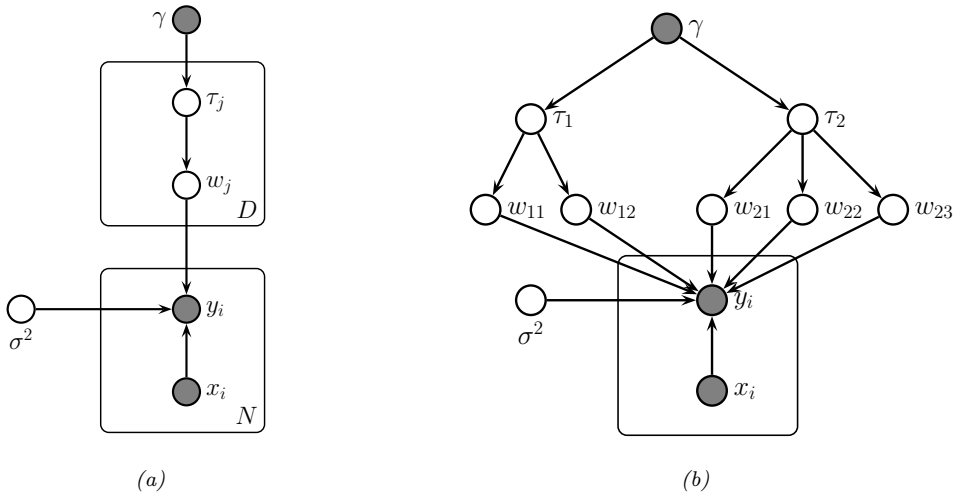


Figure 15.3: (a) Representing lasso using a Gaussian scale mixture prior. (b) Graphical model for group lasso with 2 groups, the first has size $G_1 = 2$, the second has size $G_2 = 3$.

Unfortunately, MAP estimation (not to mention full Bayesian inference) with such discrete mixture priors is computationally difficult. Various approximate inference methods have been proposed, including greedy search (see e.g., [SPZ09]) or MCMC (see e.g., [HS09]).

15.2.5 Laplace prior (Bayesian lasso)

A computationally cheap way to achieve sparsity is to perform MAP estimation with a Laplace prior by minimizing the penalized negative log likelihood:

$$\text{PNLL}(\boldsymbol{w}) = -\log p(\mathcal{D}|\boldsymbol{w}) - \log p(\boldsymbol{w}|\lambda) = ||\mathbf{X}\boldsymbol{w} - \boldsymbol{y}||_2^2 + \lambda||\boldsymbol{w}||_1 \tag{15.84}$$

where $||\boldsymbol{w}||_1 \triangleq \sum_{d=1}^{D} |w_d|$ is the ℓ_1 norm of $\boldsymbol{w}$. This method is called **lasso**, which stands for "least absolute shrinkage and selection operator" [Tib96]. See Section 11.4 of the prequel to this book, [Mur22], for details.

In this section, we discuss posterior inference with this prior; this is known as the **Bayesian lasso** [PC08]. In particular, we assume the following prior:

$$p(\boldsymbol{w}|\sigma^2) = \prod_j \frac{\lambda}{2\sqrt{\sigma^2}} e^{-\lambda|w_j|/\sqrt{\sigma^2}} \tag{15.85}$$

(Note that conditioning the prior on σ^2 is important to ensure that the full posterior is unimodal.)

To simplify inference, we will represent the Laplace prior as a Gaussian scale mixture, which we discussed in Section 28.2.3.2. In particular, one can show that the Laplace distribution is an infinite

weighted sum of Gaussians, where the precision comes from a Gamma distribution:

$$\text{Laplace}(w|0,\lambda) = \int \mathcal{N}(w|0,\tau^2)\text{Ga}(\tau^2|1,\frac{\lambda^2}{2})d\tau^2 \tag{15.86}$$

We can therefore represent the Bayesian lasso model as a hierarchical latent variable model, as shown in Figure 15.3a. The corresponding joint distribution has the following form:

$$p(\boldsymbol{y},\boldsymbol{w},\boldsymbol{\tau},\sigma^2|\mathbf{X}) = \mathcal{N}(\boldsymbol{y}|\mathbf{X}\boldsymbol{w},\sigma^2\mathbf{I}_N)\left[\prod_j \mathcal{N}(w_j|0,\sigma^2\tau_j^2)\text{Ga}(\tau_j^2|1,\lambda^2/2)\right]p(\sigma^2) \tag{15.87}$$

We can also create a GSM to match the **group lasso** prior, which sets multiple coefficients to zero at the same time:

$$\boldsymbol{w}_g|\sigma^2,\tau_g^2 \sim \mathcal{N}(\mathbf{0},\sigma^2\tau_g^2\mathbf{I}_{d_g}) \tag{15.88}$$

$$\tau_g^2 \sim \text{Ga}(\frac{d_g+1}{2},\frac{\lambda^2}{2}) \tag{15.89}$$

where d_g is the size of group g. So we see that there is one variance term per group, each of which comes from a Gamma prior, whose shape parameter depends on the group size, and whose rate parameter is controlled by γ.

Figure 15.3b gives an example, where we have 2 groups, one of size 2 and one of size 3. This picture makes it clearer why there should be a grouping effect. For example, suppose $w_{1,1}$ is small; then τ_1^2 will be estimated to be small, which will force $w_{1,2}$ to be small, due to shrinkage (c.f., Section 3.7). Conversely, suppose $w_{1,1}$ is large; then τ_1^2 will be estimated to be large, which will allow $w_{1,2}$ to be become large as well.

Given these hierachical models, we can easily derive a Gibbs sampling algorithm (Section 12.3) to sample from the posterior (see e.g., [PC08]). Unfortunately, these posterior samples are not sparse, even though the MAP estimate is sparse. This is because the prior puts infinitessimal probability on the event that each coefficient is zero.

15.2.6 Horseshoe prior

The Laplace prior is not suitable for sparse Bayesian models, because posterior samples are not sparse. The spike and slab prior does not have this problem but is often too slow to use (although see [BRG20]). Fortunately, it is possible to devise continuous priors (without discrete latent variables) that are both sparse and computationally efficient. One popular prior of this type is the **horseshoe prior** [CPS10], so-named because of the shape of its density function.

In the horseshoe prior, instead of using a Laplace prior for each weight, we use the following Gaussian scale mixture:

$$w_j \sim \mathcal{N}(0,\lambda_j^2\tau^2) \tag{15.90}$$

$$\lambda_j \sim \mathcal{C}_+(0,1) \tag{15.91}$$

$$\tau^2 \sim \mathcal{C}_+(0,1) \tag{15.92}$$

where $\mathcal{C}_+(0,1)$ is the half-Cauchy distribution (Section 2.2.2.4), λ_j is a local shrinkage factor, and τ^2 is a global shrinkage factor. The Cauchy distribution has very fat tails, so λ_j is likely to be either 0 or very far from 0, which emulates the spike and slab prior, but in a continuous way. For more details, see e.g., [Bha+19].

15.2.7 Automatic relevancy determination

An alternative to using posterior inference with a sparsity promoting prior is to use posterior inference with a Gaussian prior, $w_j \sim \mathcal{N}(0, 1/\alpha_j)$, but where we use empirical Bayes to optimize the precisions α_j. That is, we first compute $\hat{\boldsymbol{\alpha}} = \text{argmax}_{\boldsymbol{\alpha}}\, p(\boldsymbol{y}|\mathbf{X}, \boldsymbol{\alpha})$, and then compute $\hat{\boldsymbol{w}} = \text{argmax}_{\boldsymbol{w}}\, \mathcal{N}(\boldsymbol{w}|\mathbf{0}, \hat{\boldsymbol{\alpha}}^{-1})$. Perhaps surprisingly, we will see that this results in a sparse estimate, for reasons we explain in Section 15.2.7.2.

This technique is known as **sparse Bayesian learning** [Tip01] or **automatic relevancy determination** (**ARD**) [Mac95; Nea96]. It has also been called **NUV** estimation, which stands for "normal prior with unknown variance" [Loe+16]. It was originally developed for neural networks (where sparsity is applied to the first layer weights), but here we apply it to linear models.

15.2.7.1 ARD for linear models

In this section, we explain ARD in more detail, by applying it to linear regression. The likelihood is $p(y|\boldsymbol{x}, \boldsymbol{w}, \beta) = \mathcal{N}(y|\boldsymbol{w}^\mathsf{T}\boldsymbol{x}, 1/\beta)$, where $\beta = 1/\sigma^2$. The prior is $p(\boldsymbol{w}) = \mathcal{N}(\boldsymbol{w}|\mathbf{0}, \mathbf{A}^{-1})$, where $\mathbf{A} = \text{diag}(\boldsymbol{\alpha})$. The marginal likelihood can be computed analytically (using Equation (2.90)) as follows:

$$p(\boldsymbol{y}|\mathbf{X}, \boldsymbol{\alpha}, \beta) = \int \mathcal{N}(\boldsymbol{y}|\mathbf{X}\boldsymbol{w}, (1/\beta)\mathbf{I}_{N_\mathcal{D}})\mathcal{N}(\boldsymbol{w}|\mathbf{0}, \mathbf{A}^{-1})d\boldsymbol{w} \tag{15.93}$$

$$= \mathcal{N}(\boldsymbol{y}|\mathbf{0}, \beta^{-1}\mathbf{I}_{N_\mathcal{D}} + \mathbf{X}\mathbf{A}^{-1}\mathbf{X}^\mathsf{T}) \tag{15.94}$$

$$= \mathcal{N}(\boldsymbol{y}|\mathbf{0}, \mathbf{C}_{\boldsymbol{\alpha}}) \tag{15.95}$$

where $\mathbf{C}_{\boldsymbol{\alpha}} \triangleq \beta^{-1}\mathbf{I}_N + \mathbf{X}\mathbf{A}^{-1}\mathbf{X}^\mathsf{T}$. This is very similar to the marginal likelihood under the spike-and-slab prior (Section 15.2.4), which is given by

$$p(\boldsymbol{y}|\mathbf{X}, \boldsymbol{s}, \sigma_w^2, \sigma_y^2) = \int \mathcal{N}(\boldsymbol{y}|\mathbf{X}_{\boldsymbol{s}}\boldsymbol{w}_{\boldsymbol{s}}, \sigma_y^2\mathbf{I})\mathcal{N}(\boldsymbol{w}_{\boldsymbol{s}}|\mathbf{0}_{\boldsymbol{s}}, \sigma_w^2\mathbf{I})d\boldsymbol{w}_{\boldsymbol{s}} = \mathcal{N}(\boldsymbol{y}|\mathbf{0}, \mathbf{C}_{\boldsymbol{s}}) \tag{15.96}$$

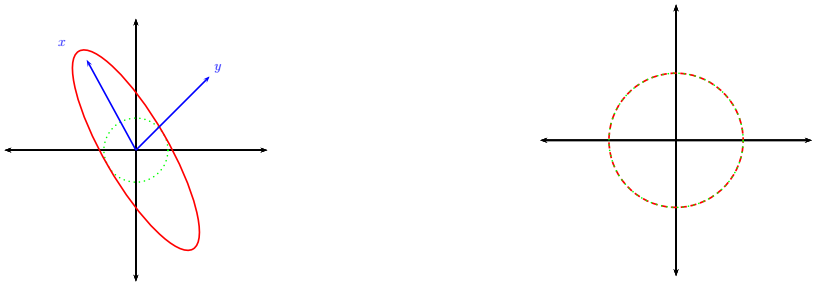
where $\mathbf{C}_{\boldsymbol{s}} = \sigma_y^2\mathbf{I}_{N_\mathcal{D}} + \sigma_w^2\mathbf{X}_{\boldsymbol{s}}\mathbf{X}_{\boldsymbol{s}}^\mathsf{T}$. (Here $\mathbf{X}_{\boldsymbol{s}}$ refers to the design matrix where we select only the columns of $\mathbf{X}$ where $s_d = 1$.) The difference is that we have replaced the binary $s_j \in \{0, 1\}$ variables with continuous $\alpha_j \in \mathbb{R}^+$, which makes the optimization problem easier.

The objective is the log marginal likelihood, given by

$$\ell(\boldsymbol{\alpha}, \beta) = -\frac{1}{2}\log p(\boldsymbol{y}|\mathbf{X}, \boldsymbol{\alpha}, \beta) = \log|\mathbf{C}_{\boldsymbol{\alpha}}| + \boldsymbol{y}^\mathsf{T}\mathbf{C}_{\boldsymbol{\alpha}}^{-1}\boldsymbol{y} \tag{15.97}$$

There are various algorithms for optimizing $\ell(\boldsymbol{\alpha}, \beta)$, some of which we discuss in Section 15.2.7.3.

ARD can be used as an alternative to ℓ_1 regularization. Although the ARD objective is not convex, it tends to give much sparser results [WW12]. In addition, it can be shown [WRN10] that the ARD objective has many fewer local optima than the ℓ_0-regularized objective, and hence is much easier to optimize.



(a) (b)

Figure 15.4: Illustration of why ARD results in sparsity. The vector of inputs $\boldsymbol{x}$ does not point towards the vector of outputs $\boldsymbol{y}$, so the feature should be removed. (a) For finite α, the probability density is spread in directions away from $\boldsymbol{y}$. (b) When $\alpha = \infty$, the probability density at $\boldsymbol{y}$ is maximized. Adapted from Figure 8 of [Tip01].

15.2.7.2 Why does ARD result in a sparse solution?

Once we have estimated $\boldsymbol{\alpha}$ and β, we can compute the posterior over the parameters using Bayes rule for Gaussians, to get $p(\boldsymbol{w}|\mathcal{D}, \hat{\boldsymbol{\alpha}}, \hat{\beta}) = \mathcal{N}(\boldsymbol{w}|\ \widehat{\boldsymbol{w}}, \widehat{\boldsymbol{\Sigma}})$, where $\widehat{\boldsymbol{\Sigma}}^{-1} = \hat{\beta}\mathbf{X}^{\mathsf{T}}\mathbf{X} + \mathbf{A}$ and $\widehat{\boldsymbol{w}} = \hat{\beta}\ \widehat{\boldsymbol{\Sigma}}\ \mathbf{X}^{\mathsf{T}}\boldsymbol{y}$. If we have $\hat{\alpha}_d \approx \infty$, then $\widehat{w}_d \approx 0$, so the solution vector will be sparse.

We now give an intuitive argument, based on [Tip01], about when such a sparse solution may be optimal. We shall assume $\beta = 1/\sigma^2$ is fixed for simplicity. Consider a 1d linear regression with 2 training examples, so $\mathbf{X} = \boldsymbol{x} = (x_1, x_2)$, and $\boldsymbol{y} = (y_1, y_2)$. We can plot $\boldsymbol{x}$ and $\boldsymbol{y}$ as vectors in the plane, as shown in Figure 15.4. Suppose the feature is irrelevant for predicting the response, so $\boldsymbol{x}$ points in a nearly orthogonal direction to $\boldsymbol{y}$. Let us see what happens to the marginal likelihood as we change α. The marginal likelihood is given by $p(\boldsymbol{y}|\boldsymbol{x}, \alpha, \beta) = \mathcal{N}(\boldsymbol{y}|\mathbf{0}, \mathbf{C}_\alpha)$, where $\mathbf{C}_\alpha = \frac{1}{\beta}\mathbf{I} + \frac{1}{\alpha}\boldsymbol{x}\boldsymbol{x}^{\mathsf{T}}$. If α is finite, the posterior will be elongated along the direction of $\boldsymbol{x}$, as in Figure 15.4(a). However, if $\alpha = \infty$, we have $\mathbf{C}_\alpha = \frac{1}{\beta}\mathbf{I}$, which is spherical, as in Figure 15.4(b). If $|\mathbf{C}_\alpha|$ is held constant, the latter assigns higher probability density to the observed response vector $\boldsymbol{y}$, so this is the preferred solution. In other words, the marginal likelihood "punishes" solutions where α_d is small but $\mathbf{X}_{:,d}$ is irrelevant, since these waste probability mass. It is more parsimonious (from the point of view of Bayesian Occam's razor) to eliminate redundant dimensions.

Another way to understand the sparsity properties of ARD is as approximate inference in a hierarchical Bayesian model [BT00]. In particular, suppose we put a conjugate prior on each precision, $\alpha_d \sim \text{Ga}(a, b)$, and on the observation precision, $\beta \sim \text{Ga}(c, d)$. Since exact inference with a Student prior is intractable, we can use variational Bayes (Section 10.2.3), with a factored posterior approximation of the form

$$q(\boldsymbol{w}, \boldsymbol{\alpha}) = q(\boldsymbol{w})q(\boldsymbol{\alpha}) \approx \mathcal{N}(\boldsymbol{w}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) \prod_d \text{Ga}(\alpha_d|\ \widehat{a}_d, \widehat{b}_d) \tag{15.98}$$

ARD approximates $q(\boldsymbol{\alpha})$ by a point estimate. However, in VB, we integrate out $\boldsymbol{\alpha}$; the resulting

posterior marginal $q(\boldsymbol{w})$ on the weights is given by

$$p(\boldsymbol{w}|\mathcal{D}) = \int \mathcal{N}(\boldsymbol{w}|\mathbf{0}, \text{diag}(\boldsymbol{\alpha})^{-1}) \prod_d \text{Ga}(\alpha_d|\, \widehat{a}_d, \widehat{b}) d\boldsymbol{\alpha} \tag{15.99}$$

This is a Gaussian scale mixture, and can be shown to be the same as a multivariate Student distribution (see Section 28.2.3.1), with non-diagonal covariance. Note that the Student has a large spike at 0, which intuitively explains why the posterior mean (which, for a Student distribution, is equal to the posterior mode) is sparse.

Finally, we can also view ARD as a MAP estimation problem with a **non-factorial prior** [WN07]. Intuitively, the dependence between the w_j parameters arises, despite the use of a diagonal Gaussian prior, because the prior precision α_j is estimated based after marginalizing out all $\boldsymbol{w}$, and hence depends on all the features. Interestingly, [WRN10] prove that MAP estimation with non-factorial priors is strictly better than MAP estimation with any possible factorial prior in the following sense: the non-factorial objective always has fewer local minima than factorial objectives, while still satisfying the property that the global optimum of the non-factorial objective corresponds to the global optimum of the ℓ_0 objective — a property that ℓ_1 regularization, which has no local minima, does not enjoy.

15.2.7.3 Algorithms for ARD

There are various algorithms for optimizing $\ell(\boldsymbol{\alpha}, \beta)$. One approach is to use EM, in which we compute $p(\boldsymbol{w}|\mathcal{D}, \boldsymbol{\alpha})$ in the E step and then maximize $\boldsymbol{\alpha}$ in the M step. In variational Bayes, we infer both $\boldsymbol{w}$ and $\boldsymbol{\alpha}$ (see [Dru08] for details). In [WN10], they present a method based on iteratively reweighted ℓ_1 estimation.

Recently, [HXW17] showed that the nested iterative computations performed these methods can emulated by a recurrent neural network (Section 16.3.4). Furthermore, by training this model, it is possible to achieve much faster convergence than manually designed optimization algorithms.

15.2.7.4 Relevance vector machines

Suppose we create a linear regression model of the form $p(y|\boldsymbol{x}; \boldsymbol{\theta}) = \mathcal{N}(y|\boldsymbol{w}^\mathsf{T}\boldsymbol{\phi}(\boldsymbol{x}), \sigma^2)$, where $\boldsymbol{\phi}(\boldsymbol{x}) = [\mathcal{K}(\boldsymbol{x}, \boldsymbol{x}_1), \ldots, \mathcal{K}(\boldsymbol{x}, \boldsymbol{x}_N)]$, where $\mathcal{K}()$ is a kernel function (Section 18.2) and $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N$ are the N training points. This is called **kernel basis function expansion**, and transforms the input from $\boldsymbol{x} \in \mathcal{X}$ to $\boldsymbol{\phi}(\boldsymbol{x}) \in \mathbb{R}^N$. Obviously this model has $O(N)$ parameters, and hence is nonparametric. However, we can use ARD to select a small subset of the exemplars. This technique is called the relevance vector machine (RVM) [Tip01; TF03].

15.2.8 Multivariate linear regression

This section is written by Xinglong Li.

In this section, we consider the **multivariate linear regression** model, which has the form

$$\mathbf{Y} = \mathbf{W}\mathbf{X} + \mathbf{E} \tag{15.100}$$

where $\mathbf{W} \in \mathbb{R}^{N_y \times N_x}$ is the matrix of regression coefficient, $\mathbf{X} \in \mathbb{R}^{N_x \times N_\mathcal{D}}$ is the matrix of input features (with each row being an input variable and each *column* being an observation), $\mathbf{Y} \in \mathbb{R}^{N_y \times N_\mathcal{D}}$

is the matrix of responses (with each row being an output variable and each *column* being an observation), and $\mathbf{E} = [\boldsymbol{e}_1, \cdots, \boldsymbol{e}_{N_\mathcal{D}}]$ is the matrix of residual errors, where $\boldsymbol{e}_i \overset{i.i.d.}{\sim} \mathcal{N}(\mathbf{0}, \boldsymbol{\Sigma})$. It can be seen from the definition that given $\boldsymbol{\Sigma}$, $\mathbf{W}$ and $\mathbf{X}$, columns of $\mathbf{Y}$ are independently random variables following multivariate normal distributions. So the likelihood of the observation is

$$p(\mathbf{Y}|\mathbf{W}, \mathbf{X}, \boldsymbol{\Sigma}) = \frac{1}{(2\pi)^{N_y \times N_\mathcal{D}} |\boldsymbol{\Sigma}|^{N_\mathcal{D}/2}} \exp\left(\sum_{i=1}^{N_\mathcal{D}} -\frac{1}{2}(\boldsymbol{y}_i - \mathbf{W}\boldsymbol{x}_i)^\mathsf{T} \boldsymbol{\Sigma}^{-1} (\boldsymbol{y}_i - \mathbf{W}\boldsymbol{x}_i)\right) \tag{15.101}$$

$$= \frac{1}{(2\pi)^{N_y \times N_\mathcal{D}} |\boldsymbol{\Sigma}|^{N_\mathcal{D}/2}} \exp\left(-\frac{1}{2}\text{trace}\left((\mathbf{Y} - \mathbf{W}\mathbf{X})^\mathsf{T} \boldsymbol{\Sigma}^{-1} (\mathbf{Y} - \mathbf{W}\mathbf{X})\right)\right) \tag{15.102}$$

$$= \mathcal{MN}(\mathbf{Y}|\mathbf{W}\mathbf{X}, \boldsymbol{\Sigma}, \mathbf{I}_{N_\mathcal{D} \times N_\mathcal{D}}), \tag{15.103}$$

The conjugate prior for this is the **matrix normal inverse Wishart** distribution,

$$\mathbf{W}, \boldsymbol{\Sigma} \sim \text{MNIW}(\mathbf{M}_0, \mathbf{V}_0, \nu_0, \boldsymbol{\Psi}_0) \tag{15.104}$$

where the MNIW is defined by

$$\mathbf{W}|\boldsymbol{\Sigma} \sim \mathcal{MN}(\mathbf{M}_0, \boldsymbol{\Sigma}_0, \mathbf{V}_0) \tag{15.105}$$

$$\boldsymbol{\Sigma} \sim \text{IW}(\nu_0, \boldsymbol{\Psi}_0), \tag{15.106}$$

where $\mathbf{V}_0 \in \mathbb{R}_{++}^{N_x \times N_x}$, $\boldsymbol{\Psi}_0 \in \mathbb{R}_{++}^{N_y \times N_y}$ and $\nu_0 > N_x - 1$ is the degree of freedom of the inverse Wishart distribution.

The posterior distribution of $\{\mathbf{W}, \boldsymbol{\Sigma}\}$ still follows a matrix normal inverse Wishart distribution. We follow the derivation in [Fox09, App.F]. Firstly, the density of the joint distribution is

$$p(\mathbf{Y}, \mathbf{W}, \boldsymbol{\Sigma}) \propto |\boldsymbol{\Sigma}|^{-(\nu_0 + N_y + 1 + N_x + N_\mathcal{D})/2} \times \exp\left\{-\frac{1}{2}\text{trace}(\boldsymbol{\Omega}_0)\right\} \tag{15.107}$$

$$\boldsymbol{\Omega}_0 \triangleq \boldsymbol{\Psi}_0 \boldsymbol{\Sigma}^{-1} + (\mathbf{Y} - \mathbf{W}\mathbf{X})^\mathsf{T} \boldsymbol{\Sigma}^{-1} (\mathbf{Y} - \mathbf{W}\mathbf{X}) + (\mathbf{W} - \mathbf{M}_0)^\mathsf{T} \boldsymbol{\Sigma}^{-1} (\mathbf{W} - \mathbf{M}_0) \mathbf{V}_0 \tag{15.108}$$

We firstly aggregate items including $\mathbf{W}$ in the exponent so that it takes the form of a matrix normal distribution. This is similar to the "completing the square" technique that we used in deriving the conjugate posterior for multivariate normal distributions in Section 3.3.3.3. Specifically,

$$\text{trace}\left[(\mathbf{Y} - \mathbf{W}\mathbf{X})^\mathsf{T} \boldsymbol{\Sigma}^{-1} (\mathbf{Y} - \mathbf{W}\mathbf{X}) + (\mathbf{W} - \mathbf{M}_0)^\mathsf{T} \boldsymbol{\Sigma}^{-1} (\mathbf{W} - \mathbf{M}_0) \mathbf{V}_0\right] \tag{15.109}$$

$$= \text{trace}\left(\boldsymbol{\Sigma}^{-1}[(\mathbf{Y} - \mathbf{W}\mathbf{X})(\mathbf{Y} - \mathbf{W}\mathbf{X})^\mathsf{T} + (\mathbf{W} - \mathbf{M}_0)\mathbf{V}_0(\mathbf{W} - \mathbf{M}_0)^\mathsf{T}]\right) \tag{15.110}$$

$$= \text{trace}\left(\boldsymbol{\Sigma}^{-1}[\mathbf{W}\mathbf{S}_{xx}\mathbf{W}^\mathsf{T} - 2\mathbf{S}_{yx}\mathbf{W}^\mathsf{T} + \mathbf{S}_{yy}]\right) \tag{15.111}$$

$$= \text{trace}\left(\boldsymbol{\Sigma}^{-1}[(\mathbf{W} - \mathbf{S}_{yx}\mathbf{S}_{xx}^{-1})\mathbf{S}_{xx}(\mathbf{W} - \mathbf{S}_{yx}\mathbf{S}_{xx}^{-1})^\mathsf{T} + \mathbf{S}_{y|x}]\right). \tag{15.112}$$

where

$$\mathbf{S}_{xx} = \mathbf{X}\mathbf{X}^\mathsf{T} + \mathbf{V}_0, \qquad \mathbf{S}_{yx} = \mathbf{Y}\mathbf{X}^\mathsf{T} + \mathbf{M}_0\mathbf{V}_0, \tag{15.113}$$

$$\mathbf{S}_{yy} = \mathbf{Y}\mathbf{Y}^\mathsf{T} + \mathbf{M}_0\mathbf{V}_0\mathbf{M}_0^\mathsf{T}, \qquad \mathbf{S}_{y|x} = \mathbf{S}_{yy} - \mathbf{S}_{yx}\mathbf{S}_{xx}^{-1}\mathbf{S}_{yx}^\mathsf{T}. \tag{15.114}$$

Therefore, it can be see from Equation (15.112) that given $\boldsymbol{\Sigma}$, $\mathbf{W}$ follows a matrix normal distribution

$$\mathbf{W}|\boldsymbol{\Sigma}, \mathbf{X}, \mathbf{Y} \sim \mathcal{MN}(\mathbf{S}_{yx}\mathbf{S}_{xx}^{-1}, \boldsymbol{\Sigma}, \mathbf{S}_{xx}). \tag{15.115}$$

Marginalizing out $\mathbf{W}$ (which corresponds to removing the terms including $\mathbf{W}$ in the exponent in Equation (15.108)), it can be shown that the posterior distribution of $\boldsymbol{\Sigma}$ is an inverse Wishart distribution. In fact, by replacing Equation (15.112) to the corresponding terms in Equation (15.108), it can be seen that the only terms left after integrating out $\mathbf{W}$ are $\boldsymbol{\Sigma}^{-1}\boldsymbol{\Psi}$ and $\boldsymbol{\Sigma}^{-1}\mathbf{S}_{y|x}$, which indicates that the scale matrix of the posterior inverse Wishart distribution is $\boldsymbol{\Psi}_0 + \mathbf{S}_{y|x}$.

In conclusion, the joint posterior distribution of $\{\mathbf{W}, \boldsymbol{\Sigma}\}$ given the observation is

$$\mathbf{W}, \boldsymbol{\Sigma}|\mathbf{X}, \mathbf{Y} \sim \text{MNIW}(\mathbf{M}_1, \mathbf{V}_1, \nu_1, \boldsymbol{\Psi}_1) \tag{15.116}$$

$$\mathbf{M}_1 = \mathbf{S}_{yx}\mathbf{S}_{xx}^{-1} \tag{15.117}$$

$$\mathbf{V}_1 = \mathbf{S}_{xx} \tag{15.118}$$

$$\nu_1 = N_{\mathcal{D}} + \nu_0 \tag{15.119}$$

$$\boldsymbol{\Psi}_1 = \boldsymbol{\Psi}_0 + \mathbf{S}_{y|x} \tag{15.120}$$

The MAP estimate of $\mathbf{W}$ and $\boldsymbol{\Sigma}$ are the mode of the posterior matrix normal inverse Wishart distribution. To derive this, notice that $\mathbf{W}$ only appears in the matrix normal density function in the posterior, so the matrix $\mathbf{W}$ maximizing the posterior density of $\{\mathbf{W}, \boldsymbol{\Sigma}\}$ is the matrix $\mathbf{W}$ that maximizes the matrix normal posterior of $\mathbf{W}$. So the MAP estimate of $\mathbf{W}$ is $\hat{\mathbf{W}} = \mathbf{M}_1 = \mathbf{S}_{yx}\mathbf{S}_{xx}^{-1}$, and this holds for any value of $\boldsymbol{\Sigma}$. By plugging $\mathbf{W} = \hat{\mathbf{W}}$ into the joint posterior of $\{\mathbf{W}, \boldsymbol{\Sigma}\}$, and taking derivatives over $\boldsymbol{\Sigma}$, it can be seen that the matrix maximizing the density is $(\nu_1 + N_y + N_x + 1)^{-1}\boldsymbol{\Psi}_1$. Since $\boldsymbol{\Psi}_1$ is positive definite, it is the MAP estimate of $\boldsymbol{\Sigma}$.

In conclusion, the MAP estimate of $\{\mathbf{W}, \boldsymbol{\Sigma}\}$ are

$$\hat{\mathbf{W}} = \mathbf{S}_{yx}\mathbf{S}_{xx}^{-1} \tag{15.121}$$

$$\hat{\boldsymbol{\Sigma}} = \frac{1}{\nu_1 + N_y + N_x + 1}(\boldsymbol{\Psi}_0 + \mathbf{S}_{y|x}) \tag{15.122}$$

15.3 Logistic regression

Logistic regression is a very widely used discriminative classification model that maps input vectors $\boldsymbol{x} \in \mathbb{R}^D$ to a distribution over class labels, $y \in \{1, \ldots, C\}$. If $C = 2$, this is known as **binary logistic regression**, and if $C > 2$, it is known as **multinomial logistic regression**, or alternatively, **multiclass logistic regression**.

15.3.1 Binary logistic regression

In the binary case, where $y \in \{0, 1\}$, the model has the following form

$$p(y|\boldsymbol{x}; \boldsymbol{\theta}) = \text{Ber}(y|\sigma(\boldsymbol{w}^{\mathsf{T}}\boldsymbol{x} + b)) \tag{15.123}$$

where $\boldsymbol{w}$ are the weights, b is the bias (offset), and σ is the **sigmoid** or **logistic** function, defined by

$$\sigma(a) \triangleq \frac{1}{1 + e^{-a}} \tag{15.124}$$

Let $\eta_n = \boldsymbol{w}^\mathsf{T}\boldsymbol{x}_n + b$ be the **logits** for example n, and $\mu_n = \sigma(\eta_n) = p(y=1|\boldsymbol{x}_n)$ be the mean of the output. Then we can write the log likelihood as the negative cross entropy:

$$\log p(\mathcal{D}|\boldsymbol{\theta}) = \log \prod_{n=1}^{N} \mu_n^{y_n}(1-\mu_n)^{1-y_n} = \sum_{n=1}^{N} y_n \log \mu_n + (1-y_n)\log(1-\mu_n) \tag{15.125}$$

We can expand this equation into a more explicit form (that is commonly seen in implementations) by performing some simple algebra. First note that

$$\mu_n = \frac{1}{1+e^{-\eta_n}} = \frac{e^{\eta_n}}{1+e^{\eta_n}},\ 1-\mu_n = 1 - \frac{e^{\eta_n}}{1+e^{\eta_n}} = \frac{1}{1+e^{\eta_n}} \tag{15.126}$$

Hence

$$\log p(\mathcal{D}|\boldsymbol{\theta}) = \sum_{n=1}^{N} y_n[\log e^{\eta_n} - \log(1+e^{\eta_n})] + (1-y_n)[\log 1 - \log(1+e^{\eta_n})] \tag{15.127}$$

$$= \sum_{n=1}^{N} y_n[\eta_n - \log(1+e^{\eta_n})] + (1-y_n)[-\log(1+e^{\eta_n})] \tag{15.128}$$

$$= \sum_{n=1}^{N} y_n\eta_n - \sum_{n=1}^{N} \log(1+e^{\eta_n}) \tag{15.129}$$

Note that the $\log(1+e^a)$ function is often implemented using `np.log1p(np.exp(a))`.

15.3.2 Multinomial logistic regression

Multinomial logistic regression is a discriminative classification model of the following form:

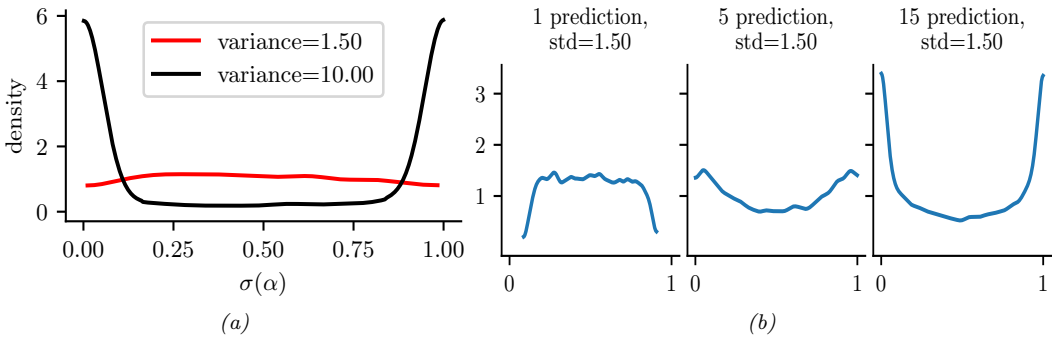
$$p(y|\boldsymbol{x};\boldsymbol{\theta}) = \text{Cat}(y|\text{softmax}(\mathbf{W}\boldsymbol{x}+\boldsymbol{b})) \tag{15.130}$$

where $\boldsymbol{x} \in \mathbb{R}^D$ is the input vector, $y \in \{1,\ldots,C\}$ is the class label, $\mathbf{W}$ is a $C \times D$ weight matrix, $\boldsymbol{b}$ is C-dimensional bias vector, and softmax() is the **softmax function**, defined as

$$\text{softmax}(\boldsymbol{a}) \triangleq \left[\frac{e^{a_1}}{\sum_{c'=1}^{C} e^{a_{c'}}}, \ldots, \frac{e^{a_C}}{\sum_{c'=1}^{C} e^{a_{c'}}}\right] \tag{15.131}$$

If we define the logits as $\boldsymbol{\eta}_n = \mathbf{W}\boldsymbol{x}_n + \boldsymbol{b}$, the probabilities as $\boldsymbol{\mu}_n = \text{softmax}(\boldsymbol{\eta}_n)$, and let $\boldsymbol{y}_n$ be the one-hot encoding of the label y_n, then the log likelihood can be written as as the negative cross entropy:

$$\log p(\mathcal{D}|\boldsymbol{\theta}) = \log \prod_{n=1}^{N}\prod_{c=1}^{C} \mu_{nc}^{y_{nc}} = \sum_{n=1}^{N}\sum_{c=1}^{C} y_{nc}\log \mu_{nc} \tag{15.132}$$



(a) *(b)*

Figure 15.5: (a) Prior on logistic regression output when using $\mathcal{N}(0, \omega)$ prior for the offset term, for $\omega = 10$ or $\omega = 1.5$. Adapted from Figure 11.3 of [McE20]. Generated by logreg_prior_offset.ipynb. (b) Distribution over the fraction of 1s we expect to see when using binary logistic regression applied to random binary feature vectors of increasing dimensionality. We use a $\mathcal{N}(0, 1.5)$ prior on the regression coefficients. Adapted from Figure 3 of [Gel+20]. Generated by logreg_prior.ipynb.

15.3.3 Priors

As with linear regression, it is standard to use Gaussian priors for the weights in a logistic regression model. It is natural to set the prior mean to 0, to reflect the fact that the output could either increase or decrease in probability depending on the input. But how do we set the prior variance? It is tempting to use a large value, to approximate a uniform distribution, but this is a bad idea. To see why, consider a binary logistic regression model with just an offset term and no features:

$$p(y|\boldsymbol{\theta}) = \text{Ber}(y|\sigma(\alpha)) \tag{15.133}$$

$$p(\alpha) = \mathcal{N}(\alpha|0, \omega) \tag{15.134}$$

If we set the prior to the large value of $\omega = 10$, the implied prior for y is an extreme distribution, with most of its density near 0 or 1, as shown in Figure 15.5a. By contrast, if we use the smaller value of $\omega = 1.5$, we get a flatter distribution, as shown.

If we have input features, the problem gets a little trickier, since the magnitude of the logits will now depend on the number and distribution of the input variables. For example, suppose we generate N random binary vectors $\boldsymbol{x}_n$, each of dimension D, where $x_{nd} \sim \text{Ber}(p)$, where $p = 0.8$. We then compute $p(y_n = 1|\boldsymbol{x}_n) = \sigma(\boldsymbol{\beta}^\mathsf{T}\boldsymbol{x}_n)$, where $\boldsymbol{\beta} \sim \mathcal{N}(\mathbf{0}, 1.5\mathbf{I})$. We sample S values of $\boldsymbol{\beta}$, and for each one, we sample a vector of labels, $\boldsymbol{y}_{1:N,s}$ from the above distribution. We then compute the fraction of positive labels, $f_s = \frac{1}{N}\sum_{n=1}^{N} \mathbb{I}(y_{n,s} = 1)$. We plot the distribution of $\{f_s\}$ as a function of D in Figure 15.5b. We see that the induced prior is initially flat, but eventually becomes skewed towards the extreme values of 0 and 1. To avoid this, we should standardize the inputs, and scale the variance of the prior by $1/\sqrt{D}$. We can also use a heavier tailed distribution, such as a Cauchy or Student [Gel+08; GLM15].

15.3.4 Posteriors

Unfortunately, there is no tractable prior that is conjugate to the logistic likelihood. Hence we cannot compute the posterior analytically, unlike with linear regression, even if we use a Gaussian prior. (This mirrors the case with MLE, where we have a closed form solution for linear regression, but not for logistic regression.) Fortunately, there are a range of approximate inference methods we can use, as we discuss in the sections below.

15.3.5 Laplace approximation

As we discuss in Section 7.4.3, the Laplace approximation approximates the posterior using a Gaussian. The mean of the Gaussian is equal to the MAP estimate $\hat{\boldsymbol{w}}$, and the covariance is equal to the inverse Hessian $\mathbf{H}$ computed at the MAP estimate, i.e., $p(\boldsymbol{w}|\mathcal{D}) \approx \mathcal{N}(\boldsymbol{w}|\hat{\boldsymbol{w}}, \mathbf{H})$, We can find the mode using a standard optimization method, and we can then compute the Hessian at the mode analytically or using automatic differentiation.

As an example, consider the binary data illustrated in Figure 15.6(a). There are many parameter settings that correspond to lines that perfectly separate the training data; we show 4 example lines. For each decision boundary in Figure 15.6(a), we plot the corresponding parameter vector as point in the log likelihood surface in Figure 15.6(b). These parameters values are $\boldsymbol{w}_1 = (3, 1)$, $\boldsymbol{w}_2 = (4, 2)$, $\boldsymbol{w}_3 = (5, 3)$, and $\boldsymbol{w}_4 = (7, 3)$. These points all approximately satisfy $\boldsymbol{w}_i(1)/\boldsymbol{w}_i(2) \approx \hat{\boldsymbol{w}}_{\text{mle}}(1)/\hat{\boldsymbol{w}}_{\text{mle}}(2)$, and hence are close to the orientation of the maximum likelihood decision boundary. The points are ordered by increasing weight norm (3.16, 4.47, 5.83, and 7.62). The unconstrained MLE has $||\boldsymbol{w}|| = \infty$, so is infinitely far to the top right.

To ensure a unique solution, we use a (spherical) Gaussian prior centered at the origin, $\mathcal{N}(\boldsymbol{w}|\mathbf{0}, \sigma^2\mathbf{I})$. The value of σ^2 controls the strength of the prior. If we set $\sigma^2 = \infty$, we force the MAP estimate to be $\boldsymbol{w} = \mathbf{0}$; this will result in maximally uncertain predictions, since all points $\boldsymbol{x}$ will produce a predictive distribution of the form $p(y = 1|\boldsymbol{x}) = 0.5$. If we set $\sigma^2 = 0$, the MAP estimate becomes the MLE, resulting in minimally uncertain predictions. (In particular, all positively labeled points will have $p(y = 1|\boldsymbol{x}) = 1.0$, and all negatively labeled points will have $p(y = 1|\boldsymbol{x}) = 0.0$, since the data is separable.) As a compromise (to make a nice illustration), we pick the value $\sigma^2 = 100$.

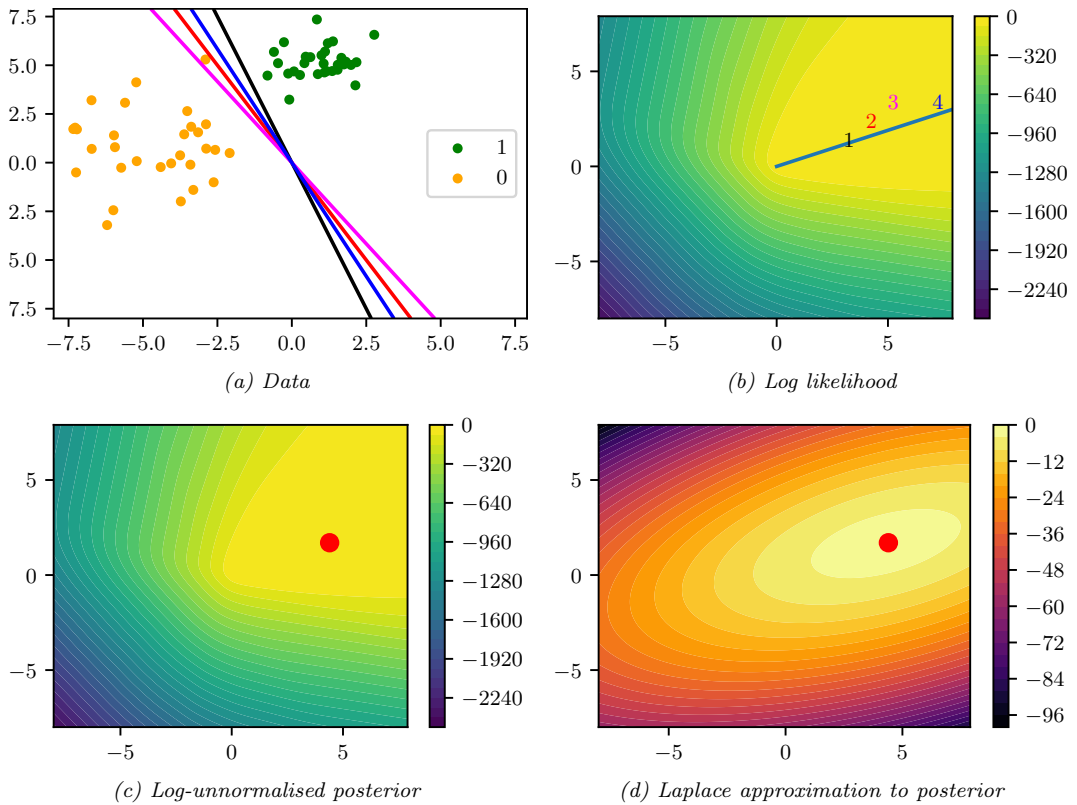
Multiplying this prior by the likelihood results in the unnormalized posterior shown in Figure 15.6(c). The MAP estimate is shown by the blue dot. The Laplace approximation to this posterior is shown in Figure 15.6(d). We see that it gets the mode correct (by construction), but the shape of the posterior is somewhat distorted. (The southwest-northeast orientation captures uncertainty about the magnitude of $\boldsymbol{w}$, and the southeast-northwest orientation captures uncertainty about the orientation of the decision boundary.)

Next we need to convert the posterior over the parameters into a posterior over predictions, as follows:

$$p(y|\boldsymbol{x}, \mathcal{D}) = \int p(y|\boldsymbol{x}, \boldsymbol{w}) p(\boldsymbol{w}|\mathcal{D}) d\boldsymbol{w} \tag{15.135}$$

The simplest way to evaluate this integral is to use a Monte Carlo approximation:

$$p(y = 1|\boldsymbol{x}, \mathcal{D}) \approx \frac{1}{S} \sum_{s=1}^{S} \sigma(\boldsymbol{w}_s^{\mathsf{T}} \boldsymbol{x}) \tag{15.136}$$



(a) Data *(b) Log likelihood*

(c) Log-unnormalised posterior *(d) Laplace approximation to posterior*

Figure 15.6: (a) Illustration of the data and some decision boundaries, where colors correspond to the points in the (b) panel. (b) Log-likelihood for a logistic regression model. The line is drawn from the origin in the direction of the MLE (which is at infinity). The numbers correspond to 4 points in parameter space, corresponding to the lines in (a). (c) Unnormalized log posterior (assuming vague spherical prior). (d) Laplace approximation to posterior. Adapted from a figure by Mark Girolami. Generated by logreg_ laplace_ demo.ipynb.

where $\boldsymbol{w}_s \sim p(\boldsymbol{w}|\mathcal{D})$.

