The corresponding n-step version of the TD update becomes

$$V(s_t) \leftarrow V(s_t) + \eta \left[G_{t:t+n} - V(s_t) \right] \tag{35.9}$$

Rather than picking a specific lookahead value, n, we can take a weighted average of all possible values, with a single parameter $\lambda \in [0, 1]$, by using

$$G_t^\lambda \triangleq (1-\lambda) \sum_{n=1}^{\infty} \lambda^{n-1} G_{t:t+n} \tag{35.10}$$

This is called the **λ-return**. The coefficient of $1 - \lambda = (1 + \lambda + \lambda^2 + \cdots)^{-1}$ in the front ensures this is a convex combination of n-step returns. See Figure 35.3 for an illustration.

An important benefit of using the geometric weighting in Equation (35.10) is that the corresponding TD learning update can be efficiently implemented, through the use of **eligibility traces**, even though G_t^λ is a sum of infinitely many terms. The method is called **TD(λ)**, and can be combined with many algorithms to be studied in the rest of the chapter. See [SB18] for a detailed discussion.

35.2.4 SARSA: on-policy TD control

TD learning is for policy evaluation, as it estimates the value function for a fixed policy. In order to find an optimal policy, we may use the algorithm as a building block inside generalized policy

iteration (Section 34.6.2). In this case, it is more convenient to work with the action-value function, Q, and a policy π that is greedy with respect to Q. The agent follows π in every step to choose actions, and upon a transition (s, a, r, s') the TD update rule is

$$Q(s,a) \leftarrow Q(s,a) + \eta \left[r + \gamma Q(s', a') - Q(s,a) \right] \tag{35.11}$$

where $a' \sim \pi(s')$ is the action the agent will take in state s'. After Q is updated (for policy evaluation), π also changes accordingly as it is greedy with respect to Q (for policy improvement). This algorithm, first proposed by [RN94], was further studied and renamed to **SARSA** by [Sut96]; the name comes from its update rule that involves an augmented transition (s, a, r, s', a').

In order for SARSA to converge to Q_*, every state-action pair must be visited infinitely often, at least in the tabular case, since the algorithm only updates $Q(s, a)$ for (s, a) that it visits. One way to ensure this condition is to use a "greedy in the limit with infinite exploration" (**GLIE**) policy. An example is the ϵ-greedy policy, with ϵ vanishing to 0 gradually. It can be shown that SARSA with a GLIE policy will converge to Q_* and π_* [Sin+00].

35.2.5 Q-learning: off-policy TD control

SARSA is an **on-policy** algorithm, which means it learns the Q-function for the policy it is currently using, which is typically not the optimal policy (except in the limit for a GLIE policy). However, with a simple modification, we can convert this to an **off-policy** algorithm that learns Q_*, even if a suboptimal policy is used to choose actions.

The idea is to replace the sampled next action $a' \sim \pi(s')$ in Equation (35.11) with a greedy action in s': $a' = \text{argmax}_b \, Q(s', b)$. This results in the following update when a transition (s, a, r, s') happens

$$Q(s,a) \leftarrow Q(s,a) + \eta \left[r + \gamma \max_b Q(s', b) - Q(s,a) \right] \tag{35.12}$$

This is the update rule of **Q-learning** for the tabular case [WD92]. The extension to work with function approximation can be done in a way similar to Equation (35.7). Since it is off-policy, the method can use (s, a, r, s') triples coming from any data source, such as older versions of the policy, or log data from an existing (non-RL) system. If every state-action pair is visited infinitely often, the algorithm provably converges to Q_* in the tabular case, with properly decayed learning rates [Ber19]. Algorithm 45 gives a vanilla implementation of Q-learning with ϵ-greedy exploration.

35.2.5.1 Example

Figure 35.4 gives an example of Q-learning applied to the simple 1d grid world from Figure 34.11, using $\gamma = 0.9$. We show the Q-functon at the start and end of each episode, after performing actions chosen by an ϵ-greedy policy. We initialize $Q(s, a) = 0$ for all entries, and use a step size of $\eta = 1$, so the update becomes $Q_*(s, a) = r + \gamma Q_*(s', a_*)$, where $a_* = \downarrow$ for all states.

35.2.5.2 Double Q-learning

Standard Q-learning suffers from a problem known as the **optimizer's curse** [SW06], or the **maximization bias**. The problem refers to the simple statistical inequality, $\mathbb{E}\left[\max_a X_a\right] \geq \max_a \mathbb{E}\left[X_a\right]$,

Algorithm 45: Q-learning with ϵ-greedy exploration

Initialize value function parameters $\boldsymbol{w}$
repeat
  Sample starting state s of new episode
  repeat
    Sample action $a = \begin{cases} \text{argmax}_b \, Q_{\boldsymbol{w}}(s,b), & \text{with probability } 1-\epsilon \\ \text{random action}, & \text{with probability } \epsilon \end{cases}$
    Observe state s', reward r
    Compute the TD error: $\delta = r + \gamma \max_{a'} Q_{\boldsymbol{w}}(s',a') - Q_{\boldsymbol{w}}(s,a)$
    $\boldsymbol{w} \leftarrow \boldsymbol{w} + \eta \delta \nabla_{\boldsymbol{w}} Q_{\boldsymbol{w}}(s,a)$
    $s \leftarrow s'$
  until *state s is terminal*
until *converged*

for a set of random variables $\{X_a\}$. Thus, if we pick actions greedily according to their random scores $\{X_a\}$, we might pick a wrong action just because random noise makes it appealing.

Figure 35.5 gives a simple example of how this can happen in an MDP. The start state is A. The right action gives a reward 0 and terminates the episode. The left action also gives a reward of 0, but then enters state B, from which there are many possible actions, with rewards drawn from $\mathcal{N}(-0.1, 1.0)$. Thus the expected return for any trajectory starting with the left action is -0.1, making it suboptimal. Nevertheless, the RL algorithm may pick the left action due to the maximization bias making B appear to have a positive value.

One solution to avoid the maximization bias is to use two separate Q-functions, Q_1 and Q_2, one for selecting the greedy action, and the other for estimating the corresponding Q-value. In particular, upon seeing a transition (s, a, r, s'), we perform the following update

$$Q_1(s,a) \leftarrow Q_1(s,a) + \eta \left[r + \gamma Q_2\big(s', \underset{a'}{\text{argmax}}\, Q_1(s',a')\big) - Q_1(s,a) \right] \tag{35.13}$$

and may repeat the same update but with the roles of Q_1 and Q_2 swapped. This technique is called **double Q-learning** [Has10]. Figure 35.5 shows the benefits of the algorithm over standard Q-learning in a toy problem.

35.2.6 Deep Q-network (DQN)

When function approximation is used, Q-learning may be hard to use in practice due to instability problems. Here, we will describe two important heuristics, popularized by the **deep Q-network** or **DQN** work [Mni+15], which was able to train agents to outperform humans at playing Atari games, using CNN-structured Q-networks.

The first technique, originally proposed in [Lin92], is to leverage an **experience replace** buffer, which stores the most recent (s, a, r, s') transition tuples. In contrast to standard Q-learning which updates the Q-function when a new transition occurs, the DQN agent also performs additional updates using transitions sampled from the buffer. This modification has two advantages. First, it

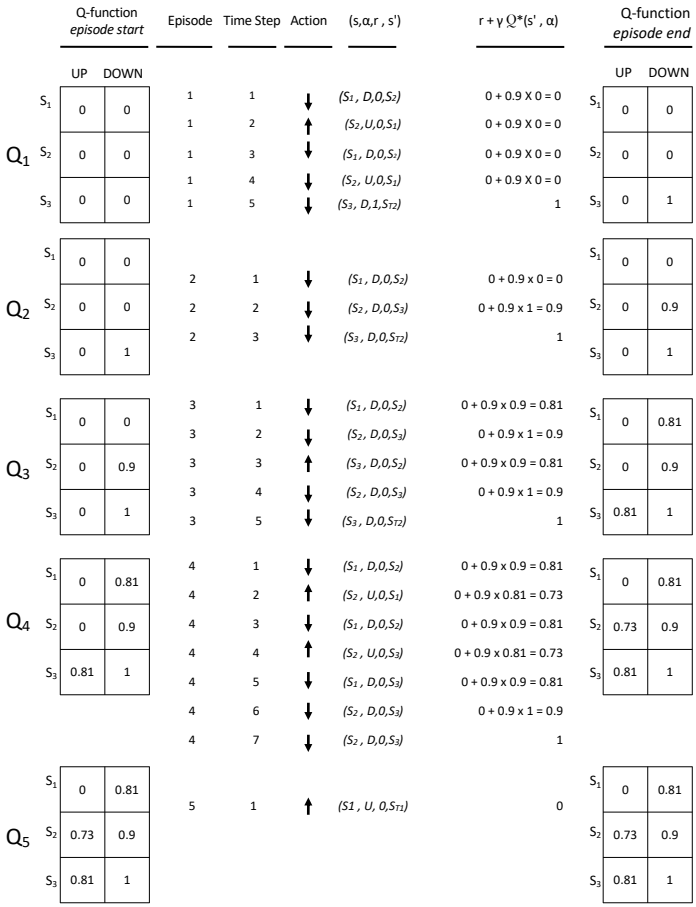


Figure 35.4: Illustration of Q learning for the 1d grid world in Figure 34.11 using ϵ-greedy exploration. At the end of episode 1, we make a transition from S_3 to S_{T2} and get a reward of $r = 1$, so we estimate $Q(S_3, \downarrow) = 1$. In episode 2, we make a transition from S_2 to S_3, so S_2 gets incremented by $\gamma Q(S_3, \downarrow) = 0.9$. Adapted from Figure 3.3 of [GK19].

improves data efficiency as every transition can be used multiple times. Second, it improves stability in training, by reducing the correlation of the data samples that the network is trained on.

The second idea to improve stability is to regress the Q-network to a "frozen" **target network** computed at an earlier iteration, rather than trying to chase a constantly moving target. Specifically, we maintain an extra, frozen copy of the Q-network, $Q_{\boldsymbol{w}^-}$, of the same structure as $Q_{\boldsymbol{w}}$. This new Q-network is to compute bootstrapping targets for training $Q_{\boldsymbol{w}}$, in which the loss function is

$$\mathcal{L}^{\text{DQN}}(\boldsymbol{w}) = \mathbb{E}_{(s,a,r,s')\sim U(\mathcal{D})}\left[\left(r + \gamma \max_{a'} Q_{\boldsymbol{w}^-}(s', a') - Q_{\boldsymbol{w}}(s, a)\right)^2\right] \tag{35.14}$$

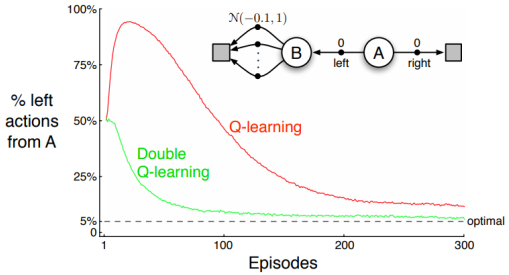


Figure 35.5: Comparison of Q-learning and double Q-learning on a simple episodic MDP using ϵ-greedy action selection with $\epsilon = 0.1$. The initial state is A, and squares denote absorbing states. The data are averaged over 10,000 runs. From Figure 6.5 of [SB18]. Used with kind permission of Rich Sutton.

where $U(\mathcal{D})$ is a uniform distribution over the replay buffer $\mathcal{D}$. We then periodically set $\boldsymbol{w}^- \leftarrow \boldsymbol{w}$, usually after a few episodes. This approach is an instance of **fitted value iteration** [SB18].

Various improvements to DQN have been proposed. One is **double DQN** [HGS16], which uses the double learning technique (Section 35.2.5.2) to remove the maximization bias. The second is to replace the uniform distribution in Equation (35.14) with one that favors more important transition tuples, resulting in the use of **prioritized experience replay** [Sch+16a]. For example, we can sample transitions from $\mathcal{D}$ with probability $p(s, a, r, s') \propto (|\delta| + \varepsilon)^\eta$, where δ is the corresponding TD error (under the current Q-function), $\varepsilon > 0$ a hyperparameter to ensure every experience is chosen with nonzero probability, and $\eta \geq 0$ controls the "inverse temperature" of the distribution (so $\eta = 0$ corresponds to uniform sampling). The third is to learn a value function $V_{\boldsymbol{w}}$ and an advantage function $A_{\boldsymbol{w}}$, with shared parameter $\boldsymbol{w}$, instead of learning $Q_{\boldsymbol{w}}$. The resulting **dueling DQN** [Wan+16] is shown to be more sample efficient, especially when there are many actions with similar Q-values.

The **rainbow** method [Hes+18] combines all three improvements, as well as others, including multi-step returns (Section 35.2.3), distributional RL [BDM17] (which predicts the distribution of returns, not just the expected return), and noisy nets [For+18b] (which adds random noise to the network weights to encourage exploration). It produces state-of-the-art results on the Atari benchmark.

35.3 Policy-based RL

In the previous section, we considered methods that estimate the action-value function, $Q(s, a)$, from which we derive a policy, which may be greedy or softmax. However, these methods have three main disadvantages: (1) they can be difficult to apply to continuous action spaces; (2) they may diverge if function approximation is used; and (3) the training of Q, often based on TD-style updates, is not directly related to the expected return garnered by the learned policy.

In this section, we discuss **policy search** methods, which directly optimize the parameters of the policy so as to maximize its expected return. However, we will see that these methods often benefit from estimating a value or advantage function to reduce the variance in the policy search process.

35.3.1 The policy gradient theorem

We start by defining the objective function for policy learning, and then derive its gradient. We consider the episodic case. A similar result can be derived for the continuing case with the average reward criterion [SB18, Sec 13.6].

