Probabilistic Artificial Intelligence

Cristian Perez Jensen

January 20, 2025

Note that these are not the official lecture notes of the course, but only notes written by a student of the course. As such, there might be mistakes. The source code can be found at `github.com/cristianpjensen/eth-cs-notes`. If you find a mistake, please create an issue or open a pull request.

Contents

List of symbols

$\doteq$	Equality by definition
$\approx$	Approximate equality
$\propto$	Proportional to
$\mathbb{N}$	Set of natural numbers
$\mathbb{R}$	Set of real numbers
$i : j$	Set of natural numbers between i and j. *I.e.*, $\{i, i+1, \ldots, j\}$
$f : A \to B$	Function f that maps elements of set A to elements of set B
$\mathbb{1}\{\text{predicate}\}$	Indicator function (1 if predicate is true, otherwise 0)
$\boldsymbol{v} \in \mathbb{R}^n$	n-dimensional vector
$\boldsymbol{M} \in \mathbb{R}^{m \times n}$	$m \times n$ matrix
$\mathbf{T} \in \mathbb{R}^{d_1 \times \cdots \times d_n}$	Tensor
$\boldsymbol{M}^\top$	Transpose of matrix $\boldsymbol{M}$
$\boldsymbol{M}^{-1}$	Inverse of matrix $\boldsymbol{M}$
$\det(\boldsymbol{M})$	Determinant of $\boldsymbol{M}$
$\frac{\mathrm{d}}{\mathrm{d}x} f(x)$	Ordinary derivative of $f(x)$ w.r.t. x at point $x \in \mathbb{R}$
$\frac{\partial}{\partial x} f(\boldsymbol{x})$	Partial derivative of $f(\boldsymbol{x})$ w.r.t. x at point $\boldsymbol{x} \in \mathbb{R}^n$
$\boldsymbol{\nabla}_{\boldsymbol{x}} f(\boldsymbol{x}) \in \mathbb{R}^n$	Gradient of $f : \mathbb{R}^n \to \mathbb{R}$ at point $\boldsymbol{x} \in \mathbb{R}^n$
$\boldsymbol{\nabla}^2_{\boldsymbol{x}} f(\boldsymbol{x}) \in \mathbb{R}^{n \times n}$	Hessian of $f : \mathbb{R}^n \to \mathbb{R}$ at point $\boldsymbol{x} \in \mathbb{R}^n$
$\boldsymbol{\theta} \in \Theta$	Parametrization of a model, where Θ is a compact subset of $\mathbb{R}^K$
$\mathcal{X}$	Input space
$\mathcal{Y}$	Output space
$\mathcal{D} \subseteq \mathcal{X} \times \mathcal{Y}$	Labeled training data

1 Probability review

Probability is formalized by a probability space $(\Omega, \mathcal{F}, P)$, where Ω is a set of atomic events, $\mathcal{F} \subseteq 2^\Omega$ is the set of non-atomic events, and $P : \mathcal{F} \to [0,1]$ is the probability measure that assigns probabilities to events.

The following axioms hold:

$$P(\Omega) = 1 \qquad \text{(Normalization)}$$

$$P(A) \geq 0 \; \forall A \in \mathcal{F} \qquad \text{(Non-negativity)}$$

$$A_1, \ldots, A_n \in \mathcal{F} \wedge \bigcap_{i=1}^{n} A_i = \emptyset \implies P\left(\bigcup_{i=1}^{n} A_i\right) = \sum_{i=1}^{n} P(A_i) \qquad (\sigma\text{-additivity}).$$

1.1 Random variables

Events are cumbersome to work with, so we can define random variables $X : \Omega \to D$ for some set D. Then, we can give a probability to X assuming state x,

$$P(X = x) = P(\{\omega \in \Omega : X(\omega) = x\}).$$

Instead of random variables X, we can also define random vectors $\boldsymbol{X} = [X_1(\omega), \ldots, X_n(\omega)]$. Then, we can specify the joint distribution $P(X_1 = x_1, \ldots, X_n = x_n) = P(\boldsymbol{X} = \boldsymbol{x})$ succinctly.

For random variables, we have the following rules,

- *Product rule*,
$$P(X_{1:n}) = P(X_1) \prod_{i=2}^{n} P(X_i \mid X_{1:i-1});$$

- *Sum rule*,
$$P(X_{1:i-1}, X_{i+1:n}) = \sum_{x_i} P(X_{1:i-1}, X_i = x_i, X_{i+1:n});$$

- *Bayes rule*, where we compute the *posterior* $P(X \mid Y)$ from the *likelihood* $P(Y \mid X)$, *prior* $P(X)$, and *marginal* $P(Y)$,
$$P(X \mid Y) = \frac{P(Y \mid X) P(X)}{P(Y)};$$

- A random variable X is *independent* from Y if the following holds for all values,
$$P_{X_1 \cdots X_n}(x_1, \ldots, x_n) = P_{X_1}(x_1) \cdots P_{X_n}(x_n);$$

- Random variables X and Y are *conditionally independent* given Z if the following holds for all x, y, z,
$$P_{XY|Z}(x, y \mid z) = P_{X|Z}(x \mid z) P_{Y|Z}(y \mid z).$$

1.2 Multivariate Gaussians

Suppose we have n binary variables, then we need $2^n - 1$ parameters.[1,2] Also, if we want to compute the joint distribution over all n variables,[3] we would have to sum up 2^{n-1} terms according to the sum rule. In conclusion, binary random variables scale poorly. Furthermore, we would need a lot of data to estimate the distribution.

[1] -1, because we do not need to specify the last parameter, since it will be whatever is remaining of the total probability.

[2] In other words, the parametrization of the distribution grows exponentially.

[3] *I.e.*, do inference.

The solution to these problems are multivariate Gaussians,

$$\mathcal{N}(\boldsymbol{x}; \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{2\pi\sqrt{\det(\boldsymbol{\Sigma})}} \exp\left(-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\mu})\right),$$

where $\boldsymbol{\mu} \in \mathbb{R}^n$ and $\boldsymbol{\Sigma} \in \mathbb{S}^n_{++}$.[4] Thus, the joint distribution over n Gaussian variables requires only $n^2 + n$ parameters.

[4] $\boldsymbol{\Sigma}$ is an $n \times n$ positive semi-definite matrix.

Let $\boldsymbol{X} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ be a d-dimensional Gaussian random vector, then the following properties hold,

- Let A be an index set, then the marginal distribution of variables indexed by A is the following,

  $$\boldsymbol{X}_A \sim \mathcal{N}(\boldsymbol{\mu}_A, \boldsymbol{\Sigma}_{AA}).$$

  Thus, it is simply a look-up to get a subset marginal distribution;

- Let A and B be index sets, then the marginal distribution of variables indexed by A, conditioned on B, is the following,

  $$\boldsymbol{X}_A \mid \boldsymbol{X}_B \sim \mathcal{N}(\boldsymbol{\mu}_{A|B}, \boldsymbol{\Sigma}_{A|B}),$$

  where

  $$\begin{aligned}\boldsymbol{\mu}_{A|B} &= \boldsymbol{\mu}_A + \boldsymbol{\Sigma}_{AB}\boldsymbol{\Sigma}_{BB}^{-1}(\boldsymbol{x}_B - \boldsymbol{\mu}_B)\\ \boldsymbol{\Sigma}_{A|B} &= \boldsymbol{\Sigma}_{AA} - \boldsymbol{\Sigma}_{AB}\boldsymbol{\Sigma}_{BB}^{-1}\boldsymbol{\Sigma}_{BA}.\end{aligned}$$

  Notice that $\boldsymbol{\Sigma}_{AB}\boldsymbol{\Sigma}_{BB}^{-1}$ removes the interdependencies of B and adds the dependencies of B with A. Further notice that the dependency of A and B added scales linearly with the mean difference $\boldsymbol{x}_B - \boldsymbol{\mu}_B$.

  Further, notice that $\boldsymbol{\Sigma}_{A|B}$ only depends on which random variables are observed, not what values those random variables are, because it does not depend on $\boldsymbol{x}_B$;

- Let $\boldsymbol{M} \in \mathbb{R}^{m \times d}$ be a matrix, then $\boldsymbol{Y} = \boldsymbol{M}\boldsymbol{X}$ is a also a Gaussian,

  $$\boldsymbol{Y} \sim \mathcal{N}\left(\boldsymbol{M}\boldsymbol{\mu}, \boldsymbol{M}\boldsymbol{\Sigma}\boldsymbol{M}^\top\right).$$

  Notice that m is not necessarily equal to d, so we can transform a d-dimensional random vector to any dimensionality m;

- Let $\boldsymbol{X}'$ be another d-dimensional Gaussian, then $\boldsymbol{Y} = \boldsymbol{X} + \boldsymbol{X}'$ is also a Gaussian,

  $$\boldsymbol{Y} \sim \mathcal{N}(\boldsymbol{\mu} + \boldsymbol{\mu}', \boldsymbol{\Sigma} + \boldsymbol{\Sigma}').$$

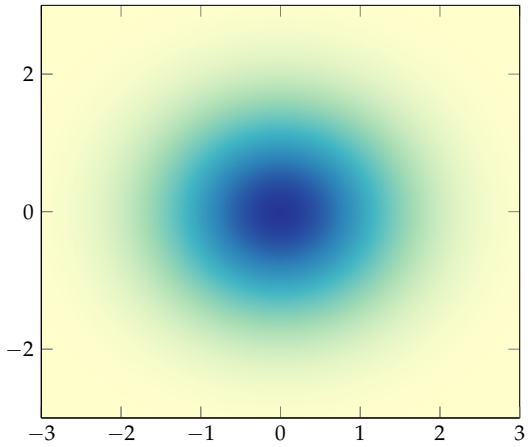


Figure 1.1. Bivariate Gaussian distribution with $\boldsymbol{\Sigma} = \boldsymbol{I}$.

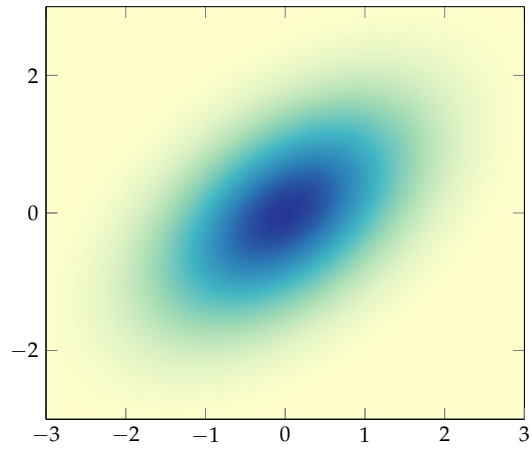


Figure 1.2. Bivariate Gaussian distribution with

$$\boldsymbol{\Sigma} = \begin{bmatrix} 1 & 1/2 \\ 1/2 & 1 \end{bmatrix}.$$

If x_1 increases, the probability of a higher x_2 increases as well.

1.3 *Kalman filters*

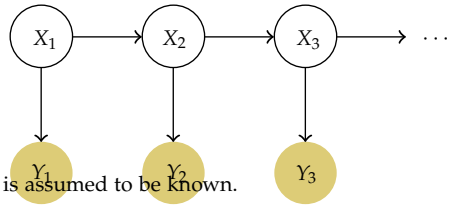
Definition 1.1 (Kalman filter)**.** A Kalman filter is specified by a Gaussian prior over the states,

$$X_0 \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma}),$$

and a conditional linear Gaussian *motion model* and *sensor model*,

$$\begin{aligned} X_{t+1} &\doteq \boldsymbol{F} X_t + \boldsymbol{\epsilon}_t & \boldsymbol{F} &\in \mathbb{R}^{d \times d}, & \boldsymbol{\epsilon}_t &\sim \mathcal{N}(\mathbf{0}, \boldsymbol{\Sigma}_x) \\ Y_t &\doteq \boldsymbol{H} X_t + \boldsymbol{\eta}_t & \boldsymbol{H} &\in \mathbb{R}^{m \times d}, & \boldsymbol{\eta}_t &\sim \mathcal{N}(\mathbf{0}, \boldsymbol{\Sigma}_y), \end{aligned}$$

respectively.



$\boldsymbol{F}$ is assumed to be known.

$\boldsymbol{H}$ is assumed to be known.

Figure 1.3. Directed graphical model of a Kalman filter with hidden states $\boldsymbol{X}_t$ and observables $\boldsymbol{Y}_t$.

From the directed graphical model in Figure 1.3, we can observe the following conditional independences,

$$\begin{aligned} X_{t+1} \perp \boldsymbol{X}_{1:t-1}, \boldsymbol{Y}_{1:t-1} \mid X_t \\ Y_t \perp \boldsymbol{X}_{1:t-1} \mid X_t \\ Y_t \perp \boldsymbol{Y}_{1:t-1} \mid X_{t-1}. \end{aligned}$$

These lead to the following factorization of the joint distribution,

$$\begin{aligned} p(\boldsymbol{x}_{1:t}, \boldsymbol{y}_{1:t}) &= \prod_{i=1}^{t} p(x_i \mid \boldsymbol{x}_{1:i-1}) p(y_i \mid \boldsymbol{x}_{1:t}, \boldsymbol{y}_{1:i-1}) \\ &= p(x_1) p(y_1 \mid x_1) \prod_{i=2}^{t} p(x_i \mid x_{i-1}) p(y_i \mid x_i). \end{aligned}$$

We now want to do *Bayesian filtering* on a Kalman filter, which involves keeping tack of an agent's state using noisy observations Y. It is described by the recursive scheme in Figure 1.4.

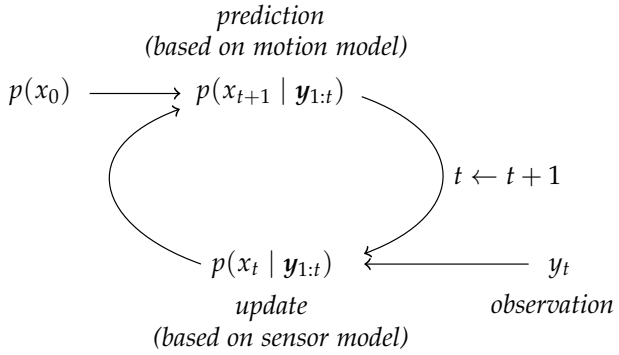


Figure 1.4. The recursive scheme of Bayesian filtering.

We can do the update by the following,

$$\begin{aligned} p(x_t \mid \boldsymbol{y}_{1:t}) &= \frac{1}{Z} p(x_t \mid \boldsymbol{y}_{1:t-1}) p(y_t \mid x_t, \boldsymbol{y}_{1:t-1}) \\ &= \frac{1}{Z} p(x_t \mid \boldsymbol{y}_{1:t-1}) p(y_t \mid x_t). \end{aligned}$$

Furthermore, we can do the prediction by the following,

$$\begin{aligned} p(x_{t+1} \mid \boldsymbol{y}_{1:t}) &= \int p(x_{t+1}, x_t \mid y_{1:t}) dx_t \\ &= \int p(x_{t+1} \mid x_t, \boldsymbol{y}_{1:t}) p(x_t \mid \boldsymbol{y}_{1:t}) dx_t \\ &= \int p(x_{t+1} \mid x_t) p(x_t \mid \boldsymbol{y}_{1:t}) dx_t. \end{aligned}$$

In general, these distributions are very complicated, but for Gaussians (as in the Kalman filter), they can be expressed in closed form. The general formula for the *Kalman update* is as follows, given the prior belief $X_t \mid \boldsymbol{y}_{1:t} \sim \mathcal{N}(\boldsymbol{\mu}_t, \boldsymbol{\Sigma}_t)$,

$$\begin{aligned} X_{t+1} \mid \boldsymbol{y}_{1:t+1} &\sim \mathcal{N}(\boldsymbol{\mu}_{t+1}, \boldsymbol{\Sigma}_{t+1}) \\ \boldsymbol{\mu}_{t+1} &\doteq \boldsymbol{F}\boldsymbol{\mu}_t + \boldsymbol{K}_{t+1}(\boldsymbol{y}_{t+1} - \boldsymbol{H}\boldsymbol{F}\boldsymbol{\mu}_t) \\ \boldsymbol{\Sigma}_{t+1} &\doteq (\boldsymbol{I} - \boldsymbol{K}_{t+1}\boldsymbol{H})(\boldsymbol{F}\boldsymbol{\Sigma}_t\boldsymbol{F}^\top + \boldsymbol{\Sigma}_x), \end{aligned}$$

where $\boldsymbol{K}_{t+1}$ is the *Kalman gain*,

$$\boldsymbol{K}_{t+1} \doteq (\boldsymbol{F}\boldsymbol{\Sigma}_t\boldsymbol{F}^\top + \boldsymbol{\Sigma}_x)\boldsymbol{H}^\top(\boldsymbol{H}(\boldsymbol{F}\boldsymbol{\Sigma}_t\boldsymbol{F}^\top + \boldsymbol{\Sigma}_x)\boldsymbol{H}^\top + \boldsymbol{\Sigma}_y)^{-1}.$$

The term $(\boldsymbol{y}_{t+1} - \boldsymbol{H}\boldsymbol{F}\boldsymbol{\mu}_t)$ measures the error in the predicted observation and the Kalman gain $\boldsymbol{K}_{t+1}$ measures the relevance of the new observation compared to the prediction.