Alternatively, we can use the deterministic **probit approxmation** first suggested in [SL90]. If we define $a = \boldsymbol{x}^\mathsf{T}\boldsymbol{w}$, and $p(\boldsymbol{w}|\mathcal{D}) = \mathcal{N}(\boldsymbol{w}|\boldsymbol{\mu}, \boldsymbol{\Sigma})$, then we can write this approximation as

$$p(y = 1|\boldsymbol{x}, \mathcal{D}) \approx \sigma(\kappa(v)m) \tag{15.137}$$

$$\kappa(v) \triangleq (1 + \pi v/8)^{-\frac{1}{2}} \tag{15.138}$$

$$v = \mathbb{V}[a] = \mathbb{V}\left[\boldsymbol{x}^\mathsf{T}\boldsymbol{w}\right] = \boldsymbol{x}^\mathsf{T}\boldsymbol{\Sigma}\boldsymbol{x} \tag{15.139}$$

$$m = \mathbb{E}[a] = \boldsymbol{x}^\mathsf{T}\boldsymbol{\mu} \tag{15.140}$$

In Figure 15.7, we show contours of the posterior predictive distribution. Figure 15.7(a) shows the plugin approximation using the MAP estimate. We see that there is no uncertainty about the decision boundary, even though we are generating probabilistic predictions over the labels. Figure 15.7(b)

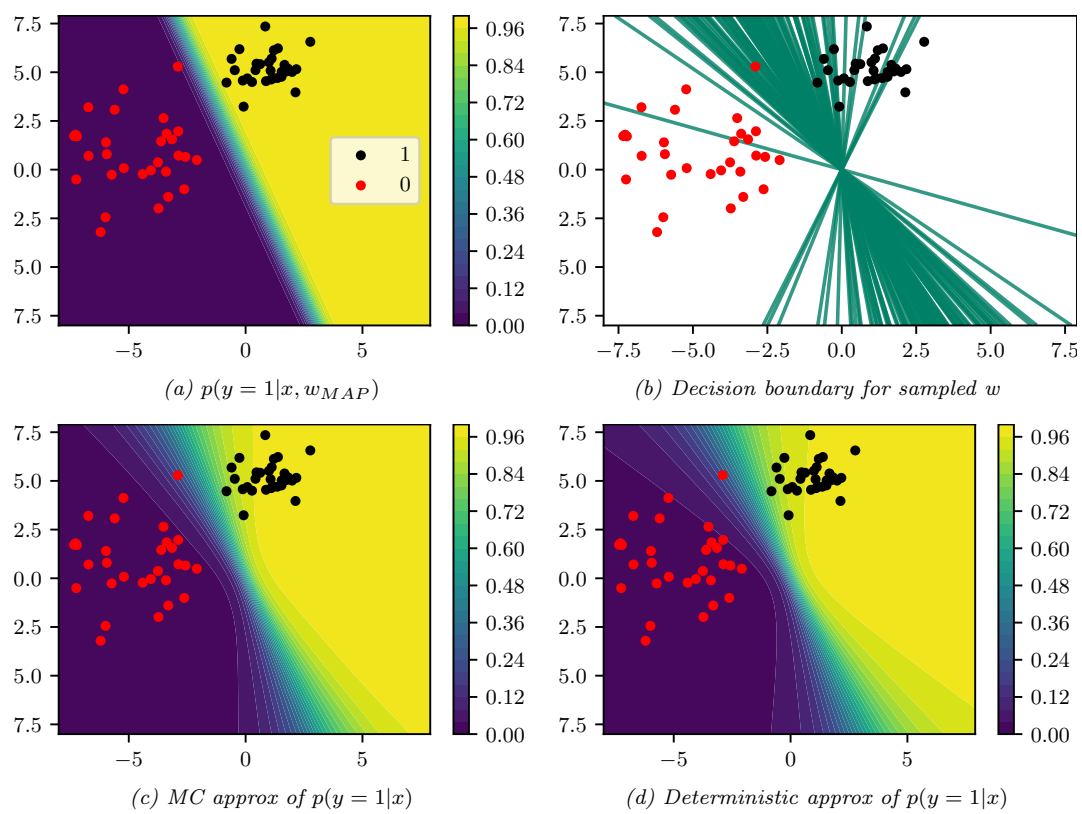


Figure 15.7: Posterior predictive distribution for a logistic regression model in 2d. (a): contours of $p(y = 1|\boldsymbol{x}, \hat{\boldsymbol{w}}_{map})$. (b): samples from the posterior predictive distribution. (c): Averaging over these samples. (d): moderated output (probit approximation). Adapted from a figure by Mark Girolami. Generated by logreg_ laplace_ demo.ipynb.

shows what happens when we plug in samples from the Gaussian posterior. Now we see that there is considerable uncertainty about the orientation of the "best" decision boundary. Figure 15.7(c) shows the average of these samples. By averaging over multiple predictions, we see that the uncertainty in the decision boundary "splays out" as we move further from the training data. Figure 15.7(d) shows that the probit approximation gives very similar results to the Monte Carlo approximation.

15.3.6 MCMC inference

Markov chain Monte Carlo, or MCMC, is often considered the "gold standard" for approximate inference, since it makes no explicit assumptions about the form of the posterior. Instead, it just approximates it non-parametrically using a set of S samples:

$$q(\boldsymbol{\theta}|\mathcal{D}) \approx \frac{1}{S} \sum_{s=1}^{S} \delta(\boldsymbol{\theta} - \boldsymbol{\theta}^s) \tag{15.141}$$

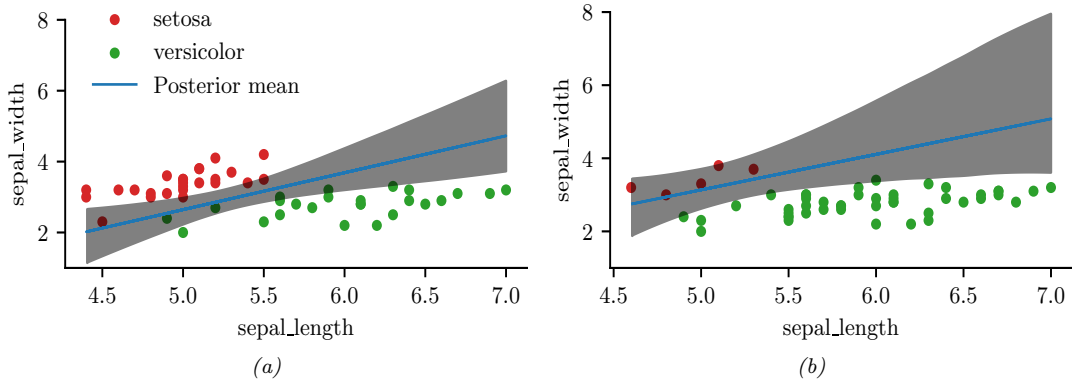


Figure 15.8: Illustration of the posterior over the decision boundary for classifying iris flowers (setosa vs versicolor) using 2 input features. (a) 25 examples per class. Adapted from Figure 4.5 of [Mar18]. (b) 5 examples of class 0, 45 examples of class 1. Adapted from Figure 4.8 of [Mar18]. Generated by logreg_iris_bayes_2d.ipynb.

where $\boldsymbol{\theta}^s \sim p(\boldsymbol{\theta}|\mathcal{D})$ are samples from the posterior.

To efficiently compute these samples, we can use the method of Hamiltonian Monte Carlo or HMC, which we describe in Section 12.5. This relies on our ability to compute the gradient of the log joint, $\nabla_{\boldsymbol{\theta}} \log p(\mathcal{D}, \boldsymbol{\theta})$, which we can compute using automatic differentiation.

Let us apply HMC to a 2-dimensional, 2-class version of the iris classification problem, where we just use two input features, sepal length and sepal width, and two classes, Virginica and Non-Virginica. The decision boundary is the set of points (x_1^*, x_2^*) such that $\sigma(b + w_1 x_1^* + w_2 x_2^*) = 0.5$. Such points must lie on the following line:

$$x_2^* = -\frac{b}{w_2} + \left(-\frac{w_1}{w_2} x_1^*\right) \tag{15.142}$$

We can therefore compute an MC approximation to the posterior over decision boundaries by sampling the parameters from the posterior, $(w_1, w_2, b) \sim p(\boldsymbol{\theta}|\mathcal{D})$, and plugging them into the above equation, to get $p(x_1^*, x_2 * |\mathcal{D})$. The results of this method (using a vague Gaussian prior for the parameters) are shown in Figure 15.8a. The solid line is the posterior mean, and the shaded interval is a 95% credible interval. As before, we see that the uncertainty about the location of the boundary is higher as we move away from the training data.

In Figure 15.8b, we show what happens to the decision boundary when we have unbalanced classes. We notice two things. First, the posterior uncertainty increases, because we have less data from the blue class. Second, we see that the posterior mean of the decision boundary shifts towards the class with less data. This follows from linear discriminant analysis, where one can show that changing the class prior changes the location of the decision boundary, so that more of the input space gets mapped to the class which is higher a priori. (See [Mur22, Sec 9.2] for details.)

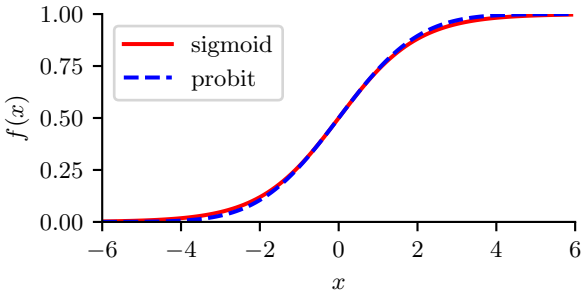


Figure 15.9: The logistic (sigmoid) function $\sigma(x)$ in solid red, with the Gaussian cdf function $\Phi(\lambda x)$ in dotted blue superimposed. Here $\lambda = \sqrt{\pi/8}$, which was chosen so that the derivatives of the two curves match at $x = 0$. Adapted from Figure 4.9 of [Bis06]. Generated by probit_plot.ipynb.

15.3.7 Variational inference

As we discuss in Section 10.1, variational inference converts approximate inference into an optimization problem. It does this by choosing an approximate distribution $q(\boldsymbol{w}; \boldsymbol{\psi})$ and optimiing the variational parameters $\boldsymbol{\psi}$ to maximize the evidence lower bound (ELBO). This has the effect of making $q(\boldsymbol{w}; \boldsymbol{\psi}) \approx p(\boldsymbol{w}|\mathcal{D})$ in the sense that the KL divergence is small. There are several ways to tackle this: use a stochastic estimate of the ELBO (see Section 10.3.3), use the conditionally conjugate VI method of Supplementary Section 10.3.1.2, or use a "local" VI method that creates a quadratic lower bound to the logistic function (see Supplementary Section 15.1).

15.3.8 Assumed density filtering

In Section 8.10.3, we discuss how to use assumed density filtering (ADF) to recursively approximate the posterior $p(\boldsymbol{w}|\mathcal{D}_{1:t})$ for a logistic regression model in an online fashion.

15.4 Probit regression

In this section, we discuss **probit regression**, which is similiar to binary logistic regression except it uses $\mu_n = \Phi(a_n)$ instead of $\mu_n = \sigma(a_n)$ as the mean function, where Φ is the cdf of the standard normal, and $a_n = \boldsymbol{w}^\mathsf{T}\boldsymbol{x}_n$. The corresponding link function is therefore $a_n = \ell(\mu_n) = \Phi^{-1}(\mu_n)$; the inverse of the Gaussian cdf is known as the **probit function**.

The Gaussian cdf Φ is very similar to the logistic function, as shown in Figure 15.9. Thus probit regression and "regular" logistic regression behave very similarly. However, probit regression has some advantages. In particular, it has a simple interpretation as a latent variable model (see Section 15.4.1), which arises from the field of **choice theory** as studied in economics (see e.g., [Koo03]). This also simplifies the task of Bayesian parameter inference.

15.4.1 Latent variable interpretation

We can interpret $a_n = \boldsymbol{w}^\mathsf{T}\boldsymbol{x}_n$ as a factor that is proportional to how likely a person is respond positively (generate $y_n = 1$) given input $\boldsymbol{x}_n$. However, typically there are other unobserved factors that

influence someone's response. Let us model these hidden factors by Gaussian noise, $\epsilon_n \sim \mathcal{N}(0,1)$. Let the combined preference for positive outcomes be represented by the latent variable $z_n = \boldsymbol{w}^\mathsf{T}\boldsymbol{x}_n + \epsilon_n$. We assume that the person will pick the positive label iff this latent factor is positive rather than negative, i.e.,

$$y_n = \mathbb{I}(z_n \geq 0) \tag{15.143}$$

When we marginalize out z_n, we recover the probit model:

$$p(y_n = 1|\boldsymbol{x}_n, \boldsymbol{w}) = \int \mathbb{I}(z_n \geq 0)\mathcal{N}(z_n|\boldsymbol{w}^\mathsf{T}\boldsymbol{x}_n, 1)dz_n \tag{15.144}$$

$$= p(\boldsymbol{w}^\mathsf{T}\boldsymbol{x}_n + \epsilon_n \geq 0) = p(\epsilon_n \geq -\boldsymbol{w}^\mathsf{T}\boldsymbol{x}_n) \tag{15.145}$$

$$= 1 - \Phi(-\boldsymbol{w}^\mathsf{T}\boldsymbol{x}_n) = \Phi(\boldsymbol{w}^\mathsf{T}\boldsymbol{x}_n) \tag{15.146}$$

Thus we can think of probit regression as a threshold function applied to noisy input.

We can interpret logistic regression in the same way. However, in that case the noise term ϵ_n comes from a **logistic distribution**, defined as follows:

$$f(y|\mu, s) \triangleq \frac{e^{-\frac{y-\mu}{s}}}{s(1+e^{-\frac{y-\mu}{s}})^2} \tag{15.147}$$

The cdf of this distribution is given by

$$F(y|\mu, s) = \frac{1}{1+e^{-\frac{y-\mu}{s}}} \tag{15.148}$$

It is clear that if we use logistic noise with $\mu = 0$ and $s = 1$ we recover logistic regression. However, it is computationally easier to deal with Gaussian noise, as we show below.

15.4.2 Maximum likelihood estimation

In this section, we discuss some methods for fitting probit regression using MLE.

15.4.2.1 MLE using SGD

We can find the MLE for probit regression using standard gradient methods. Let $\mu_n = \boldsymbol{w}^\mathsf{T}\boldsymbol{x}_n$, and let $\tilde{y}_n \in \{-1, +1\}$. Then the gradient of the log-likelihood for a single example n is given by

$$\boldsymbol{g}_n \triangleq \frac{d}{d\boldsymbol{w}} \log p(\tilde{y}_n|\boldsymbol{w}^\mathsf{T}\boldsymbol{x}_n) = \frac{d\mu_n}{d\boldsymbol{w}}\frac{d}{d\mu_n}\log p(\tilde{y}_n|\boldsymbol{w}^\mathsf{T}\boldsymbol{x}_n) = \boldsymbol{x}_n \frac{\tilde{y}_n\phi(\mu_n)}{\Phi(\tilde{y}_n\mu_n)} \tag{15.149}$$

where ϕ is the standard normal pdf, and Φ is its cdf. Similarly, the Hessian for a single case is given by

$$\mathbf{H}_n = \frac{d}{d\boldsymbol{w}^2}\log p(\tilde{y}_n|\boldsymbol{w}^\mathsf{T}\boldsymbol{x}_n) = -\boldsymbol{x}_n\left(\frac{\phi(\mu_n)^2}{\Phi(\tilde{y}_n\mu_n)^2} + \frac{\tilde{y}_n\mu_n\phi(\mu_n)}{\Phi(\tilde{y}_n\mu_n)}\right)\boldsymbol{x}_n^\mathsf{T} \tag{15.150}$$

This can be passed to any gradient-based optimizer.

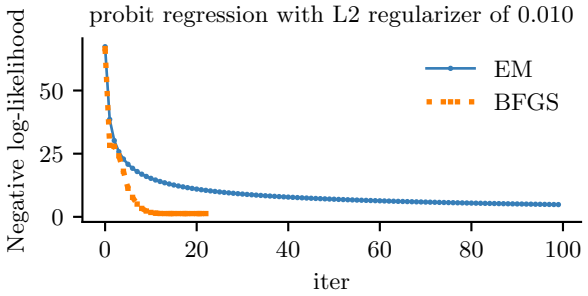


Figure 15.10: Fitting a probit regression model in 2d using a quasi-Newton method or EM. Generated by probit_reg_demo.ipynb.

15.4.2.2 MLE using EM

We can use the latent variable interpretation of probit regression to derive an elegant EM algorithm for fitting the model. The complete data log likelihood has the following form, assuming a $\mathcal{N}(\mathbf{0}, \mathbf{V}_0)$ prior on $\boldsymbol{w}$:

$$\ell(\boldsymbol{z}, \boldsymbol{w}|\mathbf{V}_0) = \log p(\boldsymbol{y}|\boldsymbol{z}) + \log \mathcal{N}(\boldsymbol{z}|\mathbf{X}\boldsymbol{w}, \mathbf{I}) + \log \mathcal{N}(\boldsymbol{w}|\mathbf{0}, \mathbf{V}_0) \tag{15.151}$$

$$= \sum_n \log p(y_n|z_n) - \frac{1}{2}(\boldsymbol{z} - \mathbf{X}\boldsymbol{w})^\mathsf{T}(\boldsymbol{z} - \mathbf{X}\boldsymbol{w}) - \frac{1}{2}\boldsymbol{w}^\mathsf{T}\mathbf{V}_0^{-1}\boldsymbol{w} \tag{15.152}$$

The posterior in the E step is a **truncated Gaussian**:

$$p(z_n|y_n, \boldsymbol{x}_n, \boldsymbol{w}) = \begin{cases} \mathcal{N}(z_n|\boldsymbol{w}^\mathsf{T}\boldsymbol{x}_n, 1)\mathbb{I}(z_n > 0) & \text{if } y_n = 1 \\ \mathcal{N}(z_n|\boldsymbol{w}^\mathsf{T}\boldsymbol{x}_n, 1)\mathbb{I}(z_n < 0) & \text{if } y_n = 0 \end{cases} \tag{15.153}$$

In Equation (15.152), we see that $\boldsymbol{w}$ only depends linearly on $\boldsymbol{z}$, so we just need to compute $\mathbb{E}[z_n|y_n, \boldsymbol{x}_n, \boldsymbol{w}]$, so we just need to compute the posterior mean. One can show that this is given by

$$\mathbb{E}[z_n|\boldsymbol{w}, \boldsymbol{x}_n] = \begin{cases} \mu_n + \frac{\phi(\mu_n)}{1-\Phi(-\mu_n)} = \mu_n + \frac{\phi(\mu_n)}{\Phi(\mu_n)} & \text{if } y_n = 1 \\ \mu_n - \frac{\phi(\mu_n)}{\Phi(-\mu_n)} = \mu_n - \frac{\phi(\mu_n)}{1-\Phi(\mu_i)} & \text{if } y_n = 0 \end{cases} \tag{15.154}$$

where $\mu_n = \boldsymbol{w}^\mathsf{T}\boldsymbol{x}_n$.

In the M step, we estimate $\boldsymbol{w}$ using ridge regression, where $\boldsymbol{\mu} = \mathbb{E}[\boldsymbol{z}]$ is the output we are trying to predict. Specifically, we have

$$\hat{\boldsymbol{w}} = (\mathbf{V}_0^{-1} + \mathbf{X}^\mathsf{T}\mathbf{X})^{-1}\mathbf{X}^\mathsf{T}\boldsymbol{\mu} \tag{15.155}$$

The EM algorithm is simple, but can be much slower than direct gradient methods, as illustrated in Figure 15.10. This is because the posterior entropy in the E step is quite high, since we only observe that z is positive or negative, but are given no information from the likelihood about its magnitude. Using a stronger regularizer can help speed convergence, because it constrains the range of plausible z values. In addition, one can use various speedup tricks, such as data augmentation [DM01].

15.4.3 Bayesian inference

It is possible to use the latent variable formulation of probit regression in Section 15.4.2.2 to derive a simple Gibbs sampling algorithm for approximating the posterior $p(\boldsymbol{w}|\mathcal{D})$ (see e.g., [AC93; HH06]).

The key idea is to use an auxiliary latent variable, which, when conditioned on, makes the whole model a conjugate linear-Gaussian model. The full conditional for the latent variables is given by

$$p(z_i|y_i, \boldsymbol{x}_i, \boldsymbol{w}) = \begin{cases} \mathcal{N}(z_i|\boldsymbol{w}^T\boldsymbol{x}_i, 1)\mathbb{I}(z_i > 0) & \text{if } y_i = 1 \\ \mathcal{N}(z_i|\boldsymbol{w}^T\boldsymbol{x}_i, 1)\mathbb{I}(z_i < 0) & \text{if } y_i = 0 \end{cases} \tag{15.156}$$

Thus the posterior is a truncated Gaussian. We can sample from a truncated Gaussian, $\mathcal{N}(z|\mu, \sigma)\mathbb{I}(a \le z \le b)$ in two steps: first sample $u \sim U(\Phi((a-\mu)/\sigma), \Phi((b-\mu)/\sigma))$, then set $z = \mu + \sigma\Phi^{-1}(u)$ [Rob95a].

The full conditional for the parameters is given by

$$p(\boldsymbol{w}|\mathcal{D}, \boldsymbol{z}, \boldsymbol{\lambda}) = \mathcal{N}(\boldsymbol{w}_N, \mathbf{V}_N) \tag{15.157}$$

$$\mathbf{V}_N = (\mathbf{V}_0^{-1} + \mathbf{X}^T\mathbf{X})^{-1} \tag{15.158}$$

$$\boldsymbol{w}_N = \mathbf{V}_N(\mathbf{V}_0^{-1}\boldsymbol{m}_0 + \mathbf{X}^T\boldsymbol{z}) \tag{15.159}$$

For further details, see e.g., [AC93; FSF10]. It is also possible to use variational Bayes, which tends to be much faster (see e.g., [GR06a; FDZ19]).

15.4.4 Ordinal probit regression

One advantage of the latent variable interpretation of probit regression is that it is easy to extend to the case where the response variable is ordered in some way, such as the outputs low, medium and high. This is called **ordinal regression**. The basic idea is as follows. If there are C output values, we introduce $C+1$ thresholds γ_j and set

$$y_n = j \quad \text{if} \quad \gamma_{j-1} < z_n \le \gamma_j \tag{15.160}$$

where $\gamma_0 \le \cdots \le \gamma_C$. For identifiability reasons, we set $\gamma_0 = -\infty$, $\gamma_1 = 0$ and $\gamma_C = \infty$. For example, if $C = 2$, this reduces to the standard binary probit model, whereby $z_n < 0$ produces $y_n = 0$ and $z_n \ge 0$ produces $y_n = 1$. If $C = 3$, we partition the real line into 3 intervals: $(-\infty, 0]$, $(0, \gamma_2]$, (γ_2, ∞). We can vary the parameter γ_2 to ensure the right relative amount of probability mass falls in each interval, so as to match the empirical frequencies of each class label. See e.g., [AC93] for further details.

Finding the MLEs for this model is a bit trickier than for binary probit regression, since we need to optimize for $\boldsymbol{w}$ and $\boldsymbol{\gamma}$, and the latter must obey an ordering constraint. See e.g., [KL09] for an approach based on EM. It is also possible to derive a simple Gibbs sampling algorithm for this model (see e.g., [Hof09, p216]).

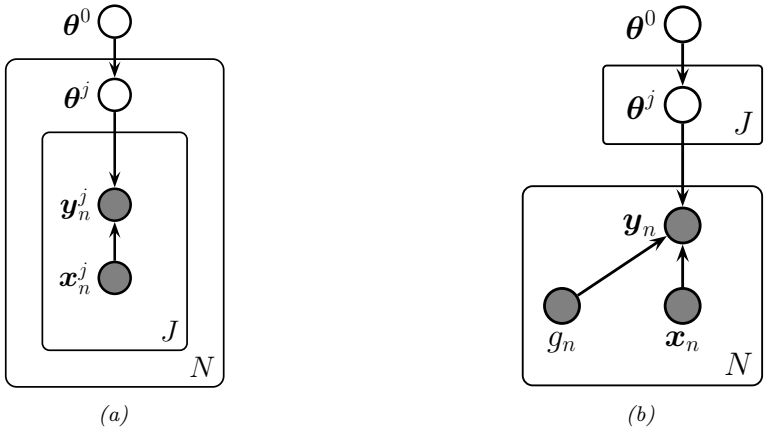


Figure 15.11: Hierarchical Bayesian discriminative models with J groups. (a) Nested formulation. (b) Non-nested formulation, with group indicator $g_n \in \{1, \ldots, J\}$.

15.4.5 Multinomial probit models

Now consider the case where the response variable can take on C unordered categorical values, $y_n \in \{1, \ldots, C\}$. The **multinomial probit** model is defined as follows:

$$z_{nc} = \boldsymbol{w}_c^\mathsf{T} \boldsymbol{x}_{nc} + \epsilon_{nc} \tag{15.161}$$

$$\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{R}) \tag{15.162}$$

$$y_n = \arg\max_c z_{nc} \tag{15.163}$$

See e.g., [DE04; GR06b; Sco09; FSF10] for more details on the model and its connection to multinomial logistic regression.

If instead of setting $y_n = \text{argmax}_c \, z_{ic}$ we use $y_{nc} = \mathbb{I}(z_{nc} > 0)$, we get a model known as **multivariate probit**, which is one way to model C correlated binary outcomes (see e.g., [TMD12]).

15.5 Multi-level (hierarchical) GLMs

Suppose we have a set of J related datasets, each of which contains a series of N_j datapoints $\mathcal{D}_j = \{(\boldsymbol{x}_n^j, \boldsymbol{y}_n^j) : n = 1 : N_j\}$. There are 3 main ways to fit models in such a setting: we could fit J separate models, $p(\boldsymbol{y}|\boldsymbol{x}; \mathcal{D}_j)$, which might result in overfitting if some $\mathcal{D}_j$ are small; we could pool all the data to get $\mathcal{D} = \cup_{j=1}^J \mathcal{D}_j$ and fit a single model, $p(\boldsymbol{y}|\boldsymbol{x}; \mathcal{D})$, which might result in underfitting; or we can use a **hierarchical Bayesian model**, also called a **mutilevel model** or **partially pooled model**, in which we assume each group has its own parameters, $\boldsymbol{\theta}^j$, but that these have something in common, as modeled by a shared global prior $p(\boldsymbol{\theta}^0)$. (Note that each group could be a single individual.) The overall model has the form

$$p(\boldsymbol{\theta}^{0:J}, \mathcal{D}) = p(\boldsymbol{\theta}^0) \prod_{j=1}^J \left[p(\boldsymbol{\theta}^j|\boldsymbol{\theta}^0) \prod_{n=1}^{N_j} p(\boldsymbol{y}_n^j|\boldsymbol{x}_n^j, \boldsymbol{\theta}^j) \right] \tag{15.164}$$

See Figure 15.11a, which represents the model using nested plate notation.

It is often more convenient to represent the model as in Figure 15.11b, which eliminates the nested plates (and hence the double indexing of variables) by associating a group indicator variable $g_n \in \{1, \ldots, J\}$, which specifies which set of parameters to use for each data point. Thus the model now has the form

$$p(\boldsymbol{\theta}^{0:J}, \mathcal{D}) = p(\boldsymbol{\theta}^0) \left[\prod_{j=1}^{J} p(\boldsymbol{\theta}^j|\boldsymbol{\theta}^0)\right] \left[\prod_{n=1}^{N} p(\boldsymbol{y}_n|\boldsymbol{x}_n, g_n, \boldsymbol{\theta})\right] \tag{15.165}$$

where

$$p(\boldsymbol{y}_n|\boldsymbol{x}_n, g_n, \boldsymbol{\theta}) = \prod_{j=1}^{J} p(\boldsymbol{y}_n|\boldsymbol{x}_n, \boldsymbol{\theta}^j)^{\mathbb{I}(g_n=j)} \tag{15.166}$$

If the likelihood function is a GLM, this hierarchical model is called a **hierarchical GLM** [LN96]. This class of models is very widely used in applied statistics. For much more details, see e.g., [GH07; GHV20; Gel+22].

15.5.1 Generalized linear mixed models (GLMMs)

Suppose that the prior on the per-group parameters is Gaussian, so $p(\boldsymbol{\theta}^j|\boldsymbol{\theta}^0) = \mathcal{N}(\boldsymbol{\theta}^j|\boldsymbol{\theta}^0, \boldsymbol{\Sigma}^j)$. If we have a GLM likelihood, the model becomes

$$p(\boldsymbol{y}_n|\boldsymbol{x}_n, g_n = j, \boldsymbol{\theta}) = p(\boldsymbol{y}_n|\ell(\eta_n)) \tag{15.167}$$

$$\eta_n = \boldsymbol{x}_n^\mathsf{T}\boldsymbol{\theta}^j = \boldsymbol{x}_n^\mathsf{T}(\boldsymbol{\theta}^0 + \boldsymbol{\epsilon}^j) = \boldsymbol{x}_n^\mathsf{T}\boldsymbol{\theta}^0 + \boldsymbol{x}_n^\mathsf{T}\boldsymbol{\epsilon}^j \tag{15.168}$$

where ℓ is the link function, and $\boldsymbol{\epsilon}^j \sim \mathcal{N}(\mathbf{0}, \boldsymbol{\Sigma})$. This is known as a **generalized linear mixed model** (**GLMM**) or **mixed effects model**. The shared (common) parameters $\boldsymbol{\theta}^0$ are called **fixed effects**, and the group-specific offsets $\boldsymbol{\epsilon}^j$ are called **random effects**.[2] We can see that the random effects model group-specific deviations or idiosyncracies away from the shared fixed parameters. Furthermore, we see that the random effects are correlated, which allows us to model dependencies between the observations that would not be captured by a standard GLM.

15.5.2 Model fitting

We can fit GLMMs, and hierarchical models more generally, using standard Bayesian inference methods. We can use a variety of algorithms, such as HMC (see e.g., [BG13]), variational Bayes (see e.g., [HOW11; TN13]), expectation propagation (see e.g., [KW18]), etc. In this section, we use HMC, since it is simple and efficient.[3]

2. Note that there are multiple definitions of the terms "fixed effects" and random effects", as explained in this blog post by Andrew Gelman: https://statmodeling.stat.columbia.edu/2005/01/25/why_i_dont_use/.

3. There are many standard software packages for HMC analysis of hierarchical GLMs, such as **Bambi** (https://github.com/bambinos/bambi), which is a Python wrapper on top of PyMC, Blackjax and numpyro samplers; **RStanARM** (https://cran.r-project.org/web/packages/rstanarm/index.html), which is an R wrapper on top of Stanl and **BRMS** (https://cran.r-project.org/web/packages/brms/index.html), which is another R wrapper on top of Stan, but which also needs a C++ compiler.

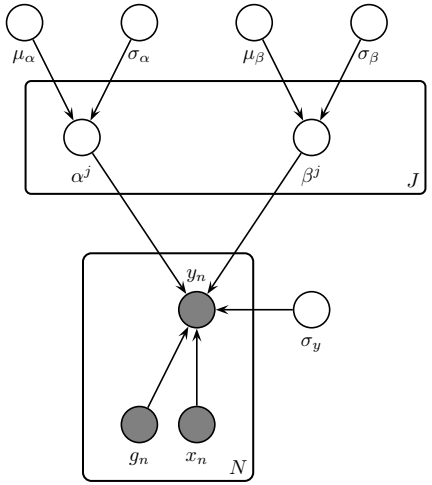


Figure 15.12: A hierarchical Bayesian linear regression model for the radon problem.

15.5.3 Example: radon regression

In this section, we give an example of a hierarchical Bayesian linear regression model. We apply it to a simplified version of the **radon** example from [Gel+14a, Sec 9.4].

Radon is known to be the highest cause of lung cancer in non-smokers, so reducing it where possible is desirable. To help with this, we fit a regression model, that predicts the (log) radon level as a function of the location of the house, as represented by a categorical feature indicating its county, and a binary feature representing whether the house has a basement or not. We use a dataset consisting of $J = 85$ counties in Minnesota; each county has between 2 and 80 measurements.

We assume the following likelihood:

$$p(y_n|x_n, g_n = j, \boldsymbol{\theta}) = \mathcal{N}(y_n|\alpha_j + \beta_j x_n, \sigma_y^2) \tag{15.169}$$

where $g_n \in \{1, \ldots, J\}$ is the county for house i, and $x_n \in \{0, 1\}$ indicates if the floor is at level 0 (i.e., in the basement) or level 1 (i.e., above ground). Intuitively we expect the radon levels to be lower in houses without basements, since they are more insulated from the earth which is the source of the radon.

Since some counties have very few data points, we use a hierarchical prior in which we assume $\alpha_j \sim \mathcal{N}(\mu_\alpha, \sigma_\alpha^2)$ and $\beta_j \sim \mathcal{N}(\mu_\beta, \sigma_\beta^2)$. We use weak priors for the parameters: $\mu_\alpha \sim \mathcal{N}(0, 1)$, $\mu_\beta \sim \mathcal{N}(0, 1)$, $\sigma_\alpha \sim \mathcal{C}_+(1)$, $\sigma_\beta \sim \mathcal{C}_+(1)$, $\sigma_y \sim \mathcal{C}_+(1)$. See Figure 15.12 for the graphical model.

15.5.3.1 Posterior inference

Figure 15.13 shows the posterior marginals for μ_α, μ_β, α_j and β_j. We see that μ_β is close to -0.6 with high probability, which confirms our suspicion that having $x = 1$ (i.e., no basement) decreases the amount of radon in the house. We also see that the distribution of the α_j parameters is quite variable, due to different base rates across the counties.

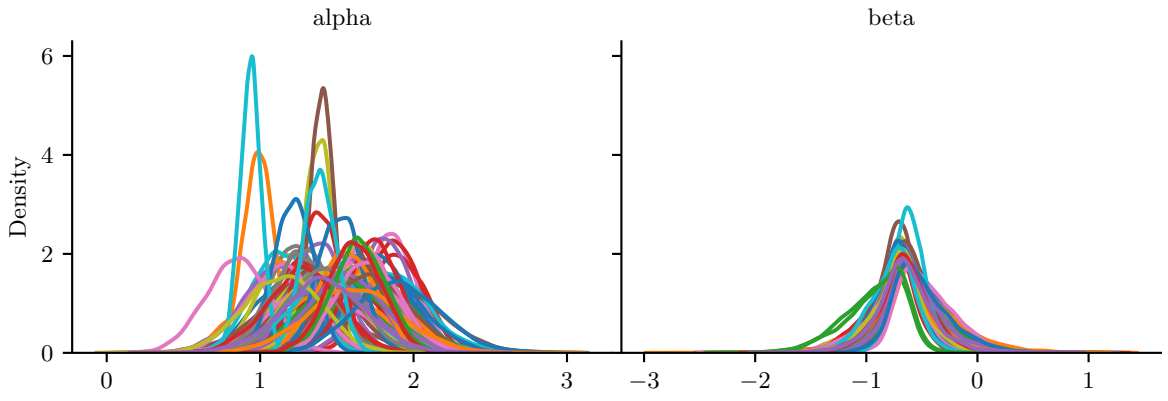


Figure 15.13: Posterior marginals for α_c and β_c for each county in the radon model. Generated by linreg_hierarchical_non_centered.ipynb.

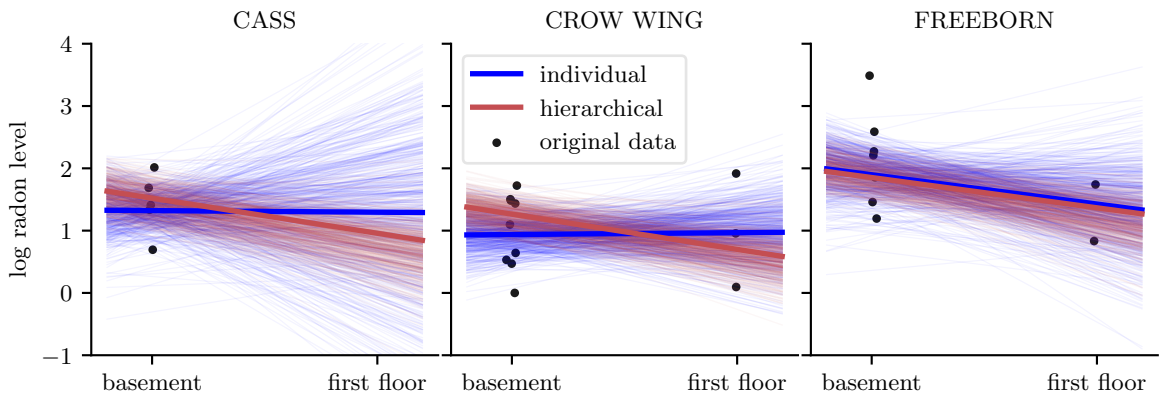
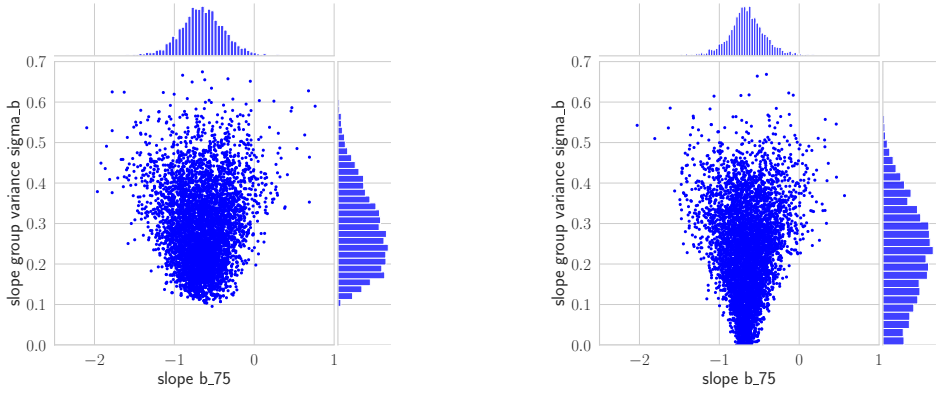


Figure 15.14: Predictions from the radon model for 3 different counties in Minnesota. Black dots are observed datapoints. Red represents results of hierarchical (shared) prior, blue represents results of non-hierarchical prior. Thick lines are the result of using the posterior mean, thin lines are the result of using posterior samples. Generated by linreg_hierarchical_non_centered.ipynb.

Figure 15.14 shows predictions from the hierarchical and non-hierarchical model for 3 different counties. We see that the predictions from the hierarchical model are more consistent across counties, and work well even if there are no examples of certain feature combinations for a given county (e.g., there are no houses without basements in the sample from Cass county). If we sample data from the posterior predictive distribution, and compare it to the real data, we find that the RMSE is 0.13 for the non-hierarchical model and 0.08 for the hierarchical model, indicating that the latter fits better.

15.5.3.2 Non-centered parameterization

One problem that frequently arises in hierarchical models is that the parameters be very correlated. This can cause computational problems when performing inference.



(a) *(b)*

Figure 15.15: (a) Bivariate posterior $p(\beta_j, \sigma_\beta|\mathcal{D})$ for the hierarchical radon model for county $j = 75$ using centered parameterization. (b) Similar to (a) except we plot $p(\tilde{\beta}_j, \sigma_\beta|\mathcal{D})$ for the non-centered parameterization. Generated by linreg_hierarchical_non_centered.ipynb.

Figure 15.15a gives an example where we plot $p(\beta_j, \sigma_\beta|\mathcal{D})$ for some specific county j. If we believe that σ_β is large, then β_c is "allowed" to vary a lot, and we get the broad distribution at the top of the figure. However, if we believe that σ_β is small, then β_j is constrained to be close to the global prior mean of μ_β, so we get the narrow distribution at the bottom of the figure. This is often called **Neal's funnel**, after a paper by Radford Neal [Nea03]. It is difficult for many algorithms (especially sampling algorithms) to explore parts of parameter space at the bottom of the funnel. This is evident from the marginal posterior for σ_β shown (as a histogram) on the right hand side of the plot: we see that it excludes the interval $[0, 0.1]$, thus ruling out models in which we shrink β_j all the way to 0. In cases where a covariate has no useful predictive role, we would like to be able to induce sparsity, so we need to overcome this problem.

A simple solution to this is to use a **non-centered parameterization** [PR03]. That is, we replace $\beta_j \sim \mathcal{N}(\mu_\beta, \sigma_\beta^2)$ with $\beta_j = \mu_\beta + \tilde{\beta}_j \sigma_\beta$, where $\tilde{\beta}_j \sim \mathcal{N}(0, 1)$ represents the *offset* from the global mean, μ_β. The correlation between $\tilde{\beta}_j$ and σ_β is much less, as shown in Figure 15.15b. See Section 12.6.5 for more details.

16 Deep neural networks

16.1 Introduction

The term "**deep neural network**" or **DNN**, in its modern usage, refers to any kind of differentiable function that can be expressed as a **computation graph**, where the nodes are primitive operations (like matrix mulitplication), and edges represent numeric data in the form of vectors, matrices, or tensors. In its simplest form, this graph can be constructed as a linear series of nodes or "**layers**". The term "deep" refers to models with many such layers.

In Section 16.2 we discuss some of the basic building blocks (node types) that are used in the field. In Section 16.3 we give examples of common architectures which are constructed from these building blocks. In Section 6.2 we show how we can efficiently compute the gradient of functions defined on such graphs. If the function computes the scalar loss of the model's predictions given a training set, we can pass this gradient to an optimization routine, such as those discussed in Chapter 6, in order to fit the model. Fitting such models to data is called "**deep learning**".

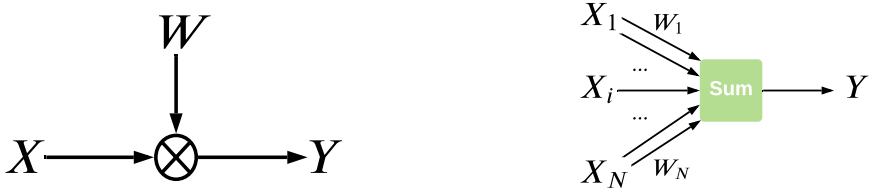
We can combine DNNs with probabilistic models in two different ways. The first is to use them to define nonlinear functions which are used inside conditional distributions. For example, we may construct a classifier using $p(y|\boldsymbol{x}, \boldsymbol{\theta}) = \text{Cat}(y|\text{softmax}(f(\boldsymbol{x}; \boldsymbol{\theta})))$, where $f(\boldsymbol{x}; \boldsymbol{\theta})$ is a neural network that maps inputs $\boldsymbol{x}$ and parameters $\boldsymbol{\theta}$ to output logits. Or we may construct a joint probability distribution over multiple variables using a directed graphical model (Chapter 4) where each CPD $p(\boldsymbol{x}_i|\text{pa}(\boldsymbol{x}_i))$ is a DNN. This lets us construct expressive probability models.

The other way we can combine DNNs and probabilistic models is to use DNNs to approximate the posterior distribution, i.e., we learn a function f to compute $q(\boldsymbol{z}|f(\mathcal{D}; \boldsymbol{\phi}))$, where $\boldsymbol{z}$ are the hidden variables (latents and/or parameters), $\mathcal{D}$ are the observed variables (data), f is an **inference network**, and $\boldsymbol{\phi}$ are its parameters; for details, see Section 10.3.6. Note that in this latter, setting the joint model $p(\boldsymbol{z}, \mathcal{D})$ may be a "traditional" model without any "neural" components. For example, it could be a complex simulator. Thus the DNN is just used for computational purposes, not statistical / modeling purposes.

More details on DNNs can be found in such books as [Zha+20a; Cho21; Gér19; GBC16], as well as a multitude of online courses. For a more theoretical treatment, see e.g., [Ber+21; Cal20; Aro+21; RY21].

16.2 Building blocks of differentiable circuits

In this section we discuss some common building blocks used in constructing neural networks. We denote the input to a block as $\boldsymbol{x}$ and the output as $\boldsymbol{y}$.



(a) (b)

Figure 16.1: An artiticial "neuron", the most basic building block of a DNN. (a) The output y is a weighted combination of the inputs $\boldsymbol{x}$, where the weights vector is denoted by $\boldsymbol{w}$. (b) Alternative depiction of the neuron's behavior. The bias term b can be emulated by defining $w_N = b$ and $X_N = 1$.

16.2.1 Linear layers

The most basic building block of a DNN is a single "**neuron**", which corresponds to a real-valued signal y computed by multiplying a vector-valued input signal $\boldsymbol{x}$ by a weight vector $\boldsymbol{w}$, and then adding a bias term b. That is,

$$y = f(\boldsymbol{x}; \boldsymbol{\theta}) = \boldsymbol{w}^\mathsf{T} \boldsymbol{x} + b \tag{16.1}$$

where $\boldsymbol{\theta} = (\boldsymbol{w}, b)$ are the parameters for the function f. This is depicted in Figure 16.1. (The bias term is omitted for clarity.)

It is common to group a set of neurons together into a **layer**. We can then represent the activations of a layer with D units as a vector $\boldsymbol{z} \in \mathbb{R}^D$. We can transform an input vector of activations $\boldsymbol{x}$ into an output vector $\boldsymbol{y}$ by multiplying by a weight matrix $\mathbf{W}$, an adding an offset vector or bias term $\boldsymbol{b}$ to get

$$\boldsymbol{y} = f(\boldsymbol{x}; \boldsymbol{\theta}) = \mathbf{W}\boldsymbol{x} + \boldsymbol{b} \tag{16.2}$$

where $\boldsymbol{\theta} = (\mathbf{W}, \boldsymbol{b})$ are the parameters for the function f. This is called a **linear layer**, or **fully connected layer**.

It is common to prepend the bias vector onto the first column of the weight matrix, and to append a 1 to the vector $\boldsymbol{x}$, so that we can write this more compactly as $\boldsymbol{x} = \tilde{\mathbf{W}}^\mathsf{T} \tilde{\boldsymbol{x}}$, where $\tilde{\mathbf{W}} = [\mathbf{W}, \boldsymbol{b}]$ and $\tilde{\boldsymbol{x}} = [\boldsymbol{x}, 1]$. This allows us to ignore the bias term from our notation if we want to.

16.2.2 Non-linearities

A stack of linear layers is equivalent to a single linear layer where we multliply together all the weight matrices. To get more expressive power we can transform each layer by passing it elementwise (pointwise) through a nonlinear function called an **activation function**. This is denoted by

$$\boldsymbol{y} = \varphi(\boldsymbol{x}) = [\varphi(x_1), \ldots, \varphi(x_D)] \tag{16.3}$$

See Table 16.1 for a list of some common activation functions, and Figure 16.2 for a visualization. For more details, see e.g., [Mur22, Sec 13.2.3].

Name	Definition	Range	Reference
Sigmoid	$\sigma(a) = \frac{1}{1+e^{-a}}$	$[0, 1]$	
Hyperbolic tangent	$\tanh(a) = 2\sigma(2a) - 1$	$[-1, 1]$	
Softplus	$\sigma_+(a) = \log(1 + e^a)$	$[0, \infty]$	[GBB11]
Rectified linear unit	$\text{ReLU}(a) = \max(a, 0)$	$[0, \infty]$	[GBB11; KSH12a]
Leaky ReLU	$\max(a, 0) + \alpha \min(a, 0)$	$[-\infty, \infty]$	[MHN13]
Exponential linear unit	$\max(a, 0) + \min(\alpha(e^a - 1), 0)$	$[-\infty, \infty]$	[CUH16]
Swish	$a\sigma(a)$	$[-\infty, \infty]$	[RZL17]
GELU	$a\Phi(a)$	$[-\infty, \infty]$	[HG16]

Table 16.1: List of some popular activation functions for neural networks.

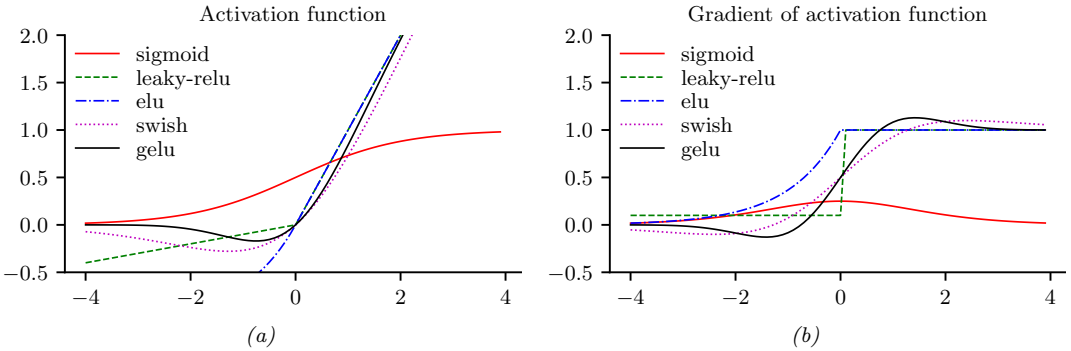


Figure 16.2: (a) Some popular activation functions. "ReLU" stands for "restricted linear unit". "GELU" stands for "Gaussian error linear unit". (b) Plot of their gradients. Generated by activation_fun_deriv.ipynb.

16.2.3 Convolutional layers

When dealing with image data, we can apply the same weight matrix to each local patch of the image, in order to reduce the number of parameters. If we "slide" this weight matrix over the image and add up the results, we get a technique known as **convolution**; in this case the weight matrix is often called a "**kernel**" or "**filter**".

More precisely, let $\mathbf{X} \in \mathbb{R}^{H \times W}$ be the input image, and $\mathbf{W} \in \mathbb{R}^{h \times w}$ be the kernel. The output is denoted by $\mathbf{Z} = \mathbf{X} \circledast \mathbf{W}$, where (ignoring boundary conditions) we have the following:[1]

$$Z_{i,j} = \sum_{u=0}^{h-1} \sum_{v=0}^{w-1} x_{i+u,j+v} w_{u,v} \tag{16.4}$$

Essentially we compare a local patch of $\boldsymbol{x}$, of size $h \times w$ and centered at (i, j), to the filter $\boldsymbol{w}$; the output just measures how similar the input patch is to the filter. We can define convolution in 1d or 3d in an analogous manner. Note that the spatial size of the outputs may be smaller than inputs,

1. Note that, technically speaking, we are using **cross correlation** rather than convolution. However, these terms are used interchangeably in deep learning.