We define the objective to be the expected return of a policy, which we aim to maximize:

$$J(\pi) \triangleq \mathbb{E}_{p_0,\pi}\left[G_0\right] = \mathbb{E}_{p_0(s_0)}\left[V_\pi(s_0)\right] = \mathbb{E}_{p_0(s_0)\pi(a_0|s_0)}\left[Q_\pi(s_0,a_0)\right] \tag{35.15}$$

We consider policies $\pi_{\boldsymbol{\theta}}$ parameterized by $\boldsymbol{\theta}$, and compute the gradient of Equation (35.15) wrt $\boldsymbol{\theta}$:

$$\nabla_{\boldsymbol{\theta}} J(\pi_{\boldsymbol{\theta}}) = \mathbb{E}_{p_0(s_0)}\left[\nabla_{\boldsymbol{\theta}}\left(\sum_{a_0}\pi_{\boldsymbol{\theta}}(a_0|s_0)Q_{\pi_{\boldsymbol{\theta}}}(s_0,a_0)\right)\right] \tag{35.16}$$

$$= \mathbb{E}_{p_0(s_0)}\left[\sum_{a_0}\nabla\pi_{\boldsymbol{\theta}}(a_0|s_0)Q_{\pi_{\boldsymbol{\theta}}}(s_0,a_0)\right] + \mathbb{E}_{p_0(s_0)\pi_{\boldsymbol{\theta}}(a_0|s_0)}\left[\nabla_{\boldsymbol{\theta}} Q_{\pi_{\boldsymbol{\theta}}}(s_0,a_0)\right] \tag{35.17}$$

Now we calculate the term $\nabla_{\boldsymbol{\theta}} Q_{\pi_{\boldsymbol{\theta}}}(s_0,a_0)$:

$$\nabla_{\boldsymbol{\theta}} Q_{\pi_{\boldsymbol{\theta}}}(s_0,a_0) = \nabla_{\boldsymbol{\theta}}\left[R(s_0,a_0) + \gamma\mathbb{E}_{p_T(s_1|s_0,a_0)}\left[V_{\pi_{\boldsymbol{\theta}}}(s_1)\right]\right] = \gamma\nabla_{\boldsymbol{\theta}}\mathbb{E}_{p_T(s_1|s_0,a_0)}\left[V_{\pi_{\boldsymbol{\theta}}}(s_1)\right] \tag{35.18}$$

The right-hand side above is in a form similar to $\nabla_{\boldsymbol{\theta}} J(\pi_{\boldsymbol{\theta}})$. Repeating the same steps as before gives

$$\nabla_{\boldsymbol{\theta}} J(\pi_{\boldsymbol{\theta}}) = \sum_{t=0}^{\infty}\gamma^t\mathbb{E}_{p_t(s)}\left[\sum_a \nabla_{\boldsymbol{\theta}}\pi_{\boldsymbol{\theta}}(a|s)Q_{\pi_{\boldsymbol{\theta}}}(s,a)\right] \tag{35.19}$$

$$= \frac{1}{1-\gamma}\mathbb{E}_{p^\infty_{\pi_{\boldsymbol{\theta}}}(s)}\left[\sum_a \nabla_{\boldsymbol{\theta}}\pi_{\boldsymbol{\theta}}(a|s)Q_{\pi_{\boldsymbol{\theta}}}(s,a)\right] \tag{35.20}$$

$$= \frac{1}{1-\gamma}\mathbb{E}_{p^\infty_{\pi_{\boldsymbol{\theta}}}(s)\pi_{\boldsymbol{\theta}}(a|s)}\left[\nabla_{\boldsymbol{\theta}}\log\pi_{\boldsymbol{\theta}}(a|s)\; Q_{\pi_{\boldsymbol{\theta}}}(s,a)\right] \tag{35.21}$$

where $p_t(s)$ is the probability of visiting s in time t if we start with $s_0 \sim p_0$ and follow $\pi_{\boldsymbol{\theta}}$, and $p^\infty_{\pi_{\boldsymbol{\theta}}}(s) = (1-\gamma)\sum_{t=0}^{\infty}\gamma^t p_t(s)$ is the normalized discounted state visitation distribution. Equation (35.21) is known as the **policy gradient theorem** [Sut+99].

In practice, estimating the policy gradient using Equation (35.21) can have a high variance. A baseline $b(s)$ can be used for variance reduction (Section 6.5.3.1):

$$\nabla_{\boldsymbol{\theta}} J(\pi_{\boldsymbol{\theta}}) = \frac{1}{1-\gamma}\mathbb{E}_{p^\infty_{\pi_{\boldsymbol{\theta}}}(s)\pi_{\boldsymbol{\theta}}(a|s)}\left[\nabla_{\boldsymbol{\theta}}\log\pi_{\boldsymbol{\theta}}(a|s)(Q_{\pi_{\boldsymbol{\theta}}}(s,a) - b(s))\right] \tag{35.22}$$

A common choice for the baseline is $b(s) = V_{\pi_{\boldsymbol{\theta}}}(s)$. We will discuss how to estimate it below.

35.3.2 REINFORCE

One way to apply the policy gradient theorem to optimize a policy is to use stochastic gradient ascent. Suppose $\boldsymbol{\tau} = (s_0, a_0, r_0, s_1, \ldots, s_T)$ is a trajectory with $s_0 \sim p_0$ and $\pi_{\boldsymbol{\theta}}$. Then,

$$\nabla_{\boldsymbol{\theta}} J(\pi_{\boldsymbol{\theta}}) = \frac{1}{1-\gamma} \mathbb{E}_{p^{\infty}_{\pi_{\boldsymbol{\theta}}}(s)\pi_{\boldsymbol{\theta}}(a|s)} \left[\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a|s) Q_{\pi_{\boldsymbol{\theta}}}(s,a)\right] \tag{35.23}$$

$$\approx \sum_{t=0}^{T-1} \gamma^t G_t \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t|s_t) \tag{35.24}$$

where the return G_t is defined in Equation (34.64), and the factor γ^t is due to the definition of $p^{\infty}_{\pi_{\boldsymbol{\theta}}}$ where the state at time t is discounted.

We can use a baseline in the gradient estimate to get the following update rule:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta \sum_{t=0}^{T-1} \gamma^t (G_t - b(s_t)) \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t|s_t) \tag{35.25}$$

This is called the **REINFORCE** algorithm [Wil92].[3] The udpate equation can be interepreted as follows: we compute the sum of discounted future rewards induced by a trajectory, compared to a baseline, and if this is positive, we increase $\boldsymbol{\theta}$ so as to make this trajectory more likely, otherwise we decrease $\boldsymbol{\theta}$. Thus, we reinforce good behaviors, and reduce the chances of generating bad ones.

We can use a constant (state-independent) baseline, or we can use a state-dependent baseline, $b(s_t)$ to further lower the variance. A natural choice is to use an estimated value function, $V_{\boldsymbol{w}}(s)$, which can be learned, e.g., with MC. Algorithm 46 gives the pseudo code where stochastic gradient updates are used with separate learning rates.

Algorithm 46: REINFORCE with value function baseline

Initialize policy parameters $\boldsymbol{\theta}$, baseline parameters $\boldsymbol{w}$
repeat
  Sample an episode $\boldsymbol{\tau} = (s_0, a_0, r_0, s_1, \ldots, s_T)$ using $\pi_{\boldsymbol{\theta}}$
  Compute G_t for all $t \in \{0, 1, \ldots, T-1\}$ using Equation (34.64)
  for $t = 0, 1, \ldots, T-1$ **do**
    $\delta = G_t - V_{\boldsymbol{w}}(s_t)$ // scalar error
    $\boldsymbol{w} \leftarrow \boldsymbol{w} + \eta_{\boldsymbol{w}} \delta \nabla_{\boldsymbol{w}} V_{\boldsymbol{w}}(s_t)$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta_{\boldsymbol{\theta}} \gamma^t \delta \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t|s_t)$
until *converged*

35.3.3 Actor-critic methods

An **actor-critic** method [BSA83] uses the policy gradient method, but where the expected return is estimated using temporal difference learning of a value function instead of MC rollouts. The term

3. The term "REINFORCE" is an acronym for "REward Increment = nonnegative Factor x Offset Reinforcement x Characteristic Eligibility". The phrase "Characteristic eligibility" refers to the $\nabla \log \pi_{\boldsymbol{\theta}}(a_t|s_t)$ term; the phrase "offset reinforcement" refers to the $G_t - b(s_t)$ term; and the phrase "nonnegative factor" refers to the learning rate η of SGD.

"actor" refers to the policy, and the term "critic" refers to the value function. The use of bootstrapping in TD updates allows more efficient learning of the value function compared to MC. In addition, it allows us to develop a fully online, incremental algorithm, that does not need to wait until the end of the trajectory before updating the parameters (as in Algorithm 46).

Concretely, consider the use of the one-step TD(0) method to estimate the return in the episodic csae, i.e., we replace G_t with $G_{t:t+1} = r_t + \gamma V_{\boldsymbol{w}}(s_{t+1})$. If we use $V_{\boldsymbol{w}}(s_t)$ as a baseline, the REINFORCE update in Equation (35.25) becomes

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta \sum_{t=0}^{T-1} \gamma^t \left(G_{t:t+1} - V_{\boldsymbol{w}}(s_t)\right) \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t|s_t) \tag{35.26}$$

$$= \boldsymbol{\theta} + \eta \sum_{t=0}^{T-1} \gamma^t \big(r_t + \gamma V_{\boldsymbol{w}}(s_{t+1}) - V_{\boldsymbol{w}}(s_t)\big) \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t|s_t) \tag{35.27}$$

35.3.3.1 A2C and A3C

Note that $r_{t+1} + \gamma V_{\boldsymbol{w}}(s_{t+1}) - V_{\boldsymbol{w}}(s_t)$ is a single sample approximation to the advantage function $A(s_t, a_t) = Q(s_t, a_t) - V(s_t)$. This method is therefore called **advantage actor critic** or **A2C** (Algorithm 47). If we run the actors in parallel and asychronously update their shared parameters, the method is called **asychrononous advantage actor critic** or **A3C** [Mni+16].

Algorithm 47: Advantage actor critic (A2C) algorithm

Initialize actor parameters $\boldsymbol{\theta}$, critic parameters $\boldsymbol{w}$
repeat
  Sample starting state s_0 of a new episode
  for $t = 0, 1, 2, \ldots$ **do**
    Sample action $a_t \sim \pi_{\boldsymbol{\theta}}(\cdot|s_t)$
    Observe next state s_{t+1} and reward r_t
    $\delta = r_t + \gamma V_{\boldsymbol{w}}(s_{t+1}) - V_{\boldsymbol{w}}(s_t)$
    $\boldsymbol{w} \leftarrow \boldsymbol{w} + \eta_{\boldsymbol{w}} \delta \nabla_{\boldsymbol{w}} V_{\boldsymbol{w}}(s_t)$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta_{\boldsymbol{\theta}} \gamma^t \delta \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t|s_t)$
until *converged*

35.3.3.2 Eligibility traces

In A2C, we use a single step rollout, and then use the value function in order to approximate the expected return for the trajectory. More generally, we can use the n-step estimate

$$G_{t:t+n} = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots + \gamma^{n-1} r_{t+n-1} + \gamma^n V_{\boldsymbol{w}}(s_{t+n}) \tag{35.28}$$

and obtain an n-step advantage estimate as follows:

$$A^{(n)}_{\pi_{\boldsymbol{\theta}}}(s_t, a_t) = G_{t:t+n} - V_{\boldsymbol{w}}(s_t) \tag{35.29}$$

The n steps of actual rewards are an unbiased sample, but have high variance. By contrast, $V_{\boldsymbol{w}}(s_{t+n+1})$ has lower variance, but is biased. By changing n, we can control the bias-variance tradeoff. Instead of using a single value of n, we can take an weighted average, with weight proportional to λ^n for $A^{(n)}_{\pi_{\boldsymbol{\theta}}}(s_t, a_t)$, as in TD($\lambda$). The average can be shown to be equivalent to

$$A^{(\lambda)}_{\pi_{\boldsymbol{\theta}}}(s_t, a_t) \triangleq \sum_{\ell=0}^{\infty} (\gamma\lambda)^{\ell} \delta_{t+l} \tag{35.30}$$

where $\delta_t = r_t + \gamma V_{\boldsymbol{w}}(s_{t+1}) - V_{\boldsymbol{w}}(s_t)$ is the TD error at time t. Here, $\lambda \in [0, 1]$ is a parameter that controls the bias-variance tradeoff: larger values decrease the bias but increase the variance, as in TD(λ). We can implement Equation (35.30) efficiently using eligibility traces, as shown in Algorithm 48, as an example of **generalized advantage estimation** (**GAE**) [Sch+16b]. See [SB18, Ch.12] for further details.

Algorithm 48: Actor critic with eligibility traces

Initialize actor parameters $\boldsymbol{\theta}$, critic parameters $\boldsymbol{w}$
repeat
  Initialize eligibility trace vectors: $\boldsymbol{z}_{\boldsymbol{\theta}} \leftarrow \mathbf{0}$, $\boldsymbol{z}_{\boldsymbol{w}} \leftarrow \mathbf{0}$
  Sample starting state s_0 of a new episode
  for $t = 0, 1, 2, \ldots$ **do**
    Sample action $a_t \sim \pi_{\boldsymbol{\theta}}(\cdot|s_t)$
    Observe state s_{t+1} and reward r_t
    Compute the TD error: $\delta = r_t + \gamma V_{\boldsymbol{w}}(s_{t+1}) - V_{\boldsymbol{w}}(s_t)$
    $\boldsymbol{z}_{\boldsymbol{w}} \leftarrow \gamma\lambda_{\boldsymbol{w}} \boldsymbol{z}_{\boldsymbol{w}} + \nabla_{\boldsymbol{w}} V_{\boldsymbol{w}}(s)$
    $\boldsymbol{z}_{\boldsymbol{\theta}} \leftarrow \gamma\lambda_{\boldsymbol{\theta}} \boldsymbol{z}_{\boldsymbol{\theta}} + \gamma^t \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t|s_t)$
    $\boldsymbol{w} \leftarrow \boldsymbol{w} + \eta_{\boldsymbol{w}} \delta \boldsymbol{z}_{\boldsymbol{w}}$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta_{\boldsymbol{\theta}} \delta \boldsymbol{z}_{\boldsymbol{\theta}}$
until *converged*

35.3.4 Bound optimization methods

In policy gradient methods, the objective $J(\boldsymbol{\theta})$ does not necessarily increase monotonically, but rather can collapse especially if the learning rate is not small enough. We now describe methods that guarantee monotonic improvement, similar to bound optimization algorithms (Section 6.6).