1.4 Entropy

Definition 1.2 (Entropy)**.** *Entropy* measures the expected surprisal of a distribution p,

$$H[p] \doteq \mathbb{E}_{x \sim p}[-\log p(x)].$$

$-\log p(x)$ is also called the surprisal value of x, because it decreases as the probability grows, and is exactly 0 if the probability is 1.

Definition 1.3 (Kullback-Leibler divergence)**.** *Kullback-Leibler divergence* (KL divergence) is a common metric that measures dissimilarity between two distributions p and q,

$$KL(p \| q) \doteq \mathbb{E}_{\theta \sim p}\left[\log \frac{p(\theta)}{q(\theta)}\right].$$

Intuitively, $KL(p \| q)$ measures the information loss when approximating p with q.

Definition 1.4 (Mutual information)**.** Given random variables X and Y, the mutual information $I(X;Y)$ quantifies how much observing Y reduces uncertainty about X, as measured by its entropy, in expectation over Y.

$$I(X;Y) \doteq H[X] - H[X \mid Y],$$

where $H[X]$ and $H[X \mid Y]$ quantify the uncertainty about X before and after observing Y.

Properties (Mutual information)**.** Mutual information is symmetric,

$$I(X;Y) = I(Y;X).$$

Mutual information is positive (*information never hurts*),

$$I(X;Y) \geq 0.$$

Example 1.5 (Mutual information of noisy Gaussian observations)**.** Let

$$\begin{aligned} \boldsymbol{X} &\sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma}) \\ \boldsymbol{Y} &= \boldsymbol{X} + \boldsymbol{\epsilon}, \qquad \boldsymbol{\epsilon} \sim \mathcal{N}(\boldsymbol{0}, \sigma_n^2 \boldsymbol{I}). \end{aligned}$$

Then,

$$\begin{aligned} I(\boldsymbol{X};\boldsymbol{Y}) &= H[\boldsymbol{Y}] - H[\boldsymbol{Y} \mid \boldsymbol{X}] \\ &= H[\boldsymbol{Y}] - H[\boldsymbol{\epsilon}] \\ &= \frac{1}{2}\log(2\pi e)^d \det\Big(\boldsymbol{\Sigma} + \sigma_n^2 \boldsymbol{I}\Big) - \frac{1}{2}\log(2\pi e)^d(\sigma_n^2 \boldsymbol{I}) \\ &= \frac{1}{2}\log\det\Big(\boldsymbol{I} + \sigma_n^{-2}\boldsymbol{\Sigma}\Big). \end{aligned}$$

Definition 1.6 (Conditional mutual information)**.**

$$\begin{aligned} I(X;Y \mid Z) &\doteq H[X \mid Z] - H[X \mid Y, Z] \\ &= I(X;Y,Z) - I(X;Z). \end{aligned}$$

2 *Bayesian linear regression*

Bayesian linear regression (BLR) is a model that is able to provide an uncertainty measure about its predictions due to a lack of data.[5] It does so by outputting a probability distribution over possible outputs $y^\star$ given input $\boldsymbol{x}^\star$. The variance of this distribution measures the uncertainty of the model at this point and the mode corresponds to its best estimate. It does this by not considering a single set of weights, but rather all possible weights, assigning each a probability via Bayes rule. Recall the posterior,

[5] This is called epistemic uncertainty.

$$p(\boldsymbol{w} \mid \boldsymbol{X}, \boldsymbol{y}) \propto p(\boldsymbol{y} \mid \boldsymbol{w}, \boldsymbol{X}) \cdot p(\boldsymbol{w}, \boldsymbol{X}),$$

where we can compute the maximum a posteriori (MAP) estimate and use that as our weights, multiplying it with the input $\boldsymbol{x}^\star$. However, by considering every plausible function and giving it a probability according to how well it models the data, we can compute the uncertainty of the model. Instead of using only the mode of the posterior, we will use the full posterior of $\boldsymbol{w}$ by taking the integral over all of them. This allows us to assign a probability distribution to any point $(\boldsymbol{x}^\star, y^\star)$,

$$\begin{aligned} p(y^\star \mid \boldsymbol{x}^\star, \boldsymbol{X}, \boldsymbol{y}) &= \int p(\boldsymbol{w}, y^\star \mid \boldsymbol{x}^\star, \boldsymbol{X}, \boldsymbol{y}) d\boldsymbol{w} && \text{Sum rule} \\ &= \int p(\boldsymbol{w} \mid \boldsymbol{x}^\star, \boldsymbol{X}, \boldsymbol{y}) \cdot p(y^\star \mid \boldsymbol{w}, \boldsymbol{x}^\star, \boldsymbol{X}, \boldsymbol{y}) d\boldsymbol{w} && \text{Product rule} \\ &= \int p(\boldsymbol{w} \mid \boldsymbol{X}, \boldsymbol{y}) \cdot p(y^\star \mid \boldsymbol{w}, \boldsymbol{x}^\star) d\boldsymbol{w}. \end{aligned}$$

Notice that each model $\boldsymbol{w}$'s prediction $p(y^\star \mid \boldsymbol{w}, \boldsymbol{x}^\star)$ is weighted by its probability $p(\boldsymbol{w} \mid \boldsymbol{X}, \boldsymbol{y})$. In general, this integral is intractable, but by assuming that the prior and likelihood are independently Gaussian,

$$\begin{aligned} \boldsymbol{w} &\sim \mathcal{N}(\boldsymbol{0}, \sigma_p^2 \boldsymbol{I}) \\ y_i \mid \boldsymbol{w}, \boldsymbol{x}_i &\sim \mathcal{N}(\boldsymbol{w}^\top \boldsymbol{x}_i, \sigma_n^2), \end{aligned}$$

it becomes tractable. Firstly, the posterior is given by the following,[6]

[6] Notice that $\bar{\boldsymbol{\Sigma}}$ does not depend on $\boldsymbol{y}$. Since the covariance matrix measures the uncertainty, this tells us that the uncertainty only depends on where we observed data, not what we observed. Intuitively, this makes a lot of sense.

$$\begin{aligned} \boldsymbol{w} \mid \boldsymbol{X}, \boldsymbol{y} &\sim \mathcal{N}(\bar{\boldsymbol{\mu}}, \bar{\boldsymbol{\Sigma}}) \\ \bar{\boldsymbol{\mu}} &= (\boldsymbol{X}^\top \boldsymbol{X} + \sigma_n^2 \boldsymbol{I})^{-1} \boldsymbol{X}^\top \boldsymbol{y} \\ \bar{\boldsymbol{\Sigma}} &= (\sigma_n^{-2} \boldsymbol{X}^\top \boldsymbol{X} + \boldsymbol{I})^{-1}. \end{aligned}$$

By taking advantage of the Gaussian distribution properties, we can compute the distribution of $y^\star$ given a point $\boldsymbol{x}^\star$. Let's say $f^\star = \boldsymbol{w}^\top \boldsymbol{x}^\star = \boldsymbol{x}^{\star\top} \boldsymbol{w}$, then

$$f^\star \mid \boldsymbol{x}^\star, \boldsymbol{X}, \boldsymbol{y} \sim \mathcal{N}(\boldsymbol{x}^{\star\top} \bar{\boldsymbol{\mu}}, \boldsymbol{x}^{\star\top} \bar{\boldsymbol{\Sigma}} \boldsymbol{x}^\star).$$

Adding aleatoric noise $y^\star = f^\star + \epsilon_n$ results in the following,

$$y^\star \mid \boldsymbol{x}^\star, \boldsymbol{X}, \boldsymbol{y} \sim \mathcal{N}(\boldsymbol{x}^{\star\top} \bar{\boldsymbol{\mu}}, \boldsymbol{x}^{\star\top} \bar{\boldsymbol{\Sigma}} \boldsymbol{x}^\star + \sigma_n^2).$$

The epistemic uncertainty $\bar{\boldsymbol{\Sigma}}$ is uncertainty about the model due to a lack of data, while the aleatoric uncertainty σ_n^2 is irreducible noise that is always present in data.

3 Gaussian processes

BLR can only make linear predictions, because it is linear in the parameters. However, we might want to make non-linear predictions. We could apply BLR on non-linearly transformed data,[7] but the computational cost would increase with the dimensionality of the feature space.

An alternative way of looking at it is considering inference directly in function space. We use *Gaussian processes* (GP) to describe distributions over functions. They are formally defined as an infinite collection of random variables, of which any finite number have a joint Gaussian distribution. GPs are specified by a mean function $\mu : \mathcal{X} \to \mathbb{R}$ and a kernel function $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$, and written as the following,[8]

$$f \sim \mathcal{GP}(\mu, k).$$

Now, we would want a finite Gaussian distribution over a marginal $\{\boldsymbol{x}_1, \dots, \boldsymbol{x}_m\} \subseteq \mathcal{X}$ (stored by matrix $\boldsymbol{X}$) of the infinite collection of random variables $\mathcal{X}$. If we assume a normal prior on the weights,

$$\boldsymbol{w} \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{I}),$$

then the distribution over $\boldsymbol{f} = \boldsymbol{w}^\top \boldsymbol{X}^\top = \boldsymbol{X}\boldsymbol{w}$ becomes the following,

$$\boldsymbol{f} \sim \mathcal{N}\Big(\boldsymbol{X}^\top \boldsymbol{0}, \boldsymbol{X}^\top \boldsymbol{I} \boldsymbol{X}\Big) = \mathcal{N}\Big(\boldsymbol{0}, \boldsymbol{X}^\top \boldsymbol{X}\Big).$$

Notice that the data points enter as inner products, thus we do not necessarily need to let them linearly depend. We could also use any kernel function,[9]

$$\boldsymbol{f} \sim \mathcal{N}(\boldsymbol{0}, k(\boldsymbol{X}, \boldsymbol{X}))$$

with

$$k(\boldsymbol{X}, \boldsymbol{X}) = \begin{bmatrix} k(\boldsymbol{x}_1, \boldsymbol{x}_1) & \cdots & k(\boldsymbol{x}_1, \boldsymbol{x}_m) \\ \vdots & \ddots & \vdots \\ k(\boldsymbol{x}_m, \boldsymbol{x}_1) & \cdots & k(\boldsymbol{x}_m, \boldsymbol{x}_m) \end{bmatrix}.$$

We can sample function realizations from a Gaussian process, by taking n equidistant points from $\mathcal{X}$ as matrix $\boldsymbol{X} \in \mathbb{R}^{n \times d}$. Then, we assume $\mu(\boldsymbol{x}) = 0$ and compute $k(\boldsymbol{X}, \boldsymbol{X})$ to make a multivariate Gaussian distribution over $\boldsymbol{f} \in \mathbb{R}^n$. Lastly, we sample a vector from this multivariate Gaussian distribution and interpolate between the points to form the function realization. Notice that GPs parametrize a probability distribution over functions.

Figure 3.1 shows samples of a prior Gaussian process with the periodic kernel,

$$k(\boldsymbol{x}, \boldsymbol{x}') = \sigma^2 \exp\left(-\frac{2}{\ell^2} \sin^2\left(\pi \frac{|\boldsymbol{x} - \boldsymbol{x}'|}{p}\right)\right),$$

where σ^2 is the overall variance, ℓ is the lengthscale, and p is the period. It also shows the periodic kernel function w.r.t. $x = 0$. As can be seen,

[7] This would mean redefining $f(\boldsymbol{x})$ to

$$f(\boldsymbol{x}) = \boldsymbol{\phi}(\boldsymbol{x})^\top \boldsymbol{w},$$

with *e.g.*

$$\boldsymbol{\phi}(x) = \begin{bmatrix} 1 & x & x^2 \end{bmatrix}^\top.$$

[8] In other words, we sample functions from a GP defined by a mean and kernel function.

[9] Formally, $k(\boldsymbol{x}, \boldsymbol{x}') = \boldsymbol{\phi}(\boldsymbol{x})^\top \boldsymbol{\phi}(\boldsymbol{x}')$ for some feature function $\boldsymbol{\phi}$. But, using a kernel function k instead makes it more computationally efficient, because the dimensionality of the Gaussian does not scale with the output dimensionality of the feature function.

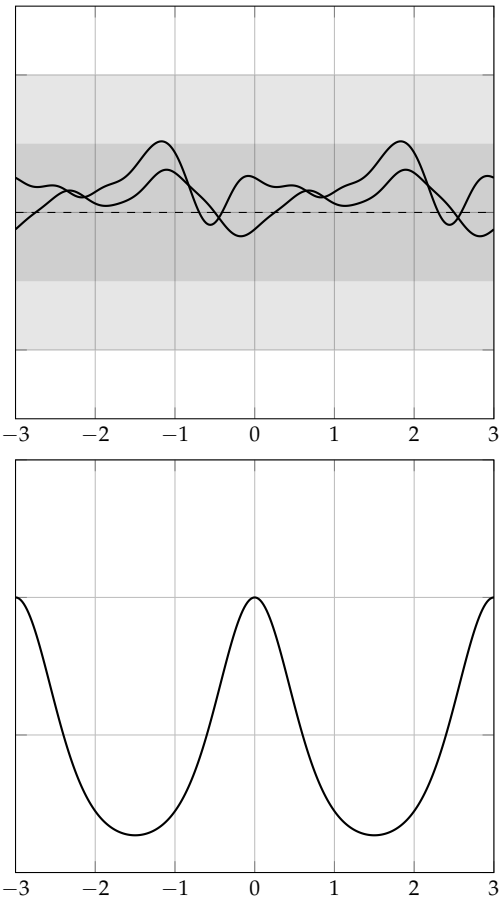


Figure 3.1. A priori samples of a Gaussian process with the periodic kernel. The second plot shows the kernel function w.r.t. $x = 0$.

pairs of points (x, x') with high covariance $k(x, x')$ have close values in all sampled function realizations. This makes sense, since k parametrizes the covariance.

3.1 Learning and inference

Adding aleatoric noise, we define the observed data $\boldsymbol{y}$ to have the following distribution,

$$\boldsymbol{y} \sim \mathcal{N}(\boldsymbol{0}, k(\boldsymbol{X}, \boldsymbol{X}) + \sigma_n^2 \boldsymbol{I}).$$

Now, suppose that we observe data $\boldsymbol{y}$ for datapoints $\boldsymbol{X}$, and want to predict the probability distribution of $y^\star$ for $\boldsymbol{x}^\star$ given the observed data. We can define the following a priori joint distribution,[10]

$$\begin{bmatrix} \boldsymbol{y} \\ f^\star \end{bmatrix} \mid \boldsymbol{x}^\star, \boldsymbol{X} \sim \mathcal{N}\left(\boldsymbol{0}, \begin{bmatrix} k(\boldsymbol{X}, \boldsymbol{X}) + \sigma_n^2 \boldsymbol{I} & k(\boldsymbol{X}, \boldsymbol{x}^\star) \\ k(\boldsymbol{x}^\star, \boldsymbol{X}) & k(\boldsymbol{x}^\star, \boldsymbol{x}^\star) \end{bmatrix}\right),$$

where

$$k(\boldsymbol{X}, \boldsymbol{x}) = \begin{bmatrix} k(\boldsymbol{x}_1, \boldsymbol{x}) \\ \vdots \\ k(\boldsymbol{x}_m, \boldsymbol{x}) \end{bmatrix}.$$

Then, we can derive the conditional distribution, using the conditional property of multivariate Gaussian distributions, as

$$f^\star \mid \boldsymbol{x}^\star, \boldsymbol{X}, \boldsymbol{y} \sim \mathcal{N}(\mu^\star, k^\star),$$

where

$$\begin{aligned} \mu^\star &= k(\boldsymbol{x}^\star, \boldsymbol{X})(k(\boldsymbol{X}, \boldsymbol{X}) + \sigma_n^2 \boldsymbol{I})^{-1} \boldsymbol{y} \\ k^\star &= k(\boldsymbol{x}^\star, \boldsymbol{x}^\star) - k(\boldsymbol{x}^\star, \boldsymbol{X})(k(\boldsymbol{X}, \boldsymbol{X}) + \sigma_n^2 \boldsymbol{I})^{-1} k(\boldsymbol{X}, \boldsymbol{x}^\star). \end{aligned}$$

Adding aleatoric noise, we get the probability distribution over $y^\star$:

$$y^\star \mid \boldsymbol{x}^\star, \boldsymbol{X}, \boldsymbol{y} \sim \mathcal{N}(\mu^\star, k^\star + \sigma_n^2).$$

3.2 Kernel functions

Suppose we have two covariance functions,

$$k_1 : \mathcal{X} \times \mathcal{X} \to \mathbb{R}, \quad k_2 : \mathcal{X} \times \mathcal{X} \to \mathbb{R},$$

$c > 0$, and f is a polynomial with positive coefficients or the exponential function. Then, the following functions are valid covariance functions,

$$\begin{aligned} k(\boldsymbol{x}, \boldsymbol{x}') &= k_1(\boldsymbol{x}, \boldsymbol{x}') + k_2(\boldsymbol{x}, \boldsymbol{x}') \\ k(\boldsymbol{x}, \boldsymbol{x}') &= k_1(\boldsymbol{x}, \boldsymbol{x}') \cdot k_2(\boldsymbol{x}, \boldsymbol{x}') \\ k(\boldsymbol{x}, \boldsymbol{x}') &= c \cdot k_1(\boldsymbol{x}, \boldsymbol{x}') \\ k(\boldsymbol{x}, \boldsymbol{x}') &= f(k_1(\boldsymbol{x}, \boldsymbol{x}')). \end{aligned}$$

[10] Notice that this is a priori because we have not observed (conditioned on) any data points yet. We are only specifying the joint distribution over $\boldsymbol{y}$ and $f^\star$.