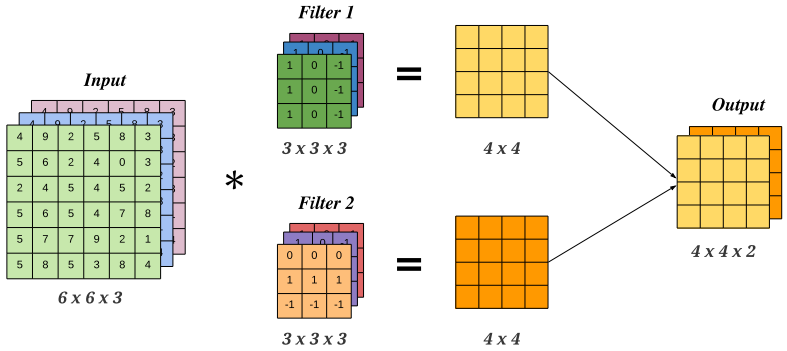


Figure 16.3: A 2d convolutional layer with 3 input channels and 2 output channels. The kernel has size 3×3 *and we use stride 1 with 0 padding, so the the* 6×6 *input gets mapped to the* 4×4 *output.*

due to boundary effects, although this can be solved by using **padding**. See [Mur22, Sec 14.2.1] for more details.

We can repeat this process for multiple layers of inputs, and by using multiple filters, we can generate multiple layers of output. In general, if we have C input channels, and we want to map it to D output (feature) channels, then we define D kernels, each of size $h \times w \times C$, where h, w are the height and width of the kernel. The d'th output feature map is obtained by convolving all C input feature maps with the d'th kernel, and then adding up the results elementwise:

$$z_{i,j,d} = \sum_{u=0}^{h-1} \sum_{v=0}^{w-1} \sum_{c=0}^{C-1} x_{i+u,j+v,c} w_{u,v,c,d} \tag{16.5}$$

This is called a **convolutional layer**, and is illustrated in Figure 16.3.

The advantage of a convolutional layer compared to using a linear layer is that the weights of the kernel are shared across locations in the input. Thus if a pattern in the input shifts locations, the corresponding output activation will also shift. This is called **shift equivariance**. In some cases, we want the output to be the same, no matter where the input pattern occurs; this is called **shift invariance**, and can be obtained by using a **pooling layer**, which computes the maximum or average value in each local patch of the input. (Note that pooling layers have no free (learnable) parameters.) Other forms of invariance can also be captured by neural networks (see e.g., [CW16; FWW21]).

16.2.4 Residual (skip) connections

If we stack a large number of nonlinear layers together, the signal may get squashed to zero or may blow up to infinity, depending on the magnitude of the weights, and the nature of the nonlinearities. Similar problems can plague gradients that are passed backwards through the network (see Section 6.2). To reduce the effect of this we can add **skip connections**, also called **residual connections**, which allow the signal to skip one or more layers, which prevents it from being modified. For example,

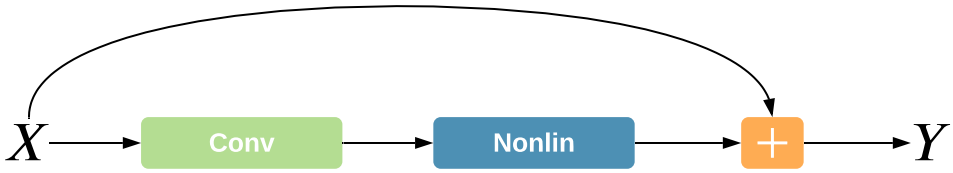


Figure 16.4: A residual connection around a convolutional layer.

Figure 16.4 illustrates a network that computes

$$\boldsymbol{y} = f(\boldsymbol{x}; \mathbf{W}) = \varphi(\text{conv}(\boldsymbol{x}; \mathbf{W})) + \boldsymbol{x} \tag{16.6}$$

Now the convolutional layer only needs to learn an offset or residual to add (or subtract) to the input to match the desired output, rather than predicting the output directly. Such residuals are often small in size, and hence are easier to learn using neurons with weights that are bounded (e.g., close to 1).

16.2.5 Normalization layers

To learn an input-output mapping, it is often best if the inputs are standardized, meaning that they have zero mean and unit standard deviation. This ensures that the required magnitude of the weights is small, and comparable across dimensions. To ensure that the internal activations have this property, it is common to add **normalization layers**.

The most common approach is to use **batch normalization** (**BN**) [IS15]. However this relies on having access to a batch of $B > 1$ input examples. Various alternatives have been proposed to overcome the need of having an input batch, such as **layer normalization** [BKH16], **instance normalization** [UVL16], **group normalization** [WH18], **filter response normalization** [SK20], etc. More details can be found in [Mur22, Sec 14.2.4].

16.2.6 Dropout layers

Neural networks often have millions of parameters, and thus can sometimes overfit, especially when trained on small datasets. There are many ways to ameliorate this effect, such as applying regularizers to the weights, or adopting a fully Bayesian approach (see Chapter 17). Another common heuristic is known as **dropout** [Sri+14a], in which edges are randomly omitted each time the network is used, as illustrated in Figure 16.5. More precisely, if w_{lij} is the weight of the edge from node i in layer $l-1$ to node j in layer $l+1$, then we replace it with $\theta_{lij} = w_{lij}\epsilon_{li}$, where $\epsilon_{li} \sim \text{Ber}(1-p)$, where p is the drop probability, and $1-p$ is the keep probability. Thus if we sample $\epsilon_{li} = 0$, then all of the weights going out of unit i in layer $l-1$ into any j in layer l will be set to 0.

During training, the gradients will be zero for the weights connected to a neuron which has been switched "off". However, since we resample ϵ_{lij} every time the network is used, different combinations of weights will be updated on each step. The result is an **ensemble** of networks, each with slightly different sparse graph structures.

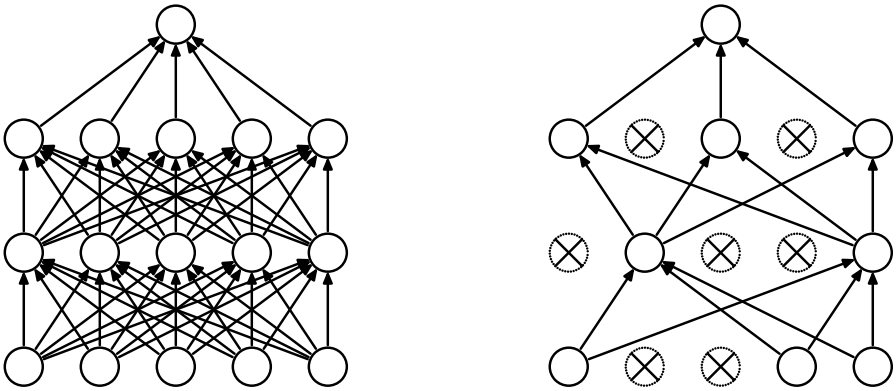


Figure 16.5: Illustration of dropout. (a) A standard neural net with 2 hidden layers. (b) An example of a thinned net produced by applying dropout with $p = 0.5$. Units that have been dropped out are marked with an x. From Figure 1 of [Sri+14a]. Used with kind permission of Geoff Hinton.

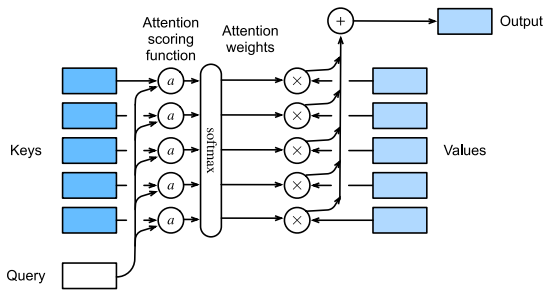


Figure 16.6: Attention layer. (a) Mapping a single query $\boldsymbol{q}$ to a single output, given a set of keys and values. From Figure 10.3.1 of [Zha+20a]. Used with kind permission of Aston Zhang.

At test time, we usually turn the dropout noise off, so the model acts deterministically. To ensure the weights have the same expectation at test time as they did during training (so the input activation to the neurons is the same, on average), at test time we should use $\mathbb{E}[\theta_{lij}] = w_{lij}\mathbb{E}[\epsilon_{li}]$. For Bernoulli noise, we have $\mathbb{E}[\epsilon] = 1 - p$, so we should multiply the weights by the keep probability, $1 - p$, before making predictions. We can, however, use dropout at test time if we wish. This is called **Monte Carlo dropout** (see Section 17.3.1).

16.2.7 Attention layers

In non-parametric kernel based prediction methods, such as Gaussian processes (Chapter 18), we compare the input $\boldsymbol{x} \in \mathbb{R}^{d_k}$ to each of the training examples $\mathbf{X} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)$ using a kernel to get a vector of similarity scores, $\boldsymbol{\alpha} = [\mathcal{K}(\boldsymbol{x}, \boldsymbol{x}_i)]_{i=1}^n$. We then use this to retrieve a weighted combination

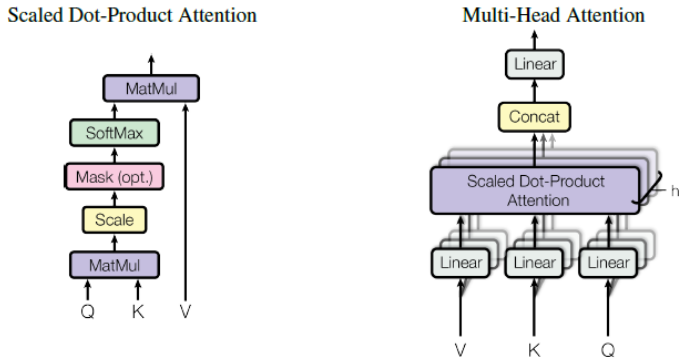


Figure 16.7: (a) Scaled dot-product attention in matrix form. (b) Multi-head attention. From Figure 2 of [Vas+17b]. Used with kind permission of Ashish Vaswani.

of the corresponding m target values $\boldsymbol{y}_i \in \mathbb{R}^{d_v}$ as follows:

$$\hat{\boldsymbol{y}} = \sum_{i=1}^{n} \alpha_i \boldsymbol{y}_i \tag{16.7}$$

See Section 18.3.7 for details.

We can make a differentiable and parametric version of this as follows (see [Tsa+19] for details). First we replace the stored examples matrix $\mathbf{X}$ with a learned embedding, to create a set of stored **keys**, $\mathbf{K} = \mathbf{W}^K\mathbf{X} \in \mathbb{R}^{n \times d_k}$. Similarly we replace the stored output matrix $\mathbf{Y}$ with a learned embedding, to create a set of stored **values**, $\mathbf{V} = \mathbf{W}^V\mathbf{Y} \in \mathbb{R}^{n \times d_v}$. Finally we embed the input to create a **query**, $\boldsymbol{q} = \mathbf{W}^Q\boldsymbol{x} \in \mathbb{R}^{d_k}$. The parameters to be learned are the three embedding matrices.

To ensure the output is a differentiable function of the input, we replace the fixed kernel function with a soft **attention layer**. More precisely, we define

$$\text{Attn}(\boldsymbol{q}, (\boldsymbol{k}_1, \boldsymbol{v}_1), \ldots, (\boldsymbol{k}_n, \boldsymbol{v}_n)) = \text{Attn}(\boldsymbol{q}, (\boldsymbol{k}_{1:n}, \boldsymbol{v}_{1:n})) = \sum_{i=1}^{n} \alpha_i(\boldsymbol{q}, \boldsymbol{k}_{1:m})\boldsymbol{v}_i \tag{16.8}$$

where $\alpha_i(\boldsymbol{q}, \boldsymbol{k}_{1:n})$ is the i'th **attention weight**; these weights satisfy $0 \leq \alpha_i(\boldsymbol{q}, \boldsymbol{k}_{1:n}) \leq 1$ for each i and $\sum_i \alpha_i(\boldsymbol{q}, \boldsymbol{k}_{1:n}) = 1$.

The attention weights can be computed from an **attention score** function $a(\boldsymbol{q}, \boldsymbol{k}_i) \in \mathbb{R}$, that computes the similarity of query $\boldsymbol{q}$ to key $\boldsymbol{k}_i$. For example, we can use (scaled) **dot product attention**, which has the form

$$a(\boldsymbol{q}, \boldsymbol{k}) = \boldsymbol{q}^\mathsf{T}\boldsymbol{k}/\sqrt{d_k} \tag{16.9}$$

(The scaling by $\sqrt{d_k}$ is to reduce the dependence of the output on the dimensionality of the vectors.) Given the scores, we can compute the attention weights using the softmax function:

$$\alpha_i(\boldsymbol{q}, \boldsymbol{k}_{1:n}) = \text{softmax}_i([a(\boldsymbol{q}, \boldsymbol{k}_1), \ldots, a(\boldsymbol{q}, \boldsymbol{k}_n)]) = \frac{\exp(a(\boldsymbol{q}, \boldsymbol{k}_i))}{\sum_{j=1}^{n} \exp(a(\boldsymbol{q}, \boldsymbol{k}_j))} \tag{16.10}$$

See Figure 16.6 for an illustration.

In some cases, we want to restrict attention to a subset of the dictionary, corresponding to valid entries. For example, we might want to pad sequences to a fixed length (for efficient minibatching), in which case we should "mask out" the padded locations. This is called **masked attention**. We can implement this efficiently by setting the attention score for the masked entries to a large negative number, such as -10^6, so that the corresponding softmax weights will be 0.

In practice, we usually deal with minibatches of n vectors at a time. Let the corresponding matrices of queries, keys and values be denoted by $\mathbf{Q} \in \mathbb{R}^{n \times d_k}$, $\mathbf{K} \in \mathbb{R}^{n \times d_k}$, $\mathbf{V} \in \mathbb{R}^{n \times d_v}$. Let

$$\boldsymbol{z}_j = \sum_{i=1}^{n} \alpha_i(\boldsymbol{q}_j, \mathbf{K})\boldsymbol{v}_i \tag{16.11}$$

be the j'th output corresponding to the j'th query. We can compute all outputs $\mathbf{Z} \in \mathbb{R}^{n \times d_v}$ in parallel using

$$\mathbf{Z} = \text{Attn}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}(\frac{\mathbf{Q}\mathbf{K}^\intercal}{\sqrt{d_k}})\mathbf{V} \tag{16.12}$$

where the softmax function softmax is applied row-wise. See Figure 16.7(left) for an illustration.

To increase the flexibility of the model, we often use a **multi-head attention** layer, as illustrated in Figure 16.7(right). Let the i'th head be

$$\boldsymbol{h}_i = \text{Attn}(\mathbf{Q}\mathbf{W}_i^Q, \mathbf{K}\mathbf{W}_i^K, \mathbf{V}\mathbf{W}_i^V) \tag{16.13}$$

where $\mathbf{W}_i^Q \in \mathbb{R}^{d \times d_k}$, $\mathbf{W}_i^K \in \mathbb{R}^{d \times d_k}$ and $\mathbf{W}_i^V \in \mathbb{R}^{d \times d_v}$ are linear projection matrices. We define the output of the MHA layer to be

$$\mathbf{Z} = \text{MHA}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{Concat}(\boldsymbol{h}_1, \ldots, \boldsymbol{h}_h)\mathbf{W}^O \tag{16.14}$$

where h is the number of heads, and $\mathbf{W}^O \in \mathbb{R}^{hd_v \times d}$. Having multiple heads can increase performance of the layer, in the event that some of the weight matrices are poorly initialized; after training, we can often remove all but one of the heads [MLN19].

When the output of one attention layer is used as input to another, the method is called **self-attention**. This is the basis of the transformer model, which we discuss in Section 16.3.5.

16.2.8 Recurrent layers

We can make the model be **stateful** by augmenting the input $\boldsymbol{x}$ with the current state $\boldsymbol{s}_t$, and then computing the output and the new state using some kind of function:

$$(\boldsymbol{y}, \boldsymbol{s}_{t+1}) = f(\boldsymbol{x}, \boldsymbol{s}_t) \tag{16.15}$$

This is called a **recurrent layer**, as shown in Figure 16.8. This forms the basis of **recurrent neural networks**, discussed in Section 16.3.4. In a vanilla RNN, the function f is a simple MLP, but it may also use attention (Section 16.2.7).

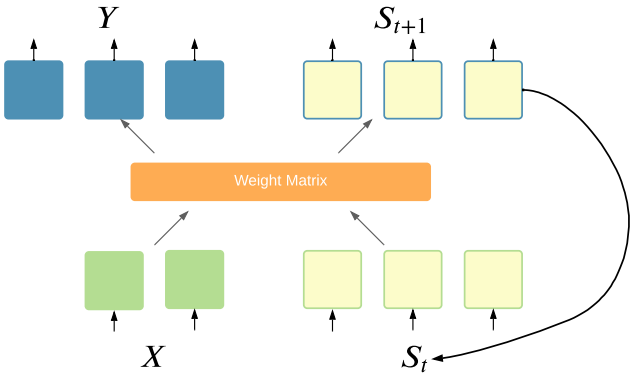


Figure 16.8: Recurrent layer.

16.2.9 Multiplicative layers

In this section, we discuss **multiplicative layers**, which are useful for combining different information sources. Our presentation follows [Jay+20].

Suppose we have inputs $\boldsymbol{x} \in \mathbb{R}^n$ and $\boldsymbol{z} \in \mathbb{R}^m$, In a linear layer (and, by extension, convolutional layers), it is common to concatenate the inputs to get $f(\boldsymbol{x}, \boldsymbol{z}) = \mathbf{W}[\boldsymbol{x}; \boldsymbol{z}]+\boldsymbol{b}$, where $\mathbf{W} \in \mathbb{R}^{k\times(m+n)}$ and $\boldsymbol{b} \in \mathbb{R}^k$. We can increase the expressive power of the model by using **multiplicative interactions**, such as the following **bilinear form**:

$$f(\boldsymbol{x}, \boldsymbol{z}) = \boldsymbol{z}^\mathsf{T}\mathbb{W}\boldsymbol{x} + \mathbf{U}\boldsymbol{z} + \mathbf{V}\boldsymbol{x} + \boldsymbol{b} \tag{16.16}$$

where $\mathbb{W} \in \mathbb{R}^{m\times n\times k}$ is a weight tensor, defined such that

$$(\boldsymbol{z}^\mathsf{T}\mathbb{W}\boldsymbol{x})_k = \sum_{ij} \boldsymbol{z}_i \mathbb{W}_{ijk} \boldsymbol{x}_j \tag{16.17}$$

That is, the k'th entry of the output is the weighted inner product of $\boldsymbol{z}$ and $\boldsymbol{x}$, where the weight matrix is the k'th "slice" of $\mathbb{W}$. The other parameters have size $\mathbf{U} \in \mathbb{R}^{k\times m}$, $\mathbf{V} \in \mathbb{R}^{k\times n}$, and $\boldsymbol{b} \in \mathbb{R}^k$.

This formulation includes many interesting special cases. In particular, a **hypernetwork** [HDL17] can be viewed in this way. A hypernetwork is a neural network that generates parameters for another neural network. In particular, we replace $f(\boldsymbol{x}; \boldsymbol{\theta})$ with $f(\boldsymbol{x}; g(\boldsymbol{z}; \boldsymbol{\phi}))$. If f and g are affine, this is equivalent to a multiplicative layer. To see this, let $\mathbf{W}' = \boldsymbol{z}^\mathsf{T}\mathbb{W} + \mathbf{V}$ and $\boldsymbol{b}' = \mathbf{U}\boldsymbol{z} + \boldsymbol{b}$. If we define $g(\boldsymbol{z}; \boldsymbol{\Phi}) = [\mathbf{W}', \boldsymbol{b}']$, and $f(\boldsymbol{x}; \boldsymbol{\theta}) = \mathbf{W}'\boldsymbol{x} + \boldsymbol{b}'$, we recover Equation (16.16).

We can also view the gating layers used in RNNs (Section 16.3.4) as a form of multiplicative interaction. In particular, if we the hypernetwork computes the diagonal matrix $\mathbf{W}' = \sigma(\boldsymbol{z}^\mathsf{T}\mathbb{W}+\mathbf{V}) = \text{diag}(a_1, \ldots, a_n)$, then we can define $f(\boldsymbol{x}, \boldsymbol{z}; \boldsymbol{\theta}) = \boldsymbol{a}(\boldsymbol{z}) \odot \boldsymbol{x}$, which is the standard gating mechanism. Attention mechanisms (Section 16.2.7) are also a form of multiplicative interaction, although they involve three-way interactions, between query, key and value.

Another variant arises if the hypernetwork just computes a scalar weight for each channel of a convolutional layer, plus a bias term:

$$f(\boldsymbol{x}, \boldsymbol{z}) = a(\boldsymbol{z}) \odot \boldsymbol{x} + \boldsymbol{b}(\boldsymbol{z}) \tag{16.18}$$

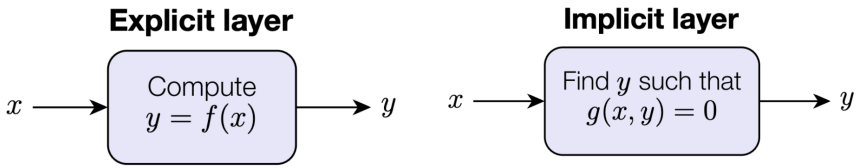


Figure 16.9: Explicit vs implicit layers.

This is called **FiLM**, which stands for "Feature-wise Linear Modulation" [Per+18]. For a detailed tutorial on the FiLm layer and its many applications, see https://distill.pub/2018/feature-wise-transformations.

16.2.10 Implicit layers

So far we have focused on **explicit layers**, which specify how to transform the input to the output using $\boldsymbol{y} = f(\boldsymbol{x})$. We can also define **implicit layers**, which specify the output indirectly, in terms of a constraint function:

$$\boldsymbol{y} \in \underset{\boldsymbol{y}}{\operatorname{argmin}} f(\boldsymbol{x}, \boldsymbol{y}) \text{ such that } g(\boldsymbol{x}, \boldsymbol{y}) = 0 \tag{16.19}$$

The details on how to find a solution to this constrained optimization problem can vary depending on the problem. For example, we may need to run an inner optimization routine, or call a differential equation solver. The main advantage of this approach is that the inner computations do not need to be stored explicitly, which saves a lot of memory. Furthermore, once the solution has been found, we can propagate gradients through the whole layer, by leveraging the implicit function theorem. This lets us use higher level primitives inside an end-to-end framework. For more details, see [GHC21] and http://implicit-layers-tutorial.org/.

16.3 Canonical examples of neural networks

In this section, we give several "canonical" examples of neural network architectures that are widely used for different tasks.

16.3.1 Multi-layer perceptrons (MLP)

A **multi-layer perceptron** (**MLP**), also called a **feedforward neural network** (**FFNN**), is one of the simplest kinds of neural networks. It consists of a series of L linear layers, combined with elementwise nonlinearities:

$$f(\boldsymbol{x}; \boldsymbol{\theta}) = \mathbf{W}_L \varphi_L(\mathbf{W}_{L-1} \varphi_{L-1}(\cdots \varphi_1(\mathbf{W}_1 \boldsymbol{x}) \cdots)) \tag{16.20}$$

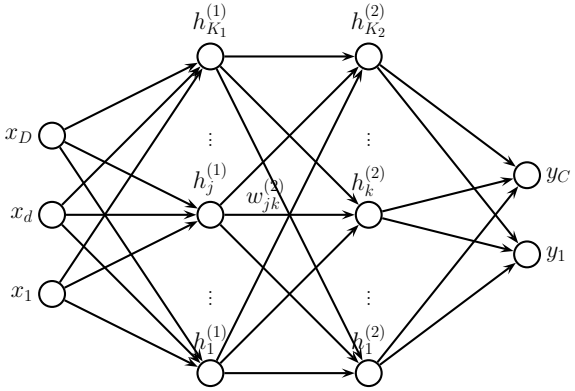


Figure 16.10: A feedforward neural network with D inputs, K_1 hidden units in layer 1, K_2 hidden units in layer 2, and C outputs. $w_{jk}^{(l)}$ is the weight of the connection from node j in layer $l-1$ to node k in layer l.

For example, Figure 16.10 shows an MLP with 1 input layer of D units, 2 hidden layers of K_1 and K_2 units, and 1 output layer with C units. The k'th hidden unit in layer l is given by

$$h_k^{(l)} = \varphi_l\left(b_k^{(l)} + \sum_{j=1}^{K_{l-1}} w_{jk}^{(l)} h_j^{(l-1)}\right) \tag{16.21}$$

where φ_l is the nonlinear activation function at layer l.

For a classification problem, the final nonlinearity is usually the softmax function. However, it is also common for the final layer to have linear activations, in which case the outputs are interpreted as logits; the loss function used during training then converts to (log) probabilities internally.

We can also use MLPs for regression. Figure 16.11 shows how we can make a model for **heteroskedastic** nonlinear regression. (The term "heteroskedastic" just means that the predicted output variance is input-dependent, rather than a constant.) This function has two outputs which compute $f_\mu(\boldsymbol{x}) = \mathbb{E}[y|\boldsymbol{x}, \boldsymbol{\theta}]$ and $f_\sigma(\boldsymbol{x}) = \sqrt{\mathbb{V}[y|\boldsymbol{x}, \boldsymbol{\theta}]}$. We can share most of the layers (and hence parameters) between these two functions by using a common "**backbone**" and two output "**heads**", as shown in Figure 16.11. For the μ head, we use a linear activation, $\varphi(a) = a$. For the σ head, we use a softplus activation, $\varphi(a) = \sigma_+(a) = \log(1 + e^a)$. If we use linear heads and a nonlinear backbone, the overall model is given by

$$p(y|\boldsymbol{x}, \boldsymbol{\theta}) = \mathcal{N}\left(y|\boldsymbol{w}_{\boldsymbol{\mu}}^\mathsf{T} f(\boldsymbol{x}; \boldsymbol{w}_{\text{shared}}), \sigma_+(\boldsymbol{w}_{\boldsymbol{\sigma}}^\mathsf{T} f(\boldsymbol{x}; \boldsymbol{w}_{\text{shared}}))\right) \tag{16.22}$$

16.3.2 Convolutional neural networks (CNN)

A vanilla **convolutional neural network** or **CNN** consists of a series of convolutional layers, pooling layers, linear layers, and nonlinearities. See Figure 16.12 for an example. More sophisticated architectures, such as the **ResNet** model [He+16a; He+16b], add skip (residual) connections, normalization layers, etc. The **ConvNeXt** model of [Liu+22] is considered the current (as of February 2022) state of the art CNN architecture for a wide variety of vision tasks. See e.g., [Mur22, Ch.14] for more details on CNNs.

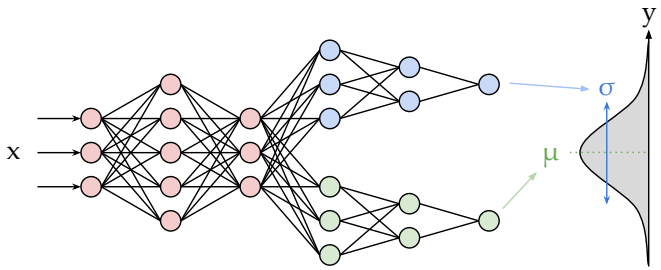


Figure 16.11: Illustration of an MLP with a shared "backbone" and two output "heads", one for predicting the mean and one for predicting the variance. From https://brendanhasz.github.io/2019/07/23/bayesian-density-net.html. Used with kind permission of Brendan Hasz.

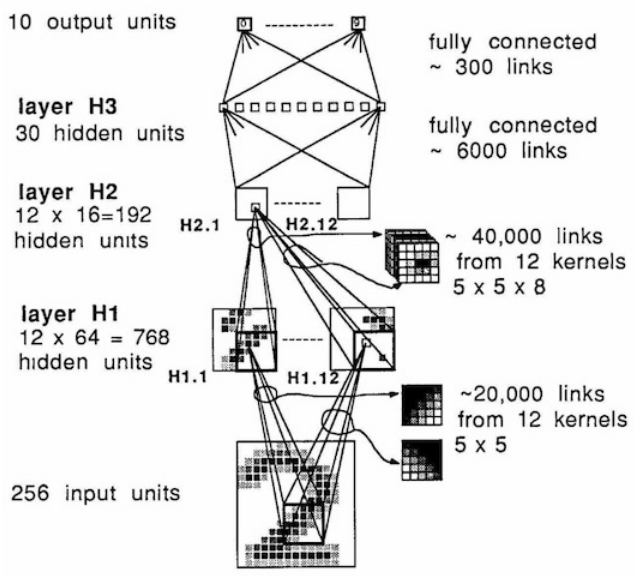


Figure 16.12: One of the first CNNs ever created, for classifying MNIST images. From Figure 3 of [LeC+89]. For a "modern" implementation, see lecun1989.ipynb.

16.3.3 Autoencoders

An **autoencoder** is a neural network that maps inputs $\boldsymbol{x}$ to a low-dimensional latent space using an **encoder**, $\boldsymbol{z} = f_e(\boldsymbol{x})$, and then attempts to reconstruct the inputs using a **decoder**, $\hat{\boldsymbol{x}} = f_d(\boldsymbol{z})$. The model is trained to minimize

$$\mathcal{L}(\boldsymbol{\theta}) = ||r(\boldsymbol{x}) - \boldsymbol{x}||_2^2 \tag{16.23}$$

where $r(\boldsymbol{x}) = f_d(f_e(\boldsymbol{x}))$. (We can also replace squared error with more general conditional log likelihoods.) See Figure 16.13 for an illustration of a 3 layer AE.

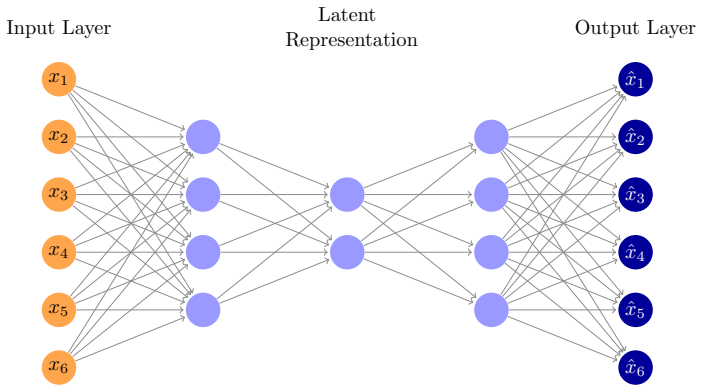
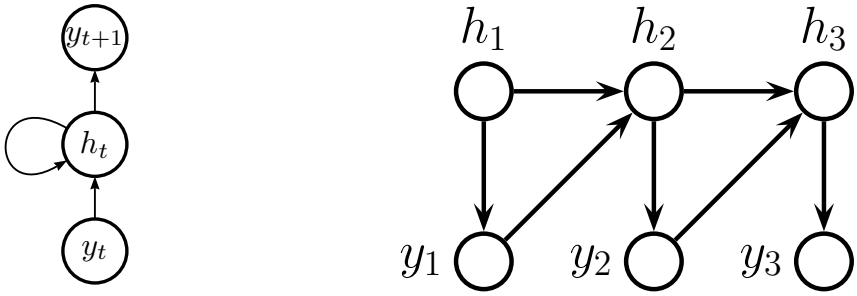


Figure 16.13: Illustration of an autoencoder with 3 hidden layers.

Figure 16.14: (a) Some MNIST digits. (b) Reconstruction of these images using a convolutional autoencoder. (c) t-SNE visualization of the 20-d embeddings. The colors correspond to class labels, which were not used during training. Generated by ae_mnist_conv_jax.ipynb.

For image data, we can make the encoder be a convolutional network, and the decoder be a transpose convolutional network. We can use this to compute low dimensional embeddings of image data. For example, suppose we fit such a model to some MNIST digits. We show the reconstruction abilities of such a model in Figure 16.14b. In Figure 16.14c, we show a 2d visualization of the 20-dimensional embedding space computed using t-SNE. The colors correspond to class labels, which were not used during training. We see fairly good separation, showing that images which are visually similar are placed close to each other in the embedding space, as desired. (See also Section 21.2.6, where we compare AEs with variational AEs.)



(a) *(b)*

Figure 16.15: Illustration of a recurrent neural network (RNN). (a) With self-loop. (b) Unrolled in time.

16.3.4 Recurrent neural networks (RNN)

A **recurrent neural network** (**RNN**) is a network with a recurrent layer, as in Equation (16.15). This is illustrated in Figure 16.15. Formally this defines the following probability distribution over sequences:

$$p(\boldsymbol{y}_{1:T}) = \sum_{\boldsymbol{h}_{1:T}} p(\boldsymbol{y}_{1:T}, \boldsymbol{h}_{1:T}) = \sum_{\boldsymbol{h}_{1:T}} \mathbb{I}\left(\boldsymbol{h}_1 = \boldsymbol{h}_1^*\right) p(\boldsymbol{y}_1|\boldsymbol{h}_1) \prod_{t=2}^{T} p(\boldsymbol{y}_t|\boldsymbol{h}_t) \mathbb{I}\left(\boldsymbol{h}_t = f(\boldsymbol{h}_{t-1}, \boldsymbol{y}_{t-1})\right) \quad (16.24)$$

where $\boldsymbol{h}_t$ is the deterministic hidden state, computed from the last hidden state and last output using $f(\boldsymbol{h}_{t-1}, \boldsymbol{y}_{t-1})$. (At training time, $\boldsymbol{y}_{t-1}$ is observed, but at prediction time, it is generated.)

In a vanilla RNN, the function f is a simple MLP. However, we can also use attention to selectively update parts of the state vector based on similarity between the input the previous state, as in the **GRU** (gated recurrent unit) model, and the **LSTM** (long short term memory) model. We can also make the model into a conditional sequence model, by feeding in extra inputs to the f function. See e.g., [Mur22, Ch. 15] for more details on RNNs.

16.3.5 Transformers

Consider the problem of classifying each word in a sentence, for example with its part of speech tag (noun, verb, etc). That is, we want to learn a mapping $f : \mathcal{X} \to \mathcal{Y}$, where $\mathcal{X} = \mathcal{V}^T$ is the set of input sequences defined over (word) vocabulary $\mathcal{V}$, T is the length of the sentence, and $\mathcal{Y} = \mathcal{T}^T$ is the set of output sequences, defined over (tag) vocabulary $\mathcal{T}$. To do well at this task, we need to learn a contextual embedding of each word. RNNs process one token at a time, so the embedding of the word at location t, $\boldsymbol{z}_t$, depends on the hidden state of the network, $\boldsymbol{s}_t$, which may be a lossy summary of all the previously seen words. We can create bidirectional RNNs so that future words can also affect the embedding of $\boldsymbol{z}_t$, but this dependence is still mediated via the hidden state. An alternative approach is to compute $\boldsymbol{z}_t$ as a direct function of all the other words in the sentence, by using the attention operator discussed in Section 16.2.7 rather than using hidden state. This is called an (encoder-only) **transformer**, and is used by models such as BERT [Dev+19]. This idea is sketched in Figure 16.16.

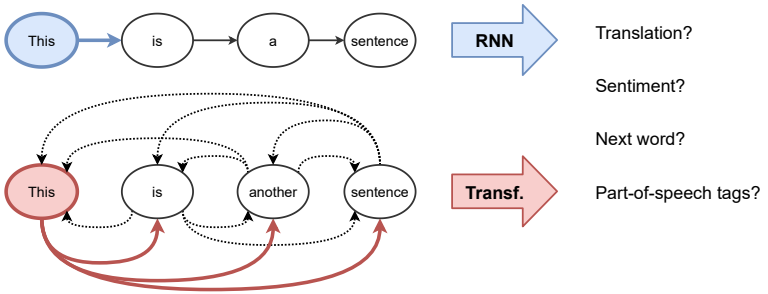


Figure 16.16: Visualizing the difference between an RNN and a transformer. From [Jos20]. Used with kind permission of Chaitanya Joshi.

It is also possible to create a decoder-only transformer, in which each output $\boldsymbol{y}_t$ only attends to all the previously generated outputs, $\boldsymbol{y}_{1:t-1}$. This can be implemented using masked attention, and is useful for generative language models, such as GPT (see Section 22.4.1). We can combine the encoder and decoder to create a conditional sequence-to-sequence model, $p(\boldsymbol{y}_{1:T_y}|\boldsymbol{x}_{1:T_x})$, as proposed in the original transformer paper [Vas+17c]. See Supplementary Section 16.1.1 for the details.

It has been found that large transformers are very flexible sequence-to-sequence function approximators, if trained on enough data (see e.g., [Lin+21a] for a review in the context of NLP, and [Kha+21; Han+20; Zan21] for reviews in the context of computer vision). The reasons why they work so well are still not very clear. However, some initial insights can be found in e.g., [Rag+21; WGY21; Nel21; BP21]. See also Supplementary Section 16.1.2.5 where we discuss the connection with graph neural networks.

16.3.6 Graph neural networks (GNNs)

It is possible to define neural networks for working with graph-structured data. These are called **graph neural networks** or **GNNs**. See Supplementary Section 16.1.2 for details.

17 Bayesian neural networks

This chapter is coauthored with Andrew Wilson.

17.1 Introduction

Deep neural networks (DNNs) are usually trained using a (penalized) maximum likelihood objective to find a single setting of parameters. However, large flexible models like neural networks can represent many functions, corresponding to different parameter settings, which fit the training data well, yet generalize in different ways, a phenoment known as **underspecification** (see e.g., [D'A+20], and Figure 17.10 for an illustration). Considering all of these different models together can lead to improved accuracy and uncertainty representation. This can be done by computing the posterior predictive distribution using Bayesian model averaging:

$$p(\boldsymbol{y}|\boldsymbol{x}, \mathcal{D}) = \int p(\boldsymbol{y}|\boldsymbol{x}, \boldsymbol{\theta}) p(\boldsymbol{\theta}|\mathcal{D}) d\boldsymbol{\theta} \tag{17.1}$$

where $p(\boldsymbol{\theta}|\mathcal{D}) \propto p(\boldsymbol{\theta}) p(\mathcal{D}|\boldsymbol{\theta})$.

The main challenges in applying Bayesian inference to DNNs are specifying suitable priors, and efficiently computing the posterior, which is challenging due to the large number of parameters and the large datasets. The application of Bayesian inference to DNNs is sometimes called **Bayesian deep learning** or **BDL**. By contrast, the term **deep Bayesian learning** or **DBL** refers to the use of deep models to help speedup Bayesian inference of "classical" models, usually by training amortized inference networks that can be used as part of a variational inference or importance sampling algorithm, as discussed in Section 10.3.6.) For more details on the topic of BDL, see e.g., [PS17; Wil20; WI20; Jos+22; Kha20].

17.2 Priors for BNNs

To perform Bayesian inference for the parameters of a DNN, we need to specify a prior $p(\boldsymbol{\theta})$. [Nal18; WI20; For21] discusses the issue of prior selection at length. Here we just give a brief summary of common approaches.

17.2.1 Gaussian priors

Consider an MLP with one hidden layer with activation function φ and a linear output:

$$f(\boldsymbol{x};\boldsymbol{\theta}) = \mathbf{W}_2\varphi(\mathbf{W}_1\boldsymbol{x} + \boldsymbol{b}_1) + \boldsymbol{b}_2 \tag{17.2}$$

(If the output is nonlinear, such as a softmax transform, we can fold it into the loss function during training.) If we have two hidden layers this becomes

$$f(\boldsymbol{x};\boldsymbol{\theta}) = \mathbf{W}_3\left(\varphi\left(\mathbf{W}_2\varphi(\mathbf{W}_1\boldsymbol{x} + \boldsymbol{b}_1) + \boldsymbol{b}_2\right)\right) + \boldsymbol{b}_3 \tag{17.3}$$

In general, with $L-1$ hidden layers and a linear output, we have

$$f(\boldsymbol{x};\boldsymbol{\theta}) = \mathbf{W}_L\left(\cdots\varphi(\mathbf{W}_1\boldsymbol{x} + \boldsymbol{b}_1)\right) + \boldsymbol{b}_L \tag{17.4}$$

We need to specify the priors for $\mathbf{W}_l$ and $\boldsymbol{b}_l$ for $l = 1 : L$. The most common choice is to use a factored Gaussian prior:

$$\mathbf{W}_\ell \sim \mathcal{N}(\mathbf{0}, \alpha_\ell^2\mathbf{I}),\ \boldsymbol{b}_\ell \sim \mathcal{N}(\mathbf{0}, \beta_\ell^2\mathbf{I}) \tag{17.5}$$

The **Xavier initialization** or **Glorot initialization**, named after the first author of [GB10], is to set

$$\alpha_\ell^2 = \frac{2}{n_{\text{in}} + n_{\text{out}}} \tag{17.6}$$

where n_{in} is the fan-in of a node in level ℓ (number of weights coming into a neuron), and n_{out} is the fan-out (number of weights going out of a neuron). **LeCun initialization**, named after Yann LeCun, corresponds to using

$$\alpha_\ell^2 = \frac{1}{n_{\text{in}}} \tag{17.7}$$

We can get a better understanding of these priors by considering the effect they have on the corresponding distribution over functions that they define. To help understand this correspondence, let us reparameterize the model as follows:

$$\mathbf{W}_\ell = \alpha_\ell\boldsymbol{\eta}_\ell,\ \boldsymbol{\eta}_\ell \sim \mathcal{N}(\mathbf{0}, \mathbf{I}),\ \boldsymbol{b}_\ell = \beta_\ell\boldsymbol{\epsilon}_\ell,\ \boldsymbol{\epsilon}_\ell \sim \mathcal{N}(\mathbf{0}, \mathbf{I}) \tag{17.8}$$

Hence every setting of the prior hyperparameters specifies the following random function:

$$f(\boldsymbol{x};\boldsymbol{\alpha},\boldsymbol{\beta}) = \alpha_L\boldsymbol{\eta}_L(\cdots\varphi(\alpha_1\boldsymbol{\eta}_1\boldsymbol{x} + \beta_1\boldsymbol{\epsilon}_1)) + \beta_L\boldsymbol{\epsilon}_L \tag{17.9}$$

To get a feeling for the effect of these hyperparameters, we can sample MLP parameters from this prior and plot the resulting random functions. We use a sigmoid nonlinearity, so $\varphi(a) = \sigma(a)$. We consider $L = 2$ layers, so $\mathbf{W}_1$ are the input-to-hidden weights, and $\mathbf{W}_2$ are the hidden-to-output weights. We assume the input and output are scalars, so we are generating random nonlinear 1d mappings $f : \mathbb{R} \to \mathbb{R}$.

Figure 17.1(a) shows some sampled functions where $\alpha_1 = 5$, $\beta_1 = 1$, $\alpha_2 = 1$, $\beta_2 = 1$. In Figure 17.1(b) we increase α_1; this allows the first layer weights to get bigger, making the sigmoid-like

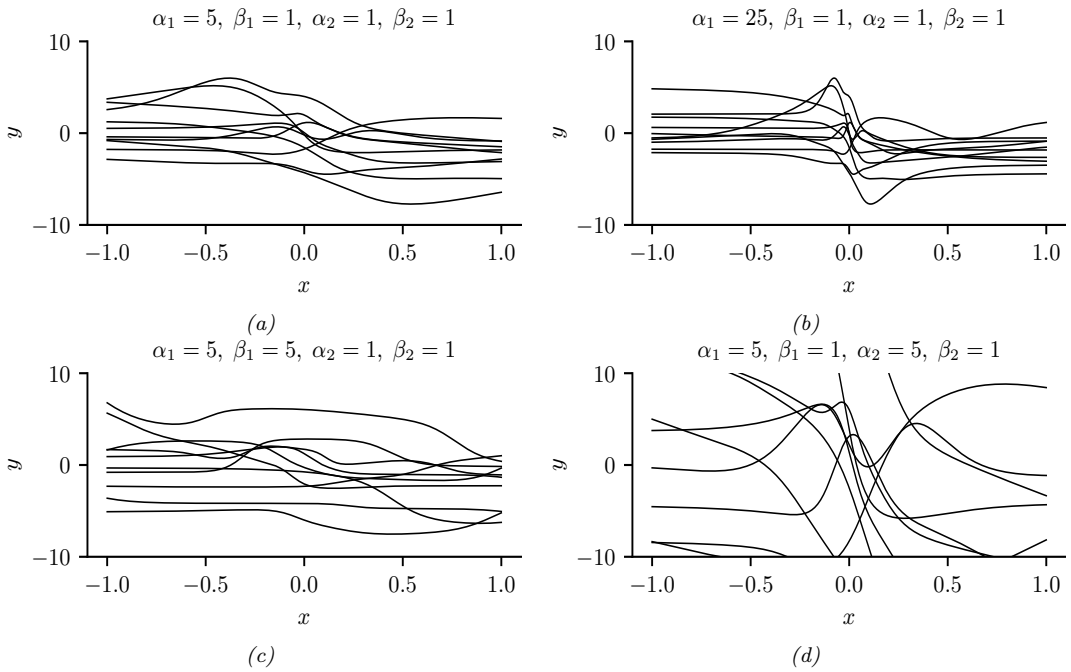


Figure 17.1: The effects of changing the hyperparameters on an MLP with one hidden layer. (a) Random functions sampled from a Gaussian prior with hyperparameters $\alpha_1 = 5$, $\beta_1 = 1$, $\alpha_2 = 1$, $\beta_2 = 1$. (b) Increasing α_1 by factor of 5. (c) Increasing β_1 by factor of 5. (d) Inreasing α_2 by factor of 5. Generated by mlp_priors_demo.ipynb.

shape of the functions steeper. In Figure 17.1(c), we increase β_1; this allows the first layer biases to get bigger, which allows the center of the sigmoid to shift left and right more, away from the origin. In Figure 17.1(d), we increase α_2; this allows the second layer linear weights to get bigger, making the functions more "wiggly" (greater sensitivity to change in the input, and hence larger dynamic range).

The above results are specific to the case of sigmoidal activation functions. ReLU units can behave differently. For example, [WI20, App. E] show that for MLPs with ReLU units, if we set $\beta_\ell = 0$, so the bias terms are all zero, the effect of changing α_ℓ is just to rescale the output. To see this, note that Equation (17.9) simplifies to

$$f(\boldsymbol{x}; \boldsymbol{\alpha}, \boldsymbol{\beta} = \mathbf{0}) = \alpha_L \boldsymbol{\eta}_L(\cdots \varphi(\alpha_1 \boldsymbol{\eta}_1 \boldsymbol{x})) = \alpha_L \cdots \alpha_1 \boldsymbol{\eta}_L(\cdots \varphi(\boldsymbol{\eta}_1 \boldsymbol{x})) \tag{17.10}$$

$$= \alpha_L \cdots \alpha_1 f(\boldsymbol{x}; (\boldsymbol{\alpha} = \mathbf{1}, \boldsymbol{\beta} = \mathbf{0})) \tag{17.11}$$

where we used the fact that for ReLU, $\varphi(\alpha z) = \alpha\varphi(z)$ for any positive α, and $\varphi(\alpha z) = 0$ for any negative α (since the pre-activation $z \geq 0$). In general, it is the ratio of α and β that matters for determining what happens to input signals as they propagate forwards and backwards through a randomly initialized model; for details, see e.g., [Bah+20].

We see that initializing the model's parameters at a particular random value is like sampling a

point from this prior over functions. In the limit of infinitely wide neural networks, we can derive this prior distribution analytically: this is known as a **neural network Gaussian process**, and is explained in Section 18.7.

17.2.2 Sparsity-promoting priors

Although Gaussian priors are simple and widely used, they are not the only option. For some applications, it is useful to use **sparsity promoting priors**, such as the Laplace, which encourage most of the weights (or channels in a CNN) to be zero (c.f., Section 15.2.5). For details, see [Hoe+21].

17.2.3 Learning the prior

We have seen how different priors for the parameters correspond to different priors over functions. We could in principle set the hyperparameters (e.g., the $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ parameters of the Gaussian prior) using grid search to optimize cross-validation loss. However, cross-validation can be slow, particularly if we allow different priors for each layer of the network, as our grid search will grow exponentially with the number of hyperparameters we wish to determine.

An alternative is to use gradient based methods to optimize the marginal likelihood

$$\log p(\mathcal{D}|\boldsymbol{\alpha}, \boldsymbol{\beta}) = \int \log p(\mathcal{D}|\boldsymbol{\theta}) p(\boldsymbol{\theta}|\boldsymbol{\alpha}, \boldsymbol{\beta}) d\boldsymbol{\theta} \tag{17.12}$$

This approach is known as empirical Bayes (Section 3.8) or **evidence maximization**, since $\log p(\mathcal{D}|\boldsymbol{\alpha}, \boldsymbol{\beta})$ is also called the evidence [Mac92a; WS93; Mac99b]. This can give rise to sparse models, as we discussed in the context of automatic relevancy determination (Section 15.2.7). Unfortunately, computing the marginal likelihood is computationally difficult for large neural networks.