We start with a useful fact that relate the policy values of two arbitrary policies [KL02]:

$$J(\pi') - J(\pi) = \frac{1}{1-\gamma} \mathbb{E}_{p^{\infty}_{\pi'}(s)} \left[\mathbb{E}_{\pi'(a|s)} \left[A_{\pi}(s, a) \right] \right] \tag{35.31}$$

where π can be interpreted as the current policy during policy optimization, and π' a candidate new policy (such as the greedy policy wrt Q_{π}). As in the policy improvement theorem (Section 34.6.2), if $\mathbb{E}_{\pi'(a|s)} \left[A_{\pi}(s, a) \right] \geq 0$ for all s, then $J(\pi') \geq J(\pi)$. However, we cannot ensure this condition to hold when function approximation is used, as such a uniformly improving policy π' may not be

representable by our parametric family, $\{\pi_{\boldsymbol{\theta}}\}_{\boldsymbol{\theta}\in\Theta}$. Therefore, nonnegativity of Equation (35.31) is not easy to ensure, when we do not have a direct way to sample states from $p^{\infty}_{\pi'}$.

One way to ensure monotonic improvement of J is to improve the policy conservatively. Define $\pi_\theta = \theta\pi' + (1-\theta)\pi$ for $\theta \in [0,1]$. It follows from the policy gradient theorem (Equation (35.21), with $\boldsymbol{\theta} = [\theta]$) that $J(\pi_\theta) - J(\pi) = \theta L(\pi') + O(\theta^2)$, where

$$L(\pi') \triangleq \frac{1}{1-\gamma}\mathbb{E}_{p^{\infty}_{\pi}(s)}\left[\mathbb{E}_{\pi'(a|s)}\left[A_\pi(s,a)\right]\right] = \frac{1}{1-\gamma}\mathbb{E}_{p^{\infty}_{\pi}(s)\pi(a|s)}\left[\frac{\pi'(a|s)}{\pi(a|s)}A_\pi(s,a)\right] \tag{35.32}$$

In the above, we have switched the state distribution from $p^{\infty}_{\pi'}$ in Equation (35.31) to p^{∞}_{π}, while at the same time introducing a higher order residual term of $O(\theta^2)$. The linear term, $\theta L(\pi')$, can be estimated and optimized based on episodes sampled by π. The higher order term can be bounded in various ways, resulting in different lower bounds of $J(\pi_\theta) - J(\pi)$. We can then optimize θ to make sure this lower bound is positive, which would imply $J(\pi_\theta) - J(\pi) > 0$. In **conservative policy iteration** [KL02], the following (slightly simplified) lower bound is used

$$J^{\mathrm{CPI}}(\pi_\theta) \triangleq J(\pi) + \theta L(\pi') - \frac{2\varepsilon\gamma}{(1-\gamma)^2}\theta^2 \tag{35.33}$$

where $\varepsilon = \max_s \left|\mathbb{E}_{\pi'(a|s)}\left[A_\pi(s,a)\right]\right|$.

This idea can be generalized to policies beyond those in the form of π_θ, where the condition of a small enough θ is replaced by a small enough divergence between π' and π. In **safe policy iteration** [Pir+13], the divergence is the maximum total variation, while in **trust region policy optimization** (**TRPO**) [Sch+15b], the divergence is the maximum KL-divergence. In the latter case, π' may be found by optimizing the following lower bound

$$J^{\mathrm{TRPO}}(\pi') \triangleq J(\pi) + L(\pi') - \frac{\varepsilon\gamma}{(1-\gamma)^2}\max_s D_{\mathbb{KL}}\left(\pi(s) \parallel \pi'(s)\right) \tag{35.34}$$

where $\varepsilon = \max_{s,a}|A_\pi(s,a)|$.

In practice, the above update rule can be overly conservative, and approximations are used. [Sch+15b] propose a version that implements two ideas: one is to replace the point-wise maximum KL-divergence by some average KL-divergence (usually averaged over $p^{\infty}_{\pi_{\boldsymbol{\theta}}}$); the second is to maximize the first two terms in Equation (35.34), with π' lying in a KL-ball centered at π. That is, we solve

$$\operatorname*{argmax}_{\pi'} L(\pi') \;\; \text{s.t.} \;\; \mathbb{E}_{p^{\infty}_{\pi}(s)}\left[D_{\mathbb{KL}}\left(\pi(s) \parallel \pi'(s)\right)\right] \le \delta \tag{35.35}$$

for some threshold $\delta > 0$.

In Section 6.4.2.1, we show that the trust region method, using a KL penalty at each step, is equivalent to natural gradient descent (see e.g., [Kak02; PS08b]). This is important, because a step of size η in parameter space does not always correspond to a step of size η in the policy space:

$$d_{\boldsymbol{\theta}}(\boldsymbol{\theta}_1, \boldsymbol{\theta}_2) = d_{\boldsymbol{\theta}}(\boldsymbol{\theta}_2, \boldsymbol{\theta}_3) \not\Rightarrow d_\pi(\pi_{\boldsymbol{\theta}_1}, \pi_{\boldsymbol{\theta}_2}) = d_\pi(\pi_{\boldsymbol{\theta}_2}, \pi_{\boldsymbol{\theta}_3}) \tag{35.36}$$

where $d_{\boldsymbol{\theta}}(\boldsymbol{\theta}_1, \boldsymbol{\theta}_2) = \|\boldsymbol{\theta}_1 - \boldsymbol{\theta}_2\|$ is the Euclidean distance, and $d_\pi(\pi_1, \pi_2) = D_{\mathbb{KL}}\left(\pi_1 \parallel \pi_2\right)$ the KL distance. In other words, the effect on the policy of any given change to the parameters depends on where we are in parameter space. This is taken into account by the natural gradient method,

resulting in faster and more robust optimization. The natural policy gradient can be approximated using the KFAC method (Section 6.4.4), as done in [Wu+17].

Other than TRPO, another approach inspired by Equation (35.34) is to use the KL-divergence as a penalty term, replacing the factor $2\varepsilon\gamma/(1-\gamma)^2$ by a tuning parameter. However, it often works better, and is simpler, by using the following clipped objective, which results in the **proximal policy optimization** or **PPO** method [Sch+17]:

$$J^{\text{PPO}}(\pi') \triangleq \frac{1}{1-\gamma}\mathbb{E}_{p_\pi^\infty(s)\pi(a|s)}\left[\kappa_\epsilon\left(\frac{\pi'(a|s)}{\pi(a|s)}\right)A_\pi(s,a)\right] \tag{35.37}$$

where $\kappa_\epsilon(x) \triangleq \text{clip}(x, 1-\epsilon, 1+\epsilon)$ ensures $|\kappa(x)-1| \leq \epsilon$. This method can be modified to ensure monotonic improvement as discussed in [WHT19], making it a true bound optimization method.

35.3.5 Deterministic policy gradient methods

In this section, we consider the case of a deterministic policy, that predicts a unique action for each state, so $a_t = \mu_{\boldsymbol{\theta}}(s_t)$, rather than $a_t \sim \pi_{\boldsymbol{\theta}}(s_t)$. We assume the states and actions are continuous, and define the objective as

$$J(\mu_{\boldsymbol{\theta}}) \triangleq \frac{1}{1-\gamma}\mathbb{E}_{p_{\mu_{\boldsymbol{\theta}}}^\infty(s)}\left[R(s,\mu_{\boldsymbol{\theta}}(s))\right] \tag{35.38}$$

The **deterministic policy gradient theorem** [Sil+14] provides a way to compute the gradient:

$$\nabla_{\boldsymbol{\theta}} J(\mu_{\boldsymbol{\theta}}) = \frac{1}{1-\gamma}\mathbb{E}_{p_{\mu_{\boldsymbol{\theta}}}^\infty(s)}\left[\nabla_{\boldsymbol{\theta}} Q_{\mu_{\boldsymbol{\theta}}}(s,\mu_{\boldsymbol{\theta}}(s))\right] \tag{35.39}$$

$$= \frac{1}{1-\gamma}\mathbb{E}_{p_{\mu_{\boldsymbol{\theta}}}^\infty(s)}\left[\nabla_{\boldsymbol{\theta}}\mu_{\boldsymbol{\theta}}(s)\nabla_a Q_{\mu_{\boldsymbol{\theta}}}(s,a)|_{a=\mu_{\boldsymbol{\theta}}(s)}\right] \tag{35.40}$$

where $\nabla_{\boldsymbol{\theta}}\mu_{\boldsymbol{\theta}}(s)$ is the $M \times N$ Jacobian matrix, and M and N are the dimensions of $\mathcal{A}$ and $\boldsymbol{\theta}$, respectively. For stochastic policies of the form $\pi_{\boldsymbol{\theta}}(a|s) = \mu_{\boldsymbol{\theta}}(s) + \text{noise}$, the standard policy gradient theorem reduces to the above form as the noise level goes to zero.

Note that the gradient estimate in Equation (35.40) integrates over the states but not over the actions, which helps reduce the variance in gradient estimation from sampled trajectories. However, since the deterministic policy does not do any exploration, we need to use an off-policy method, that collects data from a stochastic behavior policy β, whose stationary state distribution is p_β^∞. The original objective, $J(\mu_{\boldsymbol{\theta}})$, is approximated by the following:

$$J_b(\mu_{\boldsymbol{\theta}}) \triangleq \mathbb{E}_{p_\beta^\infty(s)}\left[V_{\mu_{\boldsymbol{\theta}}}(s)\right] = \mathbb{E}_{p_\beta^\infty(s)}\left[Q_{\mu_{\boldsymbol{\theta}}}(s,\mu_{\boldsymbol{\theta}}(s))\right] \tag{35.41}$$

with the off-policy deterministic policy gradient approximated by (see also Section 35.5.1.2)

$$\nabla_{\boldsymbol{\theta}} J_b(\mu_{\boldsymbol{\theta}}) \approx \mathbb{E}_{p_\beta^\infty(s)}\left[\nabla_{\boldsymbol{\theta}}\left[Q_{\mu_{\boldsymbol{\theta}}}(s,\mu_{\boldsymbol{\theta}}(s))\right]\right] = \mathbb{E}_{p_\beta^\infty(s)}\left[\nabla_{\boldsymbol{\theta}}\mu_{\boldsymbol{\theta}}(s)\nabla_a Q_{\mu_{\boldsymbol{\theta}}}(s,a)|_{a=\mu_{\boldsymbol{\theta}}(s)} ds\right] \tag{35.42}$$

where we have a dropped a term that depends on $\nabla_{\boldsymbol{\theta}} Q_{\mu_{\boldsymbol{\theta}}}(s,a)$ and is hard to estimate [Sil+14].

To apply Equation (35.42), we may learn $Q_{\boldsymbol{w}} \approx Q_{\mu_{\boldsymbol{\theta}}}$ with TD, giving rise to the following updates:

$$\delta = r_t + \gamma Q_{\boldsymbol{w}}(s_{t+1}, \mu_{\boldsymbol{\theta}}(s_{t+1})) - Q_{\boldsymbol{w}}(s_t, a_t) \tag{35.43}$$

$$\boldsymbol{w}_{t+1} \leftarrow \boldsymbol{w}_t + \eta_{\boldsymbol{w}}\delta\nabla_{\boldsymbol{w}} Q_{\boldsymbol{w}}(s_t, a_t) \tag{35.44}$$

$$\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t + \eta_{\boldsymbol{\theta}}\nabla_{\boldsymbol{\theta}}\mu_{\boldsymbol{\theta}}(s_t)\nabla_a Q_{\boldsymbol{w}}(s_t,a)|_{a=\mu_{\boldsymbol{\theta}}(s_t)} \tag{35.45}$$

This avoids importance sampling in the actor update because of the deterministic policy gradient, and avoids importance sampling in the critic update because of the use of Q-learning.

If $Q_{\boldsymbol{w}}$ is linear in $\boldsymbol{w}$, and uses features of the form $\boldsymbol{\phi}(s,a) = \boldsymbol{a}^{\mathsf{T}} \nabla_{\boldsymbol{\theta}} \mu_{\boldsymbol{\theta}}(s)$, where $\boldsymbol{a}$ is the vector representation of a, then we say the function approximator for the critic is **compatible** with the actor; in this case, one can show that the above approximation does not bias the overall gradient. The method has been extended in various ways. The **DDPG** algorithm of [Lil+16], which stands for "deep deterministic policy gradient", uses the DQN method (Section 35.2.6) to update Q that is represented by deep neural networks. The **TD3** algorithm [FHM18], which stands for "twin delayed DDPG", extends DDPG by using double DQN (Section 35.2.5.2) and other heuristics to further improve performance. Finally, the **D4PG** algorithm [BM+18], which stands for "distributed distributional DDPG", extends DDPG to handle distributed training, and to handle **distributional RL** (i.e., working with distributions of rewards instead of expected rewards [BDM17]).

35.3.6 Gradient-free methods

The policy gradient estimator computes a "zeroth order" gradient, which essentially evaluates the function with randomly sampled trajectories. Sometimes it can be more efficient to use a derivative-free optimizer (Section 6.9), that does not even attempt to estimate the gradient. For example, [MGR18] obtain good results by training linear policies with random search, and [Sal+17b] show how to use evolutionary strategies to optimize the policy of a robotic controller.

35.4 Model-based RL

Model-free approaches to RL typically need a lot of interactions with the environment to achieve good performance. For example, state of the art methods for the Atari benchmark, such as rainbow (Section 35.2.6), use millions of frames, equivalent to many days of playing at the standard frame rate. By contrast, humans can achieve the same performance in minutes [Tsi+17]. Similarly, OpenAI's robot hand controller [And+20] learns to manipulate a cube using 100 years of simulated data.

One promising approach to greater sample efficiency is **model-based RL** (**MBRL**). In this approach, we first learn the transition model and reward function, $p_T(s'|s,a)$ and $R(s,a)$, then use them to compute a near-optimal policy. This approach can significantly reduce the amount of real-world data that the agent needs to collect, since it can "try things out" in its imagination (i.e., the models), rather than having to try them out empirically.

There are several ways we can use a model, and many different kinds of model we can create. Some of the algorithms mentioned earlier, such as MBIE and UCLR2 for provably efficient exploration (Section 35.1.5.3), are examples of model-based methods. MBRL also provides a natural connection between RL and planning (Section 34.6) [Sut90]. We discuss some examples in the sections below, and refer to [MBJ20; PKP21; MH20] for more detailed reviews.