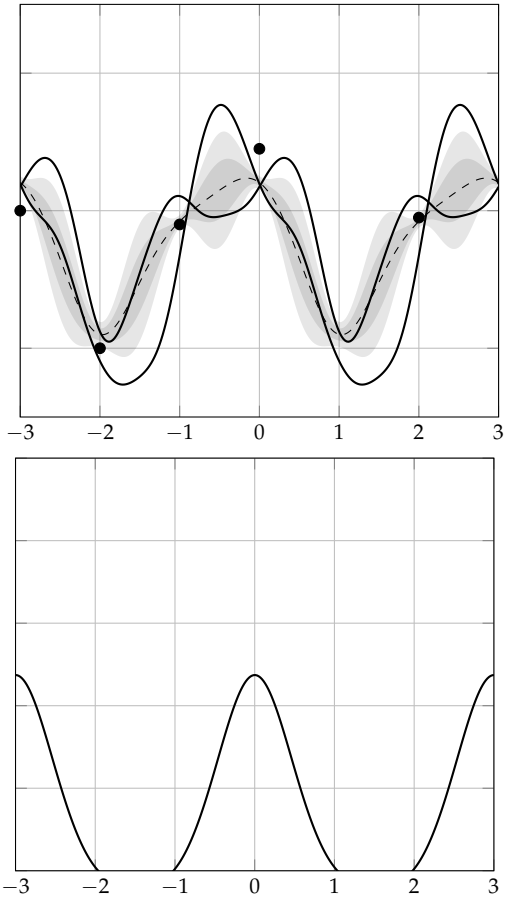


Figure 3.2. The periodic kernel function hyperparameters used are $\sigma^2 = 1$, $\ell = 1$, and $p = 3$. The second plot shows the covariance w.r.t. $x = 0$.

Definition 3.1 (Stationary and isotropic kernels)**.** A kernel function k is *stationary* if its function only depends on the difference between its arguments, *i.e.*, $k(\boldsymbol{x}, \boldsymbol{x}') = k(\boldsymbol{x} - \boldsymbol{x}')$. It is *isotropic* if it only depends on the ℓ_2-distance between its arguments, *i.e.*, $k(\boldsymbol{x}, \boldsymbol{x}') = k(\|\boldsymbol{x} - \boldsymbol{x}'\|_2)$

The following is a list of popular kernels,

- *Linear kernel*,
$$k(\boldsymbol{x}, \boldsymbol{x}') = \boldsymbol{x}^\top \boldsymbol{x}' + \sigma_0^2.$$
Its posterior can be seen in Figure 3.3;

- *Gaussian kernel* (a.k.a. *Squared Exponential* or *RBF*),
$$k(\boldsymbol{x}, \boldsymbol{x}') = \exp\left(-\frac{\|\boldsymbol{x} - \boldsymbol{x}'\|_2^2}{2\ell^2}\right).$$
Points that are close together have a high covariance, while points further away have a lower one. This is what makes it smooth, and continuous. Its posterior can be seen in Figure 3.4;

- *Exponential kernel*,
$$k(\boldsymbol{x}, \boldsymbol{x}') = \exp\left(-\frac{\|\boldsymbol{x} - \boldsymbol{x}'\|}{\ell}\right).$$
Points that are close together have a high covariance. However, because it uses the ℓ_1-distance, the covariance function is not smooth, but "linear on two sides". This is why it has so many peaks. Its posterior can be seen in Figure 3.5.

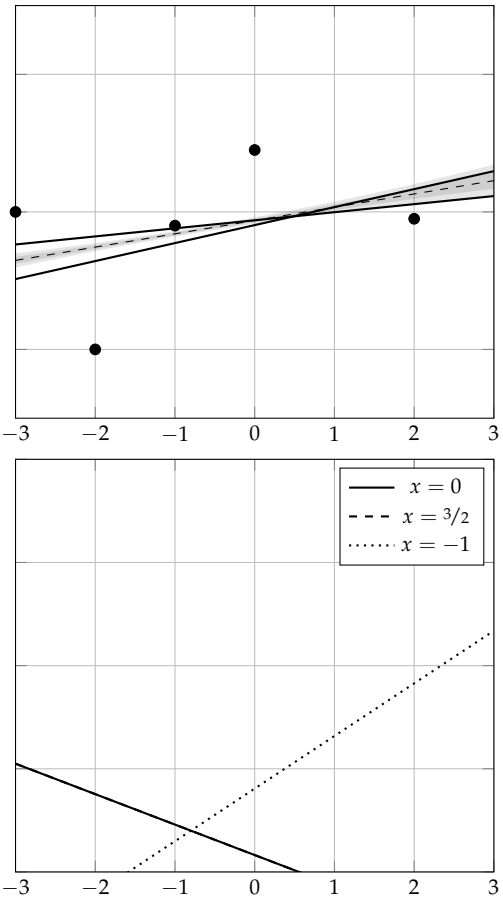


Figure 3.3. Posterior linear kernel. The second plot shows the covariance w.r.t. several points.

3.3 Model selection

As can be seen in Figures 3.2 to 3.5, the hyperparameters matter a lot for whether a GP can model the datapoints correctly. These hyperparameters can be learned by maximizing its predictive performance on the data. (Hyperparameter fitting on the training data does not cause overfitting. This will become clear.)

Suppose we have data $\{(x_i^\star, y_i^\star)\}_{i=1}^n$, then we would like to choose hyperparameters, such that the performance on this data is maximized. There are several choices for measuring predictive performance. The most naive option is mean squared error,

$$\mathrm{MSE}_\theta(y^\star, \boldsymbol{x}^\star) = (y^\star - \mu_\theta^\star(\boldsymbol{x}^\star))^2.$$

This metric ignores aleatoric (σ_n^2) and epistemic ($k(\boldsymbol{x}^\star, \boldsymbol{x}^\star)$) uncertainty, thus it will not work well for measuring the performance of GPs. Another option is to just add the variance to the loss, so it also will be minimized,

$$\tilde{\mathrm{MSE}}_\theta(y^\star, \boldsymbol{x}^\star) = \mathrm{MSE}_\theta(y^\star, \boldsymbol{x}^\star) + k_\theta^\star(\boldsymbol{x}^\star, \boldsymbol{x}^\star).$$

The problem with this is that it encourages low epistemic uncertainty which will cause the loss function to favor models that have low epistemic uncertainty without it necessarily being true. Furthermore, it still ignores the aleatoric uncertainty.

The Bayesian perspective provides an alternative approach: maximizing the likelihood of the data,

$$\begin{aligned}\ell\ell_\theta(y^\star, \boldsymbol{x}^\star) &= \mathcal{N}(y^\star; f_\theta^\star(\boldsymbol{x}^\star), \sigma_n^2) \\ &= \mathcal{N}(y^\star; \mu_\theta^\star, k_\theta^\star(\boldsymbol{x}^\star, \boldsymbol{x}^\star) + \sigma_n^2).\end{aligned}$$

We can optimize θ as follows,

$$\begin{aligned}\hat{\theta} &= \operatorname*{argmax}_\theta p(\boldsymbol{y}^\star \mid \boldsymbol{X}^\star, \theta) \\ &= \operatorname*{argmax}_\theta \int p(\boldsymbol{y}^\star \mid f, \boldsymbol{X}^\star, \theta) p(f \mid \theta) df \\ &= \operatorname*{argmax}_\theta \mathcal{N}(\boldsymbol{y}^\star; \boldsymbol{0}, k(\boldsymbol{X}^\star, \boldsymbol{X}^\star) + \sigma_n^2 \boldsymbol{I}) \\ &= \operatorname*{argmin}_\theta -\log \mathcal{N}(\boldsymbol{y}^\star; \boldsymbol{0}, k(\boldsymbol{X}^\star, \boldsymbol{X}^\star) + \sigma_n^2 \boldsymbol{I}) \\ &= \operatorname*{argmin}_\theta \frac{n}{2}\log 2\pi + \frac{1}{2}\log\det\Big(k(\boldsymbol{X}^\star, \boldsymbol{X}^\star) + \sigma_n^2 \boldsymbol{I}\Big) + \frac{1}{2}\boldsymbol{y}^{\star\top}(k(\boldsymbol{X}^\star, \boldsymbol{X}^\star) + \sigma_n^2 \boldsymbol{I})^{-1}\boldsymbol{y}^\star \\ &= \operatorname*{argmin}_\theta \underbrace{\frac{1}{2}\log\det\Big(k(\boldsymbol{X}^\star, \boldsymbol{X}^\star) + \sigma_n^2 \boldsymbol{I}\Big)}_{\text{complexity penalty}} + \underbrace{\frac{1}{2}\boldsymbol{y}^{\star\top}(k(\boldsymbol{X}^\star, \boldsymbol{X}^\star) + \sigma_n^2 \boldsymbol{I})^{-1}\boldsymbol{y}^\star}_{\text{“goodness“ of fit}}.\end{aligned}$$

As can be seen, the loss function seeks a balance between the “goodness“ of the fit and complexity. If we increase the aleatoric uncertainty σ_n^2, we increase the goodness of the fit, but increase the complexity. This is how it prevents over- and underfitting and is the reason why we do not need a validation dataset.

3.4 *Efficiency*

The time complexity of computing the posterior is $\Theta(n^3)$[11] and the space complexity of storing $k(\boldsymbol{X}, \boldsymbol{X})$ is $\Theta(n^2)$. The main approaches for accelerating GP posterior computation are exploiting parallelism (GPU),[12] local GP methods, kernel function approximations, and inducing point methods.

[11] For BLR, this is $\mathcal{O}(dn^2)$.

[12] This yields a significant speedup, but does not address the cubic scaling in n.

Local GP methods. The basic idea is that, for covariance functions that decay with distance,[13] we only need to condition on close points to $\boldsymbol{x}$. *I.e.*, to make a prediction at point $\boldsymbol{x}$, we only need to condition on points $\boldsymbol{x}'$ where $|k(\boldsymbol{x}, \boldsymbol{x}')| \geq \tau$ for some threshold τ. The problem with this method is that it is still expensive if there are many close points.

[13] Think of stationary kernels such as the Gaussian and exponential kernels.

Kernel function approximation. The key idea of approximating kernel functions is that we can construct an m-dimensional feature map with

$m \ll n$ that approximates the true kernel function,

$$k(\boldsymbol{x}, \boldsymbol{x}') \approx \boldsymbol{\phi}(\boldsymbol{x})^\top \boldsymbol{\phi}(\boldsymbol{x}') \quad \boldsymbol{\phi}(\boldsymbol{x}) \in \mathbb{R}^m.$$

Then, apply BLR. The computational cost becomes $\mathcal{O}\left(nm^2 + m^3\right)$ instead of $\mathcal{O}\left(n^3\right)$.

An example is Random Fourier Features (RFF) that reduces stationary kernels to their Fourier transform. Then, samples from this m times and defines it as the feature map $\boldsymbol{\phi}$. The problem with RFFs is that they approximate the kernel function globally, however this might not be necessary, since we only need accurate representation for the training and test points.

Inducing point methods. The idea behind inducing point methods is that we won't need all data. In areas where there are a lot of data points, we can safely throw some away. This method needs to figure out which points are safe to throw away, *i.e.*, find inducing points $\mathcal{U}$ that we need to keep to approximate well. This can be done by choosing them randomly or we could treat $\mathcal{U}$ as hyperparameters and maximize the marginal likelihood of the data.

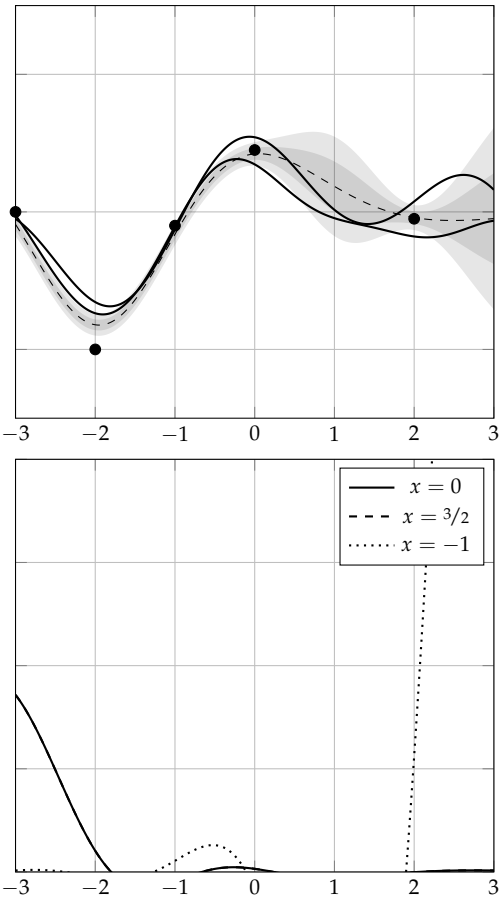


Figure 3.4. Posterior Gaussian kernel. The second plot shows the covariance w.r.t. several points.

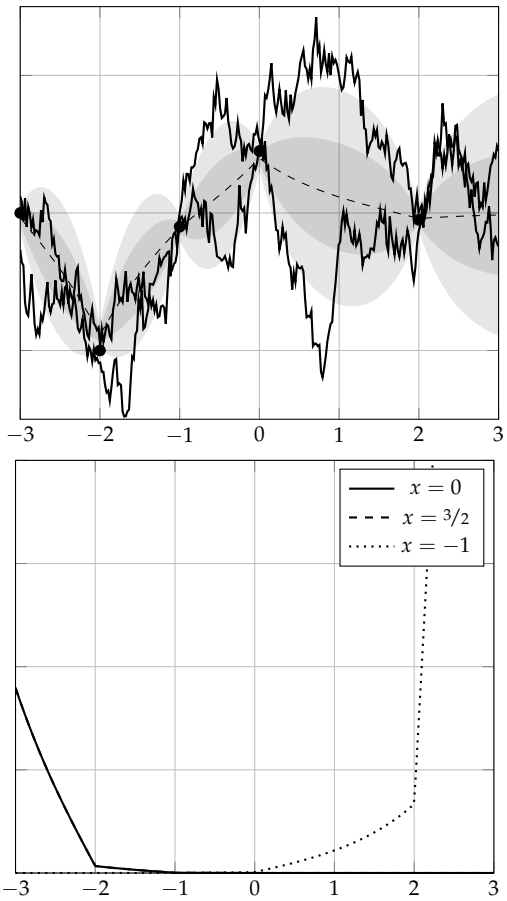


Figure 3.5. Posterior exponential kernel. The second plot shows the covariance w.r.t. several points.

4 *Variational inference*

Remark. From now on, $\boldsymbol{X}$ will be omitted and treated as a constant.

In Bayesian learning, we want to compute the following probability distribution to make predictions given the data,

$$p(y^\star \mid \boldsymbol{x}^\star, \boldsymbol{y}) = \int p(y^\star \mid \boldsymbol{x}^\star, \theta) p(\theta \mid \boldsymbol{y}) d\theta,$$

where we marginalize over all possible models θ. Furthermore, we compute the posterior as follows,

$$p(\theta \mid \boldsymbol{y}) = \frac{1}{Z} p(\theta) \prod_{i=1}^{n} p(y_i \mid \boldsymbol{x}_i, \theta).$$

However, in general, these equations are intractable.[14] Gaussian processes solved this problem by assuming that the prior and likelihood are Gaussian.[15] But, in some cases, it is not realistic to assume Gaussian distributions.[16] *Variational inference* solves this problem by approximating the intractable distribution p by a simpler one q that is "as close as possible",

$$p(\theta \mid \boldsymbol{y}) = \frac{1}{Z} p(\theta, \boldsymbol{y}) \approx q(\theta \mid \lambda) = q_\lambda(\theta),$$

where λ are called the variational parameters. Thus, we have reduced the problem to optimization, *i.e.*, maximizing the similarity between distributions p and q, where q is part of a variational family $\mathcal{Q}$ that is easy to work with.