Learning the prior is more meaningful if we can do it on a separate, but related dataset. In [SZ+22] they propose to train a model on an initial, large dataset $\mathcal{D}_1$ (possibly unsupervised) to get a point estimate, $\hat{\boldsymbol{\theta}}_1$, from which they can derive an approximate low-rank Gaussian posterior, using the SWAG method (Section 17.3.8). They then use this informative prior when fine-tuning the model on a downstream dataset $\mathcal{D}_2$/ The fine-tuning can either be a MAP estimate $\hat{\boldsymbol{\theta}}_2$ or some approximate posterior, $p(\boldsymbol{\theta}_2|\mathcal{D}_2, \mathcal{D}_1)$, e.g., computed using MCMC (Section 17.3.7). They call this technique "**Bayesian transfer learning**". (See Section 19.5.1 for more details on transfer learning.)

17.2.4 Priors in function space

Typically, the relationship between the prior distribution over parameters and the functions preferred by the prior is not transparent. In some cases, it can be possible to pick more informative priors based on principles such as desired invariances that we want the function to satisfy (see e.g., [Nal18]). [FBW21] introduces *residual pathway priors*, providing a mechanism for encoding high level concepts into prior distributions, such as locality, independencies, and symmetries, without constraining model flexibility. A different approach to encoding interpretable priors over functions leverages kernel methods such as Gaussian processes (e.g., [Sun+19a]), as we discuss in Section 18.1.

17.2.5 Architectural priors

Beyond specifying the parametric prior, it is important to note that the architecture of the model can have an even larger effect on the induced distribution over functions, as argued in Wilson and Izmailov [WI20] and Izmailov et al. [Izm+21b]. For example, a CNN architecture encode prior knowledge about translation equivariance, due to its use of convolution, and hierarchical structure, due to its use of multiple layers. Other forms of inductive bias are induced by different architectures, such as RNNs. (Models such as transformers have weaker inductive bias, but consequently often need more data to perform well.) Thus we can think of the field of **neural architecture search** (reviewed in [EMH19]) as a form of structural prior learning.

In fact, with a suitable architecture, we can often get good results using random (untrained) models. For example, Ulyanov, Vedaldi, and Lempitsky [UVL18] showed that an untrained CNN with random parameters (sampled from a Gaussian) often works very well for low-level image processing tasks, such as image denoising, super-resolution and image inpainting. The resulting prior over functions has been called the **deep image prior**. Similarly, Pinto and Cox [PC12] showed that untrained CNNs with the right structure can do well at face recognition. Moreover, Zhang et al. [Zha+17] show that randomly initialized CNNs can process data to provide features that greatly improve the performance of other models, such as kernel methods.

17.3 Posteriors for BNNs

There are a large number of different approximate inference schemes that have been applied to Bayesian neural networks, with different strengths and limitations. In the sections below, we briefly describe some of these.

17.3.1 Monte Carlo dropout

Monte Carlo dropout (MCD) [GG16; KG17] is a very simple and widely used method for approximating the Bayesian predictive distribution. Usually stochastic dropout layers are added as a form of regularization, and are "turned off" at test time, as described in Section 16.2.6, However, the idea in MCD is to also perform random sampling at test time. More precisely, we drop out each hidden unit by sampling from a Bernoulli(p) distribution; we repeat this procedure S times, to create S distinct models. We then create an equally weighted average of the predictive distributions for each of these models:

$$p(y|\boldsymbol{x}, \mathcal{D}) \approx \frac{1}{S} \sum_{s=1}^{S} p(y|\boldsymbol{x}, \boldsymbol{\theta}^s) \tag{17.13}$$

where $\boldsymbol{\theta}^s$ is a version of the MAP parameter estimate where we randomly drop out some connections.

We give an example of this process in action in Figure 17.2. We see that it succesfully captures uncertainty due to "out of distribution" inputs. (See Section 19.3.2 for more discussion of OOD detection.)

One drawback of MCD is that it is slow at test time. However this can be overcome by "distilling" the model's predictions into a deterministic "student" network, as we discuss in sec:distillation.

A more fundamental problem is that MCD does not give proper uncertainty estimates, as argued in [Osb16; LF+21]. The problem is the following. Although MCD can be viewed as a form of variational

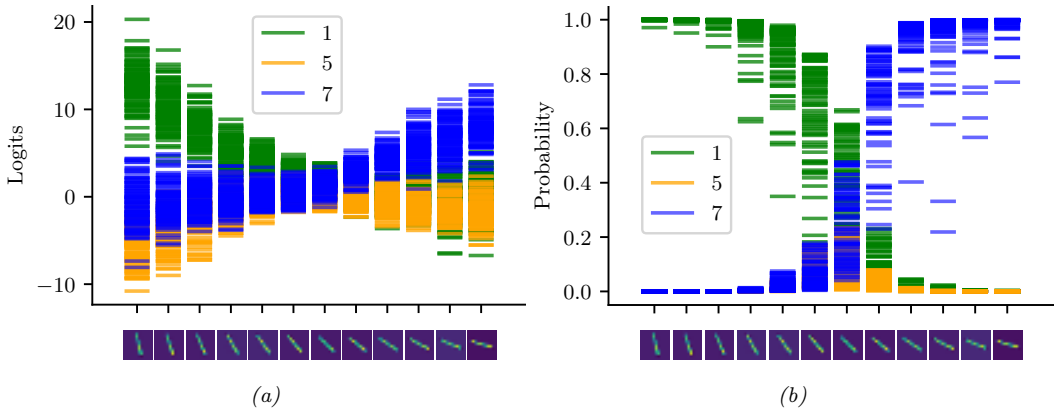


Figure 17.2: Illustration of MC dropout applied to the LeNet architecture. The inputs are some rotated images of the digit 1 from the MNIST dataset. (a) shows softmax inputs (logits) and the (b) shows the softmax outputs (proabilities). We see that the inputs are classified as digit 7 for the last three images (as shown by the probabilities), even though the model has high uncertainty (as shown by the logits). Adapted from Figure 4 of [GG16]. Generated by mnist_classification_mc_dropout.ipynb

inference [GG16], this is only true under a degenerate posterior approximation, corresponding to a mixture of two delta functions, one at 0 (for dropped out nodes) and one at the MLE. This posterior will not converge to the true posterior (which is a delta function at the MLE) even as the training set size goes to infinity, since we are always dropping out hidden nodes with a constant probability p [Osb16]. Fortunately this pathology can be fixed if the noise rate is optimized [GHK17]. For more details, see e.g., [HGMG18; NHLS19; LF+21].

17.3.2 Laplace approximation

In Section 7.4.3, we introduced the Laplace approximation, which computes a Gaussian approximation to the posterior, $p(\boldsymbol{\theta}|\mathcal{D})$, centered at the MAP estimate, $\boldsymbol{\theta}^*$. The posterior prediction matrix is equal to the Hessian of the negative log joint computed at the mode. The benefits of this approach are that it is simple, and it can be used to derive a Bayesian estimate from a pretrained model. A detailed explanation of the method in the context of DNNs can be found in [Dax+21]; here we just give a summary.

Let $\boldsymbol{f}(\boldsymbol{x}_n, \boldsymbol{\theta}) \in \mathbb{R}^C$ be the prediction function with C outputs, and $\boldsymbol{\theta} \in \mathbb{R}^P$ be the parameter vector. Let $\boldsymbol{r}(\boldsymbol{y}; \boldsymbol{f}) = \nabla_{\boldsymbol{f}} \log p(\boldsymbol{y}|\boldsymbol{f})$ be the residual[1], and $\Lambda(\boldsymbol{y}; \boldsymbol{f}) = -\nabla^2_{\boldsymbol{f}} \log p(\boldsymbol{y}|\boldsymbol{f})$ be the per-input noise term. In addition, let $\mathbf{J} \in \mathbb{R}^{C \times P}$ be the Jacobian, $[\mathbf{J}_{\boldsymbol{\theta}}(\boldsymbol{x})]_{ci} = \frac{\partial f_c(\boldsymbol{x}, \boldsymbol{\theta})}{\partial \theta_i}$, and $\mathbf{H} \in \mathbb{R}^{C \times P \times P}$ be the Hessian, $[\mathbf{H}_{\boldsymbol{\theta}}(\boldsymbol{x})]_{cij} = \frac{\partial^2 f_c(\boldsymbol{x}, \boldsymbol{\theta})}{\partial \theta_i \partial \theta_j}$. Then the gradient and Hessian of the log likelihood are given by

1. In the Gaussian case, this term becomes $\nabla_{\boldsymbol{f}} ||\boldsymbol{y} - \boldsymbol{f}||^2 = 2||\boldsymbol{y} - \boldsymbol{f}||$, so can be interpreted as a residual error.

the following [IKB21]:

$$\nabla_{\boldsymbol{\theta}} \log p(\boldsymbol{y}|\boldsymbol{f}(\boldsymbol{x},\boldsymbol{\theta})) = \mathbf{J}_{\boldsymbol{\theta}}(\boldsymbol{x})^{\mathsf{T}} \boldsymbol{r}(\boldsymbol{y};\boldsymbol{f}) \tag{17.14}$$

$$\nabla^2_{\boldsymbol{\theta}} \log p(\boldsymbol{y}|\boldsymbol{f}(\boldsymbol{x},\boldsymbol{\theta})) = \mathbf{H}_{\boldsymbol{\theta}}(\boldsymbol{x})^{\mathsf{T}} \boldsymbol{r}(\boldsymbol{y};\boldsymbol{f}) - \mathbf{J}_{\boldsymbol{\theta}}(\boldsymbol{x})^{\mathsf{T}} \boldsymbol{\Lambda}(\boldsymbol{y};\boldsymbol{f}) \mathbf{J}_{\boldsymbol{\theta}}(\boldsymbol{\theta}) \tag{17.15}$$

Since the network Hessian $\mathbf{H}$ is usually intractable to compute, it is usually dropped, leaving only the Jacobian term. This is called the **generalized Gauss-Newton** or **GGN** approximation [Sch02; Mar20]. The GGN approximation is guaranteed to be positive definite. By contrast, this is not true for the original Hessian in Equation (17.15), since the objective is not convex. Furthermore, computing the Jacobian term is cheaper to compute than the Hessian.

Putting it all together, for a Gaussian prior, $p(\boldsymbol{\theta}) = \mathcal{N}(\boldsymbol{\theta}|\boldsymbol{m}_0, \mathbf{S}_0)$, the Laplace approximation becomes $p(\boldsymbol{\theta}|\mathcal{D}) \approx (\mathcal{N}|\boldsymbol{\theta}^*, \boldsymbol{\Sigma}_{\text{GGN}})$, where

$$\boldsymbol{\Sigma}^{-1}_{\text{GGN}} = \sum_{n=1}^{N} \mathbf{J}_{\boldsymbol{\theta}^*}(\boldsymbol{x}_n)^{\mathsf{T}} \boldsymbol{\Lambda}(\boldsymbol{y}_n;\boldsymbol{f}_n) \mathbf{J}_{\boldsymbol{\theta}^*}(\boldsymbol{x}_n) + \mathbf{S}_0^{-1} \tag{17.16}$$

Unfortunately inverting this matrix takes $O(P^3)$ time, so for models with many parameters, further approximations are usually used. The simplest is to use a diagonal approximation, which takes $O(P)$ time and space. A more sophisticated approach is presented in [RBB18a], which leverages the **KFAC** (Kronecker FActored Curvature) approximation of [MG15]. This approximates the covariance of each layer using a kronecker product.

A limitation of the Laplace approximation is that the posterior covariance is derived from the Hessian evaluated at the MAP parameters. This means Laplace forms a highly *local* approximation: even if the non-Gaussian posterior could be well-described by a Gaussian distribution, the Gaussian distribution *formed using Laplace* only captures the local characteristics of the posterior at the MAP parameters — and may therefore suffer badly from local optima, providing overly compact or diffuse representations. In addition, the curvature information is only used after the model has been estimated, and not during the model optimization process. By contrast, variational inference (Section 17.3.3) can provide more accurate approximations for comparable cost.

17.3.3 Variational inference

In fixed-form variational inference (Section 10.3), we choose a distribution for the posterior approximation $q_{\boldsymbol{\psi}}(\boldsymbol{\theta})$and minimize $D_{\mathbb{KL}}\left(q \parallel p\right)$, with respect to $\boldsymbol{\psi}$. We often choose a Gaussian approximate posterior, $q_{\boldsymbol{\psi}}(\boldsymbol{\theta}) = \mathcal{N}(\boldsymbol{\theta}|\boldsymbol{\mu}, \boldsymbol{\Sigma})$, which lets us use the reparameterization trick to create a low variance estimator of the gradient of the ELBO (see Section 10.3.3). Despite the use of a Gaussian, the parameters that minimize the KL objective are often different what we would find with the Laplace approximation (Section 17.3.2).

Variational methods for neural networks date back to at least Hinton and Camp [HC93]. In deep learning, [Gra11] revisited variational methods, using a Gaussian approximation with a diagonal covariance matrix. This approximates the distribution of every parameter in the model by a univariate Gaussian, where the mean is the point estimate, and the variance captures the uncertainty, as shown in Figure 17.3. This approach was improved further in [Blu+15], who used the reparameterization trick to compute lower variance estimates of the ELBO; they called their method **Bayes by backprop** (**BBB**).

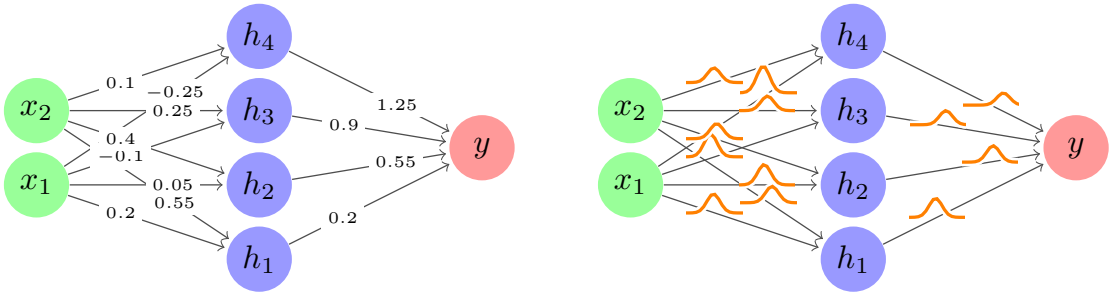


Figure 17.3: Illustration of an MLP with (Left) point estimate for each weight, (Right) a marginal distribution for each weight, corresponding to a fully factored posterior approximation.

In [Blu+15], they used a diagonal Gaussian posterior and vanilla SGD. In [Osa+19a], they used the **variational online Gauss-Newton** (**VOGN**) method of [Kha+18], for improved scalability. VOGN is a noisy version of natural gradient descent, where the extra noise emulates the effect of variational inference. In [Mis+18], they replaced the diagonal approximation with a low-rank plus diagonal approximation, and used VOGN for fitting. In [Tra+20b], they use a rank-one plus diagonal approximation known as **NAGVAC** (see Section 10.3.4.2). In this case, there are only 3 times as many parameters as when computing a point estimate (for the variational mean, variance, and rank-one vector), making the approach very scalable. In addition, in this case it is possible to analytically compute the natural gradient, which speeds up model fitting (see Section 6.4). Many other variational methods have also been proposed (see e.g., [LW16; Zha+18; Wu+19a; HHK19]). See also Section 17.5.4 for a discussion of online VI for DNNs.

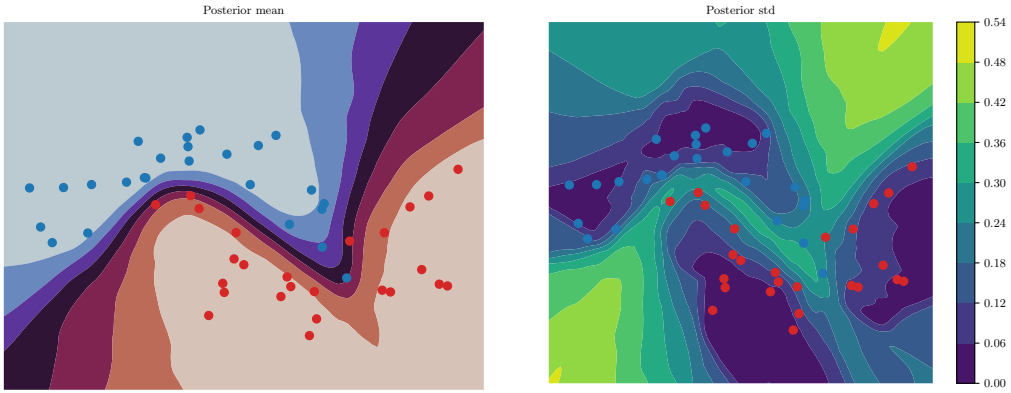
17.3.4 Expectation propagation

Expectation propagation (EP) is similar to variational inference, except it locally optimizes $D_{\mathbb{KL}}\left(p \parallel q\right)$ instead of $D_{\mathbb{KL}}\left(q \parallel p\right)$, where p is the exact posterior and q is the approximate posterior. For details, see Section 10.7.

A special case of EP is the assumed density filtering (ADF) algorithm of Section 8.10, which is equivalent to the first pass of ADF. In Section 8.10.3 we show how to apply ADF to online logistic regression. In [HLA15a], they extend ADF to the case of BNNs; they called their method probabilistic backpropagation or **PBP**. They approximate every parameter in the model by a Gaussian factor, as in Figure 17.3. See Section 17.5.3 for the details.

17.3.5 Last layer methods and SNGP

A very simple approximation is to only "be Bayesian" about the weights in the final layer, and to use MAP estimates for all the other parameters. This is called the **neural-linear** approximation [RTS18]. In [KHH20] they show this can reduce overconfidence in predictions for inputs that are far from the training data. However, this approach ignores uncertainty introduced by the earlier feature extraction layers, where most of the parameters reside. We discuss a solution to this in Section 17.3.6.



(a) (b)

Figure 17.4: Illustration of an MLP fit to the two-moons dataset using HMC. (a) Posterior mean. (b) Posterior standard derivation. The uncertainty increases as we move away from the training data. Generated by bnn_mlp_2d_hmc.ipynb.

17.3.6 SNGP

It is possible to combine DNNs with Gaussian Process (GP) models (Chapter 18), by using the DNN to act as a feature extractor, which is then fed into the kernel in the final layer. This is called "deep kernel learning" (see Section 18.6.6).

One problem with this is that the feature extractor may lose information which is not needed for classification accuracy, but which is needed for robust performance on out-of-distribution inputs (see Section 17.4.6.2). The basic problem is that, in a classification problem, there is no reduction in training accuracy (log likelihood) if points which are far away are projected close together, as long as they are on the correct side of the decision boundary. Thus the distances between two inputs can be erased by the feature extraction layers, so that OOD inputs appear to the final layer to be close to the training set.

One solution to this is to use the **SNGP** (spectrally normalized Gaussian process) method of [Liu+20d]. This constrains the feature extraction layers to be "distance preserving", so that two inputs that are far apart in input space remain far apart after many layers of feature extraction, bu using spectral normalization of the weights to bound the Lipschitz constant of the feature extractor. The overall approach ensures that information that is relevant for computing the confidence of a prediction, but which might be irrelevant to computing the label of a prediction, is not lost. This can help performance in tasks such as out-of-distribution detection (Section 17.4.6.2).

17.3.7 MCMC methods

Some of the earliest work on inference for BNNs was done by Radford Neal, who proposed to use Hamiltonian Monte Carlo (Section 12.5) to approximate the posterior [Nea96]. This is generally considered the gold standard method, since it does not make strong assumptions about the form of the posterior. For more recent work on scaling up HMC for BNNs, see e.g., [Izm+21b; CJ21].

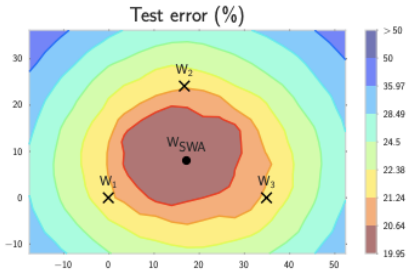


Figure 17.5: Illustration of stochastic weight averaging (SWA). The three crosses represent different SGD solutions. The star in the middle is the average of these parameter values. From Figure 1 of [Izm+18]. Used with kind permission of Andrew Wilson.

We give a simple example of vanilla HMC in Figure 17.4, where we fit a shallow MLP to a small 2d binary dataset. we plot the mean and standard deviation of the posterior predictive distribution, $p(y = 1|\boldsymbol{x}; \mathcal{D})$. We see that the uncertainty is higher as we move away from the training data. (Compare to Bayesian logistic regression in 1d in Figure 15.8a.)

However, a significant limitation of standard MCMC procedures, including HMC, is that they require access to the full training set at each step. Stochastic gradient MCMC methods, such as SGLD, operate instead using mini-batches of data, offering a scalable alternative, as we discuss in Section 12.7.1. For an example of SGLD applied to an MLP, see Section 19.3.3.1.

17.3.8 Methods based on the SGD trajectory

In [MHB17; SL18; CS18], it was shown that, under some assumptions, the iterates produced by stochastic gradient descent (SGD), when run at a fixed learning rate, correspond to samples from a Gaussian approximation to the posterior centered at a local mode, $p(\boldsymbol{\theta}|\mathcal{D}) \approx \mathcal{N}(\boldsymbol{\theta}|\hat{\boldsymbol{\theta}}, \boldsymbol{\Sigma})$. We can therefore use SGD to generate approximate posterior samples. This is similar to SG-MCMC methods, except we do not add explicit gradient noise, and the learning rate is held constant.

In [Izm+18], they noted that these SGD solutions (with fixed learning rate) surround the periphery of points of good generalization, as shown in Figure 17.5. This is in part because SGD does not converge to a local optimum unless the learning rate is annealed to 0. They therefore proposed to compute the average of several SGD samples, each one collected after a certain interval (e.g., one epoch of training), to get $\overline{\boldsymbol{\theta}} = \frac{1}{S}\sum_{s=1}^{S} \boldsymbol{\theta}_s$. They call this **stochastic weight averaging** (**SWA**). They showed that the resulting point tends to correspond to a broader local minimum than the SGD solutions (c.f., Figure 17.9), resulting in better generalization performance.

The SWA approach is related to Polyak-Ruppert averaging, which is often used in convex optimization. The difference is that Polyak-Ruppert typically assumes the learning rate decays to zero, and uses an exponential moving average (EMA) of iterates, rather than an equal average; Polyak-Ruppert averaging is mainly used to reduce variance in the SGD estimate, rather than as a method to find points of better generalization.

The SWA approach is also related to **snapshot ensembles** [Hua+17a], and **fast geometric ensembles** [Gar+18c]; these methods save the parameters $\boldsymbol{\theta}_s$ after increasing and decreasing the

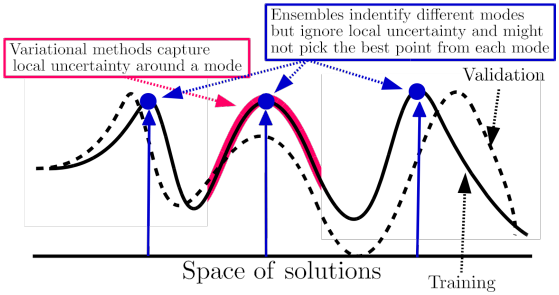


Figure 17.6: Cartoon illustration of the NLL as it varies across the parameter space. Subspace methods (red) model the local neighborhood around a local mode, whereas ensemble methods (blue) approximate the posterior using a set of distinct modes. From Figure 1 of [FHL19]. Used with kind permission of Balaji Lakshminarayanan.

learning rate multiple times in a cyclical fashion, and then computing the *average of the predictions* using $p(\boldsymbol{y}|\boldsymbol{x},\mathcal{D}) \approx \frac{1}{S}\sum_{s=1}^{S} p(\boldsymbol{y}|\boldsymbol{x},\boldsymbol{\theta}_s)$, rather than computing the *average of the parameters* and predicting with a single model (which is faster). Moreover, by finding a flat region, representing a "center or mass" in the posterior, SWA can be seen as approximating the Bayesian model average in Equation 17.1 with a single model.

In [Mad+19], they proposed to fit a Gaussian distribution to the set of samples produced by SGD near a local mode. They use the SWA solution as the mean of the Gaussian. For the covariance matrix, they use a low-rank plus diagonal approximation of the form $p(\boldsymbol{\theta}|\mathcal{D}) = \mathcal{N}(\boldsymbol{\theta}|\overline{\boldsymbol{\theta}}, \boldsymbol{\Sigma})$, where $\boldsymbol{\Sigma} = (\boldsymbol{\Sigma}_{\text{diag}} + \boldsymbol{\Sigma}_{\text{lr}})/2$, $\boldsymbol{\Sigma}_{\text{diag}} = \text{diag}(\overline{\boldsymbol{\theta}^2} - (\overline{\boldsymbol{\theta}})^2)$, $\overline{\boldsymbol{\theta}} = \frac{1}{S}\sum_{s=1}^{S}\boldsymbol{\theta}_s$, $\overline{\boldsymbol{\theta}^2} = \frac{1}{S}\sum_{s=1}^{S}\boldsymbol{\theta}_s^2$, and $\boldsymbol{\Sigma}_{\text{lr}} = \frac{1}{S}\boldsymbol{\Delta}\boldsymbol{\Delta}^{\mathsf{T}}$ is the sample covariance matrix of the last K samples of $\boldsymbol{\Delta}_i = (\boldsymbol{\theta}_i - \overline{\boldsymbol{\theta}}_i)$, where $\overline{\boldsymbol{\theta}}_i$ is the running average of the parameters from the first i samples. They call this method **SWAG**, which stands for "stochastic weight averaging with Gaussian posterior". This can be used to generate an arbitrary number of posterior samples at prediction time. They show that SWAG scales to large residual networks with millions of parameters, and large datasets such as ImageNet, with improved accuracy and calibration over conventional SGD training, and no additional training overhead.

17.3.9 Deep ensembles

Many conventional approximate inference methods focus on approximating the posterior $p(\boldsymbol{\theta}|\mathcal{D})$ in a local neighborhood around one of the posterior modes. While this is often not a major limitation in classical machine learning, modern deep neural networks have highly multi-modal posteriors, with parameters in different modes giving rise to very different functions. On the other hand, the functions in a neighborhood of a single mode may make fairly similar predictions. So using such a local approximation to compute the posterior predictive will underestimate uncertainty and generalize more poorly.

A simple alternative method is to train multiple models, and then to approximate the posterior

using an equally weighted mixture of delta functions,

$$p(\boldsymbol{\theta}|\mathcal{D}) \approx \frac{1}{M}\sum_{m=1}^{M}\delta(\boldsymbol{\theta}-\hat{\boldsymbol{\theta}}_m) \tag{17.17}$$

where M is the number of models, and $\hat{\boldsymbol{\theta}}_m$ is the MAP estimate for model m. See Figure 17.6 for a sketch. This approach is called **deep ensembles** [LPB17; FHL19].

The models can differ in terms of their random seed used for initialization [LPB17], or hyper-parameters [Wen+20c], or architecture [Zai+20], or all of the above. In addition, [DF21; TB22] discusses how to add an explicit repulsive term to ensure functional diversity between the ensemble members. This way, each member corresponds to a distinct prediction function. Combining these is more effective than combining multiple samples from the same basin of attraction, especially in the presence of dataset shift [Ova+19].

17.3.9.1 Multi-SWAG

We can further improve on this approach by fitting a Gaussian to each local mode using the SWAG method from Section 17.3.8 to get a mixture of Gaussians approximation:

$$p(\boldsymbol{\theta}|\mathcal{D}) \approx \frac{1}{M}\sum_{m=1}^{M}\mathcal{N}(\boldsymbol{\theta}|\hat{\boldsymbol{\theta}}_m, \boldsymbol{\Sigma}_m) \tag{17.18}$$

This approach is known as **MultiSWAG** [WI20]. MultiSWAG performs a Bayesian model average both across multiple basins of attraction, like deep ensembles, but also within each basin, and provides an easy way to generate an arbitrary number of posterior samples, $S > M$, in an any-time fashion.

17.3.9.2 Deep ensembles with random priors

The standard way to fit each member of a deep ensemble is to initialize them each with a different random set of parameters, but them to train them all on the same data. Unfortunately this can result in the predictions from each ensemble member being rather similar, which reduces the benefit of the approach. One way to increase diversity is to train each member on a different subset of the data; this is called **bootstrap sampling**. Another approach is to define the i'th ensemble member $g_i(\boldsymbol{x})$ to be the addition of a trainable model $t_i(\boldsymbol{x})$ and a fixed, but random, **prior network**, $p_i(\boldsymbol{x})$, to get

$$g_i(\boldsymbol{x};\boldsymbol{\theta}_i) = t_i(\boldsymbol{x};\boldsymbol{\theta}_i) + \beta p_i(\boldsymbol{x}) \tag{17.19}$$

where $\beta \geq 0$ controls the amount of data-independent variation between the members. The trainable network learns to model the residual error between the true output and the value predicted by the prior. This is called a **random prior deep ensemble** [OAC18]. See Figure 17.7 for an illustration.

17.3.9.3 Deep ensembles as approximate Bayesian inference

The posterior predictive distribution for a Bayesian neural network cannot be expressed in closed form. Therefore all Bayesian inference approaches in deep learning are approximate. In this context,

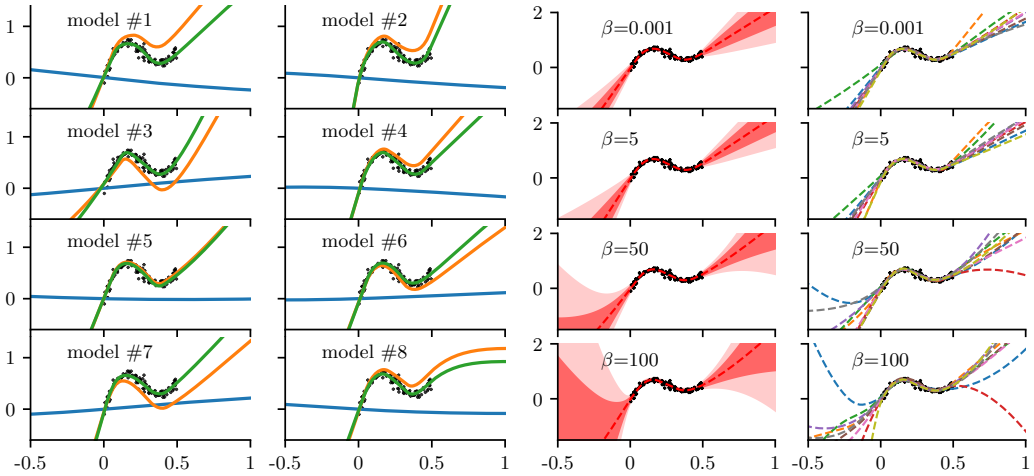


Figure 17.7: Deep ensemble with random priors. (a) Individual predictions from each member. Blue is the fixed random prior function, orange is the trainable function, green is the combination of the two. (b) Overall prediction from the ensemble, for increasingly large values of β. On the left we show (in red) the posterior mean and pointwise standard deviation, and on the right we show samples from the posterior. As β increases, we trust the random priors more, and pay less attention to the data, thus getting a more diffuse posterior. Generated by randomized_priors.ipynb.

all approximate inference procedures fall onto a spectrum, representing how closely they approximate the true posterior predictive distribution. Deep ensembles can provide better approximations to a Bayesian model average than a single basin marginalization approach, because point masses from different basins of attraction represent greater functional diversity than standard Bayesian approaches which sample within a single basin.

17.3.9.4 Deep ensembles vs classical ensembles

Note that deep ensembles is slightly different to classical ensemble methods (see e.g., [Die00]), such as bagging and random forests, which obtain diversity of its predictors by training them on different subsets of the data (created using bootstrap resampling), or on different features. This data perturbation is necessary to get diversity when the base learner is a convex problem (such as a linear model, or shallow decision tree). In the deep ensemble approach, every model is trained on the same data, and the same input features. The diversity arises due to different starting parameters, different random seeds, and SGD noise, which induces different solutions due to the nonconvex loss. It is also possible to explicitly enforce diversity of the ensemble members, which can provably improve performance [TB22].

17.3.9.5 Deep ensembles vs mixtures of experts and stacking

If we use weighted combinations of the models, $p(\boldsymbol{\theta}|\mathcal{D}) = \sum_{m=1}^{M} p(m|\mathcal{D})p(\boldsymbol{\theta}|m,\mathcal{D})$, where $p(m|\mathcal{D})$ is the marginal likelihood of model m, then, in the large sample limit, this mixture will concentrate on

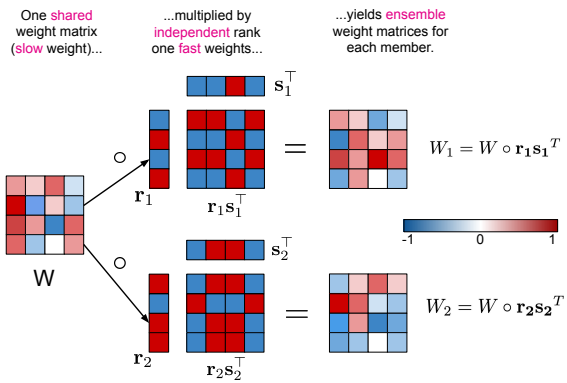


Figure 17.8: Illustration of batch ensemble with 2 ensemble members. From Figure 2 of [WTB20]. Used with kind permission of Paul Vicol.

the MAP model, so only one component will be selected. By contrast, in deep ensembles, we always use M equally weighted models. Thus we see that Bayes model averaging is not the same as model ensembling [Min00b]. Indeed, ensembling can enlarge the expressive power of the posterior predictive distribution compared to BMA [OCM21].

We can also make the mixing weights be conditional on the inputs:

$$p(y|\boldsymbol{x}, \mathcal{D}) = \sum_m w_m(\boldsymbol{x}) p(y|\boldsymbol{x}, \boldsymbol{\theta}_m) \tag{17.20}$$

If we constrain the weights to be non-zero and sum to one, this is called a **mixture of experts**. However, if we allow a general positive weighted combination, the approach is called **stacking** [Wol92; Bre96; Yao+18a; CAII20]. In stacking, the weights $w_m(\boldsymbol{x})$ are usually estimated on hold-out data, to make the method more robust to model misspecification.

17.3.9.6 Batch ensemble

Deep ensembles require M times more memory and time than a single model. One way to reduce the memory cost is to share most of the parameters — which we call **slow weights**, $\mathbf{W}$ — and then let each ensemble member m estimate its own local perturbation, which we will call **fast weights**, $\mathbf{F}_m$. We then define $\mathbf{W}_m = \mathbf{W} \odot \mathbf{F}_m$. For efficiency, we can define $\mathbf{F}_m$ to be a rank-one matrix, $\mathbf{F}_m = \boldsymbol{s}_m \boldsymbol{r}_m^\mathsf{T}$, as illustrated in Figure 17.8. This is called **batch ensemble** [WTB20].

It is clear that the memory overhead is very small compared to naive ensembles, since we just need to store $2M$ vectors ($\boldsymbol{s}_m^l$ and $\boldsymbol{r}_m^l$) for every layer, which is negligible compared to the quadratic cost of storing the shared weight matrix $\mathbf{W}^l$.

In addition to memory savings, batch ensemble can reduce the inference time by a constant factor by leveraging within-device parallelism. To see this, consider the output of one layer using ensemble m on example n:

$$y_n^m = \varphi\left(\mathbf{W}_m^\mathsf{T} \boldsymbol{x}_n\right) = \varphi\left((\mathbf{W} \odot \boldsymbol{s}_m \boldsymbol{r}_m^\mathsf{T})^\mathsf{T} \boldsymbol{x}_n\right) = \varphi\left((\mathbf{W}^\mathsf{T}(\boldsymbol{x}_n \odot \boldsymbol{s}_m) \odot \boldsymbol{r}_m\right) \tag{17.21}$$

We can vectorize this for a minibatch of inputs $\mathbf{X}$ by replicating $\boldsymbol{r}_m$ and $\boldsymbol{s}_m$ along the B rows in the batch to form matrices, giving

$$\mathbf{Y}_m = \varphi\left(((\mathbf{X} \odot \mathbf{S}_m)\mathbf{W}) \odot \mathbf{R}_m\right) \tag{17.22}$$

This applies the same ensemble parameters m to every example in the minibatch of size B. To achieve diversity during training, we can divide the minibatch into M sub-batches, and use sub-batch m to train $\mathbf{W}_m$. (Note that this reduces the batch size for training each ensemble to B/M.) At test time, when we want to average over M models, we can replicate each input M times, leading to a batch size of BM.

In [WTB20], they show that this method outperforms MC dropout at negligible extra memory cost. However, the best combination was to combine batch ensemble with MC dropout; in some cases, this approached the performance of naive ensembles.

17.3.10 Approximating the posterior predictive distibution

Once we have approximated the parameter posterior, $q(\boldsymbol{\theta}) \approx p(\boldsymbol{\theta}|\mathcal{D})$, we can use it to approximate the posterior predictive distribution:

$$p(\boldsymbol{y}|\boldsymbol{x}, \mathcal{D}) = \int q(\boldsymbol{\theta}) p(\boldsymbol{y}|\boldsymbol{x}, \boldsymbol{\theta}) d\boldsymbol{\theta} \tag{17.23}$$

We usually approximate this integral using Monte Carlo:

$$p(\boldsymbol{y}|\boldsymbol{x}, \mathcal{D}) \approx \frac{1}{S} \sum_{s=1}^{S} p(\boldsymbol{y}|\boldsymbol{f}(\boldsymbol{x}, \boldsymbol{\theta}^s)) \tag{17.24}$$

where $\boldsymbol{\theta}^s \sim q(\boldsymbol{\theta})$. We discuss some extensions of this approach below.

17.3.10.1 A linearized approximation

In [IKB21] they point out that samples from an approximate posterior, $q(\boldsymbol{\theta})$, can result in bad predictions when plugged into the model if the posterior puts probability density "in the wrong places". This is because $\boldsymbol{f}(\boldsymbol{x}; \boldsymbol{\theta})$ is a highly nonlinear function of $\boldsymbol{\theta}$ that might behave quite differently when $\boldsymbol{\theta}$ is far from the MAP estimate on which $q(\boldsymbol{\theta})$ is centered. To avoid this problem, they propose to replace $\boldsymbol{f}(\boldsymbol{x}; \boldsymbol{\theta})$ with a linear approximation centered at the MAP estimate $\boldsymbol{\theta}^*$:

$$\boldsymbol{f}_{\text{lin}}^{\boldsymbol{\theta}^*}(\boldsymbol{x}, \boldsymbol{\theta}) = \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{\theta}^*) + \mathbf{J}_{\boldsymbol{\theta}^*}(\boldsymbol{\theta} - \boldsymbol{\theta}^*) \tag{17.25}$$

Such a model is well behaved around $\boldsymbol{\theta}^*$, and so the approximation

$$p(\boldsymbol{y}|\boldsymbol{x}, \mathcal{D}) \approx \frac{1}{S} \sum_{s=1}^{S} p(\boldsymbol{y}|\boldsymbol{f}_{\text{lin}}^{\boldsymbol{\theta}^*}(\boldsymbol{x}, \boldsymbol{\theta}^s)) \tag{17.26}$$

often works better than Equation (17.24).

Note that $\boldsymbol{f}_{\text{lin}}^{\boldsymbol{\theta}^*}(\boldsymbol{x}, \boldsymbol{\theta})$ is a linear function of the parameters $\boldsymbol{\theta}$, but a nonlinear function of the inputs $\boldsymbol{x}$. Thus $p(\boldsymbol{y}|\boldsymbol{f}_{\text{lin}}^{\boldsymbol{\theta}^*}(\boldsymbol{x}, \boldsymbol{\theta}))$ is a generalized linear model (Section 15.1), so [IKB21] call this approximation the **GLM predictive distribution**.

17.3.10.2 Distillation

The MC approximation to the posterior predictive is S times slower than a standard, deterministic plug-in approximation. One way to speed this up is to use **distillation** to approximate the semi-parametric "teacher" model p_t from Equation (17.24) by a parametric "student" model p_s by minimizing $\mathbb{E}\left[D_{\mathbb{KL}}\left(p_t(\boldsymbol{y}|\boldsymbol{x}) \parallel p_s(\boldsymbol{y}|\boldsymbol{x})\right)\right]$ wrt p_s. This approach was first proposed in [HVD14], who called the technique "**dark knowledge**", because the teacher has "hidden" information in its predictive probabilities (logits) than is not apparent in the raw one-hot labels.

In [Kor+15], this idea was used to distill the predictions from a teacher whose parameter posterior was computed using HMC; this is called "**Bayesian dark knowledge**". A similar idea was used in [BPK16; GBP18], who distilled the predictive distribution derived from MC dropout (Section 17.3.1).

Since the parametric student is typically less flexible than the semi-parametric teacher, it may be overconfident, and lack diversity in its predictions. To avoid this overconfidence, it is safer to make the student be a mixture distribution c.f., [SG05]. See also [Tra+20a].

17.3.11 Tempered and cold posteriors

When working with BNNs for classification problems, the likelihood is usually taken to be

$$p(y|\boldsymbol{x}, \boldsymbol{\theta}) = \text{Cat}(y|\text{softmax}(f(\boldsymbol{x}; \boldsymbol{\theta}))) \tag{17.27}$$

where $f(\boldsymbol{x}; \boldsymbol{\theta}) \in \mathbb{R}^C$ returns the logits over the C class labels. This is the same as in multinomial logistic regression (Section 15.3.2); the only difference is that f is a nonlinear function of $\boldsymbol{\theta}$.

However, in practice, it is often found (see e.g., [Zha+18; Wen+20b; LST21; Noc+21]) that BNNs give better predictive accuracy if the likelihood function is scaled by some power α. That is, instead of targeting the posterior $p(\boldsymbol{\theta}|\mathcal{D}) \propto p(\boldsymbol{y}|\boldsymbol{x}, \boldsymbol{\theta})p(\boldsymbol{\theta})$, these methods target the **tempered posterior**, $p_{\text{tempered}}(\boldsymbol{\theta}|\mathcal{D}) \propto p(\boldsymbol{y}|\mathbf{X}, \boldsymbol{\theta})^\alpha p(\boldsymbol{\theta})$. In log space, we have

$$\log p_{\text{tempered}}(\boldsymbol{\theta}|\mathcal{D}) = \alpha \log p(\boldsymbol{y}|\mathbf{X}, \boldsymbol{\theta}) + \log p(\boldsymbol{\theta}) + \text{const} \tag{17.28}$$

This is also called an **α-posterior** or **power posterior** [Med+21].

Another common method is to target the **cold posterior**, $p_{\text{cold}}(\boldsymbol{\theta}|\mathcal{D}) \propto p(\boldsymbol{\theta}|\mathbf{X}, \boldsymbol{y})^{1/T}$, or, in log space,

$$\log p_{\text{cold}}(\boldsymbol{\theta}|\mathcal{D}) = \frac{1}{T} \log p(\boldsymbol{y}|\mathbf{X}, \boldsymbol{\theta}) + \frac{1}{T} \log p(\boldsymbol{\theta}) + \text{const} \tag{17.29}$$

If $T < 1$, we say that the posterior is "cold". Note that, in the case of a Gaussian prior, using the cold prior is the same as using the tempered prior with a different hyperparameter, since

$$\frac{1}{T} \log \mathcal{N}(\boldsymbol{\theta}|0, \sigma^2_{\text{cold}}\mathbf{I}) = -\frac{1}{2T\sigma^2_{\text{cold}}} \sum_i \theta_i^2 + \text{const} = \mathcal{N}(\boldsymbol{\theta}|0, \sigma^2_{\text{tempered}}\mathbf{I}) + \text{const} \tag{17.30}$$

where $\sigma^2_{\text{tempered}} = T\sigma^2_{\text{cold}}$. Thus both methods are effectively the same, and just reweight the likelihood.

Cold posteriors in Bayesian neural network classifiers are a consequence of underrepresenting aleatoric (label) uncertainty, as shown by [Kap+22]. On benchmarks such as CIFAR-100, we should have essentially no uncertainty about the labels of the training images, yet Bayesian classifiers with

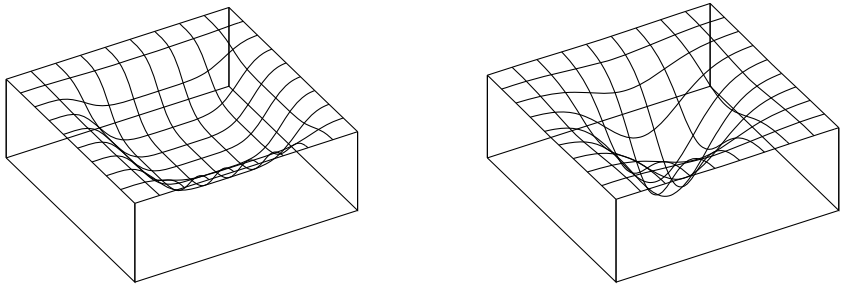


Figure 17.9: Flat vs sharp minima. From Figures 1 and 2 of [HS97]. Used with kind permission of Jürgen Schmidhuber.

softmax likelihoods have very high uncertainty for these points. Moreover, [Izm+21b] showed that the cold posterior effect in all the examples of [Wen+20b] when data augmentation is removed. [Kap+22] show that with the SGLD inference in [Wen+20b], data augmentation has the effect of raising the likelihood to a power $1/K$ for minibatches of size K. Cold posteriors exactly counteract this effect, more honestly representing our beliefs about aleatoric uncertainty, by sharpening the likelihood. However, tempering is not required, and [Kap+22] show that by using a Dirichlet observation model to explicitly represent (lack of) label noise, there is no cold posterior effect, even with data augmentation. The curation hypotheses of [Ait21] can be considered a special case of the above explanation, where curation has the effect of increasing our confidence about training labels.

In Section 14.1.3, we discuss generalized variational inference, which gives a general framework for understanding whether and how the likelihood or prior could benefit from tempering. Tempering is particularly useful if (as is usually the case) the model is misspecified [KJD21].

17.4 Generalization in Bayesian deep learning

In this section, we discuss why "being Bayesian" can improve predictive accuracy and generalization performance.

17.4.1 Sharp vs flat minima

Some optimization methods (in particular, second-order batch methods) are able to find "needles in haystacks", corresponding to narrow but deep "holes" in the loss landscape, corresponding to parameter settings with very low loss. These are known as **sharp minima**, see Figure 17.9(right). From the point of view of minimizing the empirical loss, the optimizer has done a good job. However, such solutions generally correspond to a model that has overfit the data. It is better to find points that correspond to **flat minima**, as shown in Figure 17.9(left); such solutions are more robust and generalize better. To see why, note that flat minima correspond to regions in parameter space where there is a lot of posterior uncertainty, and hence samples from this region are less able to precisely memorize irrelevant details about the training set [AS17]. Put another way, the description length for sharp minima is large, meaning you need to use many bits of precision to specify the exact location in parameter space to avoid incurring large loss, whereas the description length for flat minima is

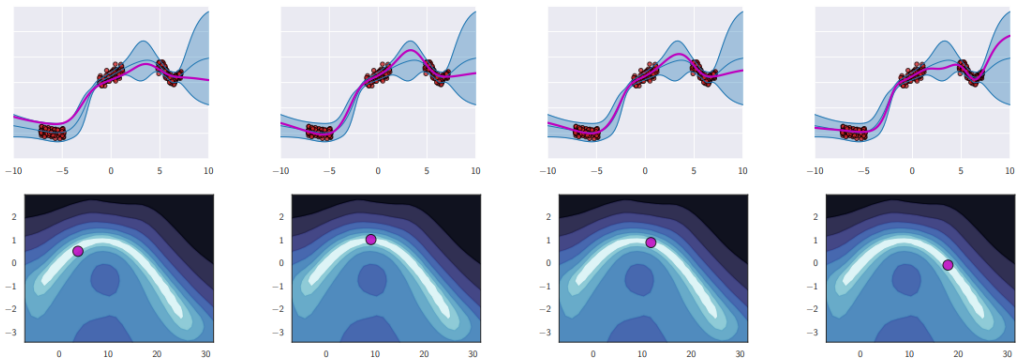


Figure 17.10: Diversity of high performing functions sampled from the posterior. Top row: we show predictions on the 1d input domain for 4 different functions. We see that they extrapolate in different ways outside of the support of the data. Bottom row: we show a 2d subspace spanning two distinct modes (MAP estimates), and connected by a low-loss curved path computed as in [Gar+18c]. From Figure 8 of [WI20]. Used with kind permission of Andrew Wilson.

less, resulting in better generalization [Mac03].