35.4.1 Model predictive control (MPC)

So far in this chapter, we have focused on trying to learn an optimal policy $\pi_*(s)$, which can then be used at run time to quickly pick the best action for any given state s. However, we can also avoid performing all this work in advance, and wait until we know what state we are in, call it s_t, and then use a model to predict future states and rewards that might follow for each possible sequence of

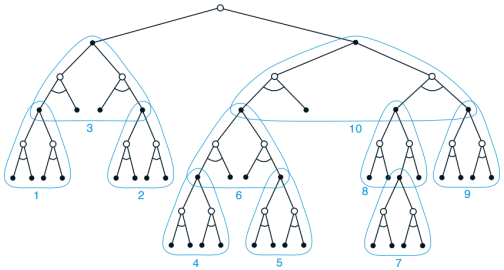


Figure 35.6: Illustration of heuristic search. In this figure, the subtrees are ordered according to a depth-first search procedure. From Figure 8.9 of [SB18]. Used with kind permission of Richard Sutton.

future actions we might pursue. We then take the action that looks most promising, and repeat the process at the next step. More precisely, we compute

$$\boldsymbol{a}^*_{t:t+H-1} = \underset{\boldsymbol{a}_{t:t+H-1}}{\operatorname{argmax}} \, \mathbb{E}\left[\sum_{h=0}^{H-1} R(s_{t+h}, a_{t+h}) + \hat{V}(s_{t+H})\right] \tag{35.46}$$

where the expectation is over state sequences that might result from executing $\boldsymbol{a}_{t:t+H-1}$ from state s_t. Here, H is called the **planning horizon**, and $\hat{V}(s_{t+H})$ is an estimate of the reward-to-go at the end of this H-step look-ahead process. This is known as **receeding horizon control** or **model predictive control** (**MPC**) [MM90; CA13]. We discuss some special cases of this below.

35.4.1.1 Heuristic search

If the state and action spaces are finite, we can solve Equation (35.46) exactly, although the time complexity will typically be exponential in H. However, in many situations, we can prune off unpromising trajectories, thus making the approach feasible in large scale problems.

In particular, consider a discrete, deterministic MDP where reward maximization corresponds to finding a shortest path to a goal state. We can expand the successors of the current state according to all possible actions, trying to find the goal state. Since the search tree grows exponentially with depth, we can use a **heuristic function** to prioritize which nodes to expand; this is called **best-first search**, as illustrated in Figure 35.6.

If the heuristic function is an optimistic lower bound on the true distance to the goal, it is called **admissable**; If we aim to maximize total rewards, admissibility means the heuristic function is an upper bound of the true value function. Admissibility ensures we will never incorrectly prune off parts of the search space. In this case, the resulting algorithm is known as A^* **search**, and is optimal. For more details on classical AI **heuristic search** methods, see [Pea84; RN19].

35.4.1.2 Monte-Carlo tree search (MCTS)

Monte-Carlo tree search or **MCTS** is similar to heuristic search, but learns a value function for each encountered state, rather than relying on a manually designed heuristic (see e.g., [Mun14] for

details). MCTSis inspired by UCB for bandits (Section 34.4.5), but applies to general sequential decision making problems including MDPs [KS06].

The MCTS method forms the basis of the famous **AlphaGo** and **AlphaZero** programs [Sil+16; Sil+18], which can play expert-level Go, chess and shogi (Japanese chess), using a known model of the environment. The **MuZero** method of [Sch+20] and the **Stochastic MuZero** method of [Ant+22] extend this to the case where the world model is also learned. The action-value functions for the intermediate nodes in the search tree are represented by deep neural networks, and updated using temporal difference methods that we discuss in Section 35.2. MCTS can also be applied to many other kinds of seqential decision problems, such as experiment design for sequentially creating molecules [SPW18].

35.4.1.3 Trajectory optimization for continuous actions

For continuous actions, we cannot enumerate all possible branches in the search tree. Instead, Equation (35.46) can be viewed as a nonlinear program, where $\boldsymbol{a}_{t:t+H-1}$ are the real-valued variables to be optimized. If the system dynamics are linear and the reward function corresponds to negative quadratic cost, the optimal action sequence can be solved mathematically, as in the **linear-quadratic-Gaussian** (**LQG**) controller (see e.g., [AM89; HR17]). However, this problem is hard in general and often solved by numerical methods such as **shooting** and **collocation** [Die+07; Rao10; Kal+11]. Many of them work in an iterative fashion, starting with an initial action sequence followed by a step to improve it. This process repeats until convergence of the cost.

An example is **differential dynamic programming** (**DDP**) [JM70; TL05]. In each iteration, DDP starts with a reference trajectory, and linearizes the system dynamics around states on the trajectory to form a locally quadratic approximation of the reward function. This system can be solved using LQG, whose optimal solution results in a new trajectory. The algorithm then moves to the next iteration, with the new trajectory as the reference trajectory.

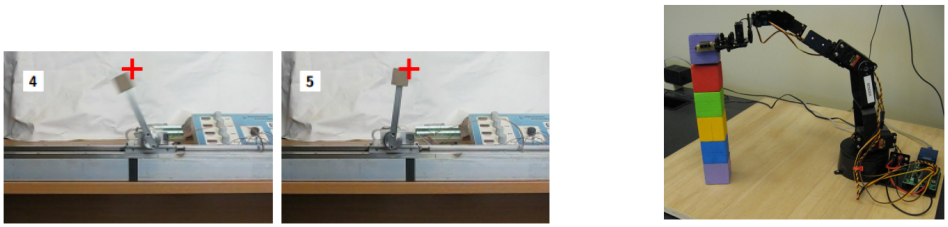
Other alternatives are possible, including black-box (gradient-free) optimization methods like the cross entropy method. (see Section 6.9.5).

35.4.2 Combining model-based and model-free

In Section 35.4.1, we discussed MPC, which uses the model to decide which action to take at each step. However, this can be slow, and can suffer from problems when the model is inaccurate. An alternative is to use the learned model to help reduce the sample complexity of policy learning.

There are many ways to do this. One approach is to generate rollouts from the model, and then train a policy or Q-function on the "hallucinated" data. This is the basis of the famous **dyna** method [Sut90]. In [Jan+19], they propose a similar method, but generate short rollouts from previously visited real states; this ensures the model only has to extrapolate locally.

In [Web+17], they train a model to predict future states and rewards, but then use the hidden states of this model as additional context for a policy-based learning method. This can help overcome partial observability. They call their method **imagination-augmented agents**. A related method appears in [Jad+17], who propose to train a model to jointly predict future rewards and other auxiliary signals, such as future states. This can help in situations when rewards are sparse or absent.



(a) (b)

Figure 35.7: (a) A cart-pole system being controlled by a policy learned by PILCO using just 17.5 seconds of real-world interaction. The goal state is marked by the red cross. The initial state is where the cart is stationary on the right edge of the workspace, and the pendulum is horizontal. For a video of the system learning, see https://bit.ly/35fpLmR. (b) A low-quality robot arm being controlled by a block-stacking policy learned by PILCO using just 230 seconds of real-world interaction. From Figures 11, 12 from [DFR15]. Used with kind permission of Marc Deisenroth.

35.4.3 MBRL using Gaussian processes

This section gives some examples of dynamics models that have been learned for low-dimensional continuous control problems. Such problems frequently arise in robotics. Since the dynamics are often nonlinear, it is useful to use a flexible and sample-efficient model family, such as Gaussian processes (Section 18.1). We will use notation like $\boldsymbol{s}$ and $\boldsymbol{a}$ for states and actions to emphasize they are vectors.

35.4.3.1 PILCO

We first describe the **PILCO** method [DR11; DFR15], which stands for "probabilistic inference for learning control". It is extremely data efficient for continuous control problems, enabling learning from scratch on real physical robots in a matter of minutes.

PILCO assumes the world model has the form $\boldsymbol{s}_{t+1} = f(\boldsymbol{s}_t, \boldsymbol{a}_t) + \boldsymbol{\epsilon}_t$, where $\boldsymbol{\epsilon}_t \sim \mathcal{N}(\mathbf{0}, \boldsymbol{\Sigma})$, and f is an unknown, continuous function.[4] The basic idea is to learn a Gaussian process (Section 18.1)) approximation of f based on some initial random trajectories, and then to use this model to generate "fantasy" rollout trajectories of length T, that can be used to evaluate the expected cost of the current policy, $J(\pi_{\boldsymbol{\theta}}) = \sum_{t=1}^{T} \mathbb{E}_{\boldsymbol{a}_t \sim \pi_{\boldsymbol{\theta}}}[c(\boldsymbol{s}_t)]$, where $\boldsymbol{s}_0 \sim p_0$. This function and its gradients wrt $\boldsymbol{\theta}$ can be computed deterministically, if a Gaussian assumption about the state distribution at each step is made, because the Gaussian belief state can be propagated deterministically through the GP model. Therefore, we can use deterministic batch optimization methods, such as Levenberg-Marquardt, to optimize the policy parameters $\boldsymbol{\theta}$, instead of applying SGD to sampled trajectories. (See https://github.com/mathDR/jax-pilco for some JAX code.)

Due to its data efficiency, it is possible to apply PILCO to real robots. Figure 35.7a shows the results of applying it to solve a **cart-pole swing-up** task, where the goal is to make the inverted pendulum swing up by applying a horizontal force to move the cart back and forth. The state of the system $\boldsymbol{s} \in \mathbb{R}^4$ consists of the position x of the cart (with $x = 0$ being the center of the track), the

4. An alternative, which often works better, is to use f to model the residual, so that $\boldsymbol{s}_{t+1} = \boldsymbol{s}_t + f(\boldsymbol{s}_t, \boldsymbol{a}_t) + \boldsymbol{\epsilon}_t$.

velocity $\dot{x}$, the angle θ of the pendulum (measured from hanging downward), and the angular velocity $\dot{\theta}$. The control signal $a \in \mathbb{R}$ is the force applied to the cart. The target state is $\boldsymbol{s}_* = (0, \star, \pi, \star)$, corresponding to the cart being in the middle and the pendulum being vertical, with velocities unspecified. The authors used an RBF controller with 50 basis functions, amounting to a total of 305 policy parameters. The controller was successfully trained using just 7 real world trials.[5]

Figure 35.7b shows the results of applying PILCO to solve a **block stacking** task using a low-quality robot arm with 6 degrees of freedom. A separate controller was trained for each block. The state space $\boldsymbol{s} \in \mathbb{R}^3$ is the 3d location of the center of the block in the arm's gripper (derived from an RGBD sensor), and the control $\boldsymbol{a} \in \mathbb{R}^4$ corresponds to the pulse widths of four servo motors. A linear policy was successfully trained using as few as 10 real world trials.

35.4.3.2 GP-MPC

[KD18a] have proposed an extension to PILCO that they call **GP-MPC**, since it combines a GP dynamics model with model predictive control (Section 35.4.1). In particular, they use an open-loop control policy to propose a sequence of actions, $\boldsymbol{a}_{t:t+H-1}$, as opposed to sampling them from a policy. They compute a Gaussian approximation to the future state trajectory, $p(\boldsymbol{s}_{t+1:t+H}|\boldsymbol{a}_{t:t+H-1}, \boldsymbol{s}_t)$, by moment matching, and use this to deterministically compute the expected reward and its gradient wrt $\boldsymbol{a}_{t:t+H-1}$ (as opposed to the policy parameters $\boldsymbol{\theta}$). Using this, they can solve Equation (35.46) to find $\boldsymbol{a}^*_{t:t+H-1}$; finally, they execute the first step of this plan, a^*_t, and repeat the whole process.

The advantage of GP-MPC over policy-based PILCO is that it can handle constraints more easily, and it can be more data efficient, since it continually updates the GP model after every step (instead of at the end of an trajectory).

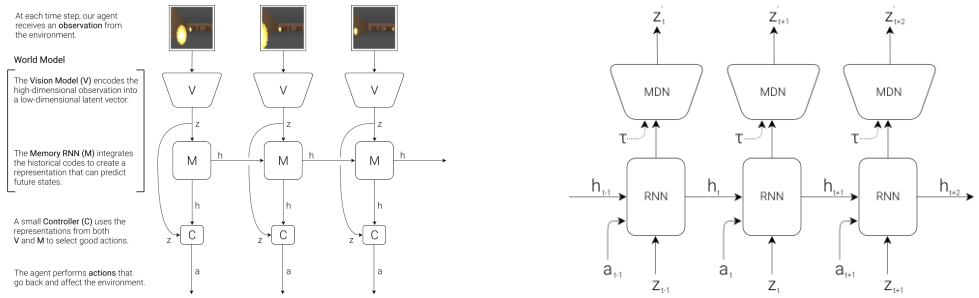
35.4.4 MBRL using DNNs

Gaussian processes do not scale well to large sample sizes and high dimensional data. Deep neural networks (DNNs) work much better in this regime. However, they do not naturally model uncertainty, which can cause MPC methods to fail. We discuss various methods for representing uncertainty with DNNs in Section 17.1. Here, we mention a few approaches that have been used for MBRL.

The **deep PILCO** method uses DNNs together with Monte Carlo dropout (Section 17.3.1) to represent uncertainty [GMAR16]. [Chu+18] proposed **probabilistic ensembles with trajectory sampling** or **PETS**, which represents uncertainty using an ensemble of DNNs (Section 17.3.9). Many other approaches are possible, depending on the details of the problem being tackled.

Since these are all stochastic methods (as opposed to the GP methods above), they can suffer from a high variance in the predicted returns, which can make it difficult for the MPC controller to pick the best action. We can reduce variance with the **common random number** trick [KSN99], where all rollouts share the same random seed, so differences in $J(\pi_{\boldsymbol{\theta}})$ can be attributed to changes in $\boldsymbol{\theta}$ but not other factors. This technique was used in **PEGASUS** [NJ00][6] and in [HMD18].

5. 2 random initial trials, each 5 seconds, and then 5 policy-generated trials, each 2.5 seconds, totalling 17.5 seeconds.
6. PEGASUS stands for "Policy Evaluation-of-Goodness And Search Using Scenarios", where the term "scenario" refers to one of the shared random samples.