[14] The integral is intractable, because distributions are not conjugate in general. The posterior is intractable, because of the normalizer Z.

[15] Because the Guassian has a conjugate prior.

[16] *E.g.* in logistic regression, the likelihood is modeled by a Bernoulli distribution.

Example 4.1 (Laplace approximation)**.** A simple way of approximating intractable integrals is *Laplace approximation* which is a Gaussian approximation to the posterior. Let's define a function $\psi(\theta) \doteq \log p(\theta \mid \boldsymbol{y})$, then the Laplace approximation of ψ can be computed from the second-order Taylor expansion around the posterior mode,

$$\begin{aligned}\psi(\theta) &\approx \psi(\hat{\theta}) + (\theta - \hat{\theta})^\top \nabla \psi(\hat{\theta}) + \frac{1}{2}(\theta - \hat{\theta})^\top \boldsymbol{H}_\psi(\hat{\theta})(\theta - \hat{\theta}) \\ &= \psi(\hat{\theta}) + \frac{1}{2}(\theta - \hat{\theta})^\top \boldsymbol{H}_\psi(\hat{\theta})(\theta - \hat{\theta}).\end{aligned}$$

Then,

$$\begin{aligned}p(\theta \mid \boldsymbol{y}) &= \exp(\psi(\theta)) \\ &\approx \exp\left(\psi(\hat{\theta}) + \frac{1}{2}(\theta - \hat{\theta})^\top \boldsymbol{H}_\psi(\hat{\theta})(\theta - \hat{\theta})\right) \\ &= \exp(\psi(\hat{\theta})) \cdot \exp\left(\frac{1}{2}(\theta - \hat{\theta})^\top \boldsymbol{H}_\psi(\hat{\theta})(\theta - \hat{\theta})\right) \\ &= \frac{1}{Z} \cdot \exp\left(-\frac{1}{2}(\theta - \hat{\theta})^\top \boldsymbol{H}_\psi^{-1}(\hat{\theta})(\theta - \hat{\theta})\right) \\ &= \mathcal{N}(\theta; \hat{\theta}, \boldsymbol{\Lambda}^{-1}) \\ &\doteq q(\theta),\end{aligned}$$

where $\hat{\theta} = \operatorname{argmax}_\theta p(\theta \mid \boldsymbol{y})$ and $\boldsymbol{\Lambda} = -\boldsymbol{H}_\theta \log p(\theta \mid \boldsymbol{y})$

Intuitively, the Laplace approximation matches the shape of the true posterior around its mode, but may not represent it accurately elsewhere. Often, this leads to extremely overconfident predictions.

4.1 Training

When training, we want to find the distribution in the variational family $\mathcal{Q}$ that minimizes the KL divergence with p,[17]

[17] When approximating p with q, we can either minimize the reverse or the forward KL divergence. The reverse KL divergence $KL(q\|p)$ typically acts more greedily and places most of its mass where p has a lot of mass, while the forward KL divergence $KL(p\|q)$ tries to cover most of the probability mass of p. Thus, the forward KL divergence is more desirable, but it requires sample from p, which is intractable (the whole reason we are doing this). Thus, we have to resort to using the reverse KL divergence.

$$
\begin{aligned}
q^\star &= \operatorname*{argmin}_{q_\lambda \in \mathcal{Q}} KL(q_\lambda \| p) \\
&= \operatorname*{argmin}_{\lambda} \int q_\lambda(\theta) \log \frac{q_\lambda(\theta)}{p(\theta \mid \boldsymbol{y})} d\theta \\
&= \operatorname*{argmin}_{\lambda} \int q_\lambda(\theta) \log \frac{q_\lambda(\theta)}{\frac{1}{Z} p(\boldsymbol{y} \mid \theta) p(\theta)} d\theta \\
&= \operatorname*{argmax}_{\lambda} \int q_\lambda(\theta) (\log p(\boldsymbol{y} \mid \theta) + \log p(\theta) - \log Z - \log q_\lambda(\theta)) d\theta \\
&= \operatorname*{argmax}_{\lambda} \int q_\lambda(\theta) \left(\log p(\boldsymbol{y} \mid \theta) - \log \frac{q_\lambda(\theta)}{p(\theta)} \right) d\theta \\
&= \operatorname*{argmax}_{\lambda} \int q_\lambda(\theta) \log p(\boldsymbol{y} \mid \theta) - \int q_\lambda(\theta) \log \frac{q_\lambda(\theta)}{p(\theta)} d\theta \\
&= \operatorname*{argmax}_{\lambda} \underbrace{\mathbb{E}_{\theta \sim q_\lambda}[\log p(\boldsymbol{y} \mid \theta)]}_{\text{expected likelihood}} - \underbrace{KL(q_\lambda \| p_{\text{prior}})}_{\text{“stay close to prior“}} .
\end{aligned}
$$

Thus, minimizing $KL(q_\lambda \| p)$ is equivalent to maximizing the expected likelihood, while remaining close to the prior distribution.

To show that minimizing the KL divergence is an adequate method of model selection, we will show that it lower bounds the evidence,

$$
\begin{aligned}
\log p(\boldsymbol{y}) &= \log \int p(\boldsymbol{y}, \theta) d\theta && \text{Sum rule} \\
&= \log \int q_\lambda(\theta) \frac{p(\boldsymbol{y}, \theta)}{q_\lambda(\theta)} d\theta \\
&= \log \mathbb{E}_{\theta \sim q_\lambda} \left[\frac{p(\boldsymbol{y}, \theta)}{q_\lambda(\theta)} \right] \\
&\geq \mathbb{E}_{\theta \sim q_\lambda} \left[\log \frac{p(\boldsymbol{y}, \theta)}{q_\lambda(\theta)} \right] && \text{Jensen's inequality} \\
&= \mathbb{E}_{\theta \sim q_\lambda} \left[\log \frac{p(\boldsymbol{y} \mid \theta) p(\theta)}{q_\lambda(\theta)} \right] \\
&= \mathbb{E}_{\theta \sim q_\lambda} [\log p(\boldsymbol{y} \mid \theta)] - KL(q_\lambda \| p_{\text{prior}}).
\end{aligned}
$$

Thus, minimizing

$$
L(\lambda) = \mathbb{E}_{\theta \sim q_\lambda} [\log p(\boldsymbol{y} \mid \theta)] - KL(q_\lambda \| p_{\text{prior}})
$$

can safely be used to find an appropriate model for the data, since a lower bound on the evidence gets maximized.

However, there is one problem: we want to compute the gradient of an expectation w.r.t. q, but q depends on λ. Thus, we cannot compute the gradient in its current form. To solve this we use the reparametrization trick.

Definition 4.2 (Reparameterization trick)**.** Suppose we have a random variable $\epsilon \sim \phi$ sampled from a base distribution, and consider $\theta \doteq g(\epsilon, \lambda)$ for some invertible function g. Then, the following holds:

$$\mathbb{E}_{\theta \sim q_\lambda}[f(\theta)] = \mathbb{E}_{\epsilon \sim \phi}[f(g(\epsilon, \lambda))].$$

Thus, after reparametrization, the expectation is w.r.t. to distribution ϕ that does not depend on λ. Thus, we can compute the gradient $\nabla_\lambda L(\lambda)$.

Example 4.3 (Reparametrization trick for Gaussians)**.** Suppose we use a Gaussian variational approximation,

$$q_\lambda(\theta) \doteq \mathcal{N}(\theta; \boldsymbol{\mu}, \boldsymbol{\Sigma}).$$

Then, we can reparametrize $\theta = g(\boldsymbol{\epsilon}, \lambda) = \boldsymbol{\Sigma}^{1/2}\boldsymbol{\epsilon} + \boldsymbol{\mu}$, where $\phi = \mathcal{N}(\mathbf{0}, \boldsymbol{I})$, because

$$\theta = \boldsymbol{\Sigma}^{1/2}\boldsymbol{\epsilon} + \boldsymbol{\mu} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma}^{1/2}\boldsymbol{I}\boldsymbol{\Sigma}^{1/2\top}) = \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma}).$$

4.2 *Inference*

To perform inference using the variational approximation, we need to compute the integral over all models θ,[18]

[18] The key insight here is that parameters θ do not matter, only their results $f^\star$.

$$\begin{aligned}
p(y^\star \mid \boldsymbol{x}^\star, \boldsymbol{y}) &= \int p(y^\star \mid \boldsymbol{x}^\star, \theta) p(\theta \mid \boldsymbol{y}) d\theta \\
&\approx \int p(y^\star \mid \boldsymbol{x}^\star, \theta) q_\lambda(\theta) d\theta \\
&= \int \int p(y^\star \mid \boldsymbol{x}^\star, \theta, f^\star) q_\lambda(\theta \mid f^\star) p(f^\star \mid \boldsymbol{x}^\star, \theta) d\theta df^\star && \text{Marginalize over } f^\star \\
&= \int \int p(y^\star \mid f^\star) p(f^\star \mid \boldsymbol{x}^\star, \theta) q_\lambda(\theta) d\theta df^\star \\
&= \int p(y^\star \mid f^\star) \int p(f^\star \mid \boldsymbol{x}^\star, \theta) q_\lambda(\theta) d\theta df^\star \\
&= \int p(y^\star \mid f^\star) q_\lambda(f^\star \mid \boldsymbol{x}^\star) df^\star. && \text{Marginalize out } \theta
\end{aligned}$$

Thus, we have reduced the high-dimensional integral over the parameters θ to a one-dimensional integral over $f^\star$. While this integral is generally still intractable, it can be approximated efficiently.

5 Markov chain Monte Carlo

Markov chain Monte Carlo (MCMC) methods seek to approximate the intractable distribution p by drawing m approximate samples from p,

$$\begin{aligned} p(y^\star \mid \boldsymbol{x}^\star, \boldsymbol{y}) &= \int p(y^\star \mid \boldsymbol{x}^\star, \theta) p(\theta \mid \boldsymbol{y}) d\theta \\ &= \mathbb{E}_{\theta \sim p(\cdot \mid \boldsymbol{y})}[p(y^\star \mid \boldsymbol{x}^\star, \theta)] \\ &\approx \frac{1}{m} \sum_{i=1}^{m} p\left(y^\star \mid \boldsymbol{x}^\star, \theta^{(i)}\right). \end{aligned}$$

Law of large numbers

Using *Hoeffding's inequality*, we can compute a bound on the error,

$$p\left(\left|\mathbb{E}_{\theta \sim p(\cdot \mid \boldsymbol{y})}[p(y^\star \mid \boldsymbol{x}^\star, \theta)] - \frac{1}{m} \sum_{i=1}^{m} p\left(y^\star \mid \boldsymbol{x}^\star, \theta^{(i)}\right)\right| > \epsilon\right) \leq 2 \exp\left(-2m\epsilon^2\right).$$

Thus, the probability of error decreases exponentially in m. To get a probability $\leq \delta$ of error $> \epsilon$, we need

$$\begin{aligned} 2\exp(-2m\epsilon^2) &\leq \delta \\ -2m\epsilon^2 &\leq \log \frac{\delta}{2} \\ m &\geq \frac{\log 2 - \log \delta}{2\epsilon^2} \end{aligned}$$

samples. Intuitively, if we want a lower error or lower probability, we need more samples.

However, we cannot sample directly from the posterior $p(\theta \mid \boldsymbol{y})$, because it is intractable. The key idea of MCMC is to construct a Markov chain with stationary distribution $p(\theta \mid \boldsymbol{y})$ and sample from that.

5.1 Markov chains

Definition 5.1 (Markov chain)**.** A Markov chain is a sequence of random variables $(X_t)_{t \in \mathbb{N}_0}$ with prior $P(X_1)$ and transition probabilities $P(X_{t+1} \mid X_t)$ independent of t. The Markov assumption is thus the following,

$$X_{t+1} \perp X_{1:t-1} \mid X_t.$$

Intuitively, this states that future behavior is independent of past states given the present state. In other words, all past information is encapsulated by the current state.

Definition 5.2 (Stationary distribution)**.** A distribution π is stationary with respect to the transition function P iff $P(X_n) = P(X_{n+1})$. In other words, the probability distribution over states remains the same between timesteps. The following must hold for all x,

$$\pi(x) = \sum_{x'} P(x \mid x') \pi(x').$$

Definition 5.3 (Ergodicity)**.** A Markov chain is ergodic iff there exists a $t \in \mathbb{N}_0$ such that for any x, x', the following holds,

$$P^{(t)}(x' \mid x) > 0,$$

where $P^{(t)}(x' \mid x)$ is the probability to reach x' from x in *exactly* t steps. Intuitively, this means that any state is reachable from another within the same amount of steps.

Remark. An easy way of ensuring that a Markov chain is ergodic is to add "self-loops" to every vertex.

Theorem 5.4 (Fundamental theorem of ergodic Markov chains)**.** An ergodic Markov chain has a unique and positive stationary distribution $\pi(X) > 0$ such that for all x, the following holds,

$$\lim_{n \to \infty} P(X_n = x) = \pi(x),$$

independent of the initial distribution $P(X_1)$. *I.e.*, ergodic Markov chains always converge to a stationary distribution.

Thus, making use of the fundamental theorem of ergodic Markov chains, we can construct an ergodic Markov chain such that its stationary distribution coincides with the posterior distribution, $\pi(x) = p(\theta \mid \boldsymbol{y})$. If we then sample "sufficiently long" from this Markov chain, X_t is drawn from a distribution that is "very close" to the stationary distribution π, which is equal to the posterior distribution.

Definition 5.5 (Detailed balance equation)**.** A Markov chain satisfies the detailed balance equation for an unnormalized distribution q iff the following holds for any x, x',

$$q(x)P(x' \mid x) = q(x')P(x \mid x').$$

Theorem 5.6. If a finite Markov chain satisfies the detailed balance equation with respect to q, then $\frac{1}{Z}q$ is a stationary distribution.

Proof. Let $p_t = q$. Then for any x, the following holds,

$$\begin{aligned} p_{t+1}(x) &= \sum_{x'} P(x \mid x') p_t(x') && \text{Markov assumption} \\ &= \sum_{x'} P(x \mid x') q(x') \\ &= \sum_{x'} P(x' \mid x) q(x) && \text{Detailed balance equation} \\ &= q(x) \sum_{x'} P(x' \mid x) \\ &= q(x). \end{aligned}$$

Thus, q is the stationary distribution. ■

5.2 Sampling

If we can show that the detailed balance equation holds for the *unnormalized* posterior distribution, then we know that the posterior distribution is the stationary distribution of the Markov chain.[19] Thus, we do not need to know the true posterior. It suffices to know its unnormalized version.[20]

[19] The only problem is that we do not know the rate of convergence to the stationary distribution.

[20] Recall that the normalizer was the intractable part of $p(\theta \mid \boldsymbol{y})$.

The *Metropolis-Hastings algorithm* constructs a Markov chain with the posterior as stationary distribution. It uses an arbitrary proposal transition distribution $R(x' \mid x)$ which, given we are in state x, proposes a new state x'.[21] Following the proposal with probability

$$\alpha(x' \mid x) \doteq \min\left\{1, \frac{q(x')R(x \mid x')}{q(x)R(x' \mid x)}\right\},$$

yields a Markov chain with the desired stationary distribution $\frac{1}{Z}q(x)$, because it makes it satisfy the detailed balance equation.

[21] The rate of convergence of this algorithm strongly depends on the choice of R.

```
1: function MetropolisHastings(R)
2:     initialize x
3:     for t = 1, ..., T do
4:         x' ~ R(x' | x)
5:         u ~ Unif([0, 1])
6:         if u ≤ α(x' | x) then
7:             x ← x'
8:         end if
9:     end for
10: end function
```

Algorithm 1. The Metropolis-Hastings algorithm. Each iteration, with a random probability, follow the proposal distribution.

5.3 Proposal distributions

We want to converge to the stationary distribution as fast as possible. The proposal distribution has a big influence on this.

Gibbs sampling. A popular example algorithm for specifying a proposal distribution R is Gibbs sampling. Gibbs sampling works by iteratively improving the variables. It starts with an initial assignment x to all variables, fixing the observed variables to their observed value. Then, iteratively uniformly pick a variable X_i to update given the rest of the set values by sampling $p(X_i \mid x_{1:i-1}, x_{i+1:n})$.[22]

[22] Sampling from this distribution is typically efficient.