SGD often finds such flat minima by virtue of the addition of noise, which prevents it from "entering" narrow regions of the loss landscape (see Section 12.5.7). In addition, in higher dimensional spaces, flat regions occupy a much greater volume, and are thus much more easily discoverable by optimization procedures. More precisely, the analysis in [SL18] shows that the probability of entering any given basin of attraction $\mathcal{A}$ around a minimum is given by $p_{SGD}(\boldsymbol{\theta} \in \mathcal{A}) \propto \int_{\mathcal{A}} e^{-\mathcal{L}(\boldsymbol{\theta})} d\boldsymbol{\theta}$. Note that this is integrating over the volume of space corresponding to $\mathcal{A}$, and hence is proportional to the model evidence (marginal likelihood) for that region, as explained in Section 3.9.1. Since the evidence is parameterization invariant (since we marginalize out the parameters), this means that SGD will avoid regions that have low evidence (corresponding to sharp minima) regardless of how we parameterize the model (contrary to the claims in [Din+17]).

In fact, several papers have shown that we can view SGD as approximately sampling from the Bayesian posterior (see Section 17.3.8). The SWA method (Section 17.3.8) can be seen as finding a center of mass in the posterior based on these SGD samples, finding solutions that generalize better than picking a single SGD point.

If we must use a single solution, a flat one will help us better approximate the Bayesian model average in the integral of Equation (17.1). However, by attempting to perform a more complete Bayesian model average, we will select for flatness without having to deal with the messiness of having to worry about flatness definitions, or the effects of reparametrization, or unknown implicit regularization, as the model average will automatically weight regions with the greatest volume.

17.4.2 Mode connectivity and the loss landscape

In DNNs there are often many low-loss solutions, which provide complementary explanations of the data. Moreover, in [Gar+18c] they showed that two independently trained SGD solutions can

be connected by a curve in a subspace, along which the training loss remains near-zero, known as **mode connectivity**. Despite having the same training loss, these different parameter settings give rise to very different functions, as illustrated in Figure 17.10, where we show predictions on a 1d regression problem coming from different points in parameter space obtained by interpolating along a mode connecting curve between two distinct MAP estimates. Using a Bayesian model average, we can combine these functions together to provide much better performance over a single flat solution [Izm+19].

Recently, it has been discovered [Ben+21b] that there are in fact large multidimensional simplexes of low loss solutions, which can be combined together for significantly improved performance. These results further motivate the Bayesian approach (Equation (17.1)), where we perform a posterior weighted model average.

17.4.3 Effective dimensionality of a model

Modern DNNs have millions of parameters, but these parameters are often not well-determined by the data, i.e., there can be a lot of posterior uncertainty. By averaging over the posterior, we reduce the chance of overfitting, because we do not use "degrees of freedom" that are not needed or warranted.

To quantify the number of degrees of freedom, or **effective dimensionality** [Mac92b], we follow [MBW20] and define

$$N_{\text{eff}}(\mathbf{H}, c) = \sum_{i=1}^{k} \frac{\lambda_i}{\lambda_i + c}, \tag{17.31}$$

where λ_i are the eigenvalues of the Hessian matrix $\mathbf{H}$ computed at a local mode, and $c > 0$ is a regularization parameter. Intuitively, the effective dimension counts the number of well-determined parameters. A "flat minimum" will have many directions in parameter space that are not well-determined, and hence will have low effective dimensionality. This means that we can perform Bayesian inference in a low dimensional subspace [Izm+19]: Since there is functional homogeneity in all directions but those defining the effective dimension, neural networks can be significantly compressed.

This compression perspective can also be used to understand why the effective dimension can be a good proxy for generalization. If two models have similar training loss, but one has lower effective dimension, then it is providing a better compression for the data at the same fidelity. In Figure 17.11 we show that for CNNs with low training loss (above the green partition), the effective dimensionality closely tracks generalization performance. We also see that the number of parameters alone is not a strong determinant of generalization. Indeed, models with more parameters can have a lower number of effective parameters. We also see that wide but shallow models overfit, while depth helps provide lower effective dimensionality, leading to a better compression of the data. It is depth that makes modern neural networks distinctive, providing hierarchical inductive biases making it possible to discover more regularity in the data.

17.4.4 The hypothesis space of DNNs

Zhang et al. [Zha+17] showed that CNNs can fit CIFAR-10 images with random labels with zero training error, but can still generalize well on the noise-free test set. It has been claimed that this

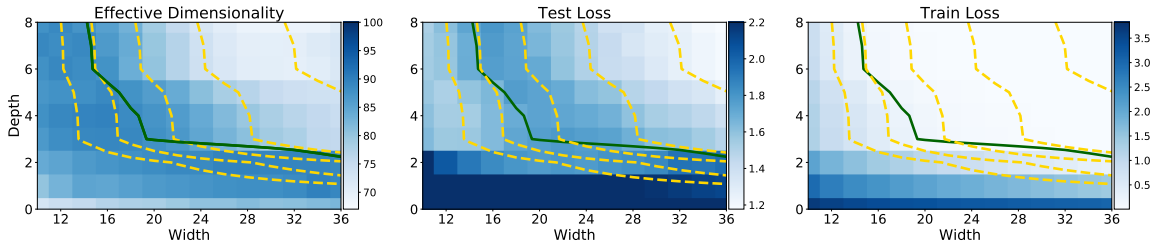


Figure 17.11: Left: Effective dimensionality as a function of model width and depth for a CNN on CIFAR-100. Center: Test loss as a function of model width and depth. Right: Train loss as a function of model width and depth. Yellow level curves represent equal parameter counts (1e5, 2e5, 4e5, 1.6e6). The green curve separates models with near-zero training loss. Effective dimensionality serves as a good proxy for generalization for models with low train loss. We see wide but shallow models overfit, providing low train loss, but high test loss and high effective dimensionality. For models with the same train loss, lower effective dimensionality can be viewed as a better compression of the data at the same fidelity. Thus depth provides a mechanism for compression, which leads to better generalization. From Figure 2 of [MBW20]. Used with kind permission of Andrew Wilson.

result contradicts a classical understanding of generalization, because it shows that neural networks are capable of significantly overfitting the data, but can still generalize well on structured inputs.

We can resolve this paradox by taking a Bayesian perspective. In particular, we know that modern CNNs are very flexible, so they can fit almost pattern (since they are in fact universal approximators). However, their architecture encodes a prior over what kinds of patterns they expect to see in the data (see Section 17.2.5). Image datasets with random labels *can* be represented by this function class, but such solutions receive very low marginal likelihood, since they strongly violate the prior assumptions [WI20]. By contrast, image datasets where the output labels are consistent with patterns in the input get much higher marginal likelihood.

This phenomenon is not unique to DNNs. For example, it also occurs with Gaussian processes (Chapter 18). Such models are also universal approximators, but they allocate most of their probability mass to a small range of solutions (depending on the chosen kernel). They can also fit image datasets with random labels, but such data receives a low marginal likelihood [WI20].

In general, we can distinguish the support of a model, i.e., the set of functions it can represent, from the distribution over that support, i.e., the inductive bias which leads it to prefer some functions over others. We would like to use models where the support is large, so we can capture the complexity of real-world data, but also where the inductive bias places probability mass on the kinds of functions we expect to see. If we succeed at this, the posterior will quickly converge on the true function after seeing a small amount of data. This idea is sketched in Figure 17.12.

17.4.5 PAC-Bayes

PAC-Bayes [McA99; LC02; Gue19; Alq21; GSZ21] provides a promising mechanism to derive non-vacuous generalization bounds for large *stochastic networks* [Ney+17; NBS18; DR17], with parameters sampled from a probability distribution. In particular, the difference between the train

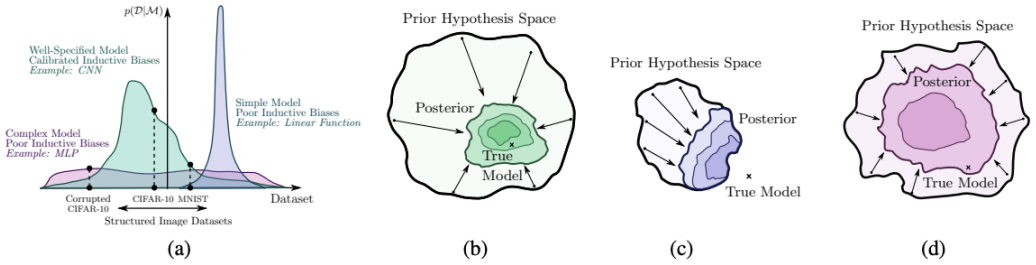


Figure 17.12: Illustration of the behavior of different kinds of model families and the prior distribution they induce over datasets. (a) The purple model is a simple linear model that has small support, and can only represent a few kinds of datasets. The pink model is an unstructured MLP: this has support over a large range of datasets with a fairly uninformative (broad) prior. Finally the green model is a CNN; this has support over a large range of datasets but the prior is more concentrated on certain kinds of datasets that have compositional structure. (b) The posterior for the green model (CNN) rapidly collapses to the true model, since it is consistent with the data. (c) The posterior for the purple model (linear) also rapidly collapses, but to a solution which cannot represent the true model. (d) The posterior for the pink model (MLP) collapses very slowly (as a function of dataset size). From Figure 2 of [WI20]. Used with kind permission of Andrew Wilson.

error and the generalization error can be expressed as

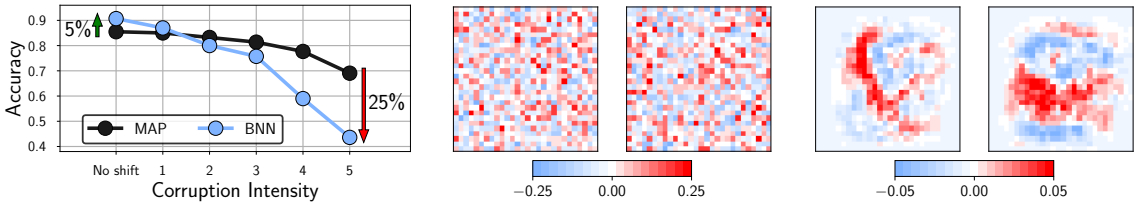
$$\sqrt{\frac{D_{\mathbb{KL}}\left(Q \parallel P\right) + c}{2(N-1)}}, \tag{17.32}$$

where c is a constant and N is the number of training points. P is the prior distribution over the parameters and Q is an arbitrary distribution, which can be chosen to optimize the bound.

The perspective in this chapter is largely complementary, and in some ways orthogonal, to the PAC-Bayes literature. Our focus has been on Bayesian marginalization, particularly multi-modal marginalization, and a prescriptive approach to model construction. In contrast, PAC-Bayes bounds are about bounding the empirical risk of a single sample, rather than marginalization, and are not currently prescriptive: what we would do to improve the bounds, such as reducing the number of model parameters, or using highly compact priors, does not typically improve generalization. Moreover, while we have seen Bayesian model averaging over multimodal posteriors has a significant effect on generalization, it has a minimal logarithmic effect on PAC-Bayes bounds. In general, because the bounds are lose, albeit non-vacuous in some cases, there is often room to make modeling choices that improve PAC-Bayes bounds without improving generalization, making it hard to derive a prescription for model construction from the bounds.

17.4.6 Out-of-Distribution generalization for BNNs

Bayesian methods are often assumed to be more robust in the context of distribution shift (discussed in Chapter 19), because they capture more uncertainty than methods based on point estimation. However, there are some subtleties, some of which we discuss below.



(a) ResNet-20, CIFAR-10-C (b) BNN weights (c) MAP weights

Figure 17.13: ***Bayesian neural networks under covariate shift.*** ***(a):*** *Performance of a ResNet-20 on the pixelate corruption in CIFAR-10-C. For the highest degree of corruption, a Bayesian model average underperforms a MAP solution by 25% (44% against 69%) accuracy. See Izmailov et al. [Izm+21b] for details.* ***(b):*** *Visualization of the weights in the first layer of a Bayesian fully-connected network on MNIST sampled via HMC.* ***(c):*** *The corresponding MAP weights. We visualize the weights connecting the input pixels to a neuron in the hidden layer as a* 28×28 *image, where each weight is shown in the location of the input pixel it interacts with. This is Figure 1 of Izmailov et al. [Izm+21a].*

17.4.6.1 BMA can give poor results with default priors

Many approximate inference methods, especially deep ensembles, are significantly less overconfident (more well calibrated) in the presence of some kinds of covariate shifts [Ova+19]. However, in [Izm+21b], it was noted that HMC, which arguably offers the most accurate approximation to the posterior, often works poorly under distribution shift.

Rather than an idiosyncracy of HMC, Izmailov et al. [Izm+21a] show this lack of robustness is a foundational issue of Bayesian model averaging under covariate shift, caused by degeneracies in the training data, and a poor choice of prior. As an illustrative special case, MNIST digits all have black cornel pixels. Weights in the first layer of a neural network connected to these pixels are multiplied by zero, and thus can take any value without affecting the outputs of the network. Classical MAP training or deep ensembles of MAP solutions with a Gaussian prior will therefore drive these parameters to zero, since they don't help with the data fit, and the resulting network will be robust to corruptions on these pixels. On the other hand, the posterior for these parameters will be the same as the prior, and so a Bayesian model average will multiply corruptions by random numbers sampled from the prior, leading to degraded predictive performance.

Figure 17.13(b, c) visualizes this example, showing the first-layer weights of a fully-connected network for the MAP solution and a BNN posterior sample, on MNIST. The MAP weights corresponding to zero intensity pixels near the boundary are near zero, while the BNN weights look noisy, sampled from a Gaussian prior.

Izmailov et al. [Izm+21a] prove that this issue is a special case of a much more general problem, whenever there are linear dependencies in the input features of the training data, both for fully-connected and convolutional networks. In this case, the data live on a hyperplane. If a covariate or domain shift, moves orthogonal to this hyperplane, the posterior will be the same as the prior in the direction of the shift. The posterior model average will thus be highly vulnerable to shifts that do not particularly affect the underlying semantic structure of the problem (such as corruptions), whereas the MAP solution will be entirely robust to such shifts.

By introducing a prior over parameters which is aligned with the principal components of the

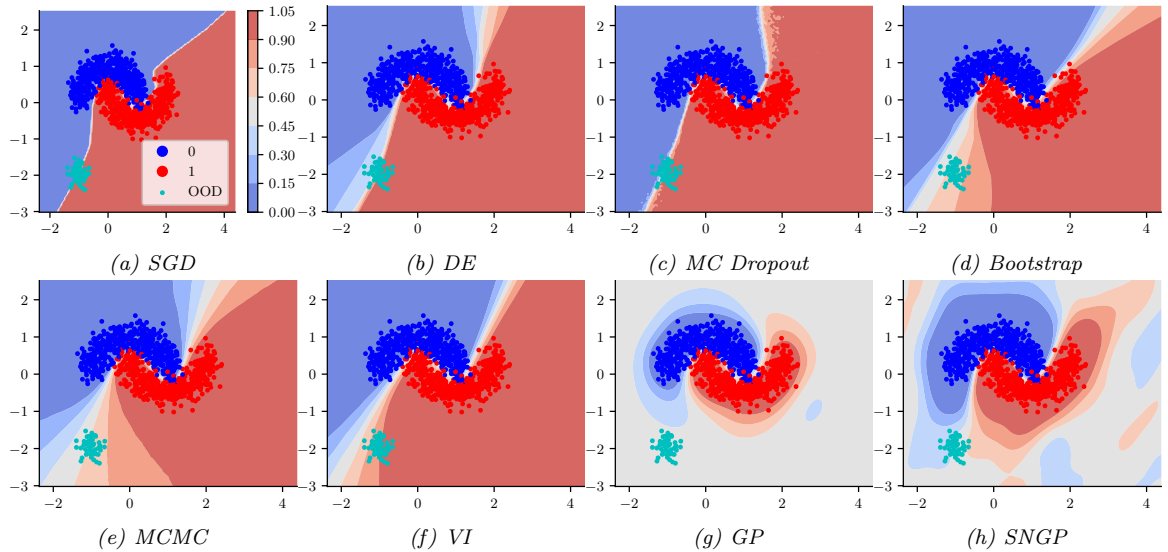


Figure 17.14: Predictions made by various (B)NNs when presented with the training data shown in blue and red. The green blob is an example of some OOD inputs. Methods are: (a) Standard SGD; (b) Deep Ensemble of 10 models with different random initializations; (c) MC Dropout with 50 samples; (d) Bootstrap training, where each of the 10 models is initialized identically but given different versions of the data, obtained by resampling with replacement; (e) MCMC using NUTS algorithm with 3000 warmup steps and 3000 samples; (f) Variational inference; (g) Gaussian process classifier using RBF kernel; (h) SNGP. The model is an MLP with 8,16,16,8 units in the hidden layers and ReLu activation. The output layer has 1 neuron with sigmoid activation. Generated by makemoons_comparison.ipynb

training inputs, we can substantially improve the generalization accuracy of Bayesian neural networks in out-of-distribution settings. Izmailov et al. [Izm+21a] propose the following *EmpCov* prior: $p(w^1) = \mathcal{N}(0, \alpha\Sigma + \epsilon I)$, where w^1 are the first layer weights, $\Sigma = \frac{1}{n-1}\sum_{i=1}^{n} x_i x_i^T$ is the empirical covariance of the training input features x_i, $\alpha > 0$ determines the scale of the prior, and ϵ is a small positive constant to ensure the covariance matrix is positive definite. With this improved prior they are able to obtain a method that is much more robust to distribution shift.

17.4.6.2 BNNs can be overconfident on OOD inputs

An important problem in practice is how a predictive model will behave when it is given an input that is "out of distribution" or OOD. Ideally we would like the model to express that it is not confident in its prediction, so that the system can abstain from predicting (see Section 19.3.3). Using "exact" inference methods, such as MCMC, for BNNs can give this behavior in some cases. For example, in Section 19.3.3.1 we showed that an MLP which was fit to MNIST using SGLD would be less overconfident than a point estimate (computed using SGD) when presented with inputs from fashion MNIST. However, this behavior does not always occur reliably.

To illustrate the problem, consider the 2d nonlinear binary classification dataset shown in Figure 17.14. In addition to the two training classes, we have highlighted (in green) a set of OOD inputs

that are far from the support of the training set. Intuitively we would expect the model to predict a probability of 0.5 (corresponding to "don't know") for such inputs that are far from the training set. However we see that the only methods that do so are the Gaussian Process (GP) classifier (see Section 18.4) and the SNGP model (Section 17.3.6), which contains a GP layer on top of the feature extractor.

The lesson we learn from this simple example is that "being Bayesian" only helps if we are using a good hypothesis class. If we only consider a single MLP classifier, with standard Gaussian priors on the weights, it is extremely unlikely that we will learn the kind of compact decision boundary shown in Figure 17.14g, because that function has negligible support under our prior (c.f. Section 17.4.4). Instead we should embrace the power of Bayes to avoid overfitting and use as complex a model class as we can afford.

17.4.7 Model Selection for BNNs

Historically, the marginal likelihood (aka Bayesian evidence) has been used for selecting between trained neural architectures and learning hyperparameters [Mac92a]. It has recently seen a resurgence for this purpose, with scalable Laplace approximations [Imm+21; Dax+21]. Moreover, the variational lower bound to the marginal likelihood, the ELBO, is often used in VAEs to train millions of parameters in the decoder network [KW14].

However, [Lot+22] show many ways in which the question the marginal likelihood answers, "what is the likelihood of generating the training data from my prior?" different from the question we care about when comparing the generalization of trained models, "what is the probability that the posterior could generate withheld points from the data distribution?". For example, eventually the posterior predictive will become insensitive to increases in prior variance, because the prior becomes weak, but the marginal likelihood will become increasingly negative — leaving room for the marginal likelihood to prefer models that have significantly worse generalization. They show how the marginal likelihood can be trivially modified to be more aligned with generalization for model selection, and hyperparameter learning. These questions apply to model selection in general, but particularly neural networks, which are often underspecified by the data, and require approximate inference.

17.5 Online inference

In Section 17.3, we have focused on batch or offline inference. However, an important application of Bayesian inference is in sequential settings, where the data arrives in a continuous stream, and the model has to "keep up". This is called **sequential Bayesian inference**, and is one approach to **online learning** (see Section 19.7.5). In this section, we discuss some algorithmic approaches to this problem in the context of DNNs. These methods are widely used for continual learning, which we discuss Section 19.7.

17.5.1 Sequential Laplace for DNNs

In [RBB18b], they extended the Laplace method of Section 17.3.2 to the sequential setting. Specifically, let $p(\boldsymbol{\theta}|\mathcal{D}_{1:t-1}) \approx \mathcal{N}(\boldsymbol{\theta}|\boldsymbol{\mu}_{t-1}, \boldsymbol{\Lambda}_{t-1}^{-1})$ be the approximate posterior from the previous step; we assume the precision matrix is Kronecker factored. We now compute the new mean by solving the MAP

problem

$$\boldsymbol{\mu}_t = \text{argmax} \log p(\mathcal{D}_t|\boldsymbol{\theta}) + \log p(\boldsymbol{\theta}|\mathcal{D}_{t-1}) \tag{17.33}$$

$$= \text{argmax} \log p(\mathcal{D}_t|\boldsymbol{\theta}) - \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\mu}_{t-1})\boldsymbol{\Lambda}_{t-1}^{-1}(\boldsymbol{\theta} - \boldsymbol{\mu}_{t-1}) \tag{17.34}$$

Once we have computed $\boldsymbol{\mu}_t$, we compute the approximate Hessian at this point, and get the new posterior precision

$$\boldsymbol{\Lambda}_t = \lambda \mathbf{H}(\boldsymbol{\mu}_t) + \boldsymbol{\Lambda}_{t-1} \tag{17.35}$$

where $\lambda \geq 0$ is a weighting factor that trades off how much the model pays attention to the new data vs old data.

Now suppose we use a diagonal approximation to the posterior prediction matrix. From Equation (17.34), we see that this amounts to adding a quadratic penalty to each new MAP estimate, to encourage it to remain close to the parameters from previous tasks. This approach is called **elastic weight consolidation** (**EWC**) [Kir+17].

17.5.2 Extended Kalman Filtering for DNNs

In Section 29.7.2, we showed how Kalman filtering can be used to incrementally compute the exact posterior for the weights of a linear regression model with known variance, i.e., we compute $p(\boldsymbol{\theta}|\mathcal{D}_{1:t}, \sigma^2)$, where $\mathcal{D}_{1:t} = \{(\boldsymbol{u}_i, y_i) : i = 1 : t\}$ is the data seen so far, and

$$p(y_t|\boldsymbol{u}_t, \boldsymbol{\theta}, \sigma^2) = \mathcal{N}(y_t|\boldsymbol{\theta}^\mathsf{T}\boldsymbol{u}_t, \sigma^2) \tag{17.36}$$

is the linear regression likelihood. The application of KF to this model is known as recursive least squares.

Now consider the case of nonlinear regression:

$$p(y_t|\boldsymbol{u}_t, \boldsymbol{\theta}, \sigma^2) = \mathcal{N}(y_t|f(\boldsymbol{\theta}, \boldsymbol{u}_t), \sigma^2) \tag{17.37}$$

where $f(\boldsymbol{\theta}, \boldsymbol{u}_t)$ is some nonlinear function, such as an MLP. We can use the extended Kalman filter (Section 8.5.2) to approximately compute $p(\boldsymbol{\theta}_t|\mathcal{D}_{1:t}, \sigma^2)$, where $\boldsymbol{\theta}_t$ is the hidden state (see e.g., [SW89; PF03]). To see this, note that we can set the dynamics model to the identity function, $f(\boldsymbol{\theta}_t) = \boldsymbol{\theta}_t$, so the parameters are propagated through unchanged, and the observation model to the input-dependent function $f(\boldsymbol{\theta}_t) = f(\boldsymbol{\theta}_t, \boldsymbol{u}_t)$. We set the observation noise to $\mathbf{R}_t = \sigma^2$, and the dynamics noise to $\mathbf{Q}_t = q\mathbf{I}$, where q is a small constant, to allow the parameters to slowly drift according to artificial **process noise**. (In practice it can be useful to anneal q from a large initial value to something near 0.)

17.5.2.1 Example

We now give an example of this process in action. We sample a synthetic dataset from the true function

$$h^*(u) = x - 10\cos(u)\sin(u) + u^3 \tag{17.38}$$

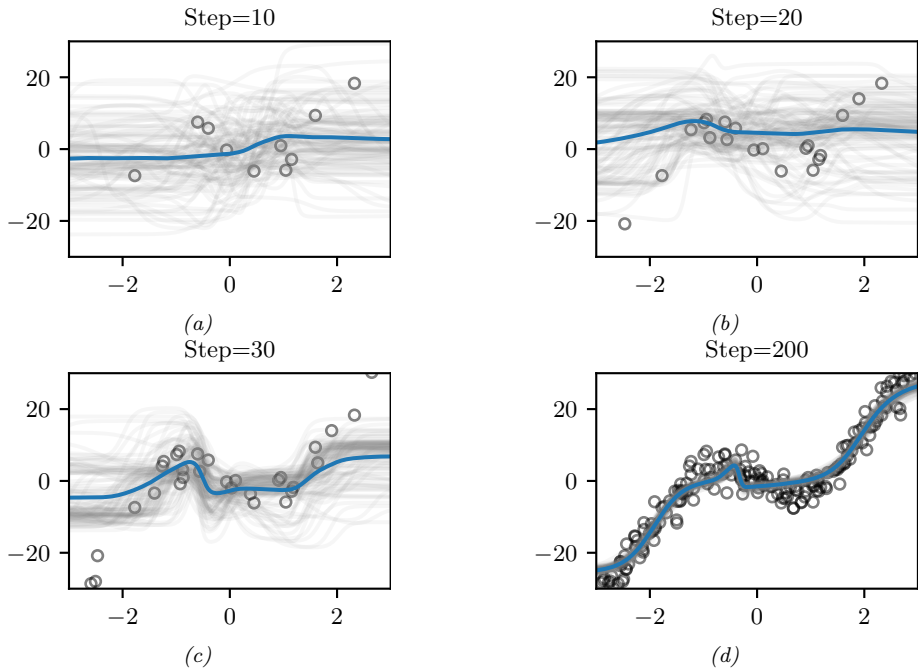


Figure 17.15: Sequential Bayesian inference for the parameters of an MLP using the extended Kalman filter. We show results after seeing the first 10, 20, 30 and 200 observations. (For a video of this, see `https://bit.ly/3wXnWaM`*.) Generated by* `ekf_mlp.ipynb`*.*

and add Gaussian noise with $\sigma = 3$. We then fit this with an MLP with one hidden layer with H hidden units, so the model has the form

$$f(\boldsymbol{\theta}, \boldsymbol{u}) = \mathbf{W}_2 \tanh(\mathbf{W}_1 \boldsymbol{u} + \boldsymbol{b}_1) + \boldsymbol{b}_2 \tag{17.39}$$

where $\mathbf{W}_1 \in \mathbb{R}^{H \times 1}$, $\boldsymbol{b}_1 \in \mathbb{R}^H$, $\mathbf{W}_2 \in \mathbb{R}^{1 \times H}$, $\boldsymbol{b}_2 \in \mathbb{R}^1$. We set $H = 6$, so there are $D = 19$ parameters in total.

Given the data, we sequentially compute the posterior, starting from a vague Gaussian prior, $p(\boldsymbol{\theta}) = \mathcal{N}(\boldsymbol{\theta}|\mathbf{0}, \boldsymbol{\Sigma}_0)$, where $\boldsymbol{\Sigma}_0 = 100\mathbf{I}$. (In practice we cannot start from the prior mean, which is $\boldsymbol{\theta}_0 = \mathbf{0}$, since linearizing the model around this point results in a zero gradient, so we use an initial random sample for $\boldsymbol{\theta}_0$.) The results are shown in Figure 17.15. We can see that the model adapts to the data, without having to specify any learning rate. In addition, we see that the predictions become gradually more confident, as the posterior concentrates on the MLE.

17.5.2.2 Setting the variance terms

In the above example, we set the variance terms by hand. In general we need to estimate the noise variance σ, which determines $\mathbf{R}_t$ and hence the learning rate, as well as the strength of the prior $\boldsymbol{\Sigma}_0$, which controls the amount of regularization. Some methods for doing this are discussed in [FNG00].

17.5.2.3 Reducing the computational complexity

The naive EKF method described above takes $O(N_z^3)$ time, which is prohibitive for large neural networks. A simple approximation, known as the **decoupled EKF**, was proposed in [PF91; SPD92] (see [PF03] for a review). This partitions the weights into G groups or blocks, and estimates the relevant matrices for each group g independently. If $G = 1$, this reduces the standard global EKF. If we put each weight into its own group, we get a fully diagonal approximation. In practice this does not work any better than SGD, since it ignores correlations between the parameters. A useful compromise is to put all the weights corresponding to each neuron into its own group; this is called "node decoupled EKF".

Another approach to increasing computational efficiency is to leverage the fact that the effective dimensionality of a DNN is often quite low (see Section 17.4.3). Indeed we can approximate the model parameters by using a low dimensional vector of coefficients that specify the point in a linear manifold corresponding to weight space; the basis set defining this linear manifold can either be chosen randomly [Li+18b; Lar+21], or can be estimated using PCA applied to the SGD iterates [Izm+19]. We can exploit this observation to perform EKF in this low-dimensional subspace, which significantly speeds up inference, as discussed in [DMKM22].

17.5.3 Assumed Density Filtering for DNNs

In Section 8.10.3, we discussed how to use assumed density filtering (ADF) to perform online (binary) logistic regression. In this section, we generalize this to nonlinear predictive models, such as DNNs. The key is to perform Gaussian moment matching of the hidden activations at each layer of the model. This provides an alternative to the EKF approach in Section 17.5.2, which is based on linearization of the network.

We will assume the following likelihood:

$$p(\boldsymbol{y}_t|\boldsymbol{u}_t, \boldsymbol{w}_t) = \text{Expfam}(\boldsymbol{y}_t|\ell^{-1}(f(\boldsymbol{u}_t; \boldsymbol{w}_t))) \tag{17.40}$$

where $f(\boldsymbol{x}; \boldsymbol{w})$ is the DNN, ℓ^{-1} is the inverse link function, and Expfam() is some exponential family distribution. For example, if f is linear and we are solving a binary classification problem, we can write

$$p(y_t|\boldsymbol{u}_t, \boldsymbol{w}_t) = \text{Ber}(y_t|\sigma(\boldsymbol{u}_t^\mathsf{T}\boldsymbol{w}_t)) \tag{17.41}$$

We discussed using ADF to fit this model in Section 8.10.3.

In [HLA15b], they propose **Probabilistic backpropagation** (**PBP**), which is an instance of ADF applied to MLPs. The basic idea is to approximate the posterior over the weights in each layer using a fully factorized distribution

$$p(\boldsymbol{w}_t|\mathcal{D}_{1:t}) \approx p_t(\boldsymbol{w}_t) = \prod_{l=1}^{L}\prod_{i=1}^{D_l}\prod_{j=1}^{D_{l-1}+1} \mathcal{N}(w_{ijl}|\mu_{ijl}^t, \tau_{ijl}^t) \tag{17.42}$$

where L is the number of layers, and D_l is the number of neurons in layer l. (The **expectation backpropagation** algorithm of [SHM14] is a special case of this, where the variances are fixed to $\tau = 1$.)

Suppose the parameters are static, so $\boldsymbol{w}_t = \boldsymbol{w}_{t-1}$. Then the new posterior, after conditioning on the t'th observation, is given by

$$\hat{p}_t(\boldsymbol{w}) = \frac{1}{Z_t} p(\boldsymbol{y}_t|\boldsymbol{u}_t, \boldsymbol{w}) \mathcal{N}(\boldsymbol{w}|\boldsymbol{\mu}^{t-1}, \boldsymbol{\Sigma}^{t-1}) \tag{17.43}$$

where $\boldsymbol{\Sigma}^{t-1} = \text{diag}(\boldsymbol{\tau}^{t-1})$. We then project $\hat{p}_t(\boldsymbol{w})$ instead the space of factored Gaussians to compute the new (approximate) posterior, $p_t(\boldsymbol{w})$. This can be done by computing the following means and variances [Min01a]:

$$\mu_{ijl}^t = \mu_{ijl}^{t-1} + \tau_{ijl}^{t-1} \frac{\partial \ln Z_t}{\partial \mu_{ijl}^{t-1}} \tag{17.44}$$

$$\tau_{ijl}^t = \tau_{ijl}^{t-1} - (\tau_{ijl}^{t-1})^2 \left[\left(\frac{\partial \ln Z_t}{\partial \mu_{ijl}^{t-1}} \right)^2 - 2 \frac{\partial \ln Z_t}{\partial \tau_{ijl}^{t-1}} \right] \tag{17.45}$$

In the forwards pass, we compute Z_t by propagating the input $\boldsymbol{u}_t$ through the model. Since we have a Gaussian distribution over the weights, instead of a point estimate, this induces an (approximately) Gaussian distribution over the values of the hidden units. For certain kinds of activation functions (such as ReLU), the relevant integrals (to compute the means and variances) can be solved analytically, as in GP-neural networks (Section 18.7). The result is that we get a Gaussian distribution over the final layer of the form $\mathcal{N}(\boldsymbol{\eta}_t|\boldsymbol{\mu}, \boldsymbol{\Sigma})$, where $\boldsymbol{\eta}_t = f(\boldsymbol{u}_t; \boldsymbol{w}_t)$ is the output of the neural network before the GLM link function induced by $p_t(\boldsymbol{w}_t)$. Hence we can approximate the partition function using

$$Z_t \approx \int p(\boldsymbol{y}_t|\boldsymbol{\eta}_t) \mathcal{N}(\boldsymbol{\eta}_t|\boldsymbol{\mu}, \boldsymbol{\Sigma}) d\boldsymbol{\eta}_t \tag{17.46}$$

We now discuss how to compute this integral. In the case of probit classification, with $y \in \{-1, +1\}$, we have $p(y|\boldsymbol{x}, \boldsymbol{w}) = \Phi(y\eta)$, where Φ is the cdf of the standard normal. We can then use the following analytical result

$$\int \Phi(y\eta) \mathcal{N}(h|\mu, \sigma) d\eta = \Phi\left(\frac{y\mu}{\sqrt{1+\sigma}} \right) \tag{17.47}$$

In the case of logistic classification, with $y \in \{0, 1\}$, we have $p(y|\boldsymbol{x}, \boldsymbol{w}) = \text{Ber}(y|\sigma(\eta))$; in this case, we can use the probit approximation from Section 15.3.5. For the multiclass case, where $\boldsymbol{y} \in \{0, 1\}^C$ (one-hot encoding), we have $p(\boldsymbol{y}|\boldsymbol{x}, \boldsymbol{w}) = \text{Cat}(\boldsymbol{y}|\text{softmax}(\boldsymbol{\eta}))$. A variational lower bound to $\log Z_t$ for this case is given in [GDFY16].

Once we have computed Z_t, we can take gradients and update the Gaussian posterior moments, before moving to the next step.

17.5.4 Online variational inference for DNNs

A natural approach to online learning is to use variational inference, where the prior is the posterior from the previous step. This is known as **streaming variational Bayes** [Bro+13]. In more detail,

at step t, we compute

$$\boldsymbol{\psi}_t = \underset{\boldsymbol{\psi}}{\operatorname{argmin}} \underbrace{\mathbb{E}_{q(\boldsymbol{\theta}|\boldsymbol{\psi})}\left[\ell_t(\boldsymbol{\theta})\right] + D_{\mathbb{KL}}\left(q(\boldsymbol{\theta}|\boldsymbol{\psi}) \parallel q(\boldsymbol{\theta}|\boldsymbol{\psi}_{t-1})\right)}_{-Ł_t(\boldsymbol{\psi})} \tag{17.48}$$

$$= \underset{\boldsymbol{\psi}}{\operatorname{argmin}}\, \mathbb{E}_{q(\boldsymbol{\theta}|\boldsymbol{\psi})}\left[\ell_t(\boldsymbol{\theta}) + \log q(\boldsymbol{\theta}|\boldsymbol{\psi}) - \log q(\boldsymbol{\theta}|\boldsymbol{\psi}_{t-1})\right] \tag{17.49}$$

where $\ell_t(\boldsymbol{\theta}) = -\log p(\mathcal{D}_t|\boldsymbol{\theta})$ is the negative log likelihood (or, more generally, some loss function) of the data batch at step t.

When applied to DNNs, this approach is called **variational continual learning** or **VCL** [Ngu+18]. (We discuss continual learning in Section 19.7.) An efficient implementation of this, known as **FOO-VB** ("Fixed-point Operator for Online Variational Bayes") is given in [Zen+21].

One problem with the VCL objective in Equation (17.48) is that the KL term can cause the model to become too sparse, which can prevent the model from adapting or learning new tasks. This problem is called **variational overpruning** [TT17]. More precisely, the reason this happens as is as follows: some weights might not be needed to fit a given dataset, so their posterior will be equal to the prior; but sampling from these high-variance weights will add noise to the likeilhood; to reduce this, the optimization method will prefer to set the bias term to a large negative value, so the corresponding unit is "turned off", and thus has no effect on the likelihood. Unfortunately, these "dead units" become stuck, so there is not enough network capacity to learn the next task.

In [LST21], they propose a solution to this, known as **generalized variational continual learning** or GVCL. The first step is to downweight the KL term by a factor $\beta < 1$ to get

$$Ł_t = \mathbb{E}_{q(\boldsymbol{\theta}|\boldsymbol{\psi})}\left[\ell_t(\boldsymbol{\theta})\right] + \beta D_{\mathbb{KL}}\left(q(\boldsymbol{\theta}|\boldsymbol{\psi}) \parallel q(\boldsymbol{\theta}|\boldsymbol{\psi}_{t-1})\right) \tag{17.50}$$

Interestingly, one can show that in the limit of $\beta \to 0$, this recovers several standard methods that use a Laplace approximation based on the Hessian. In particular if we use a diagonal variational posterior, this reduces to online EWC method of [Sch+18]; if we use a block-diagonal and Kronecker factored posterior, this reduces to the online structured Laplace method of [RBB18b]; and if we use a low-rank posterior precision matrix, this reduces to the SOLA method of [Yin+20].

The second step is to replace the prior and posterior by using tempering, which is useful when the model is misspecified, as discussed in Section 17.3.11. In the case of Gaussians, raising the distribution to the power λ is equivalent to tempering with a temperature of $\tau = 1/\lambda$, which is the same as scaling the covariance by λ^{-1}. Thus the GVCL objective becomes

$$Ł_t = \mathbb{E}_{q(\boldsymbol{\theta}|\boldsymbol{\psi})}\left[\ell_t(\boldsymbol{\theta})\right] + \beta D_{\mathbb{KL}}\left(q(\boldsymbol{\theta}|\boldsymbol{\psi})^{\lambda} \parallel q(\boldsymbol{\theta}|\boldsymbol{\psi}_{t-1})^{\lambda}\right) \tag{17.51}$$

This can be optimized using SGD, assuming the posterior is reparameterizable (see Section 10.3.3).

17.6 Hierarchical Bayesian neural networks

In some problems, we have multiple related datasets, such as a set of medical images from different hospitals. Some aspects of the data (e.g., the shape of healthy vs diseased cells) is generally the same across datasets, but other aspects may be unique or idiosyncractic (e.g., each hospital may use a different colored die for staining). To model this, we can use a hierarchical Bayesian model, in which we allow the parameters for each dataset to be different (to capture random effects), while coming from

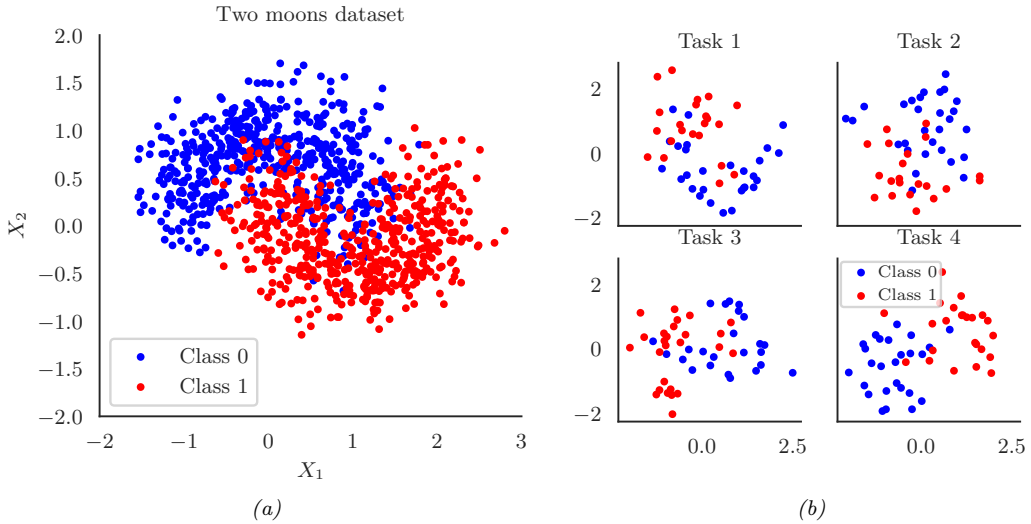


Figure 17.16: (a) Two moons synthetic dataset. (b) Multi-task version, where we rotate the data to create 18 related tasks (groups). Each dataset has 50 training and 50 test points. Here we show the first 4 tasks. Generated by bnn_hierarchical.ipynb.

a common prior (to capture shared effects). This is the setup we considered in Section 15.5, where we discuss hierarchical Bayesian GLMs. In this section, we extend this to nonlinear predictors based on neural networks. (The setup is very similar to domain generalization, discussed in Section 19.6.2, except here we care about performance on all the domains, not just a held-out target domain.)

17.6.1 Example: multi-moons classification

In this section, we consider an example[2] where we want to solve multiple related nonlinear binary classification problems coming from J different environments or distributions. We assume that each environment has its own unique decision boundary $p(y|\boldsymbol{x}, \boldsymbol{w}^j)$, so this is a form of concept shift (see Section 19.2.3). However we assume the overall shape of each boundary is similar to a common shared boundary, denote $p(y|\boldsymbol{x}, \boldsymbol{w}^0)$. We only have a small number N_j of examples from each environment, $\mathcal{D}^j = \{(\boldsymbol{x}_n^j, y_n^j) : n = 1 : N_j\}$, but we can utilise their common structure to do better than fitting J separate models.

To illustrate this, we create some synthetic 2d data for the $J = 18$ tasks. We start with the two-moons dataset, illustrated in Figure 17.16a. Each task is obtained by rotating the 2d inputs by a different amount, to create 18 related classification problems (see Figure 17.16b). See Figure 17.16b for the training data for 4 tasks.

To handle the nonlinear decision boundary, we use a multilayer perceptron. Since the dataset is low-dimensional (2d input), we use a shallow model with just 2 hidden layers, each with 5 neurons. We could fit a separate MLP to each task, but since we have limited data per task ($N_j = 50$ examples

2. This example is from https://twiecki.io/blog/2018/08/13/hierarchical_bayesian_neural_network/. For a real-world example of a similar approach applied to a gesture recognition task, see [Jos+17].

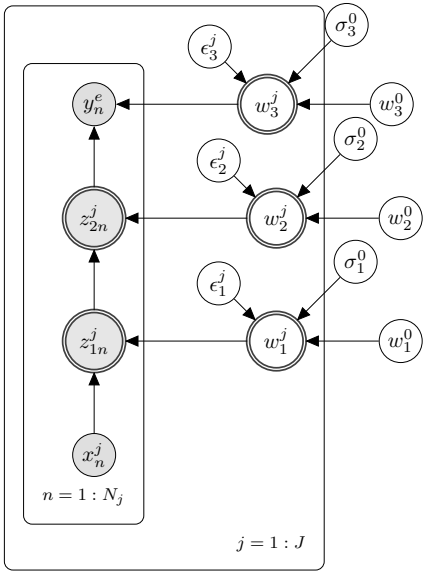


Figure 17.17: Illustration of a hierarchical Bayesian MLP with 2 hidden layers. There are J different models, each with N_j observed samples, and a common set of global shared parent parameters denoted with the 0 superscript. Nodes which are shaded are observed. Nodes with double ringed circles are deterministic functions of their parents.

for training), this works poorly, as we show below. We could also pool all the data and fit a single model, but this does even worse, since the datasets come from different underlying distributions, so mixing the data together from different "concepts" confuses the model. Instead we adopt a hierarchical Bayesian approach.

Our modeling assumptions are shown in Figure 17.17. In particular, we assume the weight from unit i to unit k in layer l for environment j, denoted $w_{i,k,l}^j$, comes from a common prior value $w_{i,k,l}^0$, with a random offset. We use the non-centered parameterization from Section 12.6.5 to write

$$w_{i,k,l}^j = w_{i,k,l}^0 + \epsilon_{i,k,l}^j \times \sigma_l^0 \tag{17.52}$$

where $\epsilon_{i,k,l}^j \sim \mathcal{N}(0,1)$. By allowing a different σ_l^0 per layer l, we let the model control the degree of shrinkage to the prior for each layer separately. (We could also make the σ_l^j parameters be environment specific, which would allow for different amounts of distribution shift from the common parent.) For the hyper-parameters, we put $\mathcal{N}(0,1)$ priors on $w_{i,k,l}^0$, and $\mathcal{N}_+(1)$ priors on σ_l^0.

We compute the posterior $p(\boldsymbol{\epsilon}_{1:L}^{1:J}, \boldsymbol{w}_{1:L}^0, \boldsymbol{\sigma}_{1:L}^0 | \mathcal{D})$ using HMC (Section 12.5). We then evaluate this model using a fresh set of labeled samples from each environment. The average classification accuracy on the train and test sets for the non-hierarchical model (one MLP per environment, fit separately) is 86% and 83%. For the hierarchical model, this improves to 91% and 89% respectively.

To see why the hierarchical model works better, we will plot the posterior predictive distribution in 2d. Figure 17.18 shows the results for the non-hierarchical models; we see that the method fails to

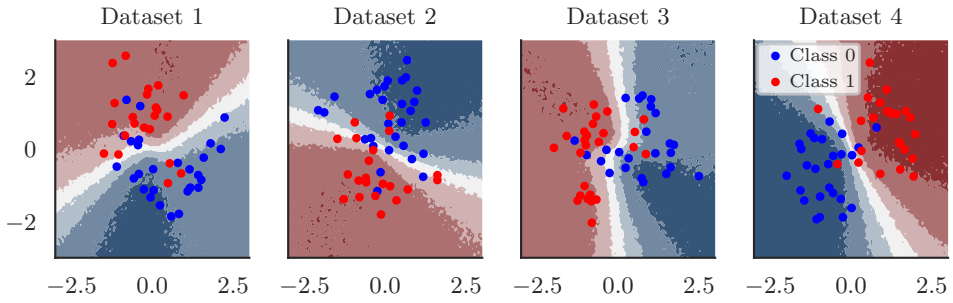


Figure 17.18: Results of fitting separate MLPs on each dataset. Generated by bnn_hierarchical.ipynb.

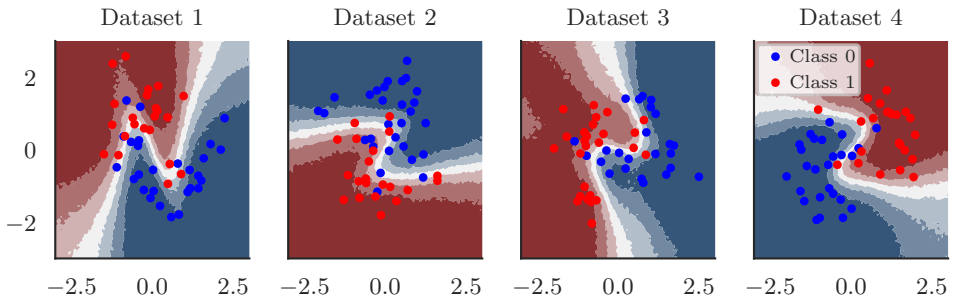


Figure 17.19: Results of fitting hierarchical MLP on all datasets jointly. Generated by bnn_hierarchical.ipynb.

learn the common underlying Z-shaped decision boundary. By contrast, Figure 17.19 shows that the hierarchical method has correctly recovered the common pattern, while still allowing group variation.