(a) (b)

Figure 35.8: (a) Illustration of an agent interacting with the VizDoom environment. (The yellow blobs represent fireballs being thrown towards the agent by various enemies.) The agent has a world model, composed of a vision system V and a memory RNN M, and has a controller C. (b) Detailed representation of the memory model. Here h_t is the deterministic hidden state of the RNN at time t, which is used to predict the next latent of the VAE, z_{t+1}, using a mixture density network (MDN). Here τ is a temperature parameter used to increase the variance of the predictions, to prevent the controller from exploiting model inaccuracies. From Figures 4, 6 of [HS18]. Used with kind permission of David Ha.

35.4.5 MBRL using latent-variable models

In this section, we describe some methods that learn latent variable models, rather than trying to predict dynamics directly in the observed space, which is hard to do when the states are images.

35.4.5.1 World models

The "world models" paper [HS18] showed how to learn a generative model of two simple video games (CarRacing and a VizDoom-like environment), such that the model can be used to train a policy entirely in simulation. The basic idea is shown in Figure 35.8. First, we collect some random experience, and use this to fit a VAE model (Section 21.2) to reduce the dimensionality of the images, $\boldsymbol{x}_t \in \mathbb{R}^{64\times64\times3}$, to a latent $\boldsymbol{z}_t \in \mathbb{R}^{64}$. Next, we train an RNN to predict $p(\boldsymbol{z}_{t+1}|\boldsymbol{z}_t, \boldsymbol{a}_t, \boldsymbol{h}_t)$, where $\boldsymbol{h}_t$ is the deterministic RNN state, and $\boldsymbol{a}_t$ is the continuous action vector (3-dimensional in both cases). The emission model for the RNN is a mixture density network, in order to model multi-modal futures. Finally, we train the controller using $\boldsymbol{z}_t$ and $\boldsymbol{h}_t$ as inputs; here $\boldsymbol{z}_t$ is a compact representation of the current frame, and $\boldsymbol{h}_t$ is a compact representation of the predicted distribution over $\boldsymbol{z}_{t+1}$.

The authors of [HS18] trained the controller using a derivative free optimizer called **CMA-ES** (covariance matrix adaptation evolutionary strategy, see Section 6.9.6.2). It can work better than policy gradient methods, as discussed in Section 35.3.6. However, it does not scale to high dimensions. To tackle this, the authors use a linear controller, which has only 867 parameters.[7] By contrast, VAE has 4.3M parameters and MDN-RNN 422k. Fortunately, these two models can be trained in an unsupervised way from random rollouts, so sample efficiency is less critical than when training the policy.

7. The input is a 32-dimensional $\boldsymbol{z}_t$ plus a 256-dimensional $\boldsymbol{h}_t$, and there are 3 outputs. So the number of parameters is $(32 + 256) \times 3 + 3 = 867$, to account for the weights and biases.

So far, we have described how to use the representation learned by the generative model as informative features for the controller, but the controller is still learned by interacting with the real world. Surprisingly, we can also train the controller entirely in "dream mode", in which the generated images from the VAE decoder at time t are fed as input to the VAE encoder at time $t+1$, and the MDN-RNN is trained to predict the next reward r_{t+1} as well as $\boldsymbol{z}_{t+1}$. Unfortunately, this method does not always work, since the model (which is trained in an unsupervised way) may fail to capture task-relevant features (due to underfitting) and may memorize task-irrelevant features (due to overfitting). The controller can learn to exploit weaknesses in the model (similar to an adversarial attack) and achieve high simulated reward, but such a controller may not work well when transferred to the real world.

One approach to combat this is to artificially increase the variance of the MDN model (by using a temperature parameter τ), in order to make the generated samples more stochastic. This forces the controller to be robust to large variations; the controller will then treat the real world as just another kind of noise. This is similar to the technique of domain randomization, which is sometimes used for sim-to-real applications; see e.g., [MAZA18].

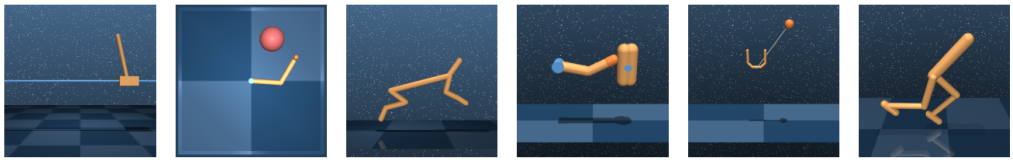
35.4.5.2 PlaNet and Dreamer

In [HS18], they first learn the world model on random rollouts, and then train a controller. On harder problems, it is necessary to iterate these two steps, so the model can be trained on data collected by the controller, in an iterative fashion.

In this section, we describe one method of this kind, known as **PlaNet** [Haf+19]. PlaNet uses a POMDP model, where $\boldsymbol{z}_t$ are the latent states, $\boldsymbol{s}_t$ are the observations, $\boldsymbol{a}_t$ are the actions, and r_t are the rewards. It fits a recurrent state space model (Section 29.13.2) of the form $p(\boldsymbol{z}_t|\boldsymbol{z}_{t-1}, \boldsymbol{a}_{t-1})p(\boldsymbol{s}_t|\boldsymbol{z}_t)p(r_t|\boldsymbol{z}_t)$ using variational inference, where the posterior is approximated by $q(\boldsymbol{z}_t|\boldsymbol{s}_{1:t}, \boldsymbol{a}_{1:t-1})$. After fitting the model to some random trajectories, the system uses the inference model to compute the current belief state, and then uses the cross entropy method to find an action sequence for the next H steps to maximize expected reward, by optimizing in latent space. The system then executes $\boldsymbol{a}_t^*$, updates the model, and repeats the whole process. To encourage the dynamics model to capture long term trajectories, they use the "latent overshooting" training method described in Section 29.13.3. The PlaNet method outperforms model-free methods, such as A3C (Section 35.3.3.1) and D4PG (Section 35.3.5), on various image-based continuous control tasks, illustrated in Figure 35.9.

Although PlaNet is sample efficient, it is not computationally efficient. For example, they use CEM with 1000 samples and 10 iterations to optimize trajectories with a horizon of length 12, which requires $120,000$ evaluations of the transition dynamics to choose a single action. [AY19] improve this by replacing CEM with differentiable CEM, and then optimize in a latent space of action sequences. This is much faster, but the results are not quite as good. However, since the whole policy is now differentiable, it can be fine-tuned using PPO (Section 35.3.4), which closes the performance gap at negligible cost.

A recent extension of the PlaNet paper, known as **Dreamer**, was proposed in [Haf+20]. In this paper, the online MPC planner is replaced by a policy network, $\pi(\boldsymbol{a}_t|\boldsymbol{z}_t)$, which is learned using gradient-based actor-critic in latent space. The inference and generative models are trained by maximizing the ELBO, as in PlaNet. The policy is trained by SGD to maximize expected total reward as predicted by the value function, and the value function is trained by SGD to minimize



(a) Cartpole (b) Reacher (c) Cheetah (d) Finger (e) Cup (f) Walker

Figure 35.9: Illustration of some image-based control problems used in the PlaNet paper. Inputs are $64 \times 64 \times 3$. (a) The cartpole swingup task has a fixed camera so the cart can move out of sight, making this a partially observable problem. (b) The reacher task has a sparse reward. (c) The cheetah running task includes both contacts and a larger number of joints. (d) The finger spinning task includes contacts between the finger and the object. (e) The cup task has a sparse reward that is only given once the ball is caught. (f) The walker task requires balance and predicting difficult interactions with the ground when the robot is lying down. From Figure 1 of [Haf+19]. Used with kind permission of Danijar Hafner.

MSE between predicted future reward and the TD-λ estimate (Section 35.2.2). They show that Dreamer gives better results than PlaNet, presumably because they learn a policy to optimize the long term reward (as estimated by the value function), rather than relying on MPC based on short-term rollouts.

35.4.6 Robustness to model errors

The main challenge with MBRL is that errors in the model can result in poor performance of the resulting policy, due to the distribution shift problem (Section 19.2). That is, the model is trained to predict states and rewards that it has seen using some behavior policy (e.g., the current policy), and then is used to compute an optimal policy under the learned model. When the latter policy is followed, the agent will experience a different distribution of states, under which the learned model may not be a good approximation of the real environment.

We require the model to generalize in a robust way to new states and actions. (This is related to the off-policy learning problem that we discuss in Section 35.5.) Failing that, the model should at least be able to quantify its uncertainty (Section 19.3). These topics are the focus of much recent research (see e.g., [Luo+19; Kur+19; Jan+19; Isl+19; Man+19; WB20; Eys+21]).

35.5 Off-policy learning

We have seen examples of off-policy methods such as Q-learning. They do not require that training data be generated by the policy it tries to evaluate or improve. Therefore, they tend to have greater data efficiency than their on-policy counterparts, by taking advantage of data generated by other policies. They are also easier to be applied in practice, especially in domains where costs and risks of following a new policy must be considered. This section covers this important topic.

A key challenge in off-policy learning is that the data distribution is typically different from the desired one, and this mismatch must be dealt with. For example, the probability of visiting a state s at time t in a trajectory depends not only on the MDP's transition model, but also on the policy that is being followed. If we are to estimate $J(\pi)$, as defined in Equation (35.15), but the trajectories

are generated by a different policy π', simply averaging rewards in the data gives us $J(\pi')$, not $J(\pi)$. We have to somehow correct for the gap, or "bias". Another challenge is that off-policy data can also make an algorithm unstable and divergent, which we will discuss in Section 35.5.3.

Removing distribution mismatches is not unique in off-policy learning, and is also needed in supervised learning to handle covariate shift (Section 19.2.3.1), and in causal effect estimation (Chapter 36), among others. Off-policy learning is also closely related to **offline reinforcement learning** (also called **batch reinforcement learning**): the former emphasizes the distributional mismatch between data and the agent's policy, and the latter emphasizes that the data is static and no further online interaction with the environment is allowed [LP03; EGW05; Lev+20]. Clearly, in the offline scenario with fixed data, off-policy learning is typically a critical technical component. Recently, several datasets have been prepared to facilitate empirical comparisons of offline RL methods (see e.g., [Gul+20; Fu+20]).

Finally, while this section focuses on MDPs, most methods can be simplified and adapted to the special case of contextual bandits (Section 34.4). In fact, off-policy methods have been successfully used in numerous industrial bandit applications (see e.g., [Li+10; Bot+13; SJ15; HLR16]).

35.5.1 Basic techniques

We start with four basic techniques, and will consider more sophisticated ones in subsequent sections. The off-policy data is assumed to be a collection of trajectories: $\mathcal{D} = \{\boldsymbol{\tau}^{(i)}\}_{1 \le i \le n}$, where each trajectory is a sequence as before: $\boldsymbol{\tau}^{(i)} = (s_0^{(i)}, a_0^{(i)}, r_0^{(i)}, s_1^{(i)} \ldots)$. Here, the reward and next states are sampled according to the reward and transition models; the actions are chosen by a **behavior policy**, denoted π_b, which is different from the **target policy**, π_e, that the agent is evaluating or improving. When π_b is unknown, we are in a **behavior-agnostic off-policy** setting.

35.5.1.1 Direct method

A natural approach to off-policy learning starts with estimating the unknown reward and transition models of the MDP from off-policy data. This can be done using regression and density estimation methods on the reward and transition models, respectively, to obtain $\hat{R}$ and $\hat{P}$; see Section 35.4 for further discussions. These estimated models then give us an inexpensive way to (approximately) simulate the original MDP, and we can apply on-policy methods on the simulated data. This method directly models the outcome of taking an action in a state, thus the name **direct method**, and is sometimes known as **regression estimator** and **plug-in estimator**.

While the direct method is natural and sometimes effective, it has a few limitations. First, a small estimation error in the simulator has a compounding effect in long-horizon problems (or equivalently, when the discount factor γ is close to 1). Therefore, an agent that is optimized against an MDP simulator may overfit the estimation errors. Unfortunately, learning the MDP model, especially the transition model, is generally difficult, making the method limited in domains where $\hat{R}$ and $\hat{P}$ can be learned to high fidelity. See Section 35.4.6 for a related discussion.

35.5.1.2 Importance sampling

The second approach relies on importance sampling (IS) (Section 11.5) to correct for distributional mismatches in the off-policy data. To demonstrate the idea, consider the problem of estimating the

target policy value $J(\pi_e)$ with a fixed horizon T. Correspondingly, the trajectories in $\mathcal{D}$ are also of length T. Then, the IS off-policy estimator, first adopted by [PSS00], is given by

$$\hat{J}_{\mathrm{IS}}(\pi_e) \triangleq \frac{1}{n}\sum_{i=1}^{n}\frac{p(\boldsymbol{\tau}^{(i)}|\pi_e)}{p(\boldsymbol{\tau}^{(i)}|\pi_b)}\sum_{t=0}^{T-1}\gamma^t r_t^{(i)} \tag{35.47}$$

It can be verified that $\mathbb{E}_{\pi_b}\left[\hat{J}_{\mathrm{IS}}(\pi_e)\right] = J(\pi_e)$, that is, $\hat{J}_{\mathrm{IS}}(\pi_e)$ is **unbiased**, provided that $p(\boldsymbol{\tau}|\pi_b) > 0$ whenever $p(\boldsymbol{\tau}|\pi_e) > 0$. The **importance ratio**, $\frac{p(\boldsymbol{\tau}^{(i)}|\pi_e)}{p(\boldsymbol{\tau}^{(i)}|\pi_b)}$, is used to compensate for the fact that the data is sampled from π_b and not π_e. Furthermore, this ratio does *not* depend on the MDP models, because for any trajectory $\boldsymbol{\tau} = (s_0, a_0, r_0, s_1, \ldots, s_T)$, we have from Equation (34.62) that

$$\frac{p(\boldsymbol{\tau}|\pi_e)}{p(\boldsymbol{\tau}|\pi_b)} = \frac{p(s_0)\prod_{t=0}^{T-1}\pi_e(a_t|s_t)p_T(s_{t+1}|s_t,a_t)p_R(r_t|s_t,a_t,s_{t+1})}{p(s_0)\prod_{t=0}^{T-1}\pi_b(a_t|s_t)p_T(s_{t+1}|s_t,a_t)p_R(r_t|s_t,a_t,s_{t+1})} = \prod_{t=0}^{T-1}\frac{\pi_e(a_t|s_t)}{\pi_b(a_t|s_t)} \tag{35.48}$$

This simplification makes it easy to apply IS, as long as the target and behavior policies are known. If the behavior policy is unknown, we can estimate it from $\mathcal{D}$ (using e.g., logistic regression or DNNs), and replace π_b by its estimate $\hat{\pi}_b$ in Equation (35.48). For convenience, define the **per-step importance ratio** at time t by $\rho_t(\boldsymbol{\tau}) \triangleq \pi_e(a_t|s_t)/\pi_b(a_t|s_t)$, and similarly, $\hat{\rho}_t(\boldsymbol{\tau}) \triangleq \pi_e(a_t|s_t)/\hat{\pi}_b(a_t|s_t)$.