Algorithm 2. Gibbs sampling.

```
function GibbsSampling
    initialize x = [x_1, ..., x_n] ∈ ℝ^n
    for t = 1, ..., T do
        uniformly sample i from {1, ..., n}
        x_{-i} ← [x_1, ..., x_{i-1}, x_{i+1}, ..., x_n]
        update x_i by sampling from p(x_i | x_{-i})
    end for
end function
```

Then, Gibbs sampling is a Metropolis-Hastings algorithm with the following proposal distribution,

$$R(x' \mid x) \doteq \begin{cases} p(x'_i \mid x'_{1:i-1}, x'_{i+1:n}) & x' \text{ differs from } x \\ 0 & \text{otherwise} \end{cases}$$

and acceptance distribution $\alpha(x' \mid x) = 1$ for all x, x'.

Gaussian. Generally, we focus on positive distributions written as the following,

$$p(x) = \frac{1}{Z} \exp(-f(x)),$$

where f is called an energy function (high energy $\equiv$ low probability, low energy $\equiv$ high probability). Then, the acceptance distribution becomes the following,

$$\alpha(x' \mid x) = \min\left\{1, \frac{R(x \mid x')}{R(x' \mid x)} \exp(f(x) - f(x'))\right\}.$$

One option for the proposal distribution is

$$R(x' \mid x) = \mathcal{N}(x'; x, \tau \boldsymbol{I}).$$

Since this R is symmetric,

$$\frac{R(x \mid x')}{R(x' \mid x)} = 1.$$

Thus, if R proposes to move to a region with lower energy, the acceptance probability will always be 1. If R proposes to move to a region with higher energy, the probability moves toward 0 dependent on how much higher the region is.

However, we want to move as quickly as possible through the function, going through all high-density areas. But, this R is "uninformed" and

thus proposes to go in any direction. We would like to propose areas with lower energy, which are high-density areas for $p(x)$. Then, we will have less iterations where the proposal simply gets rejected.

Metropolis adjusted Langevin algorithm. An improvement to the Gaussian proposal distribution is the Metropolis adjusted Langevin algorithm (MALA),

$$R(x' \mid x) = \mathcal{N}(x'; x - \tau \nabla f(x), 2\tau \boldsymbol{I}).$$

This proposes moving to high-density areas, thus it much more efficiently converges to the stationary distribution.

The problem with this is that it requires access to the full energy function to compute the gradient, which can be expensive for large datasets. This is solved by stochastically estimating the gradient $\nabla f(x)$.

6 Bayesian neural networks

So far, we have explored techniques for computing uncertainty of linear models.[23] However, in practice, we can often get better performance by considering non-linear dependencies. Thus, this is what we will explore next.

[23] The likelihood have parameters linearly dependent on the input feature.

Neural networks typically look like the following,

$$f_{\boldsymbol{\theta}}(\boldsymbol{x}) = \phi(\boldsymbol{W}_\ell \phi(\boldsymbol{W}_{\ell-1} \cdots \phi(\boldsymbol{W}_1 \boldsymbol{x}))).$$

Bayesian neural network models specify a prior distribution over the weights,

$$\boldsymbol{\theta} \sim \mathcal{N}(\mathbf{0}, \sigma_p^2 \boldsymbol{I}),$$

and use likelihood distributions parametrized by a neural network,

$$y \mid \boldsymbol{x}, \boldsymbol{\theta} \sim \mathcal{N}(f(\boldsymbol{x}; \boldsymbol{\theta}), \sigma^2),$$

which assumes homoscedastic noise.[24] However, we can also parameterize a likelihood that can model heteroscedastic noise by predicting the variance of the Gaussian,

[24] Same noise for all data points.

$$y \mid \boldsymbol{x}, \boldsymbol{\theta} \sim \mathcal{N}(f_\mu(\boldsymbol{x}; \boldsymbol{\theta}), \exp f_{\sigma^2}(\boldsymbol{x}; \boldsymbol{\theta})).$$

The MAP estimate of a BNN is the following,

$$\begin{aligned}
\hat{\boldsymbol{\theta}} &= \operatorname*{argmax}_{\boldsymbol{\theta}} p(\boldsymbol{\theta} \mid \boldsymbol{X}, \boldsymbol{y}) \\
&= \operatorname*{argmax}_{\boldsymbol{\theta}} p(\boldsymbol{\theta}) p(\boldsymbol{y} \mid \boldsymbol{X}, \boldsymbol{\theta}) \\
&= \operatorname*{argmin}_{\boldsymbol{\theta}} -\log p(\boldsymbol{\theta}) - \sum_{i=1}^{n} \log p(y_i \mid \boldsymbol{x}_i, \boldsymbol{\theta}) \\
&= \operatorname*{argmin}_{\boldsymbol{\theta}} -\lambda \|\boldsymbol{\theta}\|^2 + \sum_{i=1}^{n} -\log \mathcal{N}(y_i; f_\mu(\boldsymbol{x}_i; \boldsymbol{\theta}, f_{\sigma^2}(\boldsymbol{x}_i; \boldsymbol{\theta}))) \\
&= \operatorname*{argmin}_{\boldsymbol{\theta}} -\lambda \|\boldsymbol{\theta}\|^2 + \sum_{i=1}^{n} -\log \left(\frac{1}{2\pi f_{\sigma^2}(\boldsymbol{x}_i; \boldsymbol{\theta})} \exp\left(-\frac{1}{2 f_{\sigma^2}(\boldsymbol{x}_i; \boldsymbol{\theta})} (y_i - f_\mu(\boldsymbol{x}_i; \boldsymbol{\theta}))^2 \right) \right) \\
&= \operatorname*{argmin}_{\boldsymbol{\theta}} -\lambda \|\boldsymbol{\theta}\|^2 + \sum_{i=1}^{n} \log(2\pi) + \log(f_{\sigma^2}(\boldsymbol{x}_i; \boldsymbol{\theta})) + \frac{1}{2 f_{\sigma^2}(\boldsymbol{x}_i; \boldsymbol{\theta})} (y_i - f_\mu(\boldsymbol{x}_i; \boldsymbol{\theta}))^2 \\
&= \operatorname*{argmin}_{\boldsymbol{\theta}} -\lambda \|\boldsymbol{\theta}\|^2 + \sum_{i=1}^{n} \log(f_{\sigma^2}(\boldsymbol{x}_i; \boldsymbol{\theta})) + \frac{(y_i - f_\mu(\boldsymbol{x}_i; \boldsymbol{\theta}))^2}{2 f_{\sigma^2}(\boldsymbol{x}_i; \boldsymbol{\theta})}.
\end{aligned}$$

Thus, the MAP estimate is a balance between the mean and variance predictions. If we perfectly predict y_i with f_μ, we only need to make f_{σ^2} smaller. Otherwise, we can attenuate for the error $(y_i - f_\mu(\boldsymbol{x}_i; \boldsymbol{\theta}))^2$ with f_{σ^2} in its denominator, for which we have to pay logarithmically. Intuitively, the model can attenuate certain losses for certain datapoints by attributing the error to large variance.

However, the problem with the MAP estimate is that it does not use the entire distribution over $\boldsymbol{\theta}$. In other words, it does not account for epistemic uncertainty, only aleatoric. But, just like before, using the whole

distribution would be intractable. Thus, we need some approximation techniques to make it tractable. We will explore this in the next subsections.

6.1 Variational inference

Since BNNs are just a distribution over the weights $\boldsymbol{\theta}$, we can approximate its distribution with variational inference. Then, we can learn the distribution over $\boldsymbol{\theta}$ by optimizing the ELBO of the approximation distribution q_λ. Then, we can do inference as follows,

$$\begin{aligned}
p(y^\star \mid \boldsymbol{x}^\star, \boldsymbol{X}, \boldsymbol{y}) &= \int p(y^\star \mid \boldsymbol{x}^\star, \boldsymbol{\theta}) p(\boldsymbol{\theta} \mid \boldsymbol{X}, \boldsymbol{y}) d\boldsymbol{\theta} \\
&= \mathbb{E}_{\boldsymbol{\theta} \sim p(\cdot \mid \boldsymbol{X}, \boldsymbol{y})}[p(y^\star \mid \boldsymbol{x}^\star, \boldsymbol{\theta})] \\
&\approx \mathbb{E}_{\boldsymbol{\theta} \sim q_\lambda}[p(y^\star, \boldsymbol{x}^\star, \boldsymbol{\theta})] && \text{Variational inference} \\
&\approx \frac{1}{m} \sum_{j=1}^{m} p\left(y^\star \mid \boldsymbol{x}^\star, \boldsymbol{\theta}^{(j)}\right), \qquad \boldsymbol{\theta}^{(j)} \sim q_\lambda. && \text{Monte Carlo}
\end{aligned}$$

If q_λ is Gaussian, then the approximate predictive distribution becomes a mixture of Gaussians. The mean and variance of this distribution are the following,

$$\mathbb{E}[y^\star \mid \boldsymbol{x}^\star, \boldsymbol{X}, \boldsymbol{y}] \approx \bar{\mu}(\boldsymbol{x}^\star) \doteq \frac{1}{m} \sum_{j=1}^{m} f_\mu(\boldsymbol{x}^\star; \boldsymbol{\theta}^{(j)})$$

$$\begin{aligned}
\operatorname{Var}[y^\star \mid \boldsymbol{x}^\star, \boldsymbol{X}, \boldsymbol{y}] &= \mathbb{E}_{\boldsymbol{\theta}}[\operatorname{Var}_{y^\star}[y^\star \mid \boldsymbol{x}^\star, \boldsymbol{\theta}]] + \operatorname{Var}_{\boldsymbol{\theta}}[\mathbb{E}_{y^\star}[y^\star \mid \boldsymbol{x}^\star, \boldsymbol{\theta}]] && \text{Law of total variance} \\
&\approx \underbrace{\frac{1}{m} \sum_{j=1}^{m} f_{\sigma^2}\left(\boldsymbol{x}^\star; \boldsymbol{\theta}^{(j)}\right)}_{\text{aleatoric uncertainty}} + \underbrace{\frac{1}{m-1} \sum_{j=1}^{m} \left(f_\mu\left(\boldsymbol{x}^\star; \boldsymbol{\theta}^{(j)}\right) - \bar{\mu}(\boldsymbol{x}^\star)\right)^2}_{\text{epistemic uncertainty}}
\end{aligned}$$

6.2 Markov chain Monte Carlo

It is also possible to apply MCMC to BNNs. MCMC methods produce a sequence of weights $\boldsymbol{\theta}^{(1)}, \ldots, \boldsymbol{\theta}^{(T)}$. Using the ergodic theorem we can then make predictions with the following,

$$p(y^\star \mid \boldsymbol{x}^\star, \boldsymbol{X}, \boldsymbol{y}) \approx \frac{1}{T} \sum_{j=1}^{T} p\left(y^\star \mid \boldsymbol{x}^\star, \boldsymbol{\theta}^{(j)}\right).$$

However, models are often very large, so we cannot store T times the parameters of the network $\mathcal{O}(Td)$. Thus, we need to approximate. A simple solution is to only keep a subset of m weights. But, we can also approximate the distribution with a Gaussian,

$$\boldsymbol{\theta} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma}),$$

and keeping running averages, only requiring $\mathcal{O}(d^2)$ space complexity, where

$$\boldsymbol{\mu} = \frac{1}{T} \sum_{j=1}^{T} \boldsymbol{\theta}^{(j)}, \qquad \boldsymbol{\Sigma} = \frac{1}{T-1} \sum_{j=1}^{T} \left(\boldsymbol{\theta}^{(j)} - \boldsymbol{\mu}\right)\left(\boldsymbol{\theta}^{(j)} - \boldsymbol{\mu}\right)^\top.$$

SWAG [Maddox et al., 2019] is an example of a model that does this, but instead of an MCMC method, it uses stochastic gradient descent to sample models.

6.3 Monte Carlo dropout

Dropout regularization is often used in traditional neural networks to improve generalization. It works by randomly selecting weights to set to 0. However, using *Monte Carlo dropout*, we can view this as performing variational inference. Let p be the probability that we omit parameter, then the variational posterior is given as the following,

$$\begin{aligned} q(\boldsymbol{\theta} \mid \lambda) &= \prod_{j=1}^{d} q_j(\theta_j \mid \lambda_j) \\ q_j(\theta_j \mid \lambda_j) &= p\delta_0(\theta_j) + (1-p)\delta_{\lambda_j}(\theta_j), \end{aligned}$$

where d is the number of parameters in the neural network. Intuitively, this posterior says that the j-th weight has value 0 with probability p and value λ_j with probability $1-p$.

The difference with dropout regularization is that we also need to use dropout during inference for this to be variational inference,

$$\begin{aligned} p(y^\star \mid \boldsymbol{x}^\star, \boldsymbol{y}) &\approx \mathbb{E}_{\boldsymbol{\theta} \sim q_\lambda}[p(y^\star \mid \boldsymbol{x}^\star, \boldsymbol{\theta})] \\ &\approx \frac{1}{m} \sum_{j=1}^{m} p\left(y^\star \mid \boldsymbol{x}^\star, \boldsymbol{\theta}^{(j)}\right) \\ \boldsymbol{\theta}^{(j)} &\overset{\text{iid}}{\sim} q_\lambda. \end{aligned}$$

Intuitively, we average the distribution of m neural networks for each of which we randomly drop out weights.

6.4 Probabilistic ensembles

We have seen that variational inference can be seen as averaging the predictions of m neural networks. A natural adaptation of this idea is to learn the weights of m neural networks. The idea is to randomly choose m training subsets, each with n data points. Then, we compute m MAP estimates $\boldsymbol{\theta}^{(j)}$, yielding the following approximation,

$$p(y^\star \mid \boldsymbol{x}^\star, \boldsymbol{y}) \approx \frac{1}{m} \sum_{j=1}^{m} p(y^\star \mid \boldsymbol{x}^\star, \boldsymbol{\theta}^{(j)}).$$

6.5 Calibration

A key challenge of BNNs is *calibration*. We want models to be well-calibrated, which means that the confidence that they have in their predictions coincides with the accuracy they have over many samples. For example, let's say that we have a classification model that predicts that a

data point belongs to a certain class with 80% probability. If the model is well-calibrated, then the prediction should be correct 80% of the time. We can calibrate models by adjusting the probability estimation of models.

Reliability diagrams (Figure 6.1) are a way of determining how calibrated a model is. This diagram is constructed by making predictions on a validation dataset. These predictions are then divided into M bins according to the predicted class probability,

$$B_j = \left\{ y \;\middle|\; p(y) \in \left[\frac{j-1}{m}, \frac{j}{m} \right) \right\}.$$

Within each bin, we then compare the predicted probabilities (confidence) with how often the input actually belonged to the class (frequency),

$$\begin{aligned} \text{conf}(B_j) &= \frac{1}{|B_j|} \sum_{y \in B_j} p_{\boldsymbol{\theta}}(y) \\ \text{acc}(B_j) &= \frac{1}{|B_j|} \sum_{y \in B_j} \mathbb{1}\{y = \hat{y}\}, \end{aligned}$$

A model is well-calibrated if $\text{conf}(B_j) \approx \text{acc}(B_j)$ for all bins B_j.

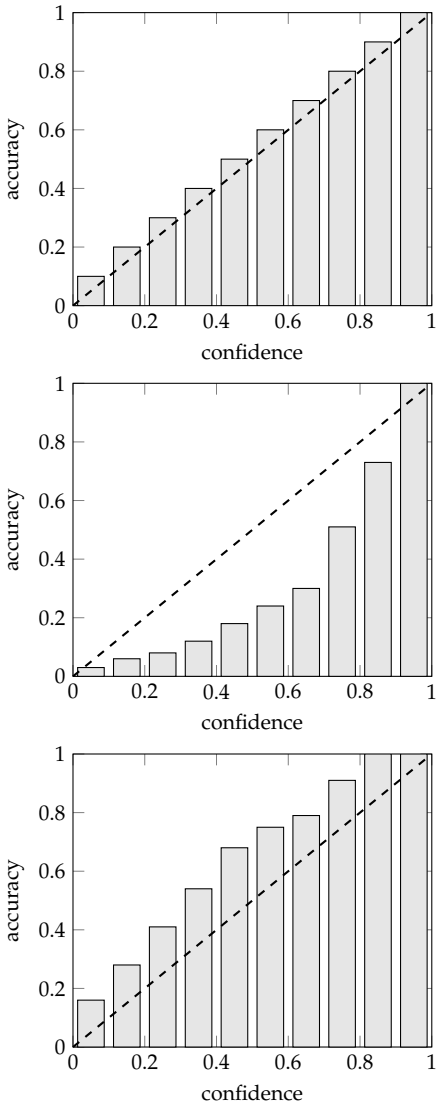


Figure 6.1. Reliability diagrams. The top diagram shows a well-calibrated model, the second diagram is overconfident, and the third diagram is underconfident.