18 Gaussian processes

This chapter is co-authored with Andrew Wilson.

18.1 Introduction

Deep neural networks are a family of flexible function approximators of the form $f(\boldsymbol{x}; \boldsymbol{\theta})$, where the dimensionality of $\boldsymbol{\theta}$ (i.e., the number of parameters) is fixed, and independent of the size N of the training set. However, such parametric models can overfit when N is small, and can underfit when N is large, due to their fixed capacity. In order to create models whose capacity automatically adapts to the amount of data, we turn to **nonparametric models**.

There are many approaches to building nonparametric models for classification and regression (see e.g., [Was06]). In this chapter, we consider a Bayesian approach in which we represent uncertainty about the input-output mapping f by defining a prior distribution over functions, and then updating it given data. In particular, we will use a **Gaussian process** to represent the prior $p(f)$; we then use Bayes rule to derive the posterior $p(f|\mathcal{D})$, which is another GP, as we explain below. More details on GPs can be found the excellent book [RW06], as well as the interative tutorial at `https://distill.pub/2019/visual-exploration-gaussian-processes`. See also Chapter 31 for other examples of Bayesian nonparametric models.

18.1.1 GPs: What and why?

To explain GPs in more detail, recall that a Gaussian random vector of length N, $\boldsymbol{f} = [f_1, \ldots, f_N]$, is defined by its mean $\boldsymbol{\mu} = \mathbb{E}[\boldsymbol{f}]$ and its covariance $\boldsymbol{\Sigma} = \text{Cov}[\boldsymbol{f}]$. Now consider a function $f : \mathcal{X} \to \mathbb{R}$ evaluated at a set of inputs, $\mathbf{X} = \{\boldsymbol{x}_n \in \mathcal{X}\}_{n=1}^N$. Let $\boldsymbol{f}_X = [f(\boldsymbol{x}_1), \ldots, f(\boldsymbol{x}_N)]$ be the set of unknown function values at these points. If $\boldsymbol{f}_X$ is jointly Gaussian for any set of $N \geq 1$ points, then we say that $f : \mathcal{X} \to \mathbb{R}$ is a **Gaussian process**. Such a process is defined by its **mean function** $m(\boldsymbol{x}) \in \mathbb{R}$ and a **covariance function**, $\mathcal{K}(\boldsymbol{x}, \boldsymbol{x}') \geq 0$, which is any positive definite **Mercer kernel** (see Section 18.2). For example, we might use an RBF kernel of the form $\mathcal{K}(\boldsymbol{x}, \boldsymbol{x}') \propto \exp(-||\boldsymbol{x} - \boldsymbol{x}'||^2)$ (see Section 18.2.1.1 for details).

We denote the corresponding GP by

$$f(\boldsymbol{x}) \sim GP(m(\boldsymbol{x}), \mathcal{K}(\boldsymbol{x}, \boldsymbol{x}')) \tag{18.1}$$

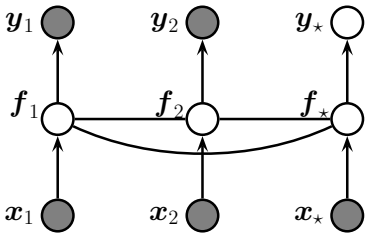


Figure 18.1: A Gaussian process for 2 training points, $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$, and 1 testing point, $\boldsymbol{x}_*$, represented as a graphical model representing $p(\boldsymbol{y}, \boldsymbol{f}_X|\mathbf{X}) = \mathcal{N}(\boldsymbol{f}_X|m(\mathbf{X}), \mathcal{K}(\mathbf{X})) \prod_i p(y_i|f_i)$. The hidden nodes $f_i = f(\boldsymbol{x}_i)$ represent the value of the function at each of the data points. These hidden nodes are fully interconnected by undirected edges, forming a Gaussian graphical model; the edge strengths represent the covariance terms $\Sigma_{ij} = \mathcal{K}(\boldsymbol{x}_i, \boldsymbol{x}_j)$. If the test point $\boldsymbol{x}_*$ is similar to the training points $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$, then the value of the hidden function f_* will be similar to f_1 and f_2, and hence the predicted output y_* will be similar to the training values y_1 and y_2.

where

$$m(\boldsymbol{x}) = \mathbb{E}\left[f(\boldsymbol{x})\right] \tag{18.2}$$

$$\mathcal{K}(\boldsymbol{x}, \boldsymbol{x}') = \mathbb{E}\left[(f(\boldsymbol{x}) - m(\boldsymbol{x}))(f(\boldsymbol{x}') - m(\boldsymbol{x}'))^{\mathsf{T}}\right] \tag{18.3}$$

This means that, for any finite set of points $\mathbf{X} = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N\}$, we have

$$p(\boldsymbol{f}_X|\mathbf{X}) = \mathcal{N}(\boldsymbol{f}_X|\boldsymbol{\mu}_X, \mathbf{K}_{X,X}) \tag{18.4}$$

where $\boldsymbol{\mu}_X = (m(\boldsymbol{x}_1), \ldots, m(\boldsymbol{x}_N))$ and $\mathbf{K}_{X,X}(i,j) \triangleq \mathcal{K}(\boldsymbol{x}_i, \boldsymbol{x}_j)$.

A GP can be used to define a prior over functions. We can evaluate this prior at any set of points we choose. However, to learn about the function from data, we have to update this prior with a likelihood function. We typically assume we have a set of N iid observations $\mathcal{D} = \{(\boldsymbol{x}_i, y_i) : i = 1 : N\}$, where $y_i \sim p(y|f(\boldsymbol{x}_i))$, as shown in Figure 18.1. If we use a Gaussian likelihood, we can compute the posterior $p(f|\mathcal{D})$ in closed form, as we discuss in Section 18.3. For other kinds of likelihoods, we will need to use approximate inference, as we discuss in Section 18.4. In many cases f is not directly observed, and instead forms part of a latent variable model, both in supervised and unsupervised settings such as in Section 28.3.7.

The generalization properties of a Gaussian process are controlled by its covariance function (kernel), which we describe in Section 18.2. These kernels live in a reproducing kernel Hilbert space (RKHS), described in Section 18.3.7.1.

GPs were originally designed for spatial data analysis, where the input is 2d. This special case is called **kriging**. However, they can be applied to higher dimensional inputs. In addition, while they have been traditionally limited to small datasets, it is now possible to apply GPs to problems with millions of points, with essentially exact inference. We discuss these scalability advances in Section 18.5.

Moreover, while Gaussian processes have historically been considered smoothing interpolators, GPs now routinely perform representation learning, through covariance function learning, and multilayer

models. These advances have clearly illustrated that GPs and neural networks are not competing, but complementary, and can be combined for better performance than would be achieved by deep learning alone. We describe GPs for representation learning in Section 18.6.

The connections between Gaussian processes and neural networks can also be further understood by considering infinite limits of neural networks that converge to Gaussian processes with particular covariance functions, which we describe in Section 18.7.

So Gaussian processes are non-parametric models which can scale and do representation learning. But why, in the age of deep learning, should we want to use a Gaussian process? There are several compelling reasons to prefer a GP, including:

- Gaussian processes typically provide well-calibrated predictive distributions, with a good characterization of epistemic (model) uncertainty — uncertainty arising from not knowing which of many solutions is correct. For example, as we move away from the data, there are a greater variety of consistent solutions, and so we expect greater uncertainty.

- Gaussian processes are often state-of-the-art for continuous regression problems, especially spatiotemporal problems, such as weather interpolation and forecasting. In regression, Gaussian process inference can also typically be performed in closed form.

- The marginal likelihood of a Gaussian process provides a powerful mechanism for flexible kernel learning. Kernel learning enables us to provide long-range extrapolations, but also tells us interpretable properties of the data that we didn't know before, towards scientific discovery.

- Gaussian processes are often used as a probabilistic surrogate for objectives in optimization, in a procedure known as **Bayesian optimization** (Section 6.8). To maximize an objective, we wish to move where there is a high expected value, but also to explore where we have large uncertainty. The ability for a Gaussian process to provide closed form inference in regression, in conjunction with high quality uncertainty representations, make them particularly impactful in this setting. Bayesian optimization has a wide range of applications, including A/B testing, experimental design, protein engineering, hyperparameter tuning, and AutoML. See Section 6.8 for details.

18.2 Mercer kernels

The generalization properties of Gaussian processes boil down to how we encode prior knowledge about the similarity of two input vectors. If we know that $\boldsymbol{x}_i$ is similar to $\boldsymbol{x}_j$, then we can encourage the model to make the predicted output at both locations (i.e., $f(\boldsymbol{x}_i)$ and $f(\boldsymbol{x}_j)$) to be similar.

To define similarity, we introduce the notion of a **kernel function**. The word "kernel" has many different meanings in mathematics; here we consider a **Mercer kernel**, also called a **positive definite kernel**. This is any symmetric function $\mathcal{K} : \mathcal{X} \times \mathcal{X} \to \mathbb{R}^+$ such that

$$\sum_{i=1}^{N}\sum_{j=1}^{N}\mathcal{K}(\boldsymbol{x}_i, \boldsymbol{x}_j)c_i c_j \geq 0 \tag{18.5}$$

for any set of N (unique) points $\boldsymbol{x}_i \in \mathcal{X}$, and any choice of numbers $c_i \in \mathbb{R}$. We assume $\mathcal{K}(\boldsymbol{x}_i, \boldsymbol{x}_j) > 0$, so that we can only achieve equality in the above equation if $c_i = 0$ for all i.

Another way to understand this condition is the following. Given a set of N datapoints, let us define the **Gram matrix** as the following $N \times N$ similarity matrix:

$$\mathbf{K} = \begin{pmatrix} \mathcal{K}(\boldsymbol{x}_1, \boldsymbol{x}_1) & \cdots & \mathcal{K}(\boldsymbol{x}_1, \boldsymbol{x}_N) \\ & \vdots & \\ \mathcal{K}(\boldsymbol{x}_{N_\mathcal{D}}, \boldsymbol{x}_1) & \cdots & \mathcal{K}(\boldsymbol{x}_N, \boldsymbol{x}_N) \end{pmatrix} \tag{18.6}$$

We say that $\mathcal{K}$ is a Mercer kernel iff the Gram matrix is positive definite for any set of (distinct) inputs $\{\boldsymbol{x}_i\}_{i=1}^N$.

The most widely used kernel for real-valued inputs is the **radial basis function** or **RBF** kernel, defined by

$$\mathcal{K}(\boldsymbol{x}, \boldsymbol{x}'; \ell) = \exp\left(-\frac{||\boldsymbol{x} - \boldsymbol{x}'||^2}{2\ell^2}\right) \tag{18.7}$$

Here ℓ corresponds to the length-scale of the kernel, i.e., the distance over which we expect differences to matter. This is also sometimes known as the **bandwidth** parameter. The RBF kernel measures similarity between two vectors in $\mathbb{R}^D$ using (scaled) Euclidean distance. In Section 18.2.1, we will discuss several other kinds of kernel.

18.2.1 Some popular Mercer kernels

In the sections below, we describe some popular Mercer kernels. More details can be found at [Wil14] and https://www.cs.toronto.edu/~duvenaud/cookbook/. See also Section 18.6 where we discuss how to learn kernels from data.

18.2.1.1 Stationary kernels for real-valued vectors

For real-valued inputs, $\mathcal{X} = \mathbb{R}^D$, it is common to use **stationary kernels**, which are functions of the form $\mathcal{K}(\boldsymbol{x}, \boldsymbol{x}') = \mathcal{K}(\boldsymbol{r})$, where $\boldsymbol{r} = \boldsymbol{x} - \boldsymbol{x}'$; thus the ouput only depends on the relative difference between the inputs. Furthermore, in many cases, all that matters is the magnitude of the difference:

$$r = ||\boldsymbol{r}||_2 = ||\boldsymbol{x} - \boldsymbol{x}'|| \tag{18.8}$$

We give some examples below. (See also Figure 18.3 and Figure 18.4 for some visualizations of these kernels.)

Squared exponential (RBF) kernel

The **squared exponential** (SE) kernel, also sometimes called the **exponentiated quadratic** kernel, is defined as

$$\mathcal{K}(r; \ell) = \exp\left(-\frac{r^2}{2\ell^2}\right) \tag{18.9}$$

Here ℓ corresponds to the length-scale of the kernel, i.e., the distance over which we expect differences to matter.

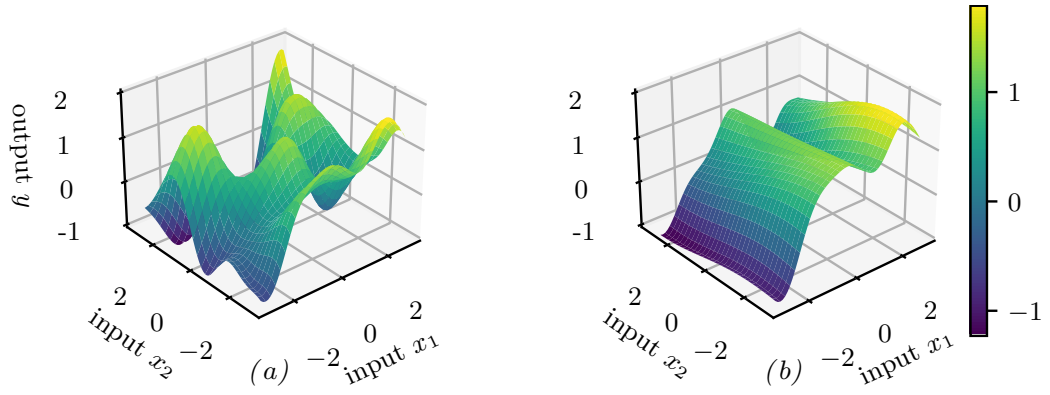


Figure 18.2: Function samples from a GP with an ARD kernel. (a) $\ell_1 = \ell_2 = 1$. Both dimensions contribute to the response. (b) $\ell_1 = 1$, $\ell_2 = 5$. The second dimension is essentially ignored. Adapted from Figure 5.1 of [RW06]. Generated by gpr_demo_ard.ipynb.

From Equation (18.8) we can rewrite this kernel as

$$\mathcal{K}(\boldsymbol{x}, \boldsymbol{x}'; \ell) = \exp\left(-\frac{||\boldsymbol{x} - \boldsymbol{x}'||^2}{2\ell^2}\right) \tag{18.10}$$

This is the RBF kernel we encountered earlier. It is also sometimes called the **Gaussian kernel**. See Figure 18.3(f) and Figure 18.4(f) for a visualization in 1D.

ARD kernel

We can generalize the RBF kernel by replacing Euclidean distance with Mahalanobis distance, as follows:

$$\mathcal{K}(\boldsymbol{r}; \boldsymbol{\Sigma}, \sigma^2) = \sigma^2 \exp\left(-\frac{1}{2}\boldsymbol{r}^\mathsf{T}\boldsymbol{\Sigma}^{-1}\boldsymbol{r}\right) \tag{18.11}$$

where $\boldsymbol{r} = \boldsymbol{x} - \boldsymbol{x}'$. If $\boldsymbol{\Sigma}$ is diagonal, this can be written as

$$\mathcal{K}(\boldsymbol{r}; \boldsymbol{\ell}, \sigma^2) = \sigma^2 \exp\left(-\frac{1}{2}\sum_{d=1}^{D}\frac{1}{\ell_d^2}r_d^2\right) = \prod_{d=1}^{D}\mathcal{K}(r_d; \ell_d, \sigma^{2/d}) \tag{18.12}$$

where

$$\mathcal{K}(r; \ell, \tau^2) = \tau^2 \exp\left(-\frac{1}{2}\frac{1}{\ell^2}r^2\right) \tag{18.13}$$

We can interpret σ^2 as the overall variance, and ℓ_d as defining the **characteristic length scale** of dimension d. If d is an irrelevant input dimension, we can set $\ell_d = \infty$, so the corresponding dimension will be ignored. This is known as **Automatic Relevance Determination** or **ARD** (Section 15.2.7). Hence the corresponding kernel is called the **ARD kernel**. See Figure 18.2 for an illustration of some 2d functions sampled from a GP using this prior.

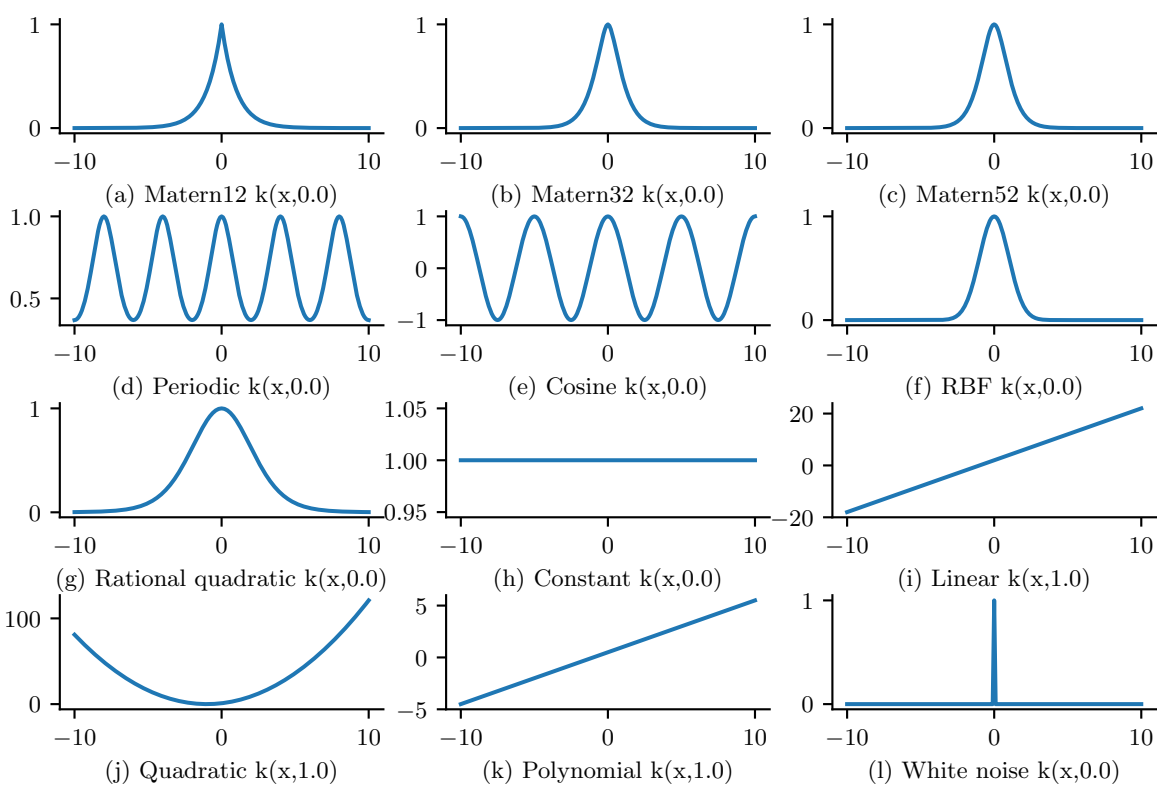


Figure 18.3: GP kernels evaluated at $k(x, 0)$ as a function of x. Generated by gpKernelPlot.ipynb.

Matern kernels

The SE kernel gives rise to functions that are infinitely differentiable, and therefore are very smooth. For many applications, it is better to use the **Matern kernel**, which gives rise to "rougher" functions, which can better model local "wiggles" without having to make the overall length scale very small.

The Matern kernel has the following form:

$$\mathcal{K}(r; \nu, \ell) = \frac{2^{1-\nu}}{\Gamma(\nu)} \left(\frac{\sqrt{2\nu} r}{\ell} \right)^{\nu} K_\nu \left(\frac{\sqrt{2\nu} r}{\ell} \right) \tag{18.14}$$

where K_ν is a modified Bessel function and ℓ is the length scale. Functions sampled from this GP are k-times differentiable iff $\nu > k$. As $\nu \to \infty$, this approaches the SE kernel.

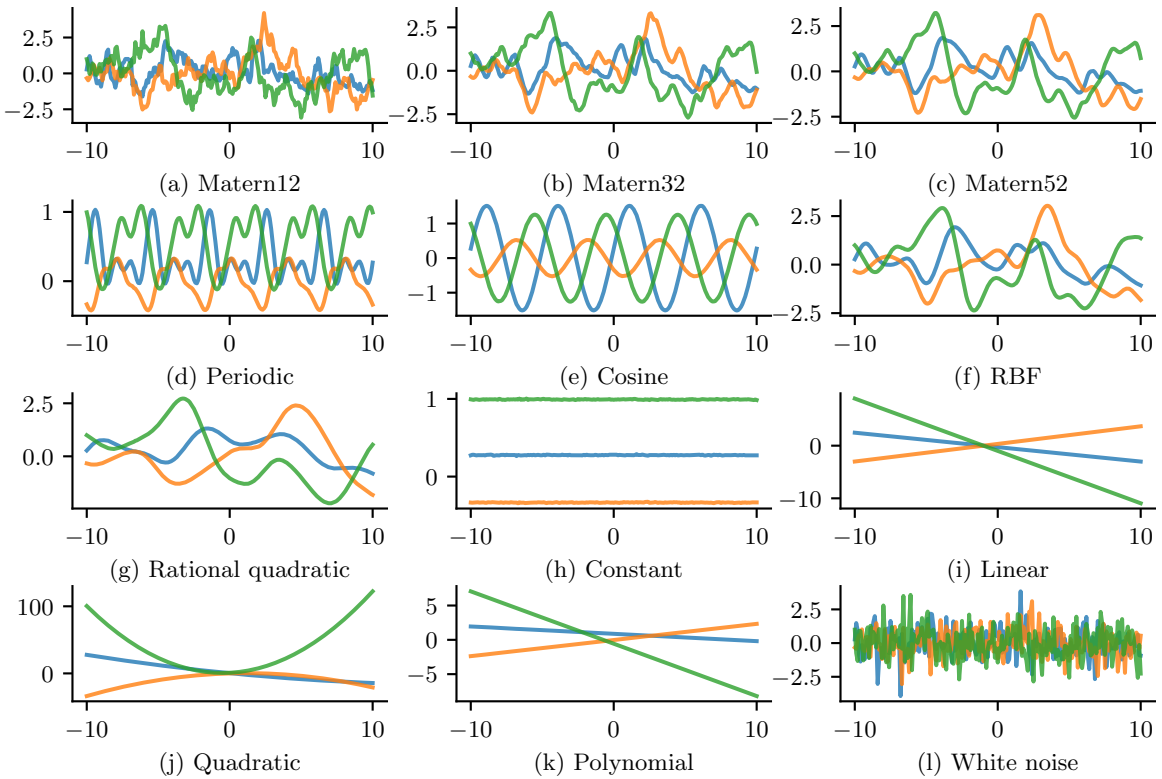


Figure 18.4: GP samples drawn using different kernels. Generated by gpKernelPlot.ipynb.

For values $\nu \in \{\frac{1}{2}, \frac{3}{2}, \frac{5}{2}\}$, the function simplifies as follows:

$$\mathcal{K}(r; \frac{1}{2}, \ell) = \exp(-\frac{r}{\ell}) \tag{18.15}$$

$$\mathcal{K}(r; \frac{3}{2}, \ell) = \left(1 + \frac{\sqrt{3}r}{\ell}\right) \exp\left(-\frac{\sqrt{3}r}{\ell}\right) \tag{18.16}$$

$$\mathcal{K}(r; \frac{5}{2}, \ell) = \left(1 + \frac{\sqrt{5}r}{\ell} + \frac{5r^2}{3\ell^2}\right) \exp\left(-\frac{\sqrt{5}r}{\ell}\right) \tag{18.17}$$

See Figure 18.3(a-c) and Figure 18.4(a-c) for a visualization.

The value $\nu = \frac{1}{2}$ corresponds to the **Ornstein-Uhlenbeck process**, which describes the velocity of a particle undergoing Brownian motion. The corresponding function is continuous but not differentiable, and hence is very "jagged".

Periodic kernels

One way to create a periodic 1d random function is to map x to the 2d space $\boldsymbol{u}(x) = (\cos(x), \sin(x))$, and then use an SE kernel in $\boldsymbol{u}$-space:

$$\mathcal{K}(x, x') = \exp\left(-\frac{2\sin^2((x-x')/2)}{\ell^2}\right) \tag{18.18}$$

which follows since $(\cos(x)-\cos(x'))^2+(\sin(x)-\sin(x'))^2 = 4\sin^2((x-x')/2)$. We can generalize this by specifying the period p to get the **periodic kernel**, also called the **exp-sine-squared kernel**:

$$\mathcal{K}_{\text{per}}(r; \ell, p) = \exp\left(-\frac{2}{\ell^2}\sin^2(\pi\frac{r}{p})\right) \tag{18.19}$$

where p is the period and ℓ is the length scale. See Figure 18.3(d-e) and Figure 18.4(d-e) for a visualization.

A related kernel is the **cosine kernel**:

$$\mathcal{K}(r; p) = \cos\left(2\pi\frac{r}{p}\right) \tag{18.20}$$

Rational quadratic kernel

We define the **rational quadratic** kernel to be

$$\mathcal{K}_{RQ}(r; \ell, \alpha) = \left(1 + \frac{r^2}{2\alpha\ell^2}\right)^{-\alpha} \tag{18.21}$$

We recognize this is proportional to a Student T density. Hence it can be interpreted as a scale mixture of SE kernels of different characteristic lengths. In particular, let $\tau = 1/\ell^2$, and assume $\tau \sim \text{Ga}(\alpha, \ell^2)$. Then one can show that

$$\mathcal{K}_{RQ}(r) = \int p(\tau|\alpha, \ell^2)\mathcal{K}_{SE}(r|\tau)d\tau \tag{18.22}$$

As $\alpha \to \infty$, this reduces to a SE kernel.

See Figure 18.3(g) and Figure 18.4(g) for a visualization.

18.2.1.2 Making new kernels from old

Given two valid kernels $\mathcal{K}_1(\boldsymbol{x}, \boldsymbol{x}')$ and $\mathcal{K}_2(\boldsymbol{x}, \boldsymbol{x}')$, we can create a new kernel using any of the following methods:

$$\mathcal{K}(\boldsymbol{x}, \boldsymbol{x}') = c\mathcal{K}_1(\boldsymbol{x}, \boldsymbol{x}'), \text{ for any constant } c > 0 \tag{18.23}$$

$$\mathcal{K}(\boldsymbol{x}, \boldsymbol{x}') = f(\boldsymbol{x})\mathcal{K}_1(\boldsymbol{x}, \boldsymbol{x}')f(\boldsymbol{x}'), \text{ for any function } f \tag{18.24}$$

$$\mathcal{K}(\boldsymbol{x}, \boldsymbol{x}') = q(\mathcal{K}_1(\boldsymbol{x}, \boldsymbol{x}')) \text{ for any function polynomial } q \text{ with nonneg. coef.} \tag{18.25}$$

$$\mathcal{K}(\boldsymbol{x}, \boldsymbol{x}') = \exp(\mathcal{K}_1(\boldsymbol{x}, \boldsymbol{x}')) \tag{18.26}$$

$$\mathcal{K}(\boldsymbol{x}, \boldsymbol{x}') = \boldsymbol{x}^{\mathsf{T}}\mathbf{A}\boldsymbol{x}', \text{ for any psd matrix } \mathbf{A} \tag{18.27}$$

For example, suppose we start with the linear kernel $\mathcal{K}(\boldsymbol{x}, \boldsymbol{x}') = \boldsymbol{x}\boldsymbol{x}'$. We know this is a valid Mercer kernel, since the corresponding Gram matrix is just the (scaled) covariance matrix of the data. From the above rules, we can see that the **polynomial kernel**

$$\mathcal{K}(\boldsymbol{x}, \boldsymbol{x}') = (\boldsymbol{x}^\mathsf{T}\boldsymbol{x}')^M \tag{18.28}$$

is a valid Mercer kernel. This contains all monomials of order M. For example, if $M = 2$ and the inputs are 2d, we have

$$(\boldsymbol{x}^\mathsf{T}\boldsymbol{x}')^2 = (x_1x_1' + x_2x_2')^2 = (x_1x_1')^2 + (x_2x_2)^2 + (x_1x_1')(x_2x_2') \tag{18.29}$$

We can generalize this to contain all terms up to degree M by using the kernel $\mathcal{K}(\boldsymbol{x}, \boldsymbol{x}') = (\boldsymbol{x}^\mathsf{T}\boldsymbol{x}' + c)^M$. For example, if $M = 2$ and the inputs are 2d, we have

$$\begin{aligned}(\boldsymbol{x}^\mathsf{T}\boldsymbol{x}' + 1)^2 = (x_1x_1')^2 + (x_1x_1')(x_2x_2') + (x_1x_1') \\ + (x_2x_2)(x_1x_1') + (x_2x_2')^2 + (x_2x_2') \\ + (x_1x_1') + (x_2x_2') + 1\end{aligned} \tag{18.30}$$

We can also use the above rules to establish that the Gaussian kernel is a valid kernel. To see this, note that

$$||\boldsymbol{x} - \boldsymbol{x}'||^2 = \boldsymbol{x}^\mathsf{T}\boldsymbol{x} + (\boldsymbol{x}')^\mathsf{T}\boldsymbol{x}' - 2\boldsymbol{x}^\mathsf{T}\boldsymbol{x}' \tag{18.31}$$

and hence

$$\mathcal{K}(\boldsymbol{x}, \boldsymbol{x}') = \exp(-||\boldsymbol{x} - \boldsymbol{x}'||^2/2\sigma^2) = \exp(-\boldsymbol{x}^\mathsf{T}\boldsymbol{x}/2\sigma^2)\exp(\boldsymbol{x}^\mathsf{T}\boldsymbol{x}'/\sigma^2)\exp(-(\boldsymbol{x}')^\mathsf{T}\boldsymbol{x}'/2\sigma^2) \tag{18.32}$$

is a valid kernel.

18.2.1.3 Combining kernels by addition and multiplication

We can also combine kernels using addition or multiplication:

$$\mathcal{K}(\boldsymbol{x}, \boldsymbol{x}') = \mathcal{K}_1(\boldsymbol{x}, \boldsymbol{x}') + \mathcal{K}_2(\boldsymbol{x}, \boldsymbol{x}') \tag{18.33}$$

$$\mathcal{K}(\boldsymbol{x}, \boldsymbol{x}') = \mathcal{K}_1(\boldsymbol{x}, \boldsymbol{x}') \times \mathcal{K}_2(\boldsymbol{x}, \boldsymbol{x}') \tag{18.34}$$

Multiplying two positive-definite kernels together always results in another positive definite kernel. This is a way to get a conjunction of the individual properties of each kernel, as illustrated in Figure 18.5.

In addition, adding two positive-definite kernels together always results in another positive definite kernel. This is a way to get a disjunction of the individual properties of each kernel, as illustrated in Figure 18.6.

For an example of combining kernels to forecast some timeseries data, see Section 18.8.1.

18.2.1.4 Kernels for structured inputs

Kernels are particularly useful when the inputs are structured objects, such as strings and graphs, since it is often hard to "featurize" variable-sized inputs. For example, we can define a **string kernel** which compares strings in terms of the number of n-grams they have in common [Lod+02; BC17].

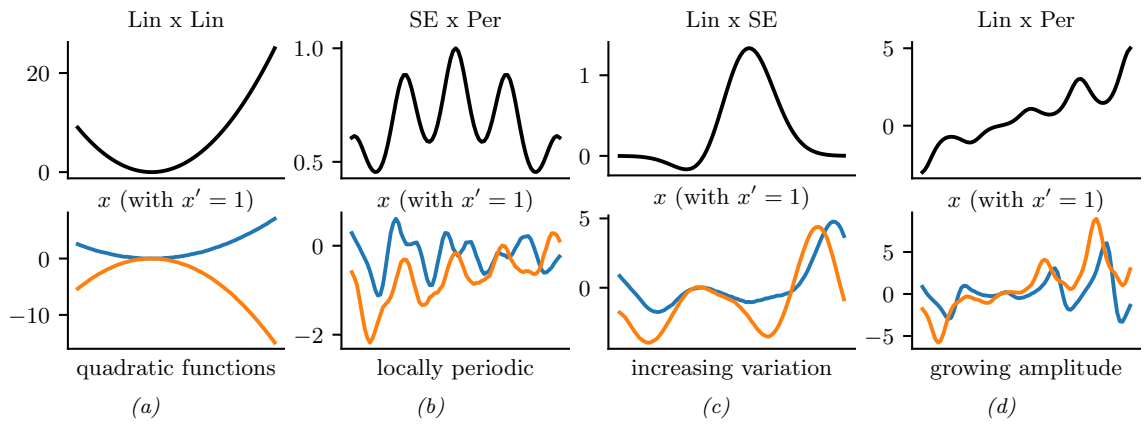


Figure 18.5: Examples of 1d structures obtained by multiplying elementary kernels. Top row shows $\mathcal{K}(x, x' = 1)$. Bottom row shows some functions sampled from $GP(f|0, \mathcal{K})$. Adapted from Figure 2.2 of [Duv14]. Generated by combining_kernels_by_multiplication.ipynb.

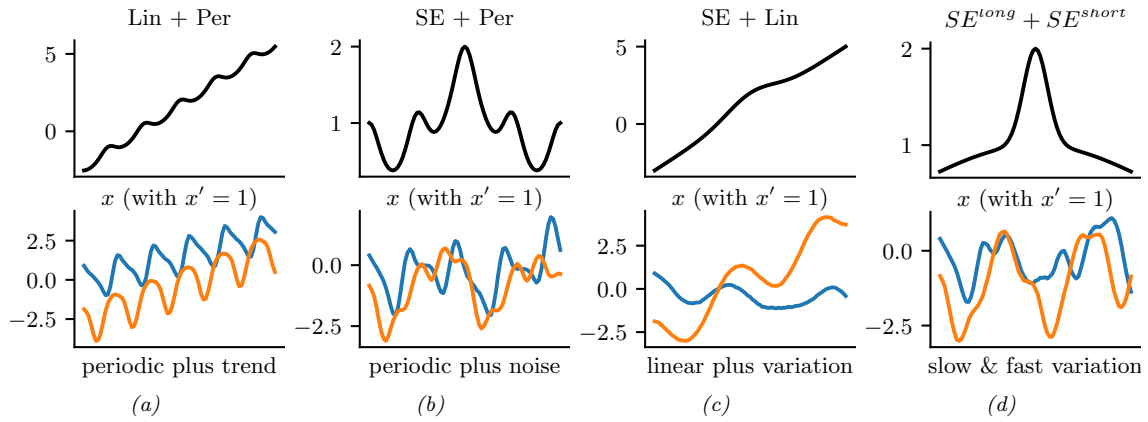


Figure 18.6: Examples of 1d structures obtained by summing elementary kernels. Top row shows $\mathcal{K}(x, x' = 1)$. Bottom row shows some functions sampled from $GP(f|0, \mathcal{K})$. Adapted from Figure 2.2 of [Duv14]. Generated by combining_kernels_by_summation.ipynb.

We can also define kernels on graphs [KJM19]. For example, the **random walk kernel** conceptually performs random walks on two graphs simultaneously, and then counts the number of paths that were produced by both walks. This can be computed efficiently as discussed in [Vis+10]. For more details on graph kernels, see [KJM19].

For a review of kernels on structured objects, see e.g., [Gär03].

18.2.2 Mercer's theorem

Recall that any positive definite matrix $\mathbf{K}$ can be represented using an eigendecomposition of the form $\mathbf{K} = \mathbf{U}^\mathsf{T}\mathbf{\Lambda}\mathbf{U}$, where $\mathbf{\Lambda}$ is a diagonal matrix of eigenvalues $\lambda_i > 0$, and $\mathbf{U}$ is a matrix containing the eigenvectors. Now consider element (i, j) of $\mathbf{K}$:

$$k_{ij} = (\mathbf{\Lambda}^{\frac{1}{2}}\mathbf{U}_{:i})^\mathsf{T}(\mathbf{\Lambda}^{\frac{1}{2}}\mathbf{U}_{:j}) \tag{18.35}$$

where $\mathbf{U}_{:i}$ is the i'th column of $\mathbf{U}$. If we define $\boldsymbol{\phi}(\boldsymbol{x}_i) = \mathbf{U}_{:i}$, then we can write

$$k_{ij} = \sum_{m=1}^{M} \lambda_m \phi_m(\boldsymbol{x}_i)\phi_m(\boldsymbol{x}_j) \tag{18.36}$$

where M is the rank of the kernel matrix. Thus we see that the entries in the kernel matrix can be computed by performing an inner product of some feature vectors that are implicitly defined by the eigenvectors of the kernel matrix.

This idea can be generalized to apply to kernel functions, not just kernel matrices, as we now show. First, we define an **eigenfunction** $\phi()$ of a kernel $\mathcal{K}$ with eigenvalue λ wrt measure μ as a function that satisfies

$$\int \mathcal{K}(\boldsymbol{x}, \boldsymbol{x}')\phi(\boldsymbol{x})d\mu(\boldsymbol{x}) = \lambda\phi(\boldsymbol{x}') \tag{18.37}$$

We usually sort the eigenfunctions in order of decreasing eigenvalue, $\lambda_1 \geq \lambda_2 \geq \cdots$. The eigenfunctions are orthogonal wrt μ:

$$\int \phi_i(\boldsymbol{x})\phi_j(\boldsymbol{x})d\mu(\boldsymbol{x}) = \delta_{ij} \tag{18.38}$$

where δ_{ij} is the Kronecker delta. With this definition in hand, we can state **Mercer's theorem**. Informally, it says that any positive definite kernel function can be represented as the following infinite sum:

$$\mathcal{K}(\boldsymbol{x}, \boldsymbol{x}') = \sum_{m=1}^{\infty} \lambda_m \phi_m(\boldsymbol{x})\phi_m(\boldsymbol{x}') \tag{18.39}$$

where ϕ_m are eigenfunctions of the kernel, and λ_m are the corresponding eigenvalues. This is the functional analog of Equation (18.36).

A **degenerate kernel** has only a finite number of non-zero eigenvalues. In this case, we can rewrite the kernel function as an inner product between two finite-length vectors. For example, consider the **quadratic kernel** $\mathcal{K}(\boldsymbol{x}, \boldsymbol{x}') = \langle \boldsymbol{x}, \boldsymbol{x}' \rangle^2$. In 2d, we have

$$\mathcal{K}(\boldsymbol{x}, \boldsymbol{x}') = (x_1x_1' + x_2x_2')^2 = x_1^2(x_1')^2 + 2x_1x_2x_1'x_2' + x_2^2(x_2')^2 \tag{18.40}$$

If we define $\boldsymbol{\phi}(x_1, x_2) = [x_1^2, \sqrt{2}x_1x_2, x_2^2] \in \mathbb{R}^3$, then we can write this as $\mathcal{K}(\boldsymbol{x}, \boldsymbol{x}') = \boldsymbol{\phi}(\boldsymbol{x})^\mathsf{T}\boldsymbol{\phi}(\boldsymbol{x})$. Thus we see that this kernel is degenerate.

Now consider the RBF kernel. In this case, the corresponding feature representation is infinite dimensional (see Section 18.2.3.1 for details). However, by working with kernel functions, we can avoid having to deal with infinite dimensional vectors.

From the anove, we see that we can replace inner product operations in an explicit (possibly infinite dimensional) feature space with a call to a kernel function, i.e., we replace $\boldsymbol{\phi}(\boldsymbol{x})^\mathsf{T}\boldsymbol{\phi}(\boldsymbol{x})$ with $\mathcal{K}(\boldsymbol{x}, \boldsymbol{x}')$. This is called the **kernel trick**.

18.2.3 Kernels from Spectral Densities

Consider the case of a **shift-invariant kernel**, also known as a *stationary kernel*, which satisfies $\mathcal{K}(\boldsymbol{x}, \boldsymbol{x}') = \mathcal{K}(\boldsymbol{\delta})$, where $\boldsymbol{\delta} = \boldsymbol{x} - \boldsymbol{x}'$, for $\boldsymbol{x}, \boldsymbol{x}' \in \mathbb{R}^d$. Let us further assume that $\mathcal{K}(\boldsymbol{\delta})$ is positive definite. In this case, **Bochner's theorem** tells us that we can represent $\mathcal{K}(\boldsymbol{\delta})$ by its Fourier transform:

$$\mathcal{K}(\boldsymbol{\delta}) = \int_{\mathbb{R}^d} p(\boldsymbol{\omega}) e^{j\boldsymbol{\omega}^\mathsf{T}\boldsymbol{\delta}} d\boldsymbol{\omega} \tag{18.41}$$

where $j = \sqrt{-1}$, $e^{j\theta} = \cos(\theta) + j\sin(\theta)$, and $p(\boldsymbol{\omega})$, the **spectral density**, is a distribution over frequencies.

We can easily derive and gain intuitions into several kernels from spectral densities. If we take the Fourier transform of an RBF kernel we find the spectral density $p(\boldsymbol{\omega}) = \sqrt{2\pi\ell^2}\exp\left(-2\pi^2\boldsymbol{\omega}^2\ell^2\right)$. Thus the spectral density is also Gaussian, but with a bandwidth *inversely* proportional to the length-scale hyperparameter ℓ. That is, as ℓ becomes large, the spectral density collapses onto a point mass. This result is intuitive: as we increase the length-scale our model treats points as correlated over large distances, and becomes very smooth and slowly varying, and thus low-frequency. In general, since the Gaussian distribution has relatively light tails, we can see that RBF kernels won't generally support high frequency solutions.

We can instead use a Student-t spectral density, which has heavy tails that will provide greater support for higher frequencies. Taking the inverse Fourier transform of this spectral density, we recover the Matern kernel, with degrees of freedom ν corresponding to the degrees of freedom in the spectral density. Indeed, the smaller we make ν the less smooth and higher frequency are the associated fits to data using a Matern kernel.

We can also derive **spectral mixture kernels** by modelling the spectral density as a scale-location mixture of Gaussians and taking the inverse Fourier transform [WA13]. Since scale-location mixtures of Gaussians are dense in the set of distributions, and can therefore approximate any spectral density, this kernel can approximate any stationary kernel to arbitrary precision. The spectral mixture kernel thus forms a powerful approach to kernel learning, which we discuss further in Section 18.6.5.

18.2.3.1 Random feature kernels

Although the power of kernels resides in the ability to avoid working with featurized representations of the inputs, such kernelized methods can take $O(N^3)$ time, in order to invert the Gram matrix $\mathbf{K}$, This can make it difficult to use such methods on large scale data. Fortunately, we can approximate the feature map for many kernels using a randomly chosen finite set of M basis functions, thus reducing the cost to $O(NM + M^3)$.

We will show how to do this for shift-invariant kernels by returning to Bochner's theorem in Eq. (18.41):

$$\mathcal{K}(\boldsymbol{\delta}) = \int_{\mathbb{R}^d} p(\boldsymbol{\omega}) e^{j\boldsymbol{\omega}^\mathsf{T}\boldsymbol{\delta}} d\boldsymbol{\omega} \tag{18.42}$$

where $j = \sqrt{-1}$, $e^{j\theta} = \cos(\theta) + j\sin(\theta)$, and the spectral density $p(\boldsymbol{\omega})$ is a distribution over frequencies.

In the case of a Gaussian RBF kernel, we have seen that the spectral density is a Gaussian distribution. Hence we can easily compute a Monte Carlo approximation to this integral by sampling

random Gaussian vectors. This yields the following approximation: $\mathcal{K}(\boldsymbol{x}, \boldsymbol{x}') \approx \boldsymbol{\phi}(\boldsymbol{x})^\mathsf{T} \boldsymbol{\phi}(\boldsymbol{x})$, where the (real-valued) feature vector is given by

$$\boldsymbol{\phi}(\boldsymbol{x}) = \sqrt{\frac{1}{D}} \left[\sin(\boldsymbol{z}_1^\mathsf{T}\boldsymbol{x}), \cdots, \sin(\boldsymbol{z}_D^\mathsf{T}\boldsymbol{x}), \cos(\boldsymbol{z}_1^\mathsf{T}\boldsymbol{x}), \cdots, \cos(\boldsymbol{z}_D^\mathsf{T}\boldsymbol{x})\right] \tag{18.43}$$

$$= \sqrt{\frac{1}{D}} \left[\sin(\mathbf{Z}^\mathsf{T}\boldsymbol{x}), \cos(\mathbf{Z}^\mathsf{T}\boldsymbol{x})\right] \tag{18.44}$$

Here $\mathbf{Z} = (1/\sigma)\mathbf{G}$, and $\mathbf{G} \in \mathbb{R}^{d \times D}$ is a random Gaussian matrix, where the entries are sampled iid from $\mathcal{N}(0, 1)$. The representation in Equation (18.44) are called **random Fourier features** (**RFF**) [RR08] or "weighted sums of random kitchen sinks" [RR09]. (One can obtain an even better approximation by ensuring that the rows of $\mathbf{Z}$ are random but orthogonal; this is called **orthogonal random features** [Yu+16].)

One can create similar random feature representations for other kinds of kernels. We can then use such features for supervised learning by defining $f(\boldsymbol{x}; \boldsymbol{\theta}) = \mathbf{W}\varphi(\mathbf{Z}\boldsymbol{x}) + \boldsymbol{b}$, where $\mathbf{Z}$ is a random Gaussian matrix, and the form of φ depends on the chosen kernel. This is equivalent to a one layer MLP with random input-to-hidden weights; since we only optimize the hidden-to-output weights $\boldsymbol{\theta} = (\mathbf{W}, \boldsymbol{b})$, the model is equivalent to a linear model with fixed random features. If we use enough random features, we can approximate the performance of a kernelized prediction model, but the computational cost is now $O(N)$ rather than $O(N^2)$.

18.3 GPs with Gaussian likelihoods

In this section, we discuss GPs for regression, using a Gaussian likelihood. In this case, all the computations can be performed in closed form, using standard linear algebra methods. We extend this framework to non-Gaussian likelihoods later in the chapter.

18.3.1 Predictions using noise-free observations

Suppose we observe a training set $\mathcal{D} = \{(\boldsymbol{x}_n, y_n) : n = 1 : N\}$, where $y_n = f(\boldsymbol{x}_n)$ is the noise-free observation of the function evaluated at $\boldsymbol{x}_n$. If we ask the GP to predict $f(\boldsymbol{x})$ for a value of $\boldsymbol{x}$ that it has already seen, we want the GP to return the answer $f(\boldsymbol{x})$ with no uncertainty. In other words, it should act as an **interpolator** of the training data. Here we assume the observed function values are noiseless. We will consider the case of noisy observations shortly.

Now we consider the case of predicting the outputs for new inputs that may not be in $\mathcal{D}$. Specifically, given a test set $\mathbf{X}_*$ of size $N_* \times D$, we want to predict the function outputs $\boldsymbol{f}_* = [f(\boldsymbol{x}_1), \ldots, f(\boldsymbol{x}_{N_*})]$. By definition of the GP, the joint distribution $p(\boldsymbol{f}_X, \boldsymbol{f}_* | \mathbf{X}, \mathbf{X}_*)$ has the following form

$$\begin{pmatrix} \boldsymbol{f}_X \\ \boldsymbol{f}_* \end{pmatrix} \sim \mathcal{N}\left(\begin{pmatrix} \boldsymbol{\mu}_X \\ \boldsymbol{\mu}_* \end{pmatrix}, \begin{pmatrix} \mathbf{K}_{X,X} & \mathbf{K}_{X,*} \\ \mathbf{K}_{X,*}^\mathsf{T} & \mathbf{K}_{*,*} \end{pmatrix} \right) \tag{18.45}$$

where $\boldsymbol{\mu}_X = (m(\boldsymbol{x}_1), \ldots, m(\boldsymbol{x}_{N_\mathcal{D}}))$, $\boldsymbol{\mu}_* = (m(\boldsymbol{x}_1^*), \ldots, m(\boldsymbol{x}_{N_*}^*))$, $\mathbf{K}_{X,X} = \mathcal{K}(\mathbf{X}, \mathbf{X})$ is $N_\mathcal{D} \times N_\mathcal{D}$, $\mathbf{K}_{X,*} = \mathcal{K}(\mathbf{X}, \mathbf{X}_*)$ is $N_\mathcal{D} \times N_*$, and $\mathbf{K}_{*,*} = \mathcal{K}(\mathbf{X}_*, \mathbf{X}_*)$ is $N_* \times N_*$. See Figure 18.7 for a static illustration, and http://www.infinitecuriosity.org/vizgp/ for an interactive visualization.