Although IS can in principle eliminate distributional mismatches, in practice its usability is often limited by its potentially high variance. Indeed, the importance ratio in Equation (35.47) can be arbitrarily large if $p(\boldsymbol{\tau}^{(i)}|\pi_e) \gg p(\boldsymbol{\tau}^{(i)}|\pi_b)$. There are many improvements to the basic IS estimator. One improvement is based on the observation that the reward r_t is independent of the trajectory beyond time t. This leads to a **per-decision importance sampling** variant that often yields lower variance (see Section 11.6.2 for a statistical motivation, and [LBB20] for a further discussion):

$$\hat{J}_{\mathrm{PDIS}}(\pi_e) \triangleq \frac{1}{n}\sum_{i=1}^{n}\sum_{t=0}^{T-1}\prod_{t'\le t}\rho_{t'}(\boldsymbol{\tau}^{(i)})\gamma^t r_t^{(i)} \tag{35.49}$$

There are many other variants such as self-normalized IS and truncated IS, both of which aim to reduce variance possibly at the cost of a small bias; precise expressions of these alternatives are found, e.g., in [Liu+18b]. In the next subsection, we will discuss another systematic way to improve IS.

IS may also be applied to improve a policy against the policy value given in Equation (35.15). However, directly applying the calculation of Equation (35.48) runs into a fundamental issue with IS, which we will discuss in Section 35.5.2. For now, we may consider the following approximation of policy value, averaging over the state distribution of the behavior policy:

$$J_b(\pi_{\boldsymbol{\theta}}) \triangleq \mathbb{E}_{p_\beta^\infty(s)}\left[V_\pi(s)\right] = \mathbb{E}_{p_\beta^\infty(s)}\left[\sum_a \pi_{\boldsymbol{\theta}}(a|s)Q_\pi(s,a)\right] \tag{35.50}$$

Differentiating this and ignoring the term $\nabla_{\boldsymbol{\theta}} Q_\pi(s,a)$, as suggested by [DWS12], gives a way to (approximately) estimate the **off-policy policy-gradient** using a one-step IS correction ratio:

$$\begin{aligned}\nabla_{\boldsymbol{\theta}} J_b(\pi_{\boldsymbol{\theta}}) &\approx \mathbb{E}_{p_\beta^\infty(s)}\left[\sum_a \nabla_{\boldsymbol{\theta}}\pi_{\boldsymbol{\theta}}(a|s)Q_\pi(s,a)\right] \\ &= \mathbb{E}_{p_\beta^\infty(s)\beta(a|s)}\left[\frac{\pi_{\boldsymbol{\theta}}(a|s)}{\beta(a|s)}\nabla_{\boldsymbol{\theta}}\log\pi_{\boldsymbol{\theta}}(a|s)Q_\pi(s,a)\right]\end{aligned}$$

Finally, we note that in the tabular MDP case, there exists a policy π_* that is optimal in all states (Section 34.5.5). This policy maximizes J and J_b simultaneously, so Equation (35.50) can be a good proxy for Equation (35.15) as long as all states are "covered" by the behavior policy π_b. The situation is similar when the set of value functions or policies under consideration is sufficiently expressive: an example is a Q-learning like algorithm called Retrace [Mun+16; ASN20]. Unfortunately, in general when we work with parametric families of value functions or policies, such a uniform optimality is lost, and the distribution of states has a direct impact on the solution found by the algorithm. We will revisit this problem in Section 35.5.2.

35.5.1.3 Doubly robust

It is possible to combine the direct and importance sampling methods discussed previously. To develop intuition, consider the problem of estimating $J(\pi_e)$ in a contextual bandit (Section 34.4), that is, when $T = 1$ in $\mathcal{D}$. The **doubly robust** (DR) estimator is given by

$$\hat{J}_{\mathrm{DR}}(\pi_e) \triangleq \frac{1}{n}\sum_{i=1}^{n}\left(\frac{\pi_e(a_0^{(i)}|s_0^{(i)})}{\hat{\pi}_b(a_0^{(i)}|s_0^{(i)})}\left(r_0^{(i)} - \hat{Q}(s_0^{(i)}, a_0^{(i)})\right) + \hat{V}(s_0^{(i)})\right) \tag{35.51}$$

where $\hat{Q}$ is an estimate of Q_{π_e}, which can be obtained using methods discussed in Section 35.2, and $\hat{V}(s) = \mathbb{E}_{\pi_e(a|s)}\left[\hat{Q}(s,a)\right]$. If $\hat{\pi}_b = \pi_b$, the term $\hat{Q}$ is canceled by $\hat{V}$ on average, and we get the IS estimate that is unbiased; if $\hat{Q} = Q_{\pi_e}$, the term $\hat{Q}$ is canceled by the reward on average, and we get the estimator as in the direct method that is also unbiased. In other words, the estimator Equation (35.51) is unbiased, as long as one of the estimates, $\hat{\pi}_b$ and $\hat{Q}$, is right. This observation justifies the name doubly robust, which has its origin in causal inference (see e.g., [BR05]).

The above DR estimator may be extended to MDPs recursively, starting from the last step. Given a length-T trajectory $\boldsymbol{\tau}$, define $\hat{J}_{\mathrm{DR}}[T] \triangleq 0$, and for $t < T$,

$$\hat{J}_{\mathrm{DR}}[t] \triangleq \hat{V}(s_t) + \hat{\rho}_t(\boldsymbol{\tau})\left(r_t + \gamma \hat{J}_{\mathrm{DR}}[t+1] - \hat{Q}(s_t, a_t)\right) \tag{35.52}$$

where $\hat{Q}(s_t, a_t)$ is the estimated cumulative reward for the remaining $T - t$ steps. The DR estimator of $J(\pi_e)$, denoted $\hat{J}_{\mathrm{DR}}(\pi_e)$, is the average of $\hat{J}_{\mathrm{DR}}[0]$ over all n trajectories in $\mathcal{D}$ [JL16]. It can be verified (as an exercise) that the recursive definition is equivalent to

$$\hat{J}_{\mathrm{DR}}[0] = \hat{V}(s_0) + \sum_{t=0}^{T-1}\left(\prod_{t'=0}^{t}\hat{\rho}_{t'}(\boldsymbol{\tau})\right)\gamma^t\left(r_t + \gamma\hat{V}(s_{t+1}) - \hat{Q}(s_t, a_t)\right) \tag{35.53}$$

This form can be easily generalized to the infinite-horizon setting by letting $T \to \infty$ [TB16]. Other than double robustness, the estimator is also shown to result in minimum variance under certain conditions [JL16]. Finally, the DR estimator can be incorporated into policy gradient for policy optimization, to reduce gradient estimation variance [HJ20].

35.5.1.4 Behavior regularized method

The three methods discussed previously do not impose any constraint on the target policy π_e. Typically, the more different π_e is from π_b, the less accurate our off-policy estimation can be.

Therefore, when we optimize a policy in offline RL, a natural strategy is to favor target policies that are "close" to the behavior policy. Similar ideas are discussed in the context of conservative policy gradient (Section 35.3.4).

One approach is to impose a hard constraint on the proximity between the two policies. For example, we may modify the loss function of DQN (Equation (35.14)) as follows

$$\mathcal{L}_1^{\text{DQN}}(\boldsymbol{w}) \triangleq \mathbb{E}_{(s,a,r,s')\sim\mathcal{D}}\left[\left(r+\gamma \max_{\pi: D(\pi,\pi_b)\leq\varepsilon} \mathbb{E}_{\pi(a'|s')}\left[Q_{\boldsymbol{w}^-}(s',a')\right] - Q_{\boldsymbol{w}}(s,a)\right)^2\right] \tag{35.54}$$

In the above, we replace the $\max_{a'}$ operation by an expectation over a policy that stays close enough to the behavior policy, measured by some distance function D. For various instantiations and further details, see e.g, [FMP19; Kum+19a].

We may also impose a soft constraint on the proximity, by penalizing target policies that are too different. The DQN loss function can be adapted accordingly:

$$\mathcal{L}_2^{\text{DQN}}(\boldsymbol{w}) \triangleq \mathbb{E}_{(s,a,r,s')\sim\mathcal{D}}\left[\left(r+\gamma \max_{\pi} \mathbb{E}_{\pi(a'|s')}\left[Q_{\boldsymbol{w}^-}(s',a')\right] - \alpha\gamma D(\pi(s'),\pi_b(s')) - Q_{\boldsymbol{w}}(s,a)\right)^2\right] \tag{35.55}$$

This idea has been used in contextual bandits [SJ15] and empirically studied in MDPs by [WTN19].

There are many choices for the function D, such as the KL-divergence, for both hard and soft constraints. More detailed discussions and examples can be found in [Lev+20].

Finally, behavior regularization and previous methods like IS can be combined, where the former ensures lower variance and greater generalization of the latter (e.g., [SJ15]). Furthermore, most proposed behavior regularized methods consider one-step difference in D, comparing $\pi(s)$ and $\pi_b(s)$ conditioned on s. In many cases, it is desired to consider the difference between the long-term distributions, p_β^∞ and p^∞, which we will discuss next.

35.5.2 The curse of horizon

The IS and DR approaches presented in the previous section all rely on an importance ratio to correct distributional mismatches. The ratio depends on the entire trajectory, and its variance grows exponentially in the trajectory length T. Correspondingly, the off-policy estimate of either the policy value or policy gradient can suffer an exponentially large variance (and thus very low accuracy), a challenge called the **curse of horizon** [Liu+18b]. Policies found by approximate algorithms like Q-learning and off-policy actor-critic often have hard-to-control error due to distribution mismatches.

This section discusses an approach to tackling this challenge, by considering corrections in the state-action distribution, rather than in the trajectory distribution. This change is critical: [Liu+18b] describes an example, where the state-action distributions under the behavior and target policies are identical, but the importance ratio of a trajectory grows exponentially large. It is now more convenient to assume the off-policy data consists of a set of transitions: $\mathcal{D} = \{(s_i, a_i, r_i, s_i')\}_{1\leq i\leq m}$, where $(s_i, a_i) \sim p_{\mathcal{D}}$ (some fixed but unknown sampling distribution, such as p_β^∞), and r_i and s_i' are sampled from the MDP's reward and transition models. Given a policy π, we aim to estimate the correction ratio $\zeta_*(s,a) = p_\pi^\infty(s,a)/p_{\mathcal{D}}(s,a)$, as it allows us to rewrite the policy value (Equation (35.15)) as

$$J(\pi) = \frac{1}{1-\gamma}\mathbb{E}_{p_\pi^\infty(s,a)}\left[R(s,a)\right] = \frac{1}{1-\gamma}\mathbb{E}_{p_\beta^\infty(s,a)}\left[\zeta_*(s,a)R(s,a)\right] \tag{35.56}$$

For simplicity, we assume the initial state distribution p_0 is known, or can be easily sampled from. This assumption is often easy to satisfy in practice.

The starting point is the following linear program formulation for any given π:

$$\max_{d \geq 0} -\mathrm{D}_\mathrm{f}\left(d \| p_{\mathcal{D}}\right)) \quad \text{s.t.} \quad d(s,a) = (1-\gamma)\mu_0(s)\pi(a|s) + \gamma \sum_{\bar{s},\bar{a}} p(s|\bar{s},\bar{a}) d(\bar{s},\bar{a})\pi(a|s) \;\; \forall (s,a) \tag{35.57}$$

where D_f is the f-divergence (Section 2.7.1). The constraint is a variant of Equation (34.81), giving similar flow conditions in the space of $\mathcal{S} \times \mathcal{A}$ under policy π. Under mild conditions, p_π^∞ is only solution that satisfies the flow constraints, so the objective does not affect the solution, but will facilitate the derivation below. We can now obtain the Lagrangian, with multipliers $\{\nu(s,a)\}$, and use the change-of-variables $\zeta(s,a) = d(s,a)/p_{\mathcal{D}}(s,a)$ to obtain the following optimization problem:

$$\begin{aligned} \max_{\zeta \geq 0} \min_{\nu} \mathcal{L}(\zeta, \nu) =& \mathbb{E}_{p_{\mathcal{D}}(s,a)}\left[-f(\zeta(s,a)\right] + (1-\gamma)\mathbb{E}_{p_0(s)\pi(a|s)}\left[\nu(s,a)\right] \\ &+ \mathbb{E}_{\pi(a'|s')p(s'|s,a)p_{\mathcal{D}}(s,a)}\left[\zeta(s,a)\left(\gamma\nu(s',a') - \nu(s,a)\right)\right] \end{aligned} \tag{35.58}$$

It can be shown that the saddle point to Equation (35.58) must coincide with the desired correction ratio ζ_*. In practice, we may parameterize ζ and ν, and apply two-timescales stochastic gradient descent/ascent on the off-policy data $\mathcal{D}$ to solve for an approximate saddle-point. This is the **DualDICE** method [Nac+19a], which is extended to **GenDICE** [Zha+20c].