7 Active learning

Until now, we have only covered how to learn/represent aleatoric and epistemic uncertainty in machine learning. *Active learning* covers how to use this measure for deciding which data to collect. Intuitively, we want to collect data points in places where we would gain the most information, *i.e.*, where the uncertainty is high. This assumes that there is some cost associated with collecting data points, making it important to be careful about which data points to pick that maximize the information obtained.

Let $\mathcal{X}$ be a set of possible observations of f, and $y_{\boldsymbol{x}}$ the observation at $\boldsymbol{x} \in \mathcal{X}$,

$$y_{\boldsymbol{x}} \doteq f(\boldsymbol{x}) + \epsilon_{\boldsymbol{x}}, \qquad \boldsymbol{\epsilon}_{\boldsymbol{x}} \in \mathcal{N}(\boldsymbol{0}, \sigma_n^2 \boldsymbol{I}).$$

Then, we want to observe a subset $S \subseteq \mathcal{X}$ of a fixed size that maximizes the information gain between the model f and $\boldsymbol{y}_S$, which yields the following maximization objective,

$$I(S) \doteq I(\boldsymbol{f}_S; \boldsymbol{y}_S) = H[\boldsymbol{f}_S] - H(\boldsymbol{f}_S \mid \boldsymbol{y}_S),$$

where $H[\boldsymbol{f}_S]$ denotes the uncertainty about $\boldsymbol{f}_S$ before obtaining the observations $\boldsymbol{y}_S$ and $H[\boldsymbol{f}_S \mid \boldsymbol{y}_S]$ corresponds to the uncertainty about $\boldsymbol{f}_S$ after obtaining observations $\boldsymbol{y}_S$. This problem is $\mathcal{NP}$-hard, thus we need to formulate a strategy.

7.1 Sampling strategies

Uncertainty sampling. The simplest strategy is to greedily pick points one by one, which entails that we pick the locations, $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$, individually by greedily finding the location with maximal mutual information. That is, if we have already picked locations $S_t = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_t\}$, then the next point, $\boldsymbol{x}_{t+1}$, maximizes its mutual information,

$$\boldsymbol{x}_{t+1} \doteq \operatorname*{argmax}_{\boldsymbol{x} \in \mathcal{X}} I(f_{\boldsymbol{x}}; y_{\boldsymbol{x}} \mid \boldsymbol{y}_{S_t}).$$

Assuming that f is modeled by a Gaussian,

$$\begin{aligned} \boldsymbol{x}_{t+1} &= \operatorname*{argmax}_{\boldsymbol{x} \in \mathcal{X}} \frac{1}{2} \log\left(1 + \frac{\sigma^2_{\boldsymbol{x}|S_t}}{\sigma_n^2(\boldsymbol{x})}\right) \\ &= \operatorname*{argmax}_{\boldsymbol{x} \in \mathcal{X}} \frac{\sigma^2_{\boldsymbol{x}|S_t}}{\sigma_n^2(\boldsymbol{x})}. \end{aligned}$$

Assuming that the label noise is independent of $\boldsymbol{x}$, *i.e.*, homoscedastic,

$$\boldsymbol{x}_{t+1} = \operatorname*{argmax}_{\boldsymbol{x} \in \mathcal{X}} \sigma^2_{\boldsymbol{x}|S_t}.$$

Thus, if f is modeled by a Gaussian and we assume homoscedastic noise, greedily maximizing mutual information corresponds to picking the point $\boldsymbol{x}$ with the largest variance.

Due to the *information never hurts* principle, mutual information is monotone submodular, which means that adding data points can only increase the mutual information in expectation. From this, it follows that the greedy algorithm provides a constant-factor approximation, which means that uncertainty sampling is near-optimal.

Heteroscedastic noise. If the data is not homoscedastic, uncertainty sampling will fail, because it fails to distinguish between epistemic and aleatoric uncertainty. Therefore, the most uncertain point is not necessarily the most informative one. Thus, maximizing the mutual information yields the following,

$$\boldsymbol{x}_{t+1} = \operatorname*{argmax}_{\boldsymbol{x} \in \mathcal{X}} \frac{\sigma^2_{\boldsymbol{x}|S_t}}{\sigma^2_n(\boldsymbol{x})}.$$

Here we make a trade-off between large epistemic uncertainty and large aleatoric uncertainty. Ideally, we find an $\boldsymbol{x}$ where the epistemic uncertainty is large, and the aleatoric uncertainty low, because we want to make sure we get a lot of information (high epistemic uncertainty), but also that we can have high confidence in the point we choose (low aleatoric uncertainty).

Bayesian active learning by disagreement. Uncertainty sampling in classification corresponds to selecting samples that maximize entropy of the predicted label, *i.e.*, points that are close to the decision boundary. However, the uncertainty in points around the decision boundary are often due to aleatoric noise. Hence, we will not learn much from observing points there.

Thus, we need to distinguish between aleatoric and epistemic uncertainty of $f_{\boldsymbol{\theta}}$,

$$\begin{aligned}
\boldsymbol{x}_{t+1} &= \operatorname*{argmax}_{\boldsymbol{x} \in \mathcal{X}} I(y_{\boldsymbol{x}}; \boldsymbol{\theta} \mid \boldsymbol{x}_{1:t}, y_{1:t}) \\
&= \operatorname*{argmax}_{\boldsymbol{x} \in \mathcal{X}} H[y_{\boldsymbol{x}} \mid \boldsymbol{x}_{1:t}, y_{1:t}] - H[y_{\boldsymbol{x}} \mid \boldsymbol{\theta}, \boldsymbol{x}_{1:t}, y_{1:t}] \\
&= \operatorname*{argmax}_{\boldsymbol{x} \in \mathcal{X}} \underbrace{H[y_{\boldsymbol{x}} \mid \boldsymbol{x}_{1:t}, y_{1:t}]}_{\substack{\text{entropy of} \\ \text{pred. posterior}}} - \underbrace{\mathbb{E}_{\boldsymbol{\theta} \mid \boldsymbol{x}_{1:t}, y_{1:t}}[H[y_{\boldsymbol{x}} \mid \boldsymbol{\theta}]]}_{\text{entropy of likelihood}}.
\end{aligned}$$

The first term measures the entropy of the average prediction, while the second term measures the average entropy of predictions. Thus, the first term looks for points where the average prediction is uncertain. In contrast, the second term penalizes points where many of the sampled models $\boldsymbol{\theta}$ are uncertain about their prediction. Thus, we want to find points $\boldsymbol{x}$ where the posterior is uncertain (epistemic uncertainty) because of all models $\boldsymbol{\theta}$ being extremely certain about their differing predictions (aleatoric uncertainty).

Since $p(\boldsymbol{\theta} \mid \boldsymbol{x}_{1:t}, y_{1:t})$ is intractable, we need to use variational inference

and Monte Carlo to approximate the second term,

$$\begin{aligned}\mathbb{E}_{\boldsymbol{\theta}|\boldsymbol{x}_{1:t},y_{1:t}}[H[y_{\boldsymbol{x}} \mid \boldsymbol{\theta}]] &\approx \mathbb{E}_{q_\lambda}[H[y_{\boldsymbol{x}} \mid \boldsymbol{\theta}]] && \text{Variational inference} \\ &\approx \frac{1}{m}\sum_{i=1}^{m} H\left[y_{\boldsymbol{x}} \mid \boldsymbol{\theta}^{(i)}\right], \quad \boldsymbol{\theta}^{(i)} \sim q_\lambda. && \text{Monte Carlo}\end{aligned}$$

We can use another approximation method, such as variational inference, Markov chain Monte Carlo, or SWAG, to approximate the predictive posterior in the first term.

8 Bayesian optimization

In *Bayesian optimization*, we do not only want to reduce uncertainty, but we also want to maximize some objective.[25] This means that we have to make a trade-off between exploration (minimizing uncertainty) and exploitation (maximizing performance).

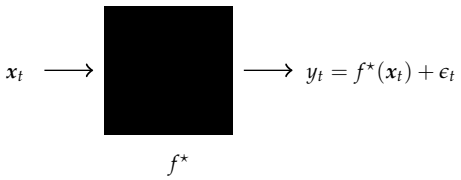


Figure 8.1. Illustration of Bayesian optimization. We pass an input x_t into the unknown function $f^\star$ to obtain a noisy observation y_t.

[25] This is in contrast with active learning. An example of this is automatically tuning the hyperparameters of a machine learning model. In this scenario, we would both want to minimize the uncertainty of the parameter space and maximize the performance of the eventual model.

Definition 8.1 (Regret)**.** The cumulative regret for a time horizon T associated with choices $x_1, \dots, x_T$ is defined as

$$R_T \doteq \sum_{t=1}^{T} \underbrace{\left(\max_x f(x) - f(x_t)\right)}_{\text{“instantaneous regret“}}.$$

The regret can be interpreted as the additive loss with respect to the maximally achievable value $\max_x f(x)$.

The goal is to find algorithms that achieve a sublinear regret,

$$\lim_{T\to\infty} \frac{R_T}{T} = 0.$$

This leads to the algorithm converging on the maximum $\max_x f(x)$. Note that using an algorithm that explores forever will result in the regret growing linearly, because we will never settle on a maximum value. In contrast, if we use an algorithm that never explores and thus only exploits, we might never find $\max_x f(x)$. Thus, achieving sublinear regret requires balancing exploration and exploitation.

8.1 Acquisition functions

Upper confidence bound. The principle of *optimism in the face of uncertainty* naturally suggests picking the point where we can hope for the optimal outcome. This corresponds to maximizing the *upper confidence bound* (UCB),[26]

$$x_t = \operatorname*{argmax}_{x\in\mathcal{X}} \mu_{t-1}(x) + \beta_t \sigma_{t-1}(x),$$

where β_t regulates how confident we are in our current model. A high β_t leads the model to explore, while a low β_t leads the model to exploit. This acquisition function naturally trades exploitation by preferring a large posterior mean with exploration by preferring a large posterior variance.

[26] This assumes that the parameters of the model (*e.g.* GP) are known. However, this is not the case in practice. Thus, in practice, we can alternate between learning the hyperparameters from the currently selected data, and selecting the next datapoint. However, this introduces a danger of overfitting. This can be solved by either placing a hyperprior on the hyperparameters, or occasionally selecting some points at random.

If f can be represented by our model and we choose β_t “correctly“,

$$R_T \in \mathcal{O}^\star\left(\sqrt{\frac{\gamma_T}{T}}\right),$$

where

$$\gamma_T = \max_{|S|\le T} I(f; y_S),$$

quantifies the maximum information gain. Thus, the maximum information gain γ_T determines the regret of the UCB algorithm.

Thompson sampling. At every iteration t, Thompson sampling draws a function realization from the Gaussian process,

$$\tilde{f}_t \sim p(f \mid \boldsymbol{x}_{1:t}, y_{1:t}),$$

and selects

$$\boldsymbol{x}_{t+1} = \operatorname*{argmax}_{\boldsymbol{x} \in \mathcal{X}} \tilde{f}_t(\boldsymbol{x}).$$

The randomness in the realization of $\tilde{f}_t$ is sufficient to trade exploration and exploitation. Similarly to UCB, it has sublinear regret bounds.

Kernel	γ_T **bound**
Linear	$\mathcal{O}(d \log T)$
Gaussian	$\mathcal{O}\big((\log T)^{d+1}\big)$
Matérn ($\nu > 1/2$)	$\mathcal{O}\Big(T^{\frac{d}{2\nu+d}} (\log T)^{\frac{2\nu}{2\nu+d}}\Big)$

Table 1. Information gain bounds of common Gaussian process kernels. These guarantee sublinear regret, which means that they are guaranteed to converge to the maximum value of the function.

9 Markov decision processes

Definition 9.1 (Markov decision process)**.** A Markov decision process $\langle \mathcal{X}, \mathcal{A}, p, r \rangle$ is specified by the following,

- Set of states $\mathcal{X}$ (not necessarily finite);
- Set of actions $\mathcal{A}$ (not necessarily finite);
- Initial state distribution $p(x_0)$;
- Transition probabilities $p(x' \mid x, a)$;
- Reward function $r(x, a)$ (or $r(x, a, x')$).

As can be seen by the form of the transition probabilities, we make the Markov assumption, where the future state only depends on the current state and action.

For now, we will assume that p and r are known, but, as we will see in section 10, in reinforcement learning, these are unknown.

In general, we want to maximize the long-term reward, either over a finite horizon T,

$$\max \mathbb{E}\left[\sum_{t=0}^{T} r(x_t, a_t)\right],$$

or over an infinite horizon where future rewards are discounted by a decaying factor $\gamma \in (0, 1)$,[27]

$$\max \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r(x_t, a_t)\right].$$

[27] We cannot maximize the summed future reward (without decaying factor), because this could be ∞ for all possible strategies.

A *policy* $\pi : \mathcal{X} \to \mathcal{A}$ is a function that maps states to their action. A policy π induces a Markov chain with $P(X_{t+1} = x' \mid X_t = x) = p(x' \mid x, \pi(x))$ as transition probabilities. In general, we want to find the policy that maximizes the expected value,

$$J(\pi) \doteq \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r(X_t, \pi(X_t))\right].$$

9.1 Bellman expectation equation

The *value function* of a state x is the expected sum of discounted future rewards, obtained from subsequent states. This is defined as the following,

$$V^\pi(x) \doteq \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r(X_t, \pi(X_t)) \;\middle|\; X_0 = x\right],$$

We can solve for the value function of a policy using a bit of linear algebra. Let $\mathcal{X} = \{1, \dots, n\}$, then we can define the following vectors and matrix,

$$\boldsymbol{v}^\pi = \begin{bmatrix} V^\pi(1) \\ \vdots \\ V^\pi(n) \end{bmatrix}$$

$$\boldsymbol{r}^\pi = \begin{bmatrix} r(1, \pi(1)) \\ \vdots \\ r(n, \pi(n)) \end{bmatrix}$$

$$\boldsymbol{T}^\pi = \begin{bmatrix} p(1 \mid 1, \pi(1)) & \cdots & p(n \mid 1, \pi(1)) \\ \vdots & \ddots & \vdots \\ p(1 \mid n, \pi(n)) & \cdots & p(n \mid n, \pi(n)) \end{bmatrix}.$$

Using the Bellman expectation equation, we have the following equality,

$$\begin{aligned} & \boldsymbol{v}^\pi = \boldsymbol{r}^\pi + \gamma \boldsymbol{T}^\pi \boldsymbol{v}^\pi \\ \iff \quad & (\boldsymbol{I} - \gamma \boldsymbol{T}^\pi)\boldsymbol{v}^\pi = \boldsymbol{r}^\pi \\ \iff \quad & \boldsymbol{v}^\pi = (\boldsymbol{I} - \gamma \boldsymbol{T}^\pi)^{-1} \boldsymbol{r}^\pi. \end{aligned}$$

However, this is computationally expensive. We could also use fixed-point iteration to obtain an (approximate) solution. This involves iteratively computing the value function of each node, given the last value function,

$$\boldsymbol{v}_t^\pi = \boldsymbol{r}^\pi + \gamma \boldsymbol{T}^\pi \boldsymbol{v}_{t-1}^\pi.$$

This is faster, because $\boldsymbol{T}^\pi$ is sparse.

which contains a recursive relationship,

$$
\begin{aligned}
V^\pi(x) &\doteq \mathbb{E}_x\left[\sum_{t=0}^{\infty}\gamma^t r(X_t,\pi(X_t)) \;\middle|\; X_0 = x\right] \\
&= \mathbb{E}_x\left[r(X_0,\pi(X_0)) + \sum_{t=1}^{\infty}\gamma^t r(X_t,\pi(X_t)) \;\middle|\; X_0 = x\right] \\
&= \mathbb{E}_x[r(X_0,\pi(X_0)) \mid X_0 = x] + \mathbb{E}_x\left[\sum_{t=0}^{\infty}\gamma^{t+1} r(X_{t+1},\pi(X_{t+1})) \;\middle|\; X_0 = x\right] \\
&= r(x,\pi(x)) + \gamma\mathbb{E}_{x'}\left[\mathbb{E}_x\left[\sum_{t=0}^{\infty}\gamma^t r(X_{t+1},\pi(X_{t+1})) \;\middle|\; X_1 = x'\right]\right] && \text{Condition on realization of } X_1 \\
&= r(x,\pi(x)) + \gamma\sum_{x'\in\mathcal{X}} p(x' \mid x,\pi(x))\cdot\mathbb{E}_{x'}\left[\sum_{t=0}^{\infty}\gamma^t r(X_{t+1},\pi(X_{t+1})) \;\middle|\; X_1 = x'\right] \\
&= r(x,\pi(x)) + \gamma\sum_{x'\in\mathcal{X}} p(x' \mid x,\pi(x))\cdot\mathbb{E}_{x'}\left[\sum_{t=0}^{\infty}\gamma^t r(X_t,\pi(X_t)) \;\middle|\; X_0 = x'\right] && \text{Stationarity of Markov chains} \\
&= r(x,\pi(x)) + \gamma\sum_{x'\in\mathcal{X}} p(x' \mid x,\pi(x))\cdot V^\pi(x').
\end{aligned}
$$

This equation is known as the *Bellman expectation equation*. Intuitively, this means that the value of the current state corresponds to the reward from the next action, plus the discounted sum of expected future rewards obtained from the subsequent states (which are their values).