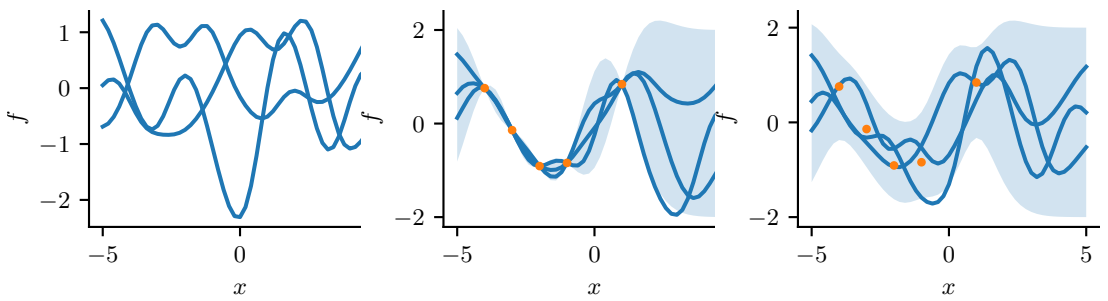


Figure 18.7: Left: some functions sampled from a GP prior with RBF kernel. Middle: some samples from a GP posterior, after conditioning on 5 noise-free observations. Right: some samples from a GP posterior, after conditioning on 5 noisy observations. The shaded area represents $\mathbb{E}[f(\boldsymbol{x})] \pm 2\sqrt{\mathbb{V}[f(\boldsymbol{x})]}$. *Adapted from Figure 2.2 of [RW06]. Generated by gpr_demo_noise_free.ipynb.*

By the standard rules for conditioning Gaussians (Section 2.2.5.4), the posterior has the following form

$$p(\boldsymbol{f}_*|\mathbf{X}_*, \mathcal{D}) = \mathcal{N}(\boldsymbol{f}_*|\boldsymbol{\mu}_{*|X}, \boldsymbol{\Sigma}_{*|X}) \tag{18.46}$$

$$\boldsymbol{\mu}_{*|X} = \boldsymbol{\mu}_* + \mathbf{K}_{X,*}^{\mathsf{T}} \mathbf{K}_{X,X}^{-1} (\boldsymbol{f}_X - \boldsymbol{\mu}_X) \tag{18.47}$$

$$\boldsymbol{\Sigma}_{*|X} = \mathbf{K}_{*,*} - \mathbf{K}_{X,*}^{\mathsf{T}} \mathbf{K}^{-1} \mathbf{K}_{X,*} \tag{18.48}$$

This process is illustrated in Figure 18.7. On the left we show some samples from the prior, $p(f)$, where we use an RBF kernel (Section 18.2.1.1) and a zero mean function. On the right, we show samples from the posterior, $p(f|\mathcal{D})$. We see that the model perfectly interpolates the training data, and that the predictive uncertainty increases as we move further away from the observed data.

Note that the cost of the above method for sampling N_* points is $O(N_*^3)$. This can be reduce to $O(N_*)$ time using the methods in [Ple+18; Wil+20a].

18.3.2 Predictions using noisy observations

In Section 18.3.1, we showed how to do GP regression when the training data was noiseless. Now let us consider the case where what we observe is a noisy version of the underlying function, $y_n = f(\boldsymbol{x}_n) + \epsilon_n$, where $\epsilon_n \sim \mathcal{N}(0, \sigma_y^2)$. In this case, the model is not required to interpolate the data, but it must come "close" to the observed data. The covariance of the observed noisy responses is

$$\mathrm{Cov}[y_i, y_j] = \mathrm{Cov}[f_i, f_j] + \mathrm{Cov}[\epsilon_i, \epsilon_j] = \mathcal{K}(\boldsymbol{x}_i, \boldsymbol{x}_j) + \sigma_y^2 \delta_{ij} \tag{18.49}$$

where $\delta_{ij} = \mathbb{I}(i = j)$. In other words

$$\mathrm{Cov}[\boldsymbol{y}|\mathbf{X}] = \mathbf{K}_{X,X} + \sigma_y^2 \mathbf{I}_N \tag{18.50}$$

The joint density of the observed data and the latent, noise-free function on the test points is given

by

$$\begin{pmatrix} \boldsymbol{y} \\ \boldsymbol{f}_* \end{pmatrix} \sim \mathcal{N}\left(\begin{pmatrix} \boldsymbol{\mu}_X \\ \boldsymbol{\mu}_* \end{pmatrix}, \begin{pmatrix} \mathbf{K}_{X,X} + \sigma_y^2 \mathbf{I} & \mathbf{K}_{X,*} \\ \mathbf{K}_{X,*}^\mathsf{T} & \mathbf{K}_{*,*} \end{pmatrix} \right) \tag{18.51}$$

Hence the posterior predictive density at a set of test points $\mathbf{X}_*$ is

$$p(\boldsymbol{f}_*|\mathcal{D}, \mathbf{X}_*) = \mathcal{N}(\boldsymbol{f}_*|\boldsymbol{\mu}_{*|X}, \boldsymbol{\Sigma}_{*|X}) \tag{18.52}$$

$$\boldsymbol{\mu}_{*|X} = \boldsymbol{\mu}_* + \mathbf{K}_{X,*}^\mathsf{T}(\mathbf{K}_{X,X} + \sigma_y^2 \mathbf{I})^{-1}(\boldsymbol{y} - \boldsymbol{\mu}_X) \tag{18.53}$$

$$\boldsymbol{\Sigma}_{*|X} = \mathbf{K}_{*,*} - \mathbf{K}_{X,*}^\mathsf{T}(\mathbf{K}_{X,X} + \sigma_y^2 \mathbf{I})^{-1}\mathbf{K}_{X,*} \tag{18.54}$$

In the case of a single test input, this simplifies as follows

$$p(f_*|\mathcal{D}, \boldsymbol{x}_*) = \mathcal{N}(f_*|m_* + \boldsymbol{k}_*^\mathsf{T}(\mathbf{K}_{X,X} + \sigma_y^2 \mathbf{I})^{-1}(\boldsymbol{y} - \boldsymbol{\mu}_X),\ k_{**} - \boldsymbol{k}_*^\mathsf{T}(\mathbf{K}_{X,X} + \sigma_y^2 \mathbf{I})^{-1}\boldsymbol{k}_*) \tag{18.55}$$

where $\boldsymbol{k}_* = [\mathcal{K}(\boldsymbol{x}_*, \boldsymbol{x}_1), \ldots, \mathcal{K}(\boldsymbol{x}_*, \boldsymbol{x}_N)]$ and $k_{**} = \mathcal{K}(\boldsymbol{x}_*, \boldsymbol{x}_*)$. If the mean function is zero, we can write the posterior mean as follows:

$$\mu_{*|X} = \boldsymbol{k}_*^\mathsf{T} \underbrace{\mathbf{K}_\sigma^{-1}\boldsymbol{y}}_{\boldsymbol{\alpha}} = \sum_{n=1}^{N} \mathcal{K}(\boldsymbol{x}_*, \boldsymbol{x}_n)\alpha_n \tag{18.56}$$

where

$$\mathbf{K}_\sigma = \mathbf{K}_{X,X} + \sigma_y^2 \mathbf{I} \tag{18.57}$$

$$\boldsymbol{\alpha} = \mathbf{K}_\sigma^{-1}\boldsymbol{y} \tag{18.58}$$

Fitting this model amounts to computing $\boldsymbol{\alpha}$ in Equation (18.58). This is usually done by computing the Cholesky decomposition of $\mathbf{K}_\sigma$, as described in Section 18.3.6. Once we have computed $\boldsymbol{\alpha}$, we can compute predictions for each test point in $O(N)$ time for the mean, and $O(N^2)$ time for the variance.

18.3.3 Weight space vs function space

In this section, we show how Bayesian linear regression is a special case of a GP.

Consider the linear regression model $y = f(\boldsymbol{x}) + \epsilon$, where $f(\boldsymbol{x}) = \boldsymbol{w}^\mathsf{T}\boldsymbol{\phi}(\boldsymbol{x})$ and $\epsilon \sim \mathcal{N}(0, \sigma_y^2)$. If we use a Gaussian prior $p(\boldsymbol{w}) = \mathcal{N}(\boldsymbol{w}|\mathbf{0}, \boldsymbol{\Sigma}_w)$, then the posterior is as follows (see Section 15.2.1 for the derivation):

$$p(\boldsymbol{w}|\mathcal{D}) = \mathcal{N}(\boldsymbol{w}|\frac{1}{\sigma_y^2}\mathbf{A}^{-1}\boldsymbol{\Phi}^T\boldsymbol{y}, \mathbf{A}^{-1}) \tag{18.59}$$

where $\boldsymbol{\Phi}$ is the $N \times D$ design matrix, and

$$\mathbf{A} = \sigma_y^{-2}\boldsymbol{\Phi}^\mathsf{T}\boldsymbol{\Phi} + \boldsymbol{\Sigma}_w^{-1} \tag{18.60}$$

The posterior predictive distribution for $f_* = f(\boldsymbol{x}_*)$ is therefore

$$p(f_*|\mathcal{D}, \boldsymbol{x}_*) = \mathcal{N}(f_*|\frac{1}{\sigma_y^2}\boldsymbol{\phi}_*^\mathsf{T}\mathbf{A}^{-1}\boldsymbol{\Phi}^\mathsf{T}\boldsymbol{y},\ \boldsymbol{\phi}_*^\mathsf{T}\mathbf{A}^{-1}\boldsymbol{\phi}_*) \tag{18.61}$$

where $\boldsymbol{\phi}_* = \boldsymbol{\phi}(\boldsymbol{x}_*)$. This views the problem of inference and prediction in **weight space**.

We now show that this is equivalent to the predictions made by a GP using a kernel of the form $\mathcal{K}(\boldsymbol{x}, \boldsymbol{x}') = \boldsymbol{\phi}(\boldsymbol{x})^\mathsf{T} \boldsymbol{\Sigma}_w \boldsymbol{\phi}(\boldsymbol{x}')$. To see this, let $\mathbf{K} = \boldsymbol{\Phi}\boldsymbol{\Sigma}_w\boldsymbol{\Phi}^\mathsf{T}$, $\boldsymbol{k}_* = \boldsymbol{\Phi}\boldsymbol{\Sigma}_w\boldsymbol{\phi}_*$, and $k_{**} = \boldsymbol{\phi}_*^\mathsf{T}\boldsymbol{\Sigma}_w\boldsymbol{\phi}_*$. Using this notation, and the matrix inversion lemma, we can rewrite Equation (18.61) as follows

$$p(f_*|\mathcal{D}, \boldsymbol{x}_*) = \mathcal{N}(f_*|\boldsymbol{\mu}_{*|X}, \boldsymbol{\Sigma}_{*|X}) \tag{18.62}$$

$$\boldsymbol{\mu}_{*|X} = \boldsymbol{\phi}_*^\mathsf{T}\boldsymbol{\Sigma}_w\boldsymbol{\Phi}^\mathsf{T}(\mathbf{K} + \sigma_y^2\mathbf{I})^{-1}\boldsymbol{y} = \boldsymbol{k}_*^\mathsf{T}(\mathbf{K}_{X,X} + \sigma_y\mathbf{I})^{-1}\boldsymbol{y} \tag{18.63}$$

$$\boldsymbol{\Sigma}_{*|X} = \boldsymbol{\phi}_*^\mathsf{T}\boldsymbol{\Sigma}_w\boldsymbol{\phi}_* - \boldsymbol{\phi}_*^\mathsf{T}\boldsymbol{\Sigma}_w\boldsymbol{\Phi}^\mathsf{T}(\mathbf{K} + \sigma_y^2\mathbf{I})^{-1}\boldsymbol{\Phi}\boldsymbol{\Sigma}_w\boldsymbol{\phi}_* = k_{**} - \boldsymbol{k}_*^\mathsf{T}(\mathbf{K}_{X,X} + \sigma_y^2\mathbf{I})^{-1}\boldsymbol{k}_* \tag{18.64}$$

which matches the results in Equation (18.55), assuming $m(\boldsymbol{x}) = 0$. A non-zero mean can be captured by adding a constant feature with value 1 to $\boldsymbol{\phi}(\boldsymbol{x})$.

Thus we can derive a GP from Bayesian linear regression. Note, however, that linear regression assumes $\boldsymbol{\phi}(\boldsymbol{x})$ is a finite length vector, whereas a GP allows us to work directly in terms of kernels, which may correspond to infinite length feature vectors (see Section 18.2.2). That is, a GP works in **function space**.

18.3.4 Semi-parametric GPs

So far, we have mostly assumed the mean of the GP is 0, and have relied on its interpolation abilities to model the mean function. Sometimes it is useful to fit a global linear model for the mean, and use the GP to model the residual errors, as follows:

$$g(\boldsymbol{x}) = f(\boldsymbol{x}) + \boldsymbol{\beta}^\mathsf{T}\boldsymbol{\phi}(\boldsymbol{x}) \tag{18.65}$$

where $f(\boldsymbol{x}) \sim \mathrm{GP}(0, \mathcal{K}(\boldsymbol{x}, \boldsymbol{x}'))$, and $\boldsymbol{\phi}()$ are some fixed basis functions. This combines a parametric and a non-parametric model, and is known as a **semi-parametric model**.

If we assume $\boldsymbol{\beta} \sim \mathcal{N}(\boldsymbol{b}, \mathbf{B})$, we can integrate these parameters out to get a new GP [O'H78]:

$$g(\boldsymbol{x}) \sim \mathrm{GP}\left(\boldsymbol{\phi}(\boldsymbol{x})^\mathsf{T}\boldsymbol{b},\ \mathcal{K}(\boldsymbol{x}, \boldsymbol{x}') + \boldsymbol{\phi}(\boldsymbol{x})^\mathsf{T}\mathbf{B}\boldsymbol{\phi}(\boldsymbol{x}')\right) \tag{18.66}$$

Let $\mathbf{H}_X = \boldsymbol{\phi}(\mathbf{X})^\mathsf{T}$ be the $D \times N$ matrix of training examples, and $\mathbf{H}_* = \boldsymbol{\phi}(\mathbf{X}_*)^\mathsf{T}$ be the $D \times N_*$ matrix of test examples. The corresponding predictive distribution for test inputs $\mathbf{X}_*$ has the following form [RW06, p28]:

$$\mathbb{E}\left[g(\mathbf{X}_*)|\mathcal{D}\right] = \mathbf{H}_*^\mathsf{T}\overline{\boldsymbol{\beta}} + \mathbf{K}_{X,*}^\mathsf{T}\mathbf{K}_\sigma^{-1}(\boldsymbol{y} - \mathbf{H}_X^\mathsf{T}\overline{\boldsymbol{\beta}}) = \mathbb{E}\left[f(\mathbf{X}_*)|\mathcal{D}\right] + \mathbf{R}^\mathsf{T}\overline{\boldsymbol{\beta}} \tag{18.67}$$

$$\mathrm{Cov}\left[g(\mathbf{X}_*)|\mathcal{D}\right] = \mathrm{Cov}\left[f(\mathbf{X}_*)|\mathcal{D}\right] + \mathbf{R}^\mathsf{T}(\mathbf{B}^{-1} + \mathbf{H}_X\mathbf{K}_\sigma^{-1}\mathbf{H}_X^\mathsf{T})^{-1}\mathbf{R} \tag{18.68}$$

$$\overline{\boldsymbol{\beta}} = (\mathbf{B}^{-1} + \mathbf{H}_X\mathbf{K}_\sigma^{-1}\mathbf{H}_X^\mathsf{T})^{-1}(\mathbf{H}_X\mathbf{K}_\sigma^{-1}\boldsymbol{y} + \mathbf{B}^{-1}\boldsymbol{b}) \tag{18.69}$$

$$\mathbf{R} = \mathbf{H}_* - \mathbf{H}_X\mathbf{K}_\sigma^{-1}\mathbf{K}_{X,*} \tag{18.70}$$

These results can be interpreted as follows: the mean is the usual mean from the GP , plus a global offset from the linear model, using $\overline{\boldsymbol{\beta}}$; and the covariance is the usual covariance from the GP, plus an additional positive term due to the uncertainty in $\boldsymbol{\beta}$.

In the limit of an uninformative prior for the regression parameters, as $\mathbf{B} \to \infty\mathbf{I}$, this simplifies to

$$\mathbb{E}\left[g(\mathbf{X}_*)|\mathcal{D}\right] = \mathbb{E}\left[f(\mathbf{X}_*)|\mathcal{D}\right] + \mathbf{R}^\mathsf{T}(\mathbf{H}_X\mathbf{K}_\sigma^{-1}\mathbf{H}_X^\mathsf{T})^{-1}\mathbf{H}_X\mathbf{K}_\sigma^{-1}\boldsymbol{y} \tag{18.71}$$

$$\mathrm{Cov}\left[g(\mathbf{X}_*)|\mathcal{D}\right] = \mathrm{Cov}\left[f(\mathbf{X}_*)|\mathcal{D}\right] + \mathbf{R}^\mathsf{T}(\mathbf{H}_X\mathbf{K}_\sigma^{-1}\mathbf{H}_X^\mathsf{T})^{-1}\mathbf{R} \tag{18.72}$$

18.3.5 Marginal likelihood

Most kernels have some free parameters. For example, the RBF-ARD kernel (Section 18.2.1.1) has the form

$$\mathcal{K}(\boldsymbol{x}, \boldsymbol{x}') = \exp\left(-\frac{1}{2}\sum_{d=1}^{D}\frac{1}{\ell_d^2}(x_d - x'_d)^2\right) = \prod_{d=1}^{D}\mathcal{K}_{\ell_d}(x_d, x'_d) \tag{18.73}$$

where each ℓ_d is a length scale for feature dimension d. Let these (and the observation noise variance σ_y^2, if present) be denoted by $\boldsymbol{\theta}$. We can compute the likelihood of these parameters as follows:

$$p(\boldsymbol{y}|\mathbf{X}, \boldsymbol{\theta}) = p(\mathcal{D}|\boldsymbol{\theta}) = \int p(\boldsymbol{y}|\boldsymbol{f}_X, \boldsymbol{\theta})p(\boldsymbol{f}_X|\mathbf{X}, \boldsymbol{\theta})d\boldsymbol{f}_X \tag{18.74}$$

Since we are integrating out the function f, we often call $\boldsymbol{\theta}$ hyperparameters, and the quantity $p(\mathcal{D}|\boldsymbol{\theta})$ the marginal likelihood.

Since f is a GP, we can compute the above integral using the marginal likelihood for the corresponding Gaussian. This gives

$$\log p(\mathcal{D}|\boldsymbol{\theta}) = -\frac{1}{2}(\boldsymbol{y} - \boldsymbol{\mu}_X)^\mathsf{T}\mathbf{K}_\sigma^{-1}(\boldsymbol{y} - \boldsymbol{\mu}_X) - \frac{1}{2}\log|\mathbf{K}_\sigma| - \frac{N}{2}\log(2\pi) \tag{18.75}$$

The first term term is the square of the Mahalanobis distance between the observations and the predicted values: better fits will have smaller distance. The second term is the log determinant of the covariance matrix, which measures model complexity: smoother functions will have smaller determinants, so $-\log|\mathbf{K}_\sigma|$ will be larger (less negative) for simpler functions. The marginal likelihood measures the tradeoff between fit and complexity.

In Section 18.6.1, we discuss how to learn the kernel parameters from data by maximizing the marginal likelihood wrt $\boldsymbol{\theta}$.

18.3.6 Computational and numerical issues

In this section, we discuss computational and numerical issues which arise when implementing the above equations. For notational simplicity, we assume the prior mean is zero, $m(\boldsymbol{x}) = 0$.

The posterior predictive mean is given by $\mu_* = \boldsymbol{k}_*^\mathsf{T}\mathbf{K}_\sigma^{-1}\boldsymbol{y}$. For reasons of numerical stability, it is unwise to directly invert $\mathbf{K}_\sigma$. A more robust alternative is to compute a Cholesky decomposition, $\mathbf{K}_\sigma = \mathbf{L}\mathbf{L}^\mathsf{T}$, which takes $O(N^3)$ time. Given this, we can compute

$$\mu_* = \boldsymbol{k}_*^\mathsf{T}\mathbf{K}_\sigma^{-1}\boldsymbol{y} = \boldsymbol{k}_*^\mathsf{T}\mathbf{L}^{-\mathsf{T}}(\mathbf{L}^{-1}\boldsymbol{y}) = \boldsymbol{k}_*^\mathsf{T}\boldsymbol{\alpha} \tag{18.76}$$

Here $\boldsymbol{\alpha} = \mathbf{L}^\mathsf{T} \setminus (\mathbf{L} \setminus \boldsymbol{y})$, where we have used the backslash operator to represent backsubstitution.

We can compute the variance in $O(N^2)$ time for each test case using

$$\sigma_*^2 = k_{**} - \boldsymbol{k}_*^\mathsf{T}\mathbf{L}^{-T}\mathbf{L}^{-1}\boldsymbol{k}_* = k_{**} - \boldsymbol{v}^\mathsf{T}\boldsymbol{v} \tag{18.77}$$

where $\boldsymbol{v} = \mathbf{L} \setminus \boldsymbol{k}_*$.

Finally, the log marginal likelihood (needed for kernel learning, Section 18.6) can be computed using

$$\log p(\boldsymbol{y}|\mathbf{X}) = -\frac{1}{2}\boldsymbol{y}^\mathsf{T}\boldsymbol{\alpha} - \sum_{n=1}^{N} \log L_{nn} - \frac{N}{2}\log(2\pi) \tag{18.78}$$

We see that overall cost is dominated by $O(N^3)$. We discuss faster, but approximate, methods in Section 18.5.

18.3.7 Kernel ridge regression

The term **ridge regression** refers to linear regression with an ℓ_2 penalty on the regression weights:

$$\boldsymbol{w}^* = \underset{\boldsymbol{w}}{\operatorname{argmin}} \sum_{n=1}^{N} (y_n - f(\boldsymbol{x}_n; \boldsymbol{w}))^2 + \lambda ||\boldsymbol{w}||_2^2 \tag{18.79}$$

where $f(\boldsymbol{x}; \boldsymbol{w}) = \boldsymbol{w}^\mathsf{T}\boldsymbol{x}$. The solution for this is

$$\boldsymbol{w}^* = (\mathbf{X}^\mathsf{T}\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}^\mathsf{T}\boldsymbol{y} = (\sum_{n=1}^{N} \boldsymbol{x}_n\boldsymbol{x}_n^\mathsf{T} + \lambda\mathbf{I})^{-1}(\sum_{n=1}^{N} \boldsymbol{x}_n y_n) \tag{18.80}$$

In this section, we consider a function space version of this:

$$f^* = \underset{f \in \mathcal{F}}{\operatorname{argmin}} \sum_{n=1}^{N} (y_n - f(\boldsymbol{x}_n))^2 + \lambda ||f||^2 \tag{18.81}$$

For this to make sense, we have to define the function space $\mathcal{F}$ and the norm $||f||$. If we use a function space derived from a positive definite kernel function $\mathcal{K}$, the resulting method is called **kernel ridge regression** (KRR). We will see that the resulting estimate $f^*(\boldsymbol{x}_*)$ is equivalent to the posterior mean of a GP. We give the details below.

18.3.7.1 Reproducing kernel Hilbert spaces

In this section, we briefly introduce the relevant mathematical "machinery" needed to explain KRR.

Let $\mathcal{F} = \{f : \mathcal{X} \to \mathbb{R}\}$ be a space of real-valued funcitons. Elements of this space (i.e., functions) can be added and scalar multiplied as if they were vectors. That is, if $f \in \mathcal{F}$ and $g \in \mathcal{F}$, then $\alpha f + \beta g \in \mathcal{F}$ for $\alpha, \beta \in \mathbb{R}$. We can also define an **inner product** for $\mathcal{F}$, which is a mapping $\langle f, g \rangle \in \mathbb{R}$ which satisfies the following:

$$\langle \alpha f_1 + \beta f_2, g \rangle = \alpha \langle f_1, g \rangle + \beta \langle f_2, g \rangle \tag{18.82}$$

$$\langle f, g \rangle = \langle g, f \rangle \tag{18.83}$$

$$\langle f, f \rangle \geq 0 \tag{18.84}$$

$$\langle f, f \rangle = 0 \text{ iff } f(x) = 0 \text{ for all } x \in \mathcal{X} \tag{18.85}$$

We define the norm of a function using

$$||f|| \triangleq \sqrt{\langle f, f \rangle} \tag{18.86}$$

A function space $\mathcal{H}$ with an inner product operator is called a **Hilbert space**. (We also require that the function space be complete, which means that every Cauchy sequence of functions $f_i \in \mathcal{H}$ has a limit that is also in $\mathcal{H}$.)

The most common Hilbert space is the space known as L^2. To define this, we need to specify a **measure** μ on the input space $\mathcal{X}$; this is a function that assigns any (suitable) subset A of $\mathcal{X}$ to a positive number, such as its volume. This can be defined in terms of the density function $w : \mathcal{X} \to \mathbb{R}$, as follows:

$$\mu(A) = \int_A w(x)dx \tag{18.87}$$

Thus we have $\mu(dx) = w(x)dx$. We can now define $L^2(\mathcal{X}, \mu)$ to be the space of functions $f : \mathcal{X} \to \mathbb{R}$ that satisfy

$$\int_{\mathcal{X}} f(x)^2 w(x)dx < \infty \tag{18.88}$$

This is known as the set of **square-integrable functions**. This space has an inner product defined by

$$\langle f, g \rangle = \int_{\mathcal{X}} f(x)g(x)w(x)dx \tag{18.89}$$

We define a **Reproducing Kernel Hilbert Space** or **RKHS** as follows. Let $\mathcal{H}$ be a Hilbert space of functions $f : \mathcal{X} \to \mathbb{R}$. We say that $\mathcal{H}$ is an RKHS endowed with inner product $\langle \cdot, \cdot \rangle_{\mathcal{H}}$ if there exists a (symmetric) **kernel function** $\mathcal{K} : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ with the following properties:

- For every $\boldsymbol{x} \in \mathcal{X}$, $\mathcal{K}(\boldsymbol{x}, \cdot) \in \mathcal{H}$.
- $\mathcal{K}$ satisfies the **reproducing property**:

$$\langle f(\cdot), \mathcal{K}(\cdot, \boldsymbol{x}') \rangle = f(\boldsymbol{x}') \tag{18.90}$$

The reason for the term "reproducing property" is as follows. Let $f(\cdot) = \mathcal{K}(\boldsymbol{x}, \cdot)$. Then we have that

$$\langle \mathcal{K}(\boldsymbol{x}, \cdot), \mathcal{K}(\cdot, \boldsymbol{x}') \rangle = \mathcal{K}(\boldsymbol{x}, \boldsymbol{x}') \tag{18.91}$$

18.3.7.2 Complexity of a function in an RKHS

The main utility of RKHS from the point of view of machine learning is that it allows us to define a notion of a function's "smoothness" or "complexity" in terms of its norm, as we now discuss.

Suppose we have a positive definite kernel function $\mathcal{K}$. From Mercer's theorem we have $\mathcal{K}(\boldsymbol{x}, \boldsymbol{x}') = \sum_{i=1}^{\infty} \lambda_i \phi_i(\boldsymbol{x}) \phi_i(\boldsymbol{x}')$. Now consider a Hilbert space $\mathcal{H}$ defined by functions of the form $f(\boldsymbol{x}) = \sum_{i=1}^{\infty} f_i \phi(\boldsymbol{x})$, with $\sum_{i=1}^{\infty} f_i^2/\lambda < \infty$. The inner product of two functions in this space is

$$\langle f, g \rangle_{\mathcal{H}} = \sum_{i=1}^{\infty} \frac{f_i g_i}{\lambda_i} \tag{18.92}$$

Hence the (squared) norm is given by

$$||f||_{\mathcal{H}}^2 = \langle f, f \rangle_{\mathcal{H}} = \sum_{i=1}^{\infty} \frac{f_i^2}{\lambda_i} \tag{18.93}$$

This is analogous to the quadratic form $\boldsymbol{f}^{\mathsf{T}}\mathbf{K}^{-1}\boldsymbol{f}$ which occurs in some GP objectives (see Equation (18.102)). Thus the smoothness of the function is controlled by the properties of the corresponding kernel.

18.3.7.3 Representer theorem

In this section, we consider the problem of (regularized) empirical risk minimization in function space. In particular, consider the following problem:

$$f^* = \underset{f \in \mathcal{H}_{\mathcal{K}}}{\operatorname{argmin}} \sum_{n=1}^{N} \ell(y_n, f(\boldsymbol{x}_n)) + \frac{\lambda}{2}||f||_{\mathcal{H}}^2 \tag{18.94}$$

where $\mathcal{H}_{\mathcal{K}}$ is an RKHS with kernel $\mathcal{K}$ and $\ell(y, \hat{y}) \in \mathbb{R}$ is a loss function. Then one can show [KW70; SHS01] the following result:

$$f^*(x) = \sum_{n=1}^{N} \alpha_n \mathcal{K}(x, x_n) \tag{18.95}$$

where $\alpha_n \in \mathbb{R}$ are some coefficients that depend on the training data. This is called the **representer theorem**.

Now consider the special case where the loss function is squared loss, and $\lambda = \sigma_y^2$. We want to minimize

$$\mathcal{L}(f) = \frac{1}{2\sigma_y^2} \sum_{n=1}^{N} (y_n - f(\boldsymbol{x}_n))^2 + \frac{1}{2}||f||_{\mathcal{H}}^2 \tag{18.96}$$

Substituting in Equation (18.95), and using the fact that $\langle \mathcal{K}(\cdot, \boldsymbol{x}_i), \mathcal{K}(\cdot, \boldsymbol{x}_j) \rangle = \mathcal{K}(\boldsymbol{x}_i, \boldsymbol{x}_j)$, we obtain

$$\mathcal{L}(f) = \frac{1}{2}\boldsymbol{\alpha}^{\mathsf{T}}\mathbf{K}\boldsymbol{\alpha} + \frac{1}{2\sigma_y^2}||\boldsymbol{y} - \mathbf{K}\boldsymbol{\alpha}||^2 \tag{18.97}$$

$$= \frac{1}{2}\boldsymbol{\alpha}^{\mathsf{T}}(\mathbf{K} + \frac{1}{\sigma_y^2}\mathbf{K}^2)\boldsymbol{\alpha} - \frac{1}{\sigma_y^2}\boldsymbol{y}^{\mathsf{T}}\mathbf{K}\boldsymbol{\alpha} + \frac{1}{2\sigma_y^2}\boldsymbol{y}^{\mathsf{T}}\boldsymbol{y} \tag{18.98}$$

Minimizing this wrt $\boldsymbol{\alpha}$ gives $\hat{\boldsymbol{\alpha}} = (\mathbf{K} + \sigma_y^2\mathbf{I})^{-1}\boldsymbol{y}$, which is the same as Equation (18.58). Furthermore, the prediction for a test point is

$$\hat{f}(\boldsymbol{x}_*) = \boldsymbol{k}_*^{\mathsf{T}}\boldsymbol{\alpha} = \boldsymbol{k}_*^{\mathsf{T}}(\mathbf{K} + \sigma_y^2\mathbf{I})^{-1}\boldsymbol{y} \tag{18.99}$$

This is known as **kernel ridge regression** [Vov13]. We see that the result matches the posterior predictive mean of a GP in Equation (18.56).

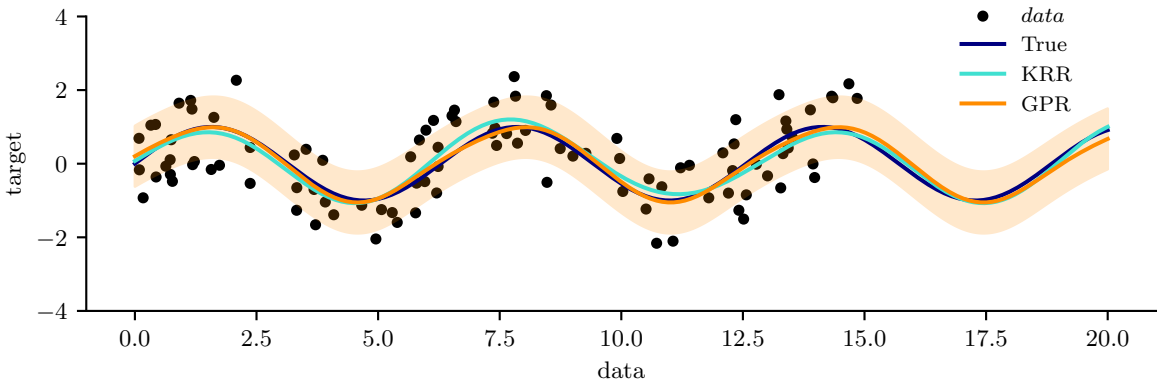


Figure 18.8: Kernel ridge regression (KRR) compared to Gaussian process regression (GPR) using the same kernel. Generated by krr_vs_gpr.ipynb.

18.3.7.4 Example of KRR vs GPR

In this section, we compare KRR with GP regression on a simple 1d problem. Since the underlying function is believed to be periodic, we use the periodic kernel from Equation (18.19). To capture the fact that the observations are noisy, we add to this a **white noise kernel**

$$\mathcal{K}(\boldsymbol{x}, \boldsymbol{x}') = \sigma_y^2 \delta(\boldsymbol{x} - \boldsymbol{x}') \tag{18.100}$$

as in Equation (18.49). Thus there are 3 GP hyper-parameters: the kernel length scale ℓ, the kernel periodicity p, and the noise level σ_y^2. We can optimize these by maximizing the marginal likelihood using gradient descent (see Section 18.6.1). For KRR, we also have 3 hyper-parameters (ℓ, p and $\lambda = \sigma_y^2$); we optimize these using grid search combined with cross validation (which in general is slower than gradient based optimization). The resulting model fits are shown in Figure 18.8, and are very similar, as is to be expected.

18.4 GPs with non-Gaussian likelihoods

So far, we have focused on GPs for regression using Gaussian likelihoods. In this case, the posterior is also a GP, and all computation can be performed analytically. However, if the likelihood is non-Gaussian, we can no longer compute the posterior exactly. We can create variety of different "classical" models by changing the form of the likelihood, as we show in Table 18.1. In the sections below, we briefly discuss some approximate inference methods.

18.4.1 Binary classification

In this section, we consider binary classification using GPs. If we use the sigmoid link function, we have $p(y_n = 1|\boldsymbol{x}_n) = \sigma(y_n f(\boldsymbol{x}_n))$. If we assume $y_n \in \{-1, +1\}$, then we have $p(y_n|\boldsymbol{x}_n) = \sigma(y_n f_n)$, since $\sigma(-z) = 1 - \sigma(z)$. If we use the probit link, we have $p(y_n = 1|\boldsymbol{x}_n) = \Phi(y_n f(\boldsymbol{x}_n))$, where $\Phi(z)$ is the cdf of the standard normal. More generally, let $p(y_n|\boldsymbol{x}_n) = \text{Ber}(y_n|\varphi(f_n))$. The overall log joint

Model	Likelihood	Section
Regression	$\mathcal{N}(f_i, \sigma_y^2)$	Section 18.3.2
Robust regression	$\mathcal{T}_\nu(f_i, \sigma_y^2)$	Section 18.4.4
Binary classification	$\text{Ber}(\sigma(f_i))$	Section 18.4.1
Multiclass classification	$\text{Cat}(\text{softmax}(\boldsymbol{f}_i))$	Section 18.4.2
Poisson regression	$\text{Poi}(\exp(f_i))$	Section 18.4.3

Table 18.1: Summary of GP models with a variety of likelihoods.

$\log p(y_i\|f_i)$	$\frac{\partial}{\partial f_i}\log p(y_i\|f_i)$	$\frac{\partial^2}{\partial f_i^2}\log p(y_i\|f_i)$
$\log \sigma(y_i f_i)$	$t_i - \pi_i$	$-\pi_i(1-\pi_i)$
$\log \Phi(y_i f_i)$	$\frac{y_i \phi(f_i)}{\Phi(y_i f_i)}$	$-\frac{\phi_i^2}{\Phi(y_i f_i)^2} - \frac{y_i f_i \phi(f_i)}{\Phi(y_i f_i)}$

Table 18.2: Likelihood, gradient and Hessian for binary logistic/ probit GP regression. We assume $y_i \in \{-1, +1\}$ and define $t_i = (y_i + 1)/2 \in \{0, 1\}$ and $\pi_i = \sigma(f_i)$ for logistic regression, and $\pi_i = \Phi(f_i)$ for probit regression. Also, ϕ and Φ are the pdf and cdf of $\mathcal{N}(0, 1)$. From [RW06, p43].

has the form

$$\mathcal{L}(\boldsymbol{f}_X) = \log p(\boldsymbol{y}|\boldsymbol{f}_X) + \log p(\boldsymbol{f}_X|\mathbf{X}) \tag{18.101}$$

$$= \log p(\boldsymbol{y}|\boldsymbol{f}_X) - \frac{1}{2}\boldsymbol{f}_X^\mathsf{T}\mathbf{K}_{X,X}^{-1}\boldsymbol{f}_X - \frac{1}{2}\log|\mathbf{K}_{X,X}| - \frac{N}{2}\log 2\pi \tag{18.102}$$

The simplest approach to approximate inference is to use a Laplace approximation (Section 7.4.3). The gradient and Hessian of the log joint are given by

$$\nabla\mathcal{L} = \nabla \log p(\boldsymbol{y}|\boldsymbol{f}_X) - \mathbf{K}_{X,X}^{-1}\boldsymbol{f}_X \tag{18.103}$$

$$\nabla^2\mathcal{L} = \nabla^2 \log p(\boldsymbol{y}|\boldsymbol{f}_X) - \mathbf{K}_{X,X}^{-1} = -\boldsymbol{\Lambda} - \mathbf{K}_{X,X}^{-1} \tag{18.104}$$

where $\boldsymbol{\Lambda} \triangleq -\nabla^2 \log p(\boldsymbol{y}|\boldsymbol{f}_X)$ is a diagonal matrix, since the likelihood factorizes across examples. Expressions for the gradient and Hessian of the log likelihood for the logit and probit case are shown in Table 18.2. At convergence, the Laplace approximation of the posterior takes the following form:

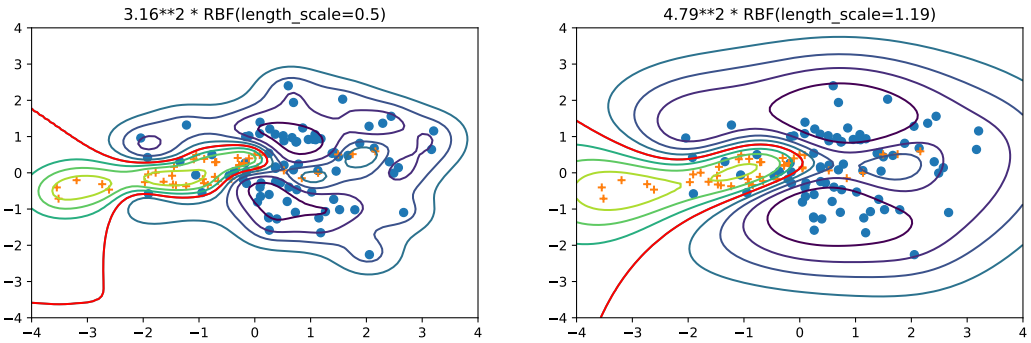
$$p(\boldsymbol{f}_X|\mathcal{D}) \approx q(\boldsymbol{f}_X) = \mathcal{N}(\hat{\boldsymbol{f}}, (\mathbf{K}_{X,X}^{-1} + \boldsymbol{\Lambda})^{-1}) \tag{18.105}$$

where $\hat{\boldsymbol{f}}$ is the MAP estimate. See [RW06, Sec 3.4] for further details.

For improved accuracy, we can use variational inference, in which we assume $q(\boldsymbol{f}_X) = \mathcal{N}(\boldsymbol{f}_X|\boldsymbol{m}, \mathbf{S})$; we then optimize $\boldsymbol{m}$ and $\mathbf{S}$ using (stochastic) gradient descent, rather than assuming $\mathbf{S}$ is the Hessian at the mode. See Section 18.5.4 for the details.

Once we have a Gaussian posterior $q(\boldsymbol{f}_X|\mathcal{D})$, we can then use standard GP prediction to compute $q(f_*|\boldsymbol{x}_*, \mathcal{D})$. Finally, we can approximate the posterior predictive distribution over binary labels using

$$\pi_* = p(y_* = 1|\boldsymbol{x}_*, \mathcal{D}) = \int p(y_* = 1|f_*) q(f_*|\boldsymbol{x}_*, \mathcal{D}) df_* \tag{18.106}$$



(a) (b)

Figure 18.9: Contours of the posterior predictive probability for a binary classifier generated by a GP with an SE kernel. (a) Manual kernel parameters: short length scale, $\ell = 0.5$, variance $3.16^2 \approx 9.98$. (b) Learned kernel parameters: long length scale, $\ell = 1.19$, variance $4.79^2 \approx 22.9$. Generated by gpc_demo_2d.ipynb.

This 1d integral can be computed using the probit approximation from Section 15.3.5. In this case we have $\pi_* \approx \sigma(\kappa(v)\mathbb{E}[f_*])$, where $v = \mathbb{V}[f_*]$ and $\kappa^2(v) = (1 + \pi v/8)^{-1}$.

In Figure 18.9, we show a synthetic binary classification problem in 2d. We use an SE kernel. On the left, we show predictions using hyper-parameters set by hand; we use a short length scale, hence the very sharp turns in the decision boundary. On the right, we show the predictions using the learned hyper-parameters; the model favors more parsimonious explanation of the data.

18.4.2 Multi-class classification

The multi-class case is somewhat harder, since the function now needs to return a vector of C logits to get $p(y_n|\boldsymbol{x}_n) = \text{Cat}(y_n|\text{softmax}(\boldsymbol{f}_n))$, where $\boldsymbol{f}_n = (f_n^1, \ldots, f_n^C)$, It is standard to assume that $f^c \sim \text{GP}(0, \mathcal{K}_c)$. Thus we have one latent function per class, which are a priori independent, and which may use different kernels.

We can derive a Laplace approximation for this model as discussed in [RW06, Sec 3.5]. Alternatively, we can use a variational approach, using the local variational bound to the multinomial softmax in [Cha12]. An alternative variational method, based on data augmentation with auxiliary variables, is described in [Wen+19b; Liu+19a; GFWO20].

18.4.3 GPs for Poisson regression (Cox process)

In this section, we illustrate Poisson regression where the underlying log rate function is modeled by a GP. This is known as a **Cox process**. We can perform approximate posterior inference in this model using Laplace, MCMC or SVI (stochastic variational inference). In Figure 18.10 we give a 1d example, where we use a Matern $\frac{5}{2}$ kernel. We apply MCMC and SVI. In the VI case, we additionally have to specify the form of the posterior; we use a Gaussian approximation for the variational GP posterior $p(\boldsymbol{f}|\mathbf{X}, \boldsymbol{y})$, and a point estimate for the kernel parameters.

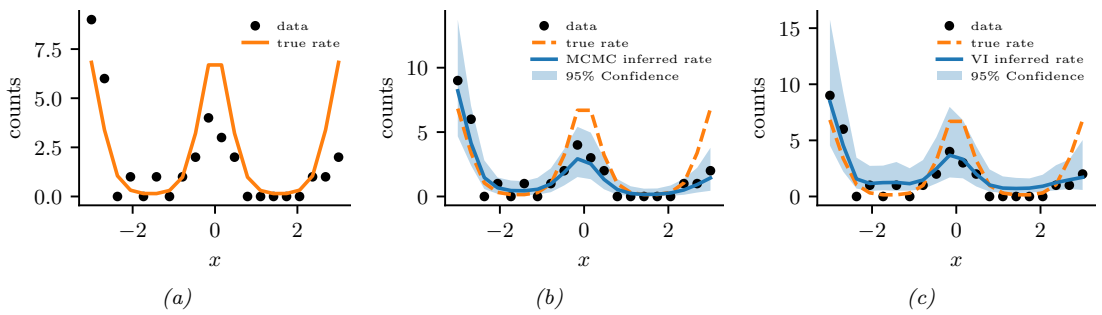


Figure 18.10: Poisson regression with a GP. (a) Observed data (black dots) and true log rate function (yellow line). (b) Posterior predictive distribution (shading shows 1 and 2 σ bands) from MCMC. (c) Posterior predictive distribution from SVI. Generated by gp_poisson_1d.ipynb.

An interesting application of this is to spatial **disease mapping**. For example, [VPV10] discuss the problem of modeling the relative risk of heart attack in different regions in Finland. The data consists of the heart attacks in Finland from 1996-2000 aggregated into 20km x 20km lattice cells. The likelihood has the following form: $y_n \sim \text{Poi}(e_n r_n)$, where e_n is the known expected number of deaths (related to the population of cell n and the overall death rate), and r_n is the **relative risk** of cell n which we want to infer. Since the data counts are small, we regularize the problem by sharing information with spatial neighbors. Hence we assume $f \triangleq \log(r) \sim \text{GP}(0, \mathcal{K})$. We use a Matern kernel (Section 18.2.1.1) with $\nu = 3/2$, and a length scale and magnitude that are estimated from data.

Figure 18.11 gives an example of this method in action (using Laplace approximation). On the left we plot the posterior mean relative risk (RR), and on the right, the posterior variance. We see that the RR is higher in Eastern Finland, which is consistent with other studies. We also see that the variance in the North is higher, since there are fewer people living there.

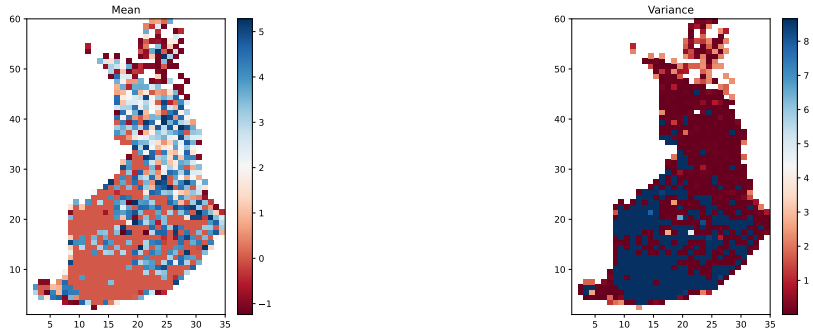
18.4.4 Other likelihoods

Many other likelihoods are possible. For example, [VJV09] uses a Student-t likelihood in order to perform robust regression. A general method for performing approximate variational inference in GPs with such non-conjugate likelihoods is discussed in [WSS21].

18.5 Scaling GP inference to large datasets

In Section 18.3.6, we saw that the best way to perform GP inference and training is to compute a Cholesky decomposition of the $N \times N$ Gram matrix. Unfortunately, this takes $O(N^3)$ time. In this section, we discuss methods to scale up GPs to handle large N. See Table 18.3 for a summary, and [Liu+20c] for more details.[1]

1. We focus on efficient methods for evaluating the marginal likelihood and the posterior predictive distribution. For an efficient method for sampling a function from the posterior, see [Wil+20a].



(a) *(b)*

Figure 18.11: We show the relative risk of heart disease in Finland using a Poisson GP fit to 911 data points. Left: posterior mean. Right: posterior variance. Generated by gp_spatial_demo.ipynb.

Method	Cost	Section
Cholesky	$O(N^3)$	Section 18.3.6
Conj. Grad.	$O(CN^2)$	Section 18.5.5
Inducing	$O(NM^2 + M^3 + DNM)$	Section 18.5.3
Variational	$O(NM^2 + M^3 + DNM)$	Section 18.5.4
SVGP	$O(BM^2 + M^3 + DNM)$	Section 18.5.4.3
KISS-GP	$O(CN + CDM^D \log M)$	Section 18.5.5.3
SKIP	$O(DLN + DLM \log M + L^3 N \log D + CL^2 N)$	Section 18.5.5.3

Table 18.3: Summary of time to compute the log marginal likelihood of a GP regression model. Notation: N is number of training examples, M is number of inducing points, B is size of minibatch, D is dimensionality of input vectors (assuming $\mathcal{X} = \mathbb{R}^D$), C is number of conjugate gradient iterations. L is number of Lanczos iterations. Based on Table 2 of [Gar+18a].