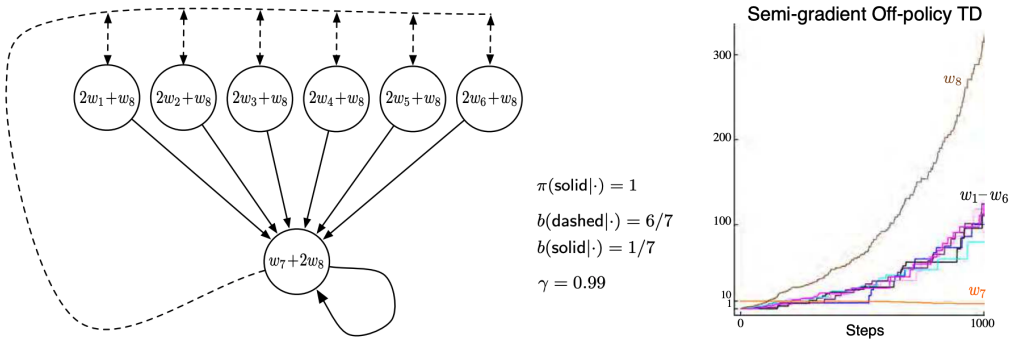
Compared to the IS or DR approaches, Equation (35.58) does not compute the importance ratio of a trajectory, thus generally has a lower variance. Furthermore, it is behavior-agnostic, without having to estimate the behavior policy, or even to assume data consists of a collection of trajectories. Finally, this approach can be extended to be doubly robust (e.g., [UHJ20]), and to optimize a policy [Nac+19b] against the true policy value $J(\pi)$ (as opposed to approximations like Equation (35.50)). For more examples along this line of approach, see [ND20] and the references therein.

35.5.3 The deadly triad

Other than introducing bias, off-policy data may also make a value-based RL method unstable and even divergent. Consider the simple MDP depicted in Figure 35.10a, due to [Bai95]. It has 7 states and 2 actions. Taking the dashed action takes the environment to the 6 upper states uniformly at random, while the solid action takes it to the bottom state. The reward is 0 in all transitions, and $\gamma = 0.99$. The value function $V_{\boldsymbol{w}}$ uses a linear parameterization indicated by the expressions shown inside the states, with $\boldsymbol{w} \in \mathbb{R}^8$. The target policies π always chooses the solid action in every state. Clearly, the true value function, $V_\pi(s) = 0$, can be exactly represented by setting $\boldsymbol{w} = \mathbf{0}$.

Suppose we use a behavior policy b to generate a trajectory, which chooses the dashed and solid actions with probabilities 6/7 and 1/7, respectively, in every state. If we apply TD(0) on this trajectory, the parameters diverge to ∞ (Figure 35.10b), even though the problem appears simple! In contrast, with on-policy data (that is, when b is the same as π), TD(0) with linear approximation can be guaranteed to converge to a good value function approximate [TR97].

The divergence behavior is demonstrated in many value-based bootstrapping methods, including TD, Q-learning, and related approximate dynamic programming algorithms, where the value function is represented either linearly (like the example above) or nonlinearly [Gor95; Ber19]. The root cause



(a) *(b)*

Figure 35.10: (a) ... (b) ... From Figures 11.1 & 11.2 of [SB18]. Used with kind permission of Richard Sutton.

of these divergence phenomena is that the contraction property in the tabular case (Equation (34.75)) may no longer hold when V is approximated by $V_{\boldsymbol{w}}$. An RL algorithm can become unstable when it has these three components: off-policy learning, bootstrapping (for faster learning, compared to MC), function approximation (for generalization in large scale MDPs). This combination is known as **the deadly triad** [SB18]. It highlights another important challenge introduced by off-policy learning, and is a subject of ongoing research (e.g., [van+18; Kum+19a]).

A general way to ensure convergence in off-policy learning is to construct an objective function function, the minimization of which leads to a good value function approximation; see [SB18, Ch. 11] for more background. A natural candidate is the discrepancy between the left and right hand sides of the Bellman optimality equation Equation (34.70), whose unique solution is V_*. However, the "max" operator is not friendly to optimization. Instead, we may introduce an entropy term to smooth the greedy policy, resulting in a differential square loss in **path consistency learning** (**PCL**) [Nac+17]:

$$\min_{V,\pi} \mathcal{L}^{\text{PCL}}(V,\pi) \triangleq \mathbb{E}\left[\frac{1}{2}\big(r + \gamma V(s') - \lambda \log \pi(a|s) - V(s)\big)^2\right] \tag{35.59}$$

where the expectation is over (s, a, r, s') tuples drawn from some off-policy distribution (e.g., uniform over $\mathcal{D}$). Minimizing this loss, however, does not result in the optimal value function and policy in general, due to an issue known as "double sampling" [SB18, Sec. 11.5].

This problem can be mitigated by introducing a dual function in the optimization [Dai+18]

$$\min_{V,\pi} \max_{\nu} \mathcal{L}^{\text{SBEED}}(V,\pi;\nu) \triangleq \mathbb{E}\left[\nu(s,a)\big(r + \gamma V(s') - \lambda \log \pi(a|s) - V(s)\big)^2 - \nu(s,a)^2/2\right] \tag{35.60}$$

where ν belongs to some function class (e.g., a DNN [Dai+18] or RKHS [FLL19]). It can be shown that optimizing Equation (35.60) forces ν to model the Bellman error. So this approach is called **smoothed Bellman error embedding**, or **SBEED**. In both PCL and SBEED, the objective can be optimized by gradient-based methods on parameterized value functions and policies.

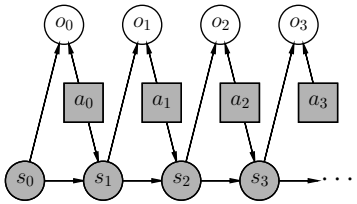


Figure 35.11: A graphical model for optimal control. States and actions are observed, while optimality variables are not. Adapted from Figure 1b of [Lev18].

35.6 Control as inference

In this section, we will discuss another approach to policy optimization, by reducing it to probabilistic inference. This is called **control as inference**, see e.g., [Att03; TS06; Tou09; BT12; KGO12; HR17; Lev18]. This approach allows one to incorporate domain knowledge in modeling, and apply powerful tools from approximate inference (see e.g., Chapter 7), in a consistent and flexible framework.

35.6.1 Maximum entropy reinforcement learning

We now describe a graphical model that exemplifies such a reduction, which results in RL algorithms that are closely related to some discussed previously. The model allows a trade-off between reward and entropy maximization, and recovers the standard RL setting when the entropy part vanishes in the trade-off. Our discussion mostly follows the approach of [Lev18].

Figure 35.11 gives a probabilistic model, which not only captures state transitions as before, but also introduces a new variable, o_t. This variable is binary, indicating whether the action at time t is optimal or not, and has the following probability distribution:

$$p(o_t = 1|s_t, a_t) = \exp(\lambda^{-1} R(s_t, a_t)) \tag{35.61}$$

for some temperature parameter $\lambda > 0$ whose role will be clear soon. In the above, we have assumed without much loss of generality that $R(s, a) < 0$, so that Equation (35.61) gives a valid probability. Furthermore, we can assume a non-informative, uniform action prior, $p(a_t|s_t)$, to simplify the exposition, for we can always push $p(a_t|s_t)$ into Equation (35.61). Under these assumptions, the likelihood of observing a length-T trajectory $\boldsymbol{\tau}$, when optimality achieved in every step, is:

$$\begin{aligned} p(\boldsymbol{\tau}|\mathbf{o}_{0:T-1} = \mathbf{1}) \propto p(\boldsymbol{\tau}, \mathbf{o}_{0:T-1} = \mathbf{1}) &\propto p(s_0) \prod_{t=0}^{T-1} p(o_t = 1|s_t, a_t) p_T(s_{t+1}|s_t, a_t) \\ &= p(s_0) \prod_{t=0}^{T-1} p_T(s_{t+1}|s_t, a_t) \exp\left(\frac{1}{\lambda} \sum_{t=0}^{T-1} R(s_t, a_t)\right) \end{aligned} \tag{35.62}$$

The intuition of Equation (35.62) is clearest when the state transitions are deterministic. In this case, $p_T(s_{t+1}|s_t, a_t)$ is either 1 or 0, depending on whether the transition is dynamically feasible or

not. Hence, $p(\boldsymbol{\tau}|\mathbf{o}_{0:T-1}=\mathbf{1})$ is either proportional to $\exp(\lambda^{-1}\sum_{t=0}^{T-1}R(s_t,a_t))$ if $\boldsymbol{\tau}$ is feasible, or 0 otherwise. Maximizing reward is equivalent to inferring a trajectory with maximum $p(\boldsymbol{\tau}|\mathbf{o}_{0:T-1}=\mathbf{1})$.

The optimal policy in this probabilistic model is given by

$$p(a_t|s_t,\mathbf{o}_{t:T-1}=\mathbf{1}) = \frac{p(s_t,a_t|\mathbf{o}_{t:T-1}=\mathbf{1})}{p(s_t|\mathbf{o}_{t:T-1}=\mathbf{1})} = \frac{p(\mathbf{o}_{t:T-1}=\mathbf{1}|s_t,a_t)p(a_t|s_t)p(s_t)}{p(\mathbf{o}_{t:T-1}=\mathbf{1}|s_t)p(s_t)} \propto \frac{p(\mathbf{o}_{t:T-1}=\mathbf{1}|s_t,a_t)}{p(\mathbf{o}_{t:T-1}=\mathbf{1}|s_t)} \tag{35.63}$$

The two probabilities in Equation (35.63) can be computed as follows, starting with $p(o_{T-1}=1|s_{T-1},a_{T-1})=\exp(\lambda^{-1}R(s_{T-1},a_{T-1}))$,

$$p(\mathbf{o}_{t:T-1}=\mathbf{1}|s_t,a_t) = \int_{\mathcal{S}} p(\mathbf{o}_{t+1:T-1}=\mathbf{1}|s_{t+1})p_T(s_{t+1}|s_t,a_t)\exp(\lambda^{-1}R(s_t,a_t))ds_{t+1} \tag{35.64}$$

$$p(\mathbf{o}_{t:T-1}=\mathbf{1}|s_t) = \int_{\mathcal{A}} p(\mathbf{o}_{t:T-1}=\mathbf{1}|s_t,a_t)p(a_t|s_t)da_t \tag{35.65}$$

The calculation above is expensive. In practice, we can approximate the optimal policy using a parametric form, $\pi_\theta(a_t|s_t)$. The resulted probability of trajectory $\boldsymbol{\tau}$ now becomes

$$p_\theta(\boldsymbol{\tau}) = p(s_1)\prod_{t=0}^{T-1} p_T(s_{t+1}|s_t,a_t)\pi_\theta(a_t|s_t) \tag{35.66}$$

If we optimize θ so that $D_{\mathbb{KL}}\left(p_\theta(\boldsymbol{\tau}) \parallel p(\boldsymbol{\tau}|\mathbf{o}_{0:T-1}=\mathbf{1})\right)$ is minimized, which can be simplified to

$$D_{\mathbb{KL}}\left(p_\theta(\boldsymbol{\tau}) \parallel p(\boldsymbol{\tau}|\mathbf{o}_{0:T-1}=\mathbf{1})\right) = -\mathbb{E}_{p_\theta}\left[\sum_{t=0}^{T-1}\lambda^{-1}R(s_t,a_t) + \mathbb{H}(\pi_{\boldsymbol{\theta}}(s_t))\right] + \text{const} \tag{35.67}$$

where the constant term only depends on the uniform action prior $p(a_t|s_t)$, but not $\boldsymbol{\theta}$. In other words, the objective is to maximize total reward, with an entropy regularization favoring more uniform policies. Thus this approach is called **maximum entropy RL**, or **MERL**. If $\pi_{\boldsymbol{\theta}}$ can represent all stochastic policies, a softmax version of the Bellman equation can be obtained for Equation (35.67):

$$Q_*(s_t,a_t) = \lambda^{-1}R(s_t,a_t) + \mathbb{E}_{p_T(s_{t+1}|s_t,a_t)}\left[\log\int_{\mathcal{A}}\exp(Q_*(s_{t+1},a_{t+1}))da\right] \tag{35.68}$$

with the convention that $Q_*(s_T,a)=0$ for all a, and the optimal policy has a softmax form: $\pi_*(a_t|s_t)\propto\exp(Q_*(s_t,a_t))$. Note that the Q_* above is different from the usual optimal Q-function (Equation (34.71)), due to the introduction of the entropy term. However, as $\lambda\to 0$, their difference vanishes, and the softmax policy becomes greedy, recovering the standard RL setting.

The **soft actor-critic** (**SAC**) algorithm [Haa+18b; Haa+18c] is an off-policy actor-critic method whose objective function is equivalent to Equation (35.67) (by taking T to ∞):

$$J^{\text{SAC}}(\boldsymbol{\theta}) \triangleq \mathbb{E}_{p^{\infty}_{\pi_{\boldsymbol{\theta}}}(s)\pi_{\boldsymbol{\theta}}(a|s)}\left[R(s,a) + \lambda\,\mathbb{H}(\pi_{\boldsymbol{\theta}}(s))\right] \tag{35.69}$$

Note that the entropy term has also the added benefit of encouraging exploration.

To compute the optimal policy, similar to other actor-critic algorithms, we will work with the "soft" state- and action-function approximations, parameterized by $\boldsymbol{w}$ and $\boldsymbol{u}$, respectively:

$$Q_{\boldsymbol{w}}(s,a) = R(s,a) + \gamma \mathbb{E}_{p_T(s'|s,a)}\left[V_{\boldsymbol{u}}(s',a') - \lambda \log \pi_{\boldsymbol{\theta}}(a'|s')\right] \tag{35.70}$$

$$V_{\boldsymbol{u}}(s,a) = \lambda \log \sum_a \exp(\lambda^{-1} Q_{\boldsymbol{w}}(s,a)) \tag{35.71}$$

This induces an improved policy (with entropy regularization): $\pi_{\boldsymbol{w}}(a|s) = \exp(\lambda^{-1}Q_{\boldsymbol{w}}(s,a))/Z_{\boldsymbol{w}}(s)$, where $Z_{\boldsymbol{w}}(s) = \sum_a \exp(\lambda^{-1}Q_{\boldsymbol{w}}(s,a))$ is the normalization constant. We then perform a soft policy improvement step to update $\boldsymbol{\theta}$ by minimizing $\mathbb{E}\left[D_{\mathbb{KL}}\left(\pi_{\boldsymbol{\theta}}(s) \parallel \pi_{\boldsymbol{w}}(s)\right)\right]$ where the expectation may be approximated by sampling s from a replay buffer D.

In [Haa+18c; Haa+18b], they show that the SAC method outperforms the off-policy DDPG algorithm (Section 35.3.5) and the on-policy PPO algorithm (Section 35.3.4) by a wide margin on various continuous control tasks. For more details, see [Haa+18c].