Theorem 9.2 (Bellman's theorem)**.** A policy $\pi^\star$ is optimal iff it is greedy w.r.t. its own value function $V^\star$.

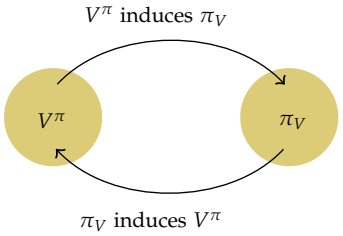


Figure 9.1. Cyclic dependency between value function and greedy policy.

9.2 *Policy iteration*

Every value function induces a policy where we greedily choose the action that maximizes the expected value,

$$\pi_V(x) = \operatorname*{argmax}_{a\in\mathcal{A}} r(x,a) + \gamma\sum_{x'\in\mathcal{X}} p(x' \mid x,a)V(x'), \tag{1}$$

while the policy induces a value function as we have seen. This leads us to *policy iteration*, where we find an optimal policy by alternating between computing the value function w.r.t. π (using fixed-point iteration) and computing the next greedy policy w.r.t. V^π (using Equation (1)), until convergence.

Policy iteration is expensive, because every iteration requires computing a value function, but it is guaranteed to converge monotonically.

9.3 *Value iteration*

Value iteration can be seen as a dynamic program that computes the optimal value function, where the state is how many timesteps t we look

```
function PolicyIteration(⟨𝒳, 𝒜, p, r⟩)
    randomly initialize π
    while not converged do
        V ← ValueFunction(π)
        π ← GreedyPolicy(V)
    end while
    return π
end function
```

Algorithm 3. Policy iteration algorithm that finds an exact solution in a polynomial number of iterations.

ahead, and $V_t(x)$ is the value function of that. The recurrence relationship is then the following,

$$V_0(x) = \max_{a \in \mathcal{A}} r(x, a)$$
$$V_t(x) = \max_{a \in \mathcal{A}} r(x, a) + \gamma \sum_{x' \in \mathcal{X}} p(x' \mid x, a) V_{t-1}(x').$$

Since V_{t-1} looks $t-1$ steps into the future, V_t looks t steps ahead. We keep iterating until ϵ-optimal convergence.[28]

[28] ϵ-optimal in the sense that the largest difference between $V^\star(x)$ and $V_t(x)$ for any x is at most ϵ.

```
function ValueIteration(⟨𝒳, 𝒜, p, r⟩)
    for x ∈ 𝒳 do                                   ▷ Initialize V_0
        V_0(x) ← max_{a∈𝒜} r(x, a)
    end for
    t ← 0
    while ‖v_t − v_{t−1}‖_∞ > ε do
        t ← t + 1                  ▷ Look one more step into the future
        for x ∈ 𝒳 do
            V_t(x) ← max_{a∈𝒜} r(x, a) + γ Σ_{x'∈𝒳} p(x' | x, a) V_{t−1}(x')
        end for
    end while
    return GreedyPolicy(V_t)
end function
```

Algorithm 4. Value iteration algorithm that finds an ϵ-optimal solution in a polynomial number of iterations. v_t is the vector containing all values of V_t. $\|x\|_\infty$ is the largest value of x.

Recall Theorem 9.2 that states that if a policy is greedy w.r.t. its own value function, it is optimal. Thus, after finding an ϵ-optimal value function, we can simply choose the policy that greedily picks according to this value function.

Value iteration is not guaranteed to converge monotonically, but it is guaranteed to converge to an ϵ-optimal policy in polynomial time. Furthermore, value iteration is inexpensive, compared to policy iteration.

10 Reinforcement learning

In *reinforcement learning* (RL), we are concerned with acting in unknown environments. These environments are still modeled by MDPs, but in RL, we do not have access to the transition probabilities p and reward function r. Thus, RL is at the intersection of probabilistic planning (MDPs) and learning, *i.e.*, everything we have learned thus far comes together here.

Remark. We will start by assuming that the state-action space is finite. Then, we will move on to potentially infinite state spaces. After that, we will learn about infinite action spaces.

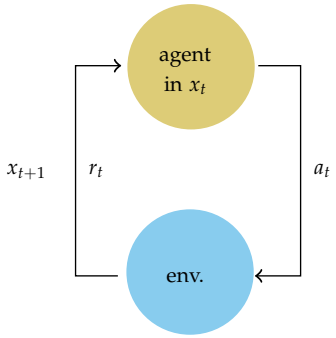


Figure 10.1. In reinforcement learning, an agent interacts with its environment. After playing an action a_t, it observes reward r_t and its new state x_{t+1}.

Since the environment is unknown, we need to explore the state-action space to find where the reward lies. However, we also want to act optimally by exploiting what we have learned thus far. This is called the exploration/exploitation dilemma and is what algorithms need to solve.

Another way that reinforcement learning differs from supervised learning is that data depends on past actions. Trajectory data looks like the following,

$$\tau = (\langle x_0, a_0, r_0, x_1 \rangle, \langle x_1, a_1, r_1, x_2 \rangle, \dots).$$

We differentiate between RL algorithms in two major ways. The first is whether the algorithm has control over its data: an algorithm is called *on-policy* if it controls its own actions from which it learns, and *off-policy* if it can learn from any data. Furthermore, we differentiate between *model-based* and *model-free* algorithms. Model-based algorithms learn the underlying MDP and solve it using value or policy iteration. Model-free algorithms only learn the value function, since, due to Bellman's theorem, that is all that is needed to act optimally.

10.1 Model-based

In model-based RL, we learn the MDP, *i.e.*, we estimate the transition probabilities $p(x' \mid x, a)$ and reward function $r(x, a)$ from the data,

$$\hat{p}(x' \mid x, a) = \frac{\text{count}(x' \mid x, a)}{\text{count}(x, a)}$$

$$\hat{r}(x, a) = \frac{1}{\text{count}(x, a)} \sum_{\substack{t=0 \\ x_t = x \\ a_t = a}}^{\infty} r_t$$

Then, we optimize the policy by value or policy iteration, based on the estimated MDP.

ϵ-greedy. At iteration t, pick random action with probability ϵ_t, or best action (according to internal MDP) with probability $1 - \epsilon_t$. Guaranteed to converge to optimal policy if the ϵ_t sequence satisfies the Robbins Monro conditions. The advantage of this method is that it is extremely simple

A sequence x_t satisfies the Robbins Monro conditions if

$$\sum_{t=0}^{\infty} x_t = \infty, \quad \sum_{t=0}^{\infty} x_t^2 < \infty.$$

E.g., $x_t = \frac{1}{t}$.

Algorithm	Classification		Space compl.
ϵ-greedy	On/off-policy	Model-based	$\mathcal{O}(\lvert\mathcal{A}\rvert \cdot \lvert\mathcal{X}\rvert^2)$
$R_{\max}$	On/off-policy	Model-based	$\mathcal{O}(\lvert\mathcal{A}\rvert \cdot \lvert\mathcal{X}\rvert^2)$
TD-learning	On-policy	Model-free	$\mathcal{O}(\lvert\mathcal{X}\rvert)$
Q-learning	Off-policy	Model-free	$\mathcal{O}(\lvert\mathcal{A}\rvert \cdot \lvert\mathcal{X}\rvert)$
Deep Q Network	Off-policy	Model-free	
REINFORCE	On-policy	Model-free	

Table 2. RL algorithms covered in this text with their types.

and has a clear interpretation w.r.t. the exploration-exploitation dilemma. The disadvantage is that it does not quickly eliminate clearly suboptimal actions. This is because it explores the state space in an uninformed manner. In other words, it explores while ignoring all past experience.

$R_{\max}$ algorithm. $R_{\max}$ solves the problem of ϵ-greedy by using the *Optimism in the face of uncertainty* principle. It assumes that any unexplored states are "fairy tale" states with high reward. More formally, if $r(x, a)$ is unknown, we set $\hat{r}(x, a) = R_{\max}$. Similarly, if $p(x' \mid x, a)$ is unknown, we set $\hat{p}(x^\star \mid x, a) = 1$ for some "fairy tale" state,

$$\begin{aligned} \hat{p}(x^\star \mid x^\star, a) &= 1 && \forall a \in \mathcal{A} \\ \hat{r}(x^\star, a) &= R_{\max} && \forall a \in \mathcal{A}. \end{aligned}$$

This gives us an algorithm that has a bias toward exploring, but once it has explored a part of the state-action space, and observed it to be suboptimal, it can quickly eliminate it. Furthermore, the algorithm does not have to explicitly choose between exploration and exploitation, because it is done by assuming that the unexplored states are optimal.

10.2 Model-free

The problem with model-based RL is that it has high space requirements for storing the MDP, *i.e.*, $\mathcal{O}(\lvert\mathcal{A}\rvert \cdot \lvert\mathcal{X}\rvert^2)$. Furthermore, it requires repeatedly solving the underlying MDP, which is expensive with policy or value iteration. In *model-free* RL, we estimate the value function directly, because that is all we need to act optimally, according to Bellman's theorem. Thus, we also do not need to do any planning, eliminating much computational complexity.

Temporal difference-learning. TD-learning directly computes the value function. Recall the Bellman expectation equation,

$$V^\pi(x) = r(x, \pi(x)) + \gamma \sum_{x' \in \mathcal{X}} p(x' \mid x, \pi(x)) V^\pi(x').$$

Since we do not have access to r and p, we have to make a Monte Carlo estimate given a single data point $\langle x, a, r, x' \rangle$,

$$\approx r + \gamma V^\pi(x').$$

The idea is that we make this approximation repeatedly as the agent collects new data, which achieves the same effect as averaging over many data points. However, there is still a problem: V^π depends on the unknown V^π.

The key idea is to use a bootstrapping estimate of the value function. In other words, instead of the true value function V^π, we will use a running estimate $\hat{V}^\pi$. However, due to relying on a single sample, the value function will have a high variance, which is why we mix the new estimate with the previous one using a learning rate α_t,

$$\hat{V}^\pi(x) \leftarrow (1-\alpha_t)\hat{V}^\pi(x) + \alpha_t(r + \gamma \hat{V}^\pi(x')).$$

If the learning rate α_t satisfies the Robbins Monro conditions and all states are chosen infinitely often, $\hat{V}^\pi$ is guaranteed to converge to the optimal value function V^π.

Note that, due to the Monte Carlo approximation w.r.t. transitions attained by following policy π, TD-learning is a fundamentally on-policy method. Further note that the space requirement of this algorithm is $\mathcal{O}(|\mathcal{X}|)$.

Q-learning. A generalization of TD-learning is Q-learning. Instead of directly learning the value function, which makes it inherently on-policy, it learns state-action values $Q(x,a)$. Then, we can compute $V^\pi(x) = \max_a Q(x,a)$. Like in TD-learning, we mix in the new estimate with the previous one according to learning rate α_t,

$$\hat{Q}(x,a) \leftarrow (1-\alpha_t)\hat{Q}(x,a) + \alpha_t\left(r + \gamma \max_{a' \in \mathcal{A}} \hat{Q}(x',a')\right).$$

The advantage of Q-learning is that it is off-policy, because the value is conditioned on the action. Thus, we can generate as much data as we need using a different algorithm, such as ϵ-greedy, and then estimate the Q-values from there.

Again, Q-learning is optimal if it satisfies the Robbins Monro conditions and all state-action pairs are chosen infinitely often. The space complexity of Q-learning is $\mathcal{O}(|\mathcal{A}| \cdot |\mathcal{X}|)$.

10.3 Model-free deep RL

In deep RL, the input could be anything that machine learning can process, *e.g.*, video game frames with CNN or language with RNN.

The problem with all previously discussed methods is that they are only feasible in a small finite domain. In continuous domains, we would need an infinite amount of memory to store all values. Thus, we need to approximate this regression problem with function approximators, a.k.a. machine learning.

Let $V^\pi(x;\boldsymbol{\theta})$ be the function approximator that approximates the value function. Just like in TD-learning, we make the following Monte Carlo

estimation for a given data point $\langle x, a, r, x' \rangle$,

$$V^\pi(x) \approx r + \gamma V^\pi(x').$$

Then, we can define the loss function of our value function as the squared error from the true value function,

$$\ell(\boldsymbol{\theta}; x, r, x') \doteq \frac{1}{2}(V^\pi(x; \boldsymbol{\theta}) - V^\pi(x)),$$

which we estimate by using the Monte Carlo estimation,

$$\approx \frac{1}{2}\Big(V^\pi(x; \boldsymbol{\theta}) - \Big(r + \gamma V^\pi\Big(x'; \boldsymbol{\theta}^{\text{old}}\Big)\Big)\Big)^2.$$

The gradient of this loss is equal to the following,

$$\boldsymbol{\nabla}_{V^\pi(x;\boldsymbol{\theta})} \ell(\boldsymbol{\theta}; x, r, x') = V^\pi(x; \boldsymbol{\theta}) - \Big(r + \gamma V^\pi\Big(x'; \boldsymbol{\theta}^{\text{old}}\Big)\Big).$$

Using stochastic gradient descent, we then get the following update rule,

$$\begin{aligned} V^\pi(x; \boldsymbol{\theta}) &\leftarrow V^\pi(x; \boldsymbol{\theta}) - \alpha_t\Big(V^\pi(x; \boldsymbol{\theta}) - \Big(r + \gamma V^\pi\Big(x'; \boldsymbol{\theta}^{\text{old}}\Big)\Big)\Big) \\ &= (1 - \alpha_t) V^\pi(x; \boldsymbol{\theta}) + \alpha_t\Big(r + \gamma V^\pi\Big(x'; \boldsymbol{\theta}^{\text{old}}\Big)\Big), \end{aligned}$$

which is the same as the TD-learning update rule. Thus, the TD-learning update rule can be viewed as gradient descent on the squared loss!

Deep Q-network. The same result holds for the Q-learning update rule, where we do gradient descent on

$$\ell(\boldsymbol{\theta}; x, a, x', r) = \frac{1}{2}\left(Q(x, a; \boldsymbol{\theta}) - \left(r + \gamma \max_{a'} Q\Big(x', a; \boldsymbol{\theta}^{\text{old}}\Big)\right)\right)^2.$$

This equation is called the *Bellman error*.

function DEEPQNETWORK(τ)
  initialize $\boldsymbol{\theta}$
  while not converged **do**
    pop $\langle x, a, r, x' \rangle$ from τ
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha_t \delta \boldsymbol{\nabla}_{\boldsymbol{\theta}} Q(x, a; \boldsymbol{\theta})$
    where $\delta \doteq \Big(Q(x, a; \boldsymbol{\theta}) - \Big(r + \gamma \max_{a'} Q\Big(x', a; \boldsymbol{\theta}^{\text{old}}\Big)\Big)\Big)$
  end while
end function

Algorithm 5. Q-learning with function approximation.

However, this algorithm is quite slow to converge. To accelerate, we clone the network and maintain a constant "target" value across episodes, which we update once in a while.

Furthermore, deep Q networks suffer from "maximization bias", which means that it tends to overestimate the actual Q value. This is caused by the max-operator used in the update rule, which also maximizes noise in the data. This is solved by the double deep Q network, where we use two target networks. We then take the minimum at each iteration of the two predictions by the two target networks.

Policy search methods. The problem with deep Q networks is that if the action-space is large or infinite, the $\max_{a'} Q(x', a; \boldsymbol{\theta})$ is no longer feasible. The solution to this is learning a parametrized policy $\pi(x; \boldsymbol{\theta})$, where the output is the action. In this case, we want to maximize the expected trajectory reward,

$$\begin{aligned} J(\boldsymbol{\theta}) &= \mathbb{E}_{x_{0:T}, a_{0:T} \sim \pi_{\boldsymbol{\theta}}} \left[\sum_{t=0}^{T} \gamma^t r(x_t, a_t) \right] \\ &= \mathbb{E}_{\tau \sim \pi_{\boldsymbol{\theta}}}[r(\tau)]. \end{aligned}$$

Theorem 10.1. The following holds,

$$\boldsymbol{\nabla}_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \boldsymbol{\nabla}_{\boldsymbol{\theta}} \mathbb{E}_{\tau \sim \pi_{\boldsymbol{\theta}}}[r(\tau)] = \mathbb{E}_{\tau \sim \pi_{\boldsymbol{\theta}}}[r(\tau) \boldsymbol{\nabla}_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(\tau)].$$

Proof.

$$\begin{aligned} \boldsymbol{\nabla}_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) &= \boldsymbol{\nabla}_{\boldsymbol{\theta}} \int \pi_{\boldsymbol{\theta}}(\tau) r(\tau) d\tau \\ &= \int \boldsymbol{\nabla}_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(\tau) r(\tau) d\tau \\ &= \int \pi_{\boldsymbol{\theta}}(\tau) r(\tau) \boldsymbol{\nabla}_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(\tau) d\tau && \text{Chain rule} \\ &= \mathbb{E}_{\tau \sim \pi_{\boldsymbol{\theta}}}[r(\tau) \boldsymbol{\nabla}_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(\tau)]. \end{aligned}$$

■

Using Theorem 10.1, we do not need to use the reparametrization trick to be able to compute gradients. We only need to compute $\boldsymbol{\nabla}_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(\tau)$. We can compute this gradient as follows,

$$\begin{aligned} \pi_{\boldsymbol{\theta}}(\tau) &= p(x_0) \prod_{t=0}^{T} \pi_{\boldsymbol{\theta}}(a_t \mid x_t) p(x_{t+1} \mid x_t, a_t) \\ \boldsymbol{\nabla}_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(\tau) &= \boldsymbol{\nabla}_{\boldsymbol{\theta}} \log p(x_0) + \sum_{t=0}^{T} \boldsymbol{\nabla}_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t \mid x_t) + \sum_{t=0}^{T} \boldsymbol{\nabla}_{\boldsymbol{\theta}} \log p(x_{t+1} \mid x_t, a_t) \\ &= \sum_{t=0}^{T} \boldsymbol{\nabla}_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t \mid x_t). \end{aligned}$$

So, to be able to compute gradients w.r.t. $\boldsymbol{\theta}$, we do not even need to know the underlying MDP! Putting this together, we get the following gradient,

$$\boldsymbol{\nabla}_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{\tau \sim \pi_{\boldsymbol{\theta}}} \left[r(\tau) \sum_{t=0}^{T} \boldsymbol{\nabla}_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t \mid x_t) \right].$$

Even though these gradients are unbiased, they typically have large variance. We can reduce the variance by introducing *baselines*,

$$\mathbb{E}_{\tau \sim \pi_{\boldsymbol{\theta}}}[r(\tau) \boldsymbol{\nabla}_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(\tau)] = \mathbb{E}_{\tau \sim \pi_{\boldsymbol{\theta}}}[(r(\tau) - b(\tau)) \boldsymbol{\nabla}_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(\tau)],$$

function REINFORCE
  initialize $\boldsymbol{\theta}$
  repeat
    generate an episode τ using $\pi_{\boldsymbol{\theta}}$
    for $t = 1, \dots, T$ **do**
      $G_t \leftarrow r_t$
      $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta \gamma^t G_t \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t \mid x_t)$
    end for
  until done
end function

Algorithm 6. The REINFORCE algorithm, where the baseline at timestep t is set to be $\sum_{t'=0}^{t-1} \gamma^{t'} r_{t'}$.

which have the same gradient. Thus, we are able to shift the reward up or down without influencing the gradient.

REINFORCE (Algorithm 6) sets its baseline adaptively to be the following,

$$b_t(\tau) = \sum_{t'=0}^{t-1} \gamma^{t'} r_{t'}.$$

Then, the gradient becomes the following,

$$\begin{aligned}
\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) &= \mathbb{E}_{t \sim \pi_{\boldsymbol{\theta}}} \left[\sum_{t=0}^{T} \left(\sum_{t'=0}^{T} \gamma^{t'} r_{t'} - \sum_{t'=0}^{t-1} \gamma^{t'} r_{t'} \right) \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t \mid x_t) \right] \\
&= \mathbb{E}_{t \sim \pi_{\boldsymbol{\theta}}} \left[\sum_{t=0}^{T} \gamma^t \left(\sum_{t'=t}^{T} \gamma^{t'-t} r_{t'} \right) \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t \mid x_t) \right] \\
&= \mathbb{E}_{t \sim \pi_{\boldsymbol{\theta}}} \left[\sum_{t=0}^{T} \gamma^t G_t \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t \mid x_t) \right],
\end{aligned}$$

where G_t is the reward to go following action a_t. REINFORCE is an on-policy algorithm, because it requires generating an episode for the data. This is necessary to be able to update its parameters correctly.

Actor-critic methods. We can reinterpret the REINFORCE gradient as follows,

$$\begin{aligned}
\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) &= \mathbb{E}_{\tau \sim \pi_{\boldsymbol{\theta}}} \left[\sum_{t=0}^{T} \gamma^t G_t \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t \mid x_t) \right] \\
&= \sum_{t=0}^{\infty} \mathbb{E}_{\tau_{t:\infty}} \left[\gamma^t G_t \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t \mid x_t) \right] && \text{Linearity of expectation} \\
&= \sum_{t=0}^{\infty} \mathbb{E}_{x_t, a_t} \left[\gamma^t \mathbb{E}_{\tau_{t:\infty}} \left[\sum_{t'=t}^{\infty} \gamma^{t'-t} r_t \,\middle|\, x_t, a_t \right] \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t \mid x_t) \right] && G_t \text{ depends on everything after } t, \text{ while the other terms depend only on } t. \\
&= \mathbb{E}_{\tau \sim \pi_{\boldsymbol{\theta}}} \left[\sum_{t=0}^{\infty} \gamma^t Q^{\pi_{\boldsymbol{\theta}}}(x_t, a_t) \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t \mid x_t) \right] && \text{The Q value is the value that we get after doing } a_t \text{ at } x_t.
\end{aligned}$$

$$r(\tau) - b_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_t \doteq G_t.$$

Intuitively, G_t is the reward to go following action a_t.

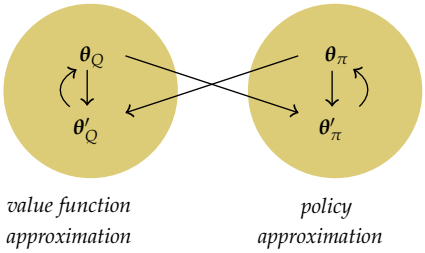


Figure 10.2. Illustration of an iteration of actor-critic methods.

Now, we can obtain the following,

$$
\begin{aligned}
&= \int \rho_{\boldsymbol{\theta}}(x) \mathbb{E}_{a \sim \pi_{\boldsymbol{\theta}}(x)}[Q(x,a) \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a \mid x)] dx \\
&= \mathbb{E}_{(x,a) \sim \pi_{\boldsymbol{\theta}}}[Q(x,a) \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a \mid x)],
\end{aligned}
$$

This is an abuse of notation, because $\rho_{\boldsymbol{\theta}}$ is an unnormalized probability distribution.

where $\rho_{\boldsymbol{\theta}}(x) \doteq \sum_{t=0}^{\infty} \gamma^t p(x_t = x)$ is the unnormalized, discounted state occupancy measure.

This result naturally suggests plugging in approximations for $Q_{\boldsymbol{\theta}}(x,a)$ for the action-value function. The idea of actor-critic networks is to parameterize an actor network that computes the policy and a critic network that computes the Q-value. They can then be used in each others' update equations,

$$
\begin{aligned}
\boldsymbol{\theta}_\pi &\leftarrow \boldsymbol{\theta}_\pi + \eta_t Q_{\boldsymbol{\theta}}(x,a) \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a \mid x) \\
\boldsymbol{\theta}_Q &\leftarrow \boldsymbol{\theta}_Q - \eta_t \big(Q_{\boldsymbol{\theta}}(x,a) - r - \gamma Q_{\boldsymbol{\theta}}(x', \pi_{\boldsymbol{\theta}}(x'))\big) \nabla_{\boldsymbol{\theta}} Q_{\boldsymbol{\theta}}(x,a).
\end{aligned}
$$

Furthermore, we can introduce baselines by adding a value network,

$$
\begin{aligned}
\boldsymbol{\theta}_\pi &\leftarrow \boldsymbol{\theta}_\pi + \eta_t (Q_{\boldsymbol{\theta}}(x,a) - V_{\boldsymbol{\theta}}(x)) \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a \mid x) \\
&= \boldsymbol{\theta}_\pi + \eta_t A(x,a) \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a \mid x),
\end{aligned}
$$

where $A(x,a) \doteq Q(x,a) - V(x)$ is the advantage function, which is positive if the chosen action is better than expected and negative if worse. Thus, intuitively, we increase the probability of the chosen action if better than expected, otherwise we decrease it. This model is called *advantage actor critic* (A2C) [Mnih et al., 2016].

All models discussed so far have been on-policy, which often causes sample inefficiency. Now, we want to move to off-policy methods. Recall that our initial motivation was that finding the maximum Q value was intractable if the action space was infinite. But, we could also replace the exact maximum by a parametrized policy,

$$
J(\boldsymbol{\theta}) = \sum_{(x,a,r,x') \in \mathcal{D}} \Big(Q_{\boldsymbol{\theta}}(x,a;\boldsymbol{\theta}_Q) - \Big(r + \gamma Q\Big(x', \pi(x';\boldsymbol{\theta}_\pi); \boldsymbol{\theta}_Q^{\text{old}}\Big)\Big)\Big)^2,
$$

where we jointly optimize over $\boldsymbol{\theta}_Q$ and $\boldsymbol{\theta}_\pi$. We want to follow the greedy policy w.r.t. the Q function, *i.e.*, we want $\pi_{\boldsymbol{\theta}} \approx \pi_Q = \operatorname{argmax}_{a \in \mathcal{A}} Q(x,a;\boldsymbol{\theta}_Q)$. The key idea is that if we use a "rich enough" parametrization of policies, selecting the greedy policy w.r.t. Q is equivalent to the following,

$$
\boldsymbol{\theta}_\pi^\star = \operatorname*{argmax}_{\boldsymbol{\theta}_\pi} \mathbb{E}_{x \sim \mu}[Q(x, \pi(x;\boldsymbol{\theta}_\pi);\boldsymbol{\theta}_Q)],
$$

where $\mu(x) > 0$ is an *exploration distribution* over states with full support. If we then use differentiable approximations of Q and a differentiable deterministic policy π, we can use backpropagation to obtain gradients,

$$
\nabla_{\boldsymbol{\theta}_\pi} Q(x, \pi(x;\boldsymbol{\theta}_\pi);\boldsymbol{\theta}_Q) = \nabla_{\pi(x;\boldsymbol{\theta}_\pi)} Q(x, \boldsymbol{\theta}_\pi) \nabla_{\boldsymbol{\theta}_\pi} \pi(x;\boldsymbol{\theta}_\pi).
$$

However, policy gradient methods rely on randomized policies for exploration, but we have deterministic policies. To encourage exploration, we can inject additional Gaussian action noise to encourage exploration, akin to ϵ-greedy exploration. This is called the *deep deterministic policy gradients* (DDPG) algorithm [Lillicrap et al., 2015].

Twin delayed deep deterministic (TD3) further improves this by introducing a second critic network to address maximization bias [Fujimoto et al., 2018]. *Soft-actor critic* (SAC) further improves this by adding entropy regularization,

$$J_\lambda(\boldsymbol{\theta}) = J(\boldsymbol{\theta}) + \lambda H(\pi_{\boldsymbol{\theta}}),$$

which encouraging exploration by giving preference to high-entropy actions [Haarnoja et al., 2018].

10.4 Model-based deep RL

So far, we have only discussed deep model-free methods. However, if we have an accurate model of the environment, we can use it for planning. The main benefit of this is that it dramatically reduces the sample complexity, compared to model-free techniques. In other words, we need much less data to find a good policy.

Planning in a known model. We assume a continuous state and action space with non-linear transitions, without constraints. Thus, this is quite a bit more complex than solving MDPs. We have a deterministic transition function f and a reward function r. The objective then becomes the following,

$$J_\infty(a_{0:\infty}) = \sum_{t=0}^{\infty} \gamma^t r(x_t, a_t). \qquad \text{such that } x_{t+1} = f(x_t, a_t) \text{ for every } x_t.$$

However, we cannot plan over an infinite horizon. The key idea is to plan over a finite horizon H, carry out the first action, then replan with a horizon H. Thus, we first optimize over the following

$$J_H(a_{t:t+H-1}) = \sum_{t'=t}^{t+H-1} \gamma^{t'-t} r(x_t, a_t), \qquad \text{such that } x_{t+1} = f(x_t, a_t) \text{ for every } x_t.$$

carry out action a_t and then replan. We can optimize this function using gradient methods (backpropagation through time) if the actions, rewards, and dynamics are differentiable. However, there are often many local minima, and vanishing/exploding gradients are a problem, because we do backpropagation through time.[29] Thus, we often use heuristic global optimization methods. The *random shooting* method generates m sets of random samples $a_{t:t+H-1}$, and then picks the sequence that optimizes the objective.

[29] This is the same problem that recurrent neural networks have.

However, if we have sparse rewards, this will inherently not work, because we might not look far enough into the future. But, if we have

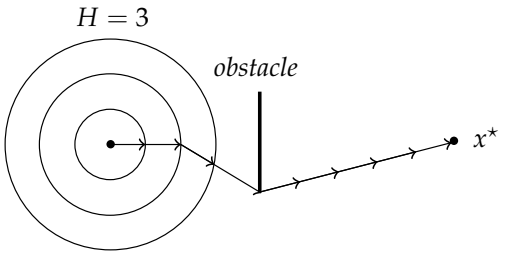


Figure 10.3. Illustration of the effect of a finite horizon. Only after two steps, the agent could "see" the obstacle within its horizon.

access to a value function estimate, we can use that to see beyond the finite horizon, giving us the following object function,

$$J_H(a_{t:t+H-1}) = \sum_{t'=t:t+H-1} \gamma^{t'-t} r_t + \gamma^H V(x_{t+H}),$$

where, intuitively, $V(x_{t+H})$ summarizes the remaining infinite timesteps. For $H = 1$, this is equal to the greedy policy w.r.t. V, but if we use larger H, this converges faster.

If the transition function is stochastic, rather than deterministic, we have to optimize over the expected performance,

$$J_H(a_{t:t+H-1}) = \mathbb{E}_{x_{t+1:t+H}}\left[\sum_{t'=t}^{t+H-1} \gamma^{t'-t} r_t + \gamma^H V(x_{t+H}) \,\middle|\, a_{t:t+H-1}\right].$$

However, the problem is that this expectation requires solving a high-dimensional integral. A common solution is to use *Monte Carlo trajectory sampling*. For this we use the reparametrization trick to obtain unbiased Monte Carlo estimates,

$$\hat{J}_H(a_{t:t+H-1}) = \frac{1}{m}\sum_{i=1}^{m}\sum_{t'=t}^{t+H-1} \gamma^{t'-t} r_t(x_t\left(a_{t:t'-1}, \epsilon^{(i)}_{t:t'-1}\right), a_t) + \gamma^H V(x_{t+H}).$$

The state is a function of the previous actions $a_{t:t'-1}$ and noise $\epsilon^{(i)}_{t:t'-1}$ that is needed for the Gaussian reparameterization trick.

Instead of optimizing over the actions, we could also optimize over parametrized policies $\pi_{\boldsymbol{\theta}}$. This replaces expensive online planning by offline training of a policy that is fast to evaluate online. The objective then becomes the following,

$$J(\boldsymbol{\theta}) = \mathbb{E}_{x_0\sim\mu}\left[\sum_{t=0}^{H-1} \gamma^t r_t + \gamma^H Q(x_H, \pi_{\boldsymbol{\theta}}(x_H))\right].$$

For $H = 0$, this is equivalent to the DDPG objective.

Learning the model. Until now, we have assumed known f and r. In RL, these are unknown, so we have to learn them. The key insight is that due to the Markovian structure of the MDP, the observed transitions and rewards are conditionally independent. We can estimate them off-policy with standard supervised learning techniques from a dataset of

trajectories. We train a model with inputs (x_t, a_t) and the model outputs (r_t, x_{t+1}). For continuous state spaces, this is essentially just a regression problem.

We could use an MAP estimate, but errors in the model estimate compound, which the planning algorithms exploit, which results in poor performance. This can easily be remedied by capturing uncertainty (Gaussian processes, Bayesian neural networks) in the estimated model and taking it into account in planning.

Exploration and exploitation. An algorithm that we can use to balance exploration and exploitation is Thompson sampling, which we have already seen before. We sample a model, plan a new policy according to it, roll out policy to collect more data, and finally update the posterior.

References

Scott Fujimoto, Herke Hoof, and David Meger. Addressing function approximation error in actor-critic methods. In *International conference on machine learning*, pages 1587–1596. PMLR, 2018.

Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International conference on machine learning*, pages 1861–1870. PMLR, 2018.

Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

Wesley J Maddox, Pavel Izmailov, Timur Garipov, Dmitry P Vetrov, and Andrew Gordon Wilson. A simple baseline for bayesian uncertainty in deep learning. *Advances in neural information processing systems*, 32, 2019.

Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937. PMLR, 2016.