There is a variant of soft actor-critic, which only requires to model the action-value function. It is based on the observation that both the policy and soft value function can be induced by the soft action-value function as follows:

$$V_{\boldsymbol{w}}(s) = \lambda \log \sum_a \exp\left(\lambda^{-1} Q_{\boldsymbol{w}}(s,a)\right) \tag{35.72}$$

$$\pi_{\boldsymbol{w}}(a|s) = \exp\left(\lambda^{-1}(Q_{\boldsymbol{w}}(s,a) - V_{\boldsymbol{w}}(s))\right) \tag{35.73}$$

We then only need to learn $\boldsymbol{w}$, using approaches similar to DQN (Section 35.2.6). The resulting algorithm, **soft Q-learning** [SAC17], is convenient if the number of actions is small (when $\mathcal{A}$ is discrete), or if the integral in obtaining $V_{\boldsymbol{w}}$ from $Q_{\boldsymbol{w}}$ is easy to compute (when $\mathcal{A}$ is continuous).

It is interesting to see that algorithms derived in the maximum entropy RL framework bears a resemblance to PCL and SBEED in Section 35.5.3, both of which were to minimize an objective function resulting from the entropy-smoothed Bellman equation.

35.6.2 Other approaches

VIREL is an alternative model to maximum entropy RL [Fel+19]. Similar to soft actor-critic, it uses an approximate action-value function, $Q_{\boldsymbol{w}}$, a stochastic policy, $\pi_{\boldsymbol{\theta}}$, and a binary optimality random variable o_t at time t. A different probability model for o_t is used

$$p(o_t = 1|s_t, a_t) = \exp\left(\frac{Q_{\boldsymbol{w}}(s_t,a_t) - \max_a Q_{\boldsymbol{w}}(s_t,a)}{\lambda_{\boldsymbol{w}}}\right) \tag{35.74}$$

The temperature parameter $\lambda_{\boldsymbol{w}}$ is also part of the parameterization, and can be updated from data.

An EM method can be used to maximize the objective

$$\mathcal{L}(\boldsymbol{w}, \boldsymbol{\theta}) = \mathbb{E}_{p(s)}\left[\mathbb{E}_{\pi_{\boldsymbol{\theta}}(a|s)}\left[\frac{Q_{\boldsymbol{w}}(s,a)}{\lambda_{\boldsymbol{w}}}\right] + \mathbb{H}(\pi_{\boldsymbol{\theta}}(s))\right] \tag{35.75}$$

for some distribution p that can be conveniently sampled from (e.g., in a replay buffer). The algorithm may be interpreted as an instance of actor-critic. In the E-step, the critic parameter $\boldsymbol{w}$ is fixed, and the actor parameter $\boldsymbol{\theta}$ is updated using gradient accent with stepsize $\eta_{\boldsymbol{\theta}}$ (for policy improvement):

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta_{\boldsymbol{\theta}} \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{w}, \boldsymbol{\theta}) \tag{35.76}$$

In the M-step, the actor parameter is fixed, and the critic parameter is updated (for policy evaluation):

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + \eta_{\boldsymbol{w}} \nabla_{\boldsymbol{w}} \mathcal{L}(\boldsymbol{w}, \boldsymbol{\theta}) \tag{35.77}$$

Finally, there are other possibilities of reducing optimal control to probabilistic inference, in addition to MERL and VIREL. For example, we may aim to maximize the expectation of the trajectory return G, by optimizing the policy parameter $\boldsymbol{\theta}$:

$$J(\pi_{\boldsymbol{\theta}}) = \int G(\boldsymbol{\tau}) p(\boldsymbol{\tau}|\boldsymbol{\theta}) d\boldsymbol{\tau} \tag{35.78}$$

It can be interpreted as a pseudo-likelihood function, when the $G(\boldsymbol{\tau})$ is treated as probability density, and solved (approximately) by a range of algorithms (see e.g., [PS07; Neu11; Abd+18]). Interestingly, some of these methods have a similar objective as MERL (Equation (35.67)), although the distribution involving $\boldsymbol{\theta}$ appears in the second argument of $\mathrm{D}_{\mathbb{KL}}$. As discussed in Section 2.7.1, this forward KL-divergence is mode-covering, which in the context of RL is argued to be less preferred than the mode-seeking, reverse KL-divergence used by MERL. For more details and references, see [Lev18].

Control as inference is also closely related to **active inference**; this is based on the **free energy principle** which is popular in neuroscience (see e.g., [Fri09; Buc+17; SKM18; Ger19; Maz+22]). The FEP is equivalent to using variational inference (see Section 10.1) to perform state estimation (perception) and parameter estimation (learning). In particular, consider a latent variable model with hidden states $\boldsymbol{s}$, observations $\boldsymbol{y}$ and parameters $\boldsymbol{\theta}$. Following Section 10.1.1, we define the variational free energy to be $\mathcal{F}(\boldsymbol{o}) = D_{\mathbb{KL}}\left(q(\boldsymbol{s}, \boldsymbol{\theta}|\boldsymbol{y}) \parallel p(\boldsymbol{s}, \boldsymbol{y}, \boldsymbol{\theta})\right)$. State estimation corresponds to solving $\min_{q(\boldsymbol{s}|\boldsymbol{y})} \mathcal{F}(\boldsymbol{y})$, and parameter estimation corresponds to solving $\min_{q(\boldsymbol{\theta}|\boldsymbol{y})} \mathcal{F}(\boldsymbol{y})$, just as in variational Bayes EM (Section 10.2.5). (Minimizing the VFE for certain hierarchical Gaussian models also forms the foundation of predictive coding, which we discuss in Supplementary Section 8.1.3.)

To extend this to decision making problems we can define the **expected free energy** as $\overline{\mathcal{F}}(a) = \mathbb{E}_{q(\boldsymbol{y}|\boldsymbol{a})}\left[\mathcal{F}(\boldsymbol{y})\right]$, where $q(\boldsymbol{y}|\boldsymbol{a})$ is the posterior predictive distribution over observations given actions sequence $\boldsymbol{a}$. The connection to control as inference is explained in [Mil+20; WIP20; LÖW21].

35.6.3 Imitation learning

In previous sections, an RL agent is to learn an optimal sequential decision making policy so that the total reward is maximized. **Imitation learning** (IL), also known as **apprenticeship learning** and **learning from demonstration** (LfD), is a different setting, in which the agent does not observe rewards, but has access to a collection $\mathcal{D}_{\text{exp}}$ of trajectories generated by an expert policy π_{exp}; that is, $\boldsymbol{\tau} = (s_0, a_0, s_1, a_1, \ldots, s_T)$ and $a_t \sim \pi_{\text{exp}}(s_t)$ for $\boldsymbol{\tau} \in \mathcal{D}_{\text{exp}}$. The goal is to learn a good policy by imitating the expert, in the absence of reward signals. IL finds many applications in scenarios where we have demonstrations of experts (often humans) but designing a good reward function is not easy, such as car driving and conversational systems. See [Osa+18] for a survey up to 2018.

35.6.3.1 Imitation learning by behavior cloning

A natural method is **behavior cloning**, which reduces IL to supervised learning; see [Pom89] for an early application to autonomous driving. It interprets a policy as a classifier that maps states (inputs) to actions (labels), and finds a policy by minimizing the imitation error, such as

$$\min_{\pi} \mathbb{E}_{p^{\infty}_{\pi_{\text{exp}}}(s)}\left[D_{\mathbb{KL}}\left(\pi_{\text{exp}}(s) \parallel \pi(s)\right)\right] \tag{35.79}$$

where the expectation wrt $p^{\infty}_{\pi_{\text{exp}}}$ may be approximated by averaging over states in $\mathcal{D}_{\text{exp}}$. A challenge with this method is that the loss does not consider the sequential nature of IL: future state distribution is not fixed but instead depends on earlier actions. Therefore, if we learn a policy $\hat{\pi}$ that has a low imitation error under distribution $p^{\infty}_{\pi_{\text{exp}}}$, as defined in Equation (35.79), it may still incur a large error under distribution $p^{\infty}_{\hat{\pi}}$ (when the policy $\hat{\pi}$ is actually run). Further expert demonstrations or algorithmic augmentations are often needed to handle the distribution mismatch (see e.g., [DLM09; RGB11]).

35.6.3.2 Imitation learning by inverse reinforcement learning

An effective approach to IL is **inverse reinforcement learning** (IRL) or **inverse optimal control** (IOC). Here, we first infer a reward function that "explains" the observed expert trajectories, and then compute a (near-)optimal policy against this learned reward using any standard RL algorithms studied in earlier sections. The key step of reward learning (from expert trajectories) is the opposite of standard RL, thus called inverse RL [NR00a].

It is clear that there are infinitely many reward functions for which the expert policy is optimal, for example by several optimality-preserving transformations [NHR99]. To address this challenge, we can follow the maximum entropy principle (Section 2.3.7), and use an energy-based probability model to capture how expert trajectories are generated [Zie+08]:

$$p(\boldsymbol{\tau}) \propto \exp\big(\sum_{t=0}^{T-1} R_{\boldsymbol{\theta}}(s_t, a_t)\big) \tag{35.80}$$

where $R_{\boldsymbol{\theta}}$ is an unknown reward function with parameter $\boldsymbol{\theta}$. Abusing notation slightly, we denote by $R_{\boldsymbol{\theta}}(\boldsymbol{\tau}) = \sum_{t=0}^{T-1} R_{\boldsymbol{\theta}}(s_t, a_t))$ the cumulative reward along the trajectory $\boldsymbol{\tau}$. This model assigns exponentially small probabilities to trajectories with lower cumulative rewards. The partition function, $Z_{\boldsymbol{\theta}} \triangleq \int_{\boldsymbol{\tau}} \exp(R_{\boldsymbol{\theta}}(\boldsymbol{\tau}))$, is in general intractable to compute, and must be approximated. Here, we can take a sample-based approach. Let $\mathcal{D}_{\text{exp}}$ and $\mathcal{D}$ be the sets of trajectories generated by an expert, and by some known distribution q, respectively. We may infer $\boldsymbol{\theta}$ by maximizing the likelihood, $p(\mathcal{D}_{\text{exp}}|\boldsymbol{\theta})$, or equivalently, minimizing the negative log-likelihood loss

$$\mathcal{L}(\boldsymbol{\theta}) = -\frac{1}{|\mathcal{D}_{\text{exp}}|} \sum_{\boldsymbol{\tau} \in \mathcal{D}_{\text{exp}}} R_{\boldsymbol{\theta}}(\boldsymbol{\tau}) + \log \frac{1}{|\mathcal{D}|} \sum_{\boldsymbol{\tau} \in \mathcal{D}} \frac{\exp(R_{\boldsymbol{\theta}}(\boldsymbol{\tau}))}{q(\boldsymbol{\tau})} \tag{35.81}$$

The term inside the log of the loss is an importance sampling estimate of Z that is unbiased as long as $q(\boldsymbol{\tau}) > 0$ for all $\boldsymbol{\tau}$. However, in order to reduce the variance, we can choose q adaptively as $\boldsymbol{\theta}$ is being updated. The optimal sampling distribution (Section 11.5), $q_*(\boldsymbol{\tau}) \propto \exp(R_{\boldsymbol{\theta}}(\boldsymbol{\tau}))$, is hard to obtain. Instead, we may find a policy $\hat{\pi}$ which induces a distribution that is close to q_*, for instance, using methods of maximum entropy RL discussed in Section 35.6.1. Interestingly, the process above produces the inferred reward $R_{\boldsymbol{\theta}}$ as well as an approximate optimal policy $\hat{\pi}$. This approach is used by **guided cost learning** [FLA16], and found effective in robotics applications.

35.6.3.3 Imitation learning by divergence minimization

We now discuss a different, but related, approach to IL. Recall that the reward function depends only on the state and action in an MDP. It implies that if we can find a policy π, so that $p^{\infty}_{\pi}(s, a)$

and $p^{\infty}_{\pi_{\text{exp}}}(s,a)$ are close, then π receives similar long-term reward as π_{exp}, and is a good imitation of π_{exp} in this regard. A number of IL algorithms find π by minimizing the divergence between p^{∞}_{π} and $p^{\infty}_{\pi_{\text{exp}}}$. We will largely follow the exposition of [GZG19]; see [Ke+19b] for a similar derivation.

Let f be a convex function, and $\mathrm{D_f}$ the f-divergence (Section 2.7.1). From the above intuition, we want to minimize $\mathrm{D_f}\left(p^{\infty}_{\pi_{\text{exp}}} \middle\| p^{\infty}_{\pi}\right)$. Then, using a variational approximation of $\mathrm{D_f}$ [NWJ10a], we can solve the following optimization problem for π:

$$\min_{\pi} \max_{\boldsymbol{w}} \mathbb{E}_{p^{\infty}_{\pi_{\text{exp}}}(s,a)}\left[T_{\boldsymbol{w}}(s,a)\right] - \mathbb{E}_{p^{\infty}_{\pi}(s,a)}\left[f^*(T_{\boldsymbol{w}}(s,a))\right] \tag{35.82}$$

where $T_{\boldsymbol{w}} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is a function parameterizd by $\boldsymbol{w}$. The first expectation can be estimated using $\mathcal{D}_{\text{exp}}$, as in behavior cloning, and the second can be estimated using trajectories generated by policy π. Furthermore, to implement this algorithm, we often use a parametric policy representation $\pi_{\boldsymbol{\theta}}$, and then perform stochastic gradient updates to find a saddle-point to Equation (35.82).

With different choices of the convex function f, we can obtain many existing IL algorithms, such as **generative adversarial imitation learning** (**GAIL**) [HE16b] and **adversarial inverse RL** (**AIRL**) [FLL18], as well as new algorithms like **f-Divergence Max-Ent IRL** (f-**MAX**) and **forward adversarial inverse RL** (**FAIRL**) [GZG19; Ke+19b].

Finally, the algorithms above typically require running the learned policy π to approximate the second expectation in Equation (35.82). In risk- or cost-sensitive scenarios, collecting more data is not always possible, Instead, we are in the off-policy IL setting, working with trajectories collected by some policy other than π. Hence, we need to correct the mismatch between p^{∞}_{π} and the off-policy trajectory distribution, for which techniques from Section 35.5 can be used. An example is **ValueDICE** [KNT20], which uses a similar distribution correction method of DualDICE (Section 35.5.2).