18.5.1 Subset of data

The simplest approach to speeding up GP inference is to throw away some of the data. Suppose we keep a subset of M examples. In this case, exact inference will take $O(M^3)$ time. This is called the **susbet-of-data** approach.

The key question is: how should we choose the subset? The simplest approach is to pick random examples (this method was recently analysed in [HIY19]). However, intuitively it makes more sense to try to pick a subset that in some sense "covers" the original data, so it contains approximately the same information (up to some tolerance) without the redundancy. Clustering algorithms are one heuristic approach, but we can also use coreset methods, which can provably find such an information-preserving subset (see e.g., [Hug+19] for an application of this idea to GPs).

18.5.1.1 Informative vector machine

Clustering and coreset methods are unsupervised, in that they only look at the features $\boldsymbol{x}_i$ and not the labels y_i, which can be suboptimal. The **informative vector machine** [HLS03] uses a greedy strategy to iteratively add the labeled example $(\boldsymbol{x}_j, y_j)$ that maximally reduces the entropy of the function's posterior, $\Delta_j = \mathbb{H}(p(f_j)) - \mathbb{H}(p^{\text{new}}(f_j))$, where $p^{\text{new}}(f_j)$ is the posterior of f at $\boldsymbol{x}_j$ after conditioning on y_j. (This is very similar to active learning.) To compute Δ_j, let $p(f_j) = \mathcal{N}(\mu_j, v_j)$, and $p(f_j|y_j) \propto p(f_j)\mathcal{N}(y_j|f_j, \sigma^2) = \mathcal{N}(f_j|\mu_j^{\text{new}}, v_j^{\text{new}})$, where $(v_j^{\text{new}})^{-1} = v_j^{-1} + \sigma^{-2}$. Since $\mathbb{H}(\mathcal{N}(\mu, v)) = \log(2\pi ev)/2$, we have $\Delta_j = 0.5\log(1 + v_j/\sigma^2)$. Since this is a monotonic function of v_j, we can maximize it by choosing the site with the largest variance. (In fact, entropy is a submodular function, so we can use submodular optimization algorithms to improve on the IVM, as shown in [Kra+08].)

18.5.1.2 Discussion

The main problem with the subset of data approach is that it ignores some of the data, which can reduce predictive accuracy and increase uncertainty about the true function. Fortunately there are other scalable methods that avoid this problem, essentially by approximately representing (or compressing) the training data, as we discuss below.

18.5.2 Nyström approximation

Suppose we had a rank M approximation to the $N \times N$ matrix gram matrix of the following form:

$$\mathbf{K}_{X,X} \approx \mathbf{U}\boldsymbol{\Lambda}\mathbf{U}^{\mathsf{T}} \tag{18.107}$$

where $\boldsymbol{\Lambda}$ is a diagonal matrix of the M leading eigenvalues, and $\mathbf{U}$ is the matrix of the corresponding M eigenvectors, each of size N. In this case, we can use the matrix inversion lemma to write

$$\mathbf{K}_\sigma^{-1} = (\mathbf{K}_{X,X} + \sigma^2\mathbf{I}_N)^{-1} \approx \sigma^{-2}\mathbf{I}_N + \sigma^{-2}\mathbf{U}(\sigma^2\boldsymbol{\Lambda}^{-1} + \mathbf{U}^{\mathsf{T}}\mathbf{U})^{-1}\mathbf{U}^{\mathsf{T}} \tag{18.108}$$

which takes $O(NM^2)$ time. Similarly, one can show (using the Sylvester determinant lemma) that

$$|\mathbf{K}_\sigma| \approx |\boldsymbol{\Lambda}||\sigma^2\boldsymbol{\Lambda}^{-1} + \mathbf{U}^{\mathsf{T}}\mathbf{U}| \tag{18.109}$$

which also takes $O(NM^2)$ time.

Unfortunately, directly computing such an eigendecomposition takes $O(N^3)$ time, which does not help. However, suppose we pick a subset Z of $M < N$ points. We can partition the Gram matrix as follows (where we assume the chosen points come first, and then the remaining points):

$$\mathbf{K}_{X,X} = \begin{pmatrix} \mathbf{K}_{Z,Z} & \mathbf{K}_{Z,X-Z} \\ \mathbf{K}_{X-Z,Z} & \mathbf{K}_{X-Z,X-Z} \end{pmatrix} \tag{18.110}$$

Let $\mathbf{K}_{Z,X}$ denote the top $M \times N$ block, and $\mathbf{K}_{X,Z}$ denote its transpose. We now compute an eigendecomposition of $\mathbf{K}_{Z,Z}$ to get the eigenvalues $\{\lambda_i\}_{i=1}^M$ and eigenvectors $\{\boldsymbol{u}_i\}_{i=1}^M$. We now use these to approximate the full matrix as shown below, where the scaling constants are chosen so that

$\|\tilde{\boldsymbol{u}}_i\| \approx 1$:

$$\tilde{\lambda}_i \triangleq \frac{N}{M}\lambda_i \tag{18.111}$$

$$\tilde{\boldsymbol{u}} \triangleq \sqrt{\frac{M}{N}}\frac{1}{\lambda_i}\mathbf{K}_{X,Z}\boldsymbol{u}_i \tag{18.112}$$

$$\mathbf{K}_{X,X} \approx \sum_{i=1}^{M} \tilde{\lambda}_i \tilde{\boldsymbol{u}}_i \tilde{\boldsymbol{u}}_i^{\mathsf{T}} \tag{18.113}$$

$$= \sum_{i=1}^{M} \frac{N}{M}\lambda_i \sqrt{\frac{M}{N}}\frac{1}{\lambda_i}\mathbf{K}_{X,Z}\boldsymbol{u}_i \;\; \sqrt{\frac{M}{N}}\frac{1}{\lambda_i}\boldsymbol{u}_i^{\mathsf{T}}\mathbf{K}_{X,Z}^{\mathsf{T}} \tag{18.114}$$

$$= \mathbf{K}_{X,Z}\left(\sum_{i=1}^{M}\frac{1}{\lambda_i}\boldsymbol{u}_i\boldsymbol{u}_i^{\mathsf{T}}\right)\mathbf{K}_{Z,X} \tag{18.115}$$

$$= \mathbf{K}_{X,Z}\mathbf{K}_{Z,Z}^{-1}\mathbf{K}_{Z,X} \tag{18.116}$$

This is known as the **Nyström approximation** [WS01]. If we define

$$\mathbf{Q}_{A,B} \triangleq \mathbf{K}_{A,Z}\mathbf{K}_{Z,Z}^{-1}\mathbf{K}_{Z,B} \tag{18.117}$$

then we can write the approximate Gram matrix as $\mathbf{Q}_{X,X}$. We can then replace $\mathbf{K}_\sigma$ with $\hat{\mathbf{Q}}_{X,X} = \mathbf{Q}_{X,X} + \sigma^2\mathbf{I}_N$. Computing the eigendecomposition takes $O(M^3)$ time, and computing $\hat{\mathbf{Q}}_{X,X}^{-1}$ takes $O(NM^2)$ time. Thus complexity is now linear in N instead of cubic.

If we are approximating *only* $\hat{\mathbf{K}}_{X,X}$ in $\boldsymbol{\mu}_{*|X}$ in Equation (18.53) and $\boldsymbol{\Sigma}_{*|X}$ in Equation (18.54) $\hat{\mathbf{Q}}_{X,X}$, then this is inconsistent with the other un-approximated kernel function evaluations in these formulae, and can result in the predictive variance being negative. One solution to this is to use the same $\mathbf{Q}$ approximation for all terms.

18.5.3 Inducing point methods

In this section, we discuss an approximation method based on **inducing points**, also called **pseudo inputs**, which are like a learned summary of the training data that we can condition on, rather than conditioning on all of it.

Let $\mathbf{X}$ be the observed inputs, and $\boldsymbol{f}_X = f(\mathbf{X})$ be the unknown vector of function values (for which we have noisy observations $\boldsymbol{y}$). Let $\boldsymbol{f}_*$ be the unknown function values at one or more test points $\mathbf{X}_*$. Finally, let us assume we have M additional inputs, $\mathbf{Z}$, with unknown function values $\boldsymbol{f}_Z$ (often denoted by $\boldsymbol{u}$). The exact joint prior has the form

$$p(\boldsymbol{f}_X, \boldsymbol{f}_*) = \int p(\boldsymbol{f}_*, \boldsymbol{f}_X, \boldsymbol{f}_Z)d\boldsymbol{f}_Z = \int p(\boldsymbol{f}_*, \boldsymbol{f}_X|\boldsymbol{f}_Z)p(\boldsymbol{f}_Z)d\boldsymbol{f}_Z = \mathcal{N}\left(\mathbf{0}, \begin{pmatrix}\mathbf{K}_{X,X} & \mathbf{K}_{X,*} \\ \mathbf{K}_{*,X} & \mathbf{K}_{*,*}\end{pmatrix}\right) \tag{18.118}$$

(We write $p(\boldsymbol{f}_X, \boldsymbol{f}_*)$ instead of $p(\boldsymbol{f}_X, \boldsymbol{f}_*|\mathbf{X}, \mathbf{X}_*)$, since the inputs can be thought of as just indices into the random function f.)

We will choose $\boldsymbol{f}_Z$ in such a way that it acts as a sufficient statistic for the data, so that we can predict $\boldsymbol{f}_*$ just using $\boldsymbol{f}_Z$ instead of $\boldsymbol{f}_X$, i.e., we assume $\boldsymbol{f}_* \perp \boldsymbol{f}_X|\boldsymbol{f}_Z$. Thus we approximate the prior

Figure 18.12: Illustration of the graphical model for a GP on n observations, $\boldsymbol{f}_{1:n}$, and one test case, f_*, with inducing variables $\boldsymbol{u}$. The thick lines indicate that all variables are fully interconnected. The observations y_i (not shown) are locally connected to each f_i. (a) no approximations are made. (b) we assume f_* is conditionally independent of $\boldsymbol{f}_X$ given $\boldsymbol{u}$. From Figure 1 of [QCR05]. Used with kind permission of Joaquin Quinonero-Candela.

as follows:

$$p(\boldsymbol{f}_*, \boldsymbol{f}_X, \boldsymbol{f}_Z) = p(\boldsymbol{f}_*|\boldsymbol{f}_X, \boldsymbol{f}_Z)p(\boldsymbol{f}_X|\boldsymbol{f}_Z)p(\boldsymbol{f}_Z) \approx p(\boldsymbol{f}_*|\boldsymbol{f}_Z)p(\boldsymbol{f}_X|\boldsymbol{f}_Z)p(\boldsymbol{f}_Z) \tag{18.119}$$

See Figure 18.12 for an illustration of this assumption, and Section 18.5.3.4 for details on how to choose the inducing set $\mathbf{Z}$. (Note that this method is often called a "**sparse GP**", because it makes predictions for $\boldsymbol{f}_*$ using a subset of the training data, namely $\boldsymbol{f}_Z$, instead of all of it, $\boldsymbol{f}_X$.)

From this, we can derive the following train and test conditionals

$$p(\boldsymbol{f}_X|\boldsymbol{f}_Z) = \mathcal{N}(\boldsymbol{f}_X|\mathbf{K}_{X,Z}\mathbf{K}_{Z,Z}^{-1}\boldsymbol{f}_Z, \mathbf{K}_{X,X} - \mathbf{Q}_{X,X}) \tag{18.120}$$

$$p(\boldsymbol{f}_*|\boldsymbol{f}_Z) = \mathcal{N}(\boldsymbol{f}_*|\mathbf{K}_{*,Z}\mathbf{K}_{Z,Z}^{-1}\boldsymbol{f}_Z, \mathbf{K}_{*,*} - \mathbf{Q}_{*,*}) \tag{18.121}$$

The above equations can be seen as exact inference on noise-free observations $\boldsymbol{f}_Z$. To gain computational speedups, we will make further approximations of the form $\tilde{\mathbf{Q}}_{X,X} \approx \mathbf{K}_{X,X} - \mathbf{Q}_{X,X}$ and $\tilde{\mathbf{Q}}_{*,*} \approx \mathbf{K}_{*,*} - \mathbf{Q}_{*,*}$. We consider various choices for these values below. All of these choices result in an initial training cost of $O(M^3 + NM^2)$, and then take $O(M)$ time for the predictive mean for each test case, and $O(M^2)$ time for the predictive variance. (Compare this to $O(N^3)$ training time and $O(N)$ and $O(N^2)$ testing time for exact inference.)

18.5.3.1 SOR/ DIC

Suppose we assume $\tilde{\mathbf{Q}}_{X,X} = \mathbf{0}$ and $\tilde{\mathbf{Q}}_{*,*} = \mathbf{0}$, so the conditionals are deterministic. This is called the **deterministic inducing conditional** (**DIC**) approximation [QCR05], or the **subset of regressors** (**SOR**) approximation [Sil85; SB01]. The corresponding joint prior has the form

$$q_{\text{SOR}}(\boldsymbol{f}_X, \boldsymbol{f}_*) = \mathcal{N}(\mathbf{0}, \begin{pmatrix} \mathbf{Q}_{X,X} & \mathbf{Q}_{X,*} \\ \mathbf{Q}_{*,X} & \mathbf{Q}_{*,*} \end{pmatrix}) \tag{18.122}$$

Consequently the predictive distribution is

$$q_{\text{SOR}}(\boldsymbol{f}_*|\boldsymbol{y}) = \mathcal{N}(\boldsymbol{f}_*|\mathbf{Q}_{*,X}\hat{\mathbf{Q}}_{X,X}^{-1}\boldsymbol{y}, \mathbf{Q}_{*,*} - \mathbf{Q}_{*,X}\hat{\mathbf{Q}}_{X,X}^{-1}\mathbf{Q}_{X,*}) \tag{18.123}$$

$$= \mathcal{N}(\boldsymbol{f}_*|\sigma^{-2}\mathbf{K}_{*,Z}\boldsymbol{\Sigma}\mathbf{K}_{Z,X}\boldsymbol{y}, \mathbf{K}_{*,Z}\boldsymbol{\Sigma}\mathbf{K}_{Z,*}) \tag{18.124}$$

where we have defined $\hat{\mathbf{Q}}_{X,X} = \mathbf{Q}_{X,X} + \sigma^2 \mathbf{I}_N$, and $\mathbf{\Sigma} = (\sigma^{-2}\mathbf{K}_{Z,X}\mathbf{K}_{X,Z} + \mathbf{K}_{Z,Z})^{-1}$.

This predictive distribution is equivalent to the usual one for GPs except we have replaced $\mathbf{K}_{X,X}$ by $\mathbf{Q}_{X,X}$. This is equivalent to performing GP inference with the following kernel function

$$\mathcal{K}_{\text{SOR}}(\boldsymbol{x}_i, \boldsymbol{x}_j) = \mathcal{K}(\boldsymbol{x}_i, \mathbf{Z})\mathbf{K}_{Z,Z}^{-1}\mathcal{K}(\mathbf{Z}, \boldsymbol{x}_j) \tag{18.125}$$

The kernel matrix has rank M, so the GP is degenerate. Furthermore, the kernel will be near 0 when $\boldsymbol{x}_i$ or $\boldsymbol{x}_j$ is far from one of the chosen points $\mathbf{Z}$, which can result in an underestimate of the predictive variance.

18.5.3.2 DTC

One way to overcome the overconfidence of DIC is to only assume $\tilde{\mathbf{Q}}_{X,X} = \mathbf{0}$, but let $\tilde{\mathbf{Q}}_{*,*} = \mathbf{K}_{*,*} - \mathbf{Q}_{*,*}$ be exact. This is called the **deterministic training conditional** or **DTC** method [SWL03].

The corresponding joint prior has the form

$$q_{\text{dtc}}(\boldsymbol{f}_X, \boldsymbol{f}_*) = \mathcal{N}(\mathbf{0}, \begin{pmatrix} \mathbf{Q}_{X,X} & \mathbf{Q}_{X,*} \\ \mathbf{Q}_{*,X} & \mathbf{K}_{*,*} \end{pmatrix}) \tag{18.126}$$

Hence the predictive distribution becomes

$$q_{\text{dtc}}(\boldsymbol{f}_*|\boldsymbol{y}) = \mathcal{N}(\boldsymbol{f}_*|\mathbf{Q}_{*,X}\hat{\mathbf{Q}}_{X,X}^{-1}\boldsymbol{y}, \mathbf{K}_{*,*} - \mathbf{Q}_{*,X}\hat{\mathbf{Q}}_{X,X}^{-1}\mathbf{Q}_{X,*}) \tag{18.127}$$

$$= \mathcal{N}(\boldsymbol{f}_*|\sigma^{-2}\mathbf{K}_{*,Z}\mathbf{\Sigma}\mathbf{K}_{Z,X}\boldsymbol{y}, \mathbf{K}_{*,*} - \mathbf{Q}_{*,*} + \mathbf{K}_{*,Z}\mathbf{\Sigma}\mathbf{K}_{Z,*}) \tag{18.128}$$

The predictive mean is the same as in SOR, but the variance is larger (since $\mathbf{K}_{*,*} - \mathbf{Q}_{*,*}$ is positive definite) due to the uncertainty of $\boldsymbol{f}_*$ given $\boldsymbol{f}_Z$.

18.5.3.3 FITC

A widely used approximation assumes $q(\boldsymbol{f}_X|\boldsymbol{f}_Z)$ is fully factorized, i.e,

$$q(\boldsymbol{f}_X|\boldsymbol{f}_Z) = \prod_{n=1}^{N} p(f_n|\boldsymbol{f}_Z) = \mathcal{N}(\boldsymbol{f}_X|\mathbf{K}_{X,Z}\mathbf{K}_{Z,Z}^{-1}\boldsymbol{f}_Z, \text{diag}(\mathbf{K}_{X,X} - \mathbf{Q}_{X,X})) \tag{18.129}$$

This is called the **fully independent training conditional** or **FITC** assumption, and was first proposed in [SG06a]. This throws away less uncertainty that the SOR and DTC methods, since it does not make any deterministic assumptions about the relationship between $\boldsymbol{f}_X$ and $\boldsymbol{f}_Z$.

The joint prior has the form

$$q_{\text{fitc}}(\boldsymbol{f}_X, \boldsymbol{f}_*) = \mathcal{N}(\mathbf{0}, \begin{pmatrix} \mathbf{Q}_{X,X} - \text{diag}(\mathbf{Q}_{X,X} - \mathbf{K}_{X,X}) & \mathbf{Q}_{X,*} \\ \mathbf{Q}_{*,X} & \mathbf{K}_{*,*} \end{pmatrix}) \tag{18.130}$$

The predictive distribution for a single test case is given by

$$q_{\text{fitc}}(f_*|\boldsymbol{y}) = \mathcal{N}(f_*|\boldsymbol{k}_{*,Z}\mathbf{\Sigma}\mathbf{K}_{Z,X}\mathbf{\Lambda}^{-1}\boldsymbol{y}, k_{**} - q_{**} + \boldsymbol{k}_{*,Z}\mathbf{\Sigma}\boldsymbol{k}_{Z,*}) \tag{18.131}$$

where $\Lambda \triangleq \text{diag}(\mathbf{K}_{X,X} - \mathbf{Q}_{X,X} + \sigma^2 \mathbf{I}_N)$, and $\mathbf{\Sigma} \triangleq (\mathbf{K}_{Z,Z} + \mathbf{K}_{Z,X}\mathbf{\Lambda}^{-1}\mathbf{K}_{X,Z})^{-1}$. If we have a batch of test cases, we can assume they are conditionally independent (an approach known as **fully independent conditional** or **FIC**), and multiply the above equation.

The computational cost is the same as for SOR and DTC, but the approach avoids some of the pathologies due to a non-degenerate kernel. In particular, one can show that the FIC method is equivalent to exact GP inference with the following non-degenerate kernel:

$$\mathcal{K}_{\text{fic}}(\boldsymbol{x}_i, \boldsymbol{x}_j) = \begin{cases} \mathcal{K}(\boldsymbol{x}_i, \boldsymbol{x}_j) & \text{if } i = j \\ \mathcal{K}_{\text{SOR}}(\boldsymbol{x}_i, \boldsymbol{x}_j) & \text{if } i \neq j \end{cases} \tag{18.132}$$

18.5.3.4 Learning the inducing points

So far, we have not specified how to choose the inducing points or pseudo inputs $\mathbf{Z}$. We can treat these like kernel hyperparameters, and choose them so as to maximize the log marginal likelihood, given by

$$\log q(\boldsymbol{y}|\mathbf{X}, \mathbf{Z}) = \log \int \int p(\boldsymbol{y}|\boldsymbol{f}_X) q(\boldsymbol{f}_X|\mathbf{X}, \boldsymbol{f}_Z) p(\boldsymbol{f}_Z|\mathbf{Z}) d\boldsymbol{f}_Z d\boldsymbol{f} \tag{18.133}$$

$$= \log \int p(\boldsymbol{y}|\boldsymbol{f}_X) q(\boldsymbol{f}_X|\mathbf{X}, \mathbf{Z}) d\boldsymbol{f}_X \tag{18.134}$$

$$= -\frac{1}{2}\log|\mathbf{Q}_{X,X} + \mathbf{\Lambda}| - \frac{1}{2}\boldsymbol{y}^{\mathsf{T}}(\mathbf{Q}_{X,X} + \mathbf{\Lambda})^{-1}\boldsymbol{y} - \frac{n}{2}\log(2\pi) \tag{18.135}$$

where the definition of $\mathbf{\Lambda}$ depends on the method, namely $\mathbf{\Lambda}_{\text{SOR}} = \mathbf{\Lambda}_{\text{dtc}} = \sigma^2 \mathbf{I}_N$, and $\mathbf{\Lambda}_{\text{fitc}} = \text{diag}(\mathbf{K}_{X,X} - \mathbf{Q}_{X,X}) + \sigma^2 \mathbf{I}_N$.

If the input domain is $\mathbb{R}^d$, we can optimize $\mathbf{Z} \in \mathbb{R}^{Md}$ using gradient methods. However, one of the appeals of kernel methods is that they can handle structured inputs, such as strings and graphs (see Section 18.2.1.4). In this case, we cannot use gradient methods to select the inducing points. A simple approach is to select the inducing points from the training set, as in the subset of data approach in Section 18.5.1, or using the efficient selection mechanism in [Cao+15]. However, we can also use discrete optimization methods, such as simulated annealing (Section 12.9.1), as discussed in [For+18a]. See Figure 18.13 for an illustration.

18.5.4 Sparse variational methods

In this section, we discuss a variational approach to GP inference that is similar to the inducing point methods in Section 18.5.3, but which generalizes it to also handle non-conjugate likelihoods. This is called the **sparse variational GP** or **SVGP** approximation. For more details, see [Lei+20]. See also [WKS21] for connections between SVGP and the Nyström method.)

To explain the idea behind SVGP, let us assume, for simplicity, that the function f is defined over a finite set $\mathcal{X}$ of possible inputs, which we partition into three subsets: the training set $\mathbf{X}$, a set of inducing points $\mathbf{Z}$, and all other points (which we can think of as the test set), $\mathbf{X}_*$. (We assume these sets are disjoint.) Let $\boldsymbol{f}_X$, $\boldsymbol{f}_Z$ and $\boldsymbol{f}_*$ represent the corresponding unknown function values on these points, and let $\boldsymbol{f} = [\boldsymbol{f}_X, \boldsymbol{f}_Z, \boldsymbol{f}_*]$ be all the unknowns. (Here we work with a fixed-length vector $\boldsymbol{f}$, but the result generalizes to Gaussian processes, as explained in [Mat+16].) We assume the function is sampled from a GP, so $p(\boldsymbol{f}) = \mathcal{N}(m(\mathcal{X}), \mathcal{K}(\mathcal{X}, \mathcal{X}))$.

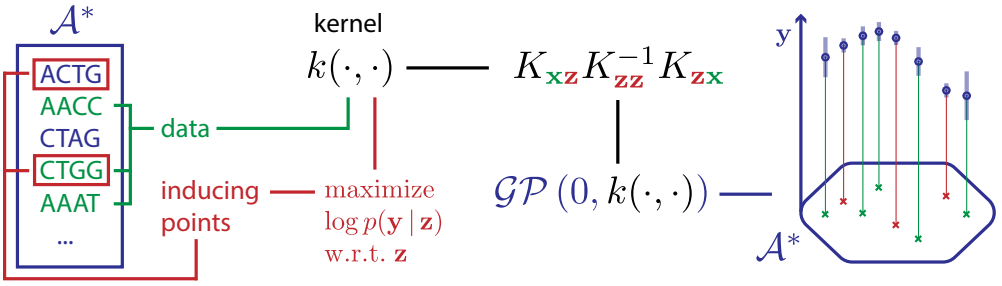


Figure 18.13: Illustration of how to choose inducing points from a discrete input domain (here DNA sequences of length 4) to maximize the log marginal likelihood. From Figure 1 of [For+18a]. Used with kind permission of Vincent Fortuin.

The inducing point methods in Section 18.5.3 approximates the GP prior by assuming $p(\boldsymbol{f}_*, \boldsymbol{f}_X, \boldsymbol{f}_Z) \approx p(\boldsymbol{f}_*|\boldsymbol{f}_Z)p(\boldsymbol{f}_X|\boldsymbol{f}_Z)p(\boldsymbol{f}_Z)$. The inducing points $\boldsymbol{f}_Z$ are chosen to maximize the likelihood of the observed data. We then perform exact inference in this approximate model. By contrast, in this section, we will keep the model unchanged, but we will instead approximate the posterior $p(\boldsymbol{f}|\boldsymbol{y})$ using variational inference. This approach is known as the **variational free energy** (**VFE**) method for GPs [Tit09; Mat+16]; it is also called SVGP.

In the VFE view, the inducing points $\mathbf{Z}$ and inducing variables $\boldsymbol{f}_Z$ are variational parameters, rather than model parameters, which avoids the risk of overfitting. Furthermore, one can show that as the number of inducing points m increases, the quality of the posterior consistently improves, eventually recovering exact inference. By contrast, in the classical inducing point method, increasing m does not always result in better performance [BWR16].

In more detail, the VFE approach tries to find an approximate posterior $q(\boldsymbol{f})$ to minimize $D_{\mathbb{KL}}\left(q(\boldsymbol{f}) \parallel p(\boldsymbol{f}|\boldsymbol{y})\right)$. The key assumption is that $q(\boldsymbol{f}) = q(\boldsymbol{f}_*, \boldsymbol{f}_X, \boldsymbol{f}_Z) = p(\boldsymbol{f}_*, \boldsymbol{f}_X|\boldsymbol{f}_Z)q(\boldsymbol{f}_Z)$, where $p(\boldsymbol{f}_*, \boldsymbol{f}_X|\boldsymbol{f}_Z)$ is computed exactly using the GP prior, and $q(\boldsymbol{f}_Z)$ is learned, by minimizing $\mathcal{K}(q) = D_{\mathbb{KL}}\left(q(\boldsymbol{f}) \parallel p(\boldsymbol{f}|\boldsymbol{y})\right)$.[2] Intuitively, $q(\boldsymbol{f}_Z)$ acts as a "bottleneck" which "absorbs" all the observations from $\boldsymbol{y}$; posterior predictions for elements of $\boldsymbol{f}_X$ or $\boldsymbol{f}_*$ are then made via their dependence on $\boldsymbol{f}_Z$, rather than their dependence on each other.

2. One can show that $D_{\mathbb{KL}}\left(q(\boldsymbol{f}) \parallel p(\boldsymbol{f}|\boldsymbol{y})\right) = D_{\mathbb{KL}}\left(q(\boldsymbol{f}_X, \boldsymbol{f}_Z) \parallel p(\boldsymbol{f}_X, \boldsymbol{f}_Z|\boldsymbol{y})\right)$, which is the original objective from [Tit09].

We can derive the form of the loss, which is used to compute the posterior $q(\boldsymbol{f}_Z)$, as follows:

$$\mathcal{K}(q) = D_{\mathbb{KL}}\left(q(\boldsymbol{f}_*, \boldsymbol{f}_X, \boldsymbol{f}_Z) \parallel p(\boldsymbol{f}_*, \boldsymbol{f}_X, \boldsymbol{f}_Z|\boldsymbol{y})\right) \tag{18.136}$$

$$= \int q(\boldsymbol{f}_*, \boldsymbol{f}_X, \boldsymbol{f}_Z) \log \frac{q(\boldsymbol{f}_*, \boldsymbol{f}_X, \boldsymbol{f}_Z)}{p(\boldsymbol{f}_*, \boldsymbol{f}_X, \boldsymbol{f}_Z|\boldsymbol{y})} d\boldsymbol{f}_* \, d\boldsymbol{f}_X \, d\boldsymbol{f}_Z \tag{18.137}$$

$$= \int p(\boldsymbol{f}_*, \boldsymbol{f}_X|\boldsymbol{f}_Z) q(\boldsymbol{f}_Z) \log \frac{\cancel{p(\boldsymbol{f}_*|\boldsymbol{f}_X, \boldsymbol{f}_Z)} \cancel{p(\boldsymbol{f}_X|\boldsymbol{f}_Z)} q(\boldsymbol{f}_Z) p(\boldsymbol{y})}{\cancel{p(\boldsymbol{f}_*|\boldsymbol{f}_X, \boldsymbol{f}_Z)} \cancel{p(\boldsymbol{f}_X|\boldsymbol{f}_Z)} p(\boldsymbol{f}_Z) p(\boldsymbol{y}|\boldsymbol{f}_X)} d\boldsymbol{f}_* \, d\boldsymbol{f}_X \, d\boldsymbol{f}_Z \tag{18.138}$$

$$= \int p(\boldsymbol{f}_*, \boldsymbol{f}_X|\boldsymbol{f}_Z) q(\boldsymbol{f}_Z) \log \frac{q(\boldsymbol{f}_Z) p(\boldsymbol{y})}{p(\boldsymbol{f}_Z) p(\boldsymbol{y}|\boldsymbol{f}_X)} d\boldsymbol{f}_* \, d\boldsymbol{f}_X \, d\boldsymbol{f}_Z \tag{18.139}$$

$$= \int q(\boldsymbol{f}_Z) \log \frac{q(\boldsymbol{f}_Z)}{p(\boldsymbol{f}_Z)} d\boldsymbol{f}_Z - \int p(\boldsymbol{f}_X|\boldsymbol{f}_Z) q(\boldsymbol{f}_Z) \log p(\boldsymbol{y}|\boldsymbol{f}_X) d\boldsymbol{f}_X \, d\boldsymbol{f}_Z + C \tag{18.140}$$

$$= D_{\mathbb{KL}}\left(q(\boldsymbol{f}_Z) \parallel p(\boldsymbol{f}_Z)\right) - \mathbb{E}_{q(\boldsymbol{f}_X)}\left[\log p(\boldsymbol{y}|\boldsymbol{f}_X)\right] + C \tag{18.141}$$

where $C = \log p(\boldsymbol{y})$ is an irrelevant constant.

We can alternatively write the objective as an evidence lower bound that we want to maximize:

$$\log p(\boldsymbol{y}) = \mathcal{K}(q) + \mathbb{E}_{q(\boldsymbol{f}_X)}\left[\log p(\boldsymbol{y}|\boldsymbol{f}_X)\right] - D_{\mathbb{KL}}\left(q(\boldsymbol{f}_Z) \parallel p(\boldsymbol{f}_Z)\right) \tag{18.142}$$

$$\geq \mathbb{E}_{q(\boldsymbol{f}_X)}\left[\log p(\boldsymbol{y}|\boldsymbol{f}_X)\right] - D_{\mathbb{KL}}\left(q(\boldsymbol{f}_Z) \parallel p(\boldsymbol{f}_Z)\right) \triangleq \mathcal{L}(q) \tag{18.143}$$

Now suppose we choose a Gaussian posterior approximation, $q(\boldsymbol{f}_Z) = \mathcal{N}(\boldsymbol{f}_Z|\boldsymbol{m}, \mathbf{S})$. Since $p(\boldsymbol{f}_Z) = \mathcal{N}(\boldsymbol{f}_Z|\mathbf{0}, \mathcal{K}(\mathbf{Z}, \mathbf{Z}))$, we can compute the KL term in closed form using the formula for KL divergence between Gaussians (Equation (5.80)).

As for the expected log-likelihood term, we need to compute $q(\boldsymbol{f}_X)$. Since $q(\boldsymbol{f}_Z)$ is Gaussian, this can be done in closed form as follows:

$$q(\boldsymbol{f}_X|\boldsymbol{m}, \mathbf{S}) = \int p(\boldsymbol{f}_X|\boldsymbol{f}_Z, \mathbf{X}, \mathbf{Z}) q(\boldsymbol{f}_Z|\boldsymbol{m}, \mathbf{S}) d\boldsymbol{f}_Z = \mathcal{N}(\boldsymbol{f}_X|\tilde{\boldsymbol{\mu}}, \tilde{\boldsymbol{\Sigma}}) \tag{18.144}$$

$$\tilde{\mu}_i = m(\boldsymbol{x}_i) + \boldsymbol{\alpha}(\boldsymbol{x}_i)^\mathsf{T}(\boldsymbol{m} - m(\mathbf{Z})) \tag{18.145}$$

$$\tilde{\Sigma}_{ij} = \mathcal{K}(\boldsymbol{x}_i, \boldsymbol{x}_j) - \boldsymbol{\alpha}(\boldsymbol{x}_i)^\mathsf{T}(\mathcal{K}(\mathbf{Z}, \mathbf{Z}) - \mathbf{S})\boldsymbol{\alpha}(\boldsymbol{x}_j) \tag{18.146}$$

$$\boldsymbol{\alpha}(\boldsymbol{x}_i) = \mathcal{K}(\mathbf{Z}, \mathbf{Z})^{-1} \mathcal{K}(\mathbf{Z}, \boldsymbol{x}_i) \tag{18.147}$$

Hence $q(f_n) = \mathcal{N}(f_n|\tilde{\mu}_n, \tilde{\Sigma}_{nn})$, which we can use to compute the expected log likelihood:

$$\mathbb{E}_{q(\boldsymbol{f}_X)}\left[\log p(\boldsymbol{y}|\boldsymbol{f}_X)\right] = \sum_{n=1}^{N} \mathbb{E}_{q(f_n)}\left[\log p(y_n|f_n)\right] \tag{18.148}$$

We discuss how to compute the expected loglikelihood below.

18.5.4.1 Gaussian likelihood

If we have a Gaussian observation model, we can compute the expected log likelihood in closed form. In particular, if we assume $m(\boldsymbol{x}) = \mathbf{0}$, we have

$$\mathbb{E}_{q(f_n)}\left[\log \mathcal{N}(y_n|f_n, \beta^{-1})\right] = \log \mathcal{N}(y_n|\boldsymbol{k}_n^\mathsf{T} \mathbf{K}_{Z,Z}^{-1} \boldsymbol{m}, \beta^{-1}) - \frac{1}{2}\beta \tilde{k}_{nn} - \frac{1}{2}\mathrm{tr}(\mathbf{S}\boldsymbol{\Lambda}_n) \tag{18.149}$$

where $\tilde{k}_{nn} = k_{nn} - \boldsymbol{k}_n^\mathsf{T} \mathbf{K}_{Z,Z}^{-1} \boldsymbol{k}_n$, $\boldsymbol{k}_n$ is the n'th column of $\mathbf{K}_{Z,X}$ and $\boldsymbol{\Lambda}_n = \beta \mathbf{K}_{Z,Z}^{-1} \boldsymbol{k}_n \boldsymbol{k}_n^\mathsf{T} \mathbf{K}_{Z,Z}^{-1}$.

Hence the overall ELBO has the form

$$\mathcal{L}(q) = \log \mathcal{N}(\boldsymbol{y}|\mathbf{K}_{X,Z}\mathbf{K}_{Z,Z}^{-1}\boldsymbol{m}, \beta^{-1}\mathbf{I}_N) - \frac{1}{2}\beta \mathrm{tr}(\mathbf{K}_{X,Z}\mathbf{K}_{Z,Z}^{-1}\mathbf{S}\mathbf{K}_{Z,Z}^{-1}\mathbf{K}_{Z,X}) \tag{18.150}$$

$$- \frac{1}{2}\beta \mathrm{tr}(\mathbf{K}_{X,X} - \mathbf{Q}_{X,X}) - D_{\mathbb{KL}}\left(q(\boldsymbol{f}_Z) \parallel p(\boldsymbol{f}_Z)\right) \tag{18.151}$$

To compute the gradients of this, we leverage the following result [OA09]:

$$\frac{\partial}{\partial \mu}\mathbb{E}_{\mathcal{N}(x|\mu,\sigma^2)}\left[h(x)\right] = \mathbb{E}_{\mathcal{N}(x|\mu,\sigma^2)}\left[\frac{\partial}{\partial x}h(x)\right] \tag{18.152}$$

$$\frac{\partial}{\partial \sigma^2}\mathbb{E}_{\mathcal{N}(x|\mu,\sigma^2)}\left[h(x)\right] = \frac{1}{2}\mathbb{E}_{\mathcal{N}(x|\mu,\sigma^2)}\left[\frac{\partial^2}{\partial x^2}h(x)\right] \tag{18.153}$$

We then substitute $h(x)$ with $\log p(y_n|f_n)$. Using this, one can show

$$\nabla_{\boldsymbol{m}}\mathcal{L}(q) = \beta \mathbf{K}_{Z,Z}^{-1}\mathbf{K}_{Z,X}\boldsymbol{y} - \boldsymbol{\Lambda}\boldsymbol{m} \tag{18.154}$$

$$\nabla_{\mathbf{S}}\mathcal{L}(q) = \frac{1}{2}\mathbf{S}^{-1} - \frac{1}{2}\boldsymbol{\Lambda} \tag{18.155}$$

Setting the derivatives to zero gives the optimal solution:

$$\mathbf{S} = \boldsymbol{\Lambda}^{-1} \tag{18.156}$$

$$\boldsymbol{\Lambda} = \beta \mathbf{K}_{Z,Z}^{-1}\mathbf{K}_{Z,X}\mathbf{K}_{X,Z}\mathbf{K}_{Z,Z}^{-1} + \mathbf{K}_{Z,Z}^{-1} \tag{18.157}$$

$$\boldsymbol{m} = \beta \boldsymbol{\Lambda}^{-1}\mathbf{K}_{Z,Z}^{-1}\mathbf{K}_{Z,X}\boldsymbol{y} \tag{18.158}$$

This is called **sparse GP regression** or **SGPR** [Tit09].

With these parameters, the lower bound on the log marginal likelihood is given by

$$\log p(\boldsymbol{y}) \geq \log \mathcal{N}(\boldsymbol{y}|\mathbf{0}, \mathbf{K}_{X,Z}\mathbf{K}_{Z,Z}^{-1}\mathbf{K}_{Z,X} + \beta^{-1}\mathbf{I}) - \frac{1}{2}\beta \mathrm{tr}(\mathbf{K}_{X,X} - \mathbf{Q}_{X,X}) \tag{18.159}$$

where $\mathbf{Q}_{X,X} = \mathbf{K}_{X,Z}\mathbf{K}_{Z,Z}^{-1}\mathbf{K}_{Z,X}$. (This is called the "collapsed" lower bound, since we have marginalized out $\boldsymbol{f}_Z$.) If $Z = X$, then $\mathbf{K}_{Z,Z} = \mathbf{K}_{Z,X} = \mathbf{K}_{X,X}$, so the bound becomes tight, and we have $\log p(\boldsymbol{y}) = \log \mathcal{N}(\boldsymbol{y}|\mathbf{0}, \mathbf{K}_{X,X} + \beta^{-1}\mathbf{I})$.

Equation (18.159) is almost the same as the log marginal likelihood for the DTC model in Equation (18.135), except for the trace term; it is this latter term that prevents overfitting, due to the fact that we treat $\boldsymbol{f}_Z$ as variational parameters of the posterior rather than model parameters of the prior.

18.5.4.2 Non-Gaussian likelihood

In this section, we briefly consider the case of non-Gaussian likelihoods, which arise when using GPs for classification or for count data (see Section 18.4). We can compute the gradients of the expected log likelihood by defining $h(f_n) = \log p(y_n|f_n)$ and then using a Monte Carlo approximation to Equation (18.152) and Equation (18.153). In the case of a binary classifier, we can use the results in Table 18.2 to compute the inner $\frac{\partial}{\partial f_n}h(f_n)$ and $\frac{\partial^2}{\partial f_n^2}h(f_n)$ terms.

18.5.4.3 Minibatch SVI

Computing the optimal variational solution in Section 18.5.4.1 requires solving a batch optimization problem, which takes $O(M^3 + NM^2)$ time. This may still be too slow if N is large, unless M is small, which compromises accuracy.

An alternative approach is to perform stochastic optimization of the VFE objective, instead of batch optimization. This is known as stochastic variational inference (see Section 10.3.1). The key observation is that the log likelihood in Equation (18.148) is a sum of N terms, which we can approximate with minibatch sampling to compute noisy estimates of the gradient, as proposed in [HFL13].

In more detail, the objective becomes

$$\mathcal{L}(q) = \left[\frac{N}{B}\sum_{b=1}^{B}\frac{1}{|\mathcal{B}_b|}\sum_{n\in\mathcal{B}_b}\mathbb{E}_{q(f_n)}\left[\log p(y_n|f_n)\right]\right] - D_{\mathbb{KL}}\left(q(\boldsymbol{f}_Z) \parallel (p(\boldsymbol{f}_Z)\right) \tag{18.160}$$

where $\mathcal{B}_b$ is the b'th batch, and B is the number of batches. Since the GP model (with Gaussian likelihoods) is in the exponential family, we can efficiently compute the natural gradient (Section 6.4) of Equation (18.160) wrt the canonical parameters of $q(\boldsymbol{f}_Z)$; this converges much faster than following the standard gradient. See [HFL13] for details.

18.5.5 Exploiting parallelization and structure via kernel matrix multiplies

It takes $O(N^3)$ time to compute the Cholesky decomposition of $\mathbf{K}_{X,X}$, which is needed to solve the linear system $\mathbf{K}_\sigma\boldsymbol{\alpha} = \boldsymbol{y}$ and to compute $|\mathbf{K}_{X,X}|$. An alternative to Cholesky decomposition is to use linear algebra methods, often called **Krylov subspace methods** based just on **matrix vector multiplication** or **MVM**. These approaches are often much faster.

In short, if the kernel matrix $\mathbf{K}_{X,X}$ has special algebraic structure, which is often the case through either the choice of kernel or the structure of the inputs, then it is typically easier to exploit this structure in performing fast matrix multiplies. Moreover, even if the kernel matrix **does not** have special structure, matrix multiplies are trivial to parallelize, and can thus be greatly accelerated by GPUs, unlike Cholesky based methods which are largely sequential. Algorithms based on matrix multiplies are in harmony with modern hardware advances, which enable significant parallelization.

18.5.5.1 Using conjugate gradient and Lanczos methods

We can solve the linear system $\mathbf{K}_\sigma\boldsymbol{\alpha} = \boldsymbol{y}$ using conjugate gradients (CG). The key computational step in CG is the ability to perform MVMs. Let $\tau(\mathbf{K}_\sigma)$ be the time complexity of a single MVM with $\mathbf{K}_\sigma$. For a dense $n \times n$ matrix, we have $\tau(\mathbf{K}_\sigma) = n^2$; however, we can speed this up if $\mathbf{K}_\sigma$ is sparse or structured, as we discuss below.

Even if $\mathbf{K}_\sigma$ is dense, we may still be able to save time by solving the linear system approximately. In particular, if we perform C iterations, CG will take $O(C\tau(\mathbf{K}_\sigma))$ time. If we run for $C = n$, and $\tau(\mathbf{K}_\sigma) = n^2$, it gives the exact solution in $O(n^3)$ time. However, often we can use fewer iterations and still get good accuracy, depending on the condition number of $\mathbf{K}_\sigma$.

We can compute the log determinant of a matrix using the MVM primitive with a similar iterative method known as **stochastic Lanczos quadrature** [UCS17; Don+17a]. This takes $O(L\tau(\mathbf{K}_\sigma))$ time for L iterations.

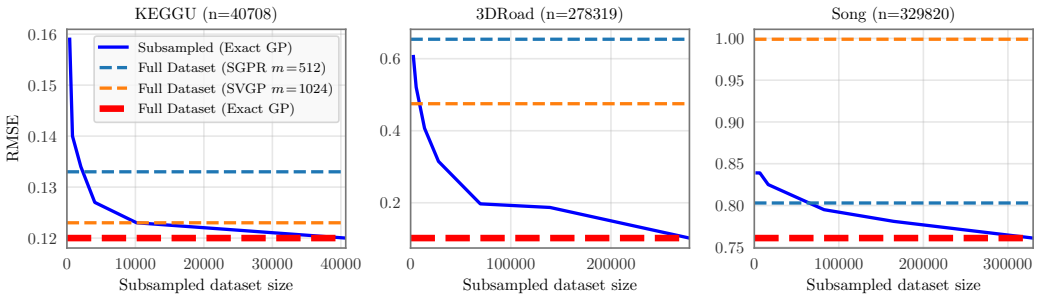


Figure 18.14: RMSE on test set as a function of training set size using a GP with Matern 3/2 kernel with shared lengthscale across all dimensions. Solid lines: exact inference. Dashed blue: SGPR method (closed-form batch solution to the Gaussian variational approximation) of Section 18.5.4.1 with $M = 512$ inducing points. Dashed orange: SVGP method (SGD on Gaussian variational approxiation) of Section 18.5.4.3 with $M = 1024$ inducing points. Number of input dimensions: KEGGU $D = 27$, 3DRoad $D = 3$, Song $D = 90$. From Figure 4 of [Wan+19a]. Used with kind permission of Andrew Wilson.

These methods have been used in the **blackbox matrix-matrix multiplication** (**BBMM**) inference procedure of [Gar+18a], which formulates a batch approach to CG that can be effectively parallelized on GPUs. Using 8 GPUs, this enabled the authors of [Wan+19a] to perform exact inference for a GP regression model on $N \sim 10^4$ datapoints in seconds, $N \sim 10^5$ datapoints in minutes, and $N \sim 10^6$ datapoints in hours.

Interestingly, Figure 18.14 shows that exact GP inference on a subset of the data can often outperform approximate inference on the full data. We also see that performance of exact GPs continues to significantly improve as we increase the size of the data, suggesting that GPs are not only useful in the small-sample setting. In particular, the BBMM is an exact method, and so will preserve the non-parametric representation of a GP with a non-degenerate kernel. By contrast, standard scalable approximations typically operate by replacing the exact kernel with an approximation that corresponds to a parametric model. The non-parametric GPs are able to grow their capacity with more data, benefiting more significantly from the structure present in large datasets.

18.5.5.2 Kernels with compact support

Suppose we use a kernel with **compact support**, where $\mathcal{K}(\boldsymbol{x}, \boldsymbol{x}') = 0$ if $\|\boldsymbol{x} - \boldsymbol{x}'\| > \epsilon$ for some threshold ϵ (see e.g., [MR09]), then $\mathbf{K}_\sigma$ will be sparse, so $\tau(\mathbf{K}_\sigma)$ will be $O(N)$. We can also induce sparsity and structure in other ways, as we discuss in Section 18.5.5.3.

18.5.5.3 KISS

One way to ensure that MVMs are fast is to force the kernel matrix to have structure. The **structured kernel interpolation** (**SKI**) method of [WN15] does this as follows. First it assumes we have a set of inducing points, with Gram matrix $\mathbf{K}_{Z,Z}$. It then interpolates these values to predict the entries of the full kernel matrix using

$$\mathbf{K}_{X,X} \approx \mathbf{W}_\mathbf{X} \mathbf{K}_{Z,Z} \mathbf{W}_\mathbf{X}^\mathsf{T} \tag{18.161}$$

where $\mathbf{W}_\mathbf{X}$ is a sparse matrix containing interpolation weights. If we use cubic interpolation, each row only has 4 nonzeros. Thus we can compute $(\mathbf{W}_\mathbf{X}\mathbf{K}_{Z,Z}\mathbf{W}_\mathbf{X}^\mathsf{T})\boldsymbol{v}$ for any vector $\boldsymbol{v}$ in $O(N + M^2)$ time.

Note that the **SKI** approach generalizes all inducing point methods. For example, we can recover the subset of regressors method (SOR) method by setting the interpolation weights to $\mathbf{W} = \mathbf{K}_{X,Z}\mathbf{K}_{Z,Z}^{-1}$. We can identify this procedure as performing a global Gaussian process interpolation strategy on the user specified kernel. See [WN15] and [WDN15] for more details.

In 1d, we can further reduce the running time by choosing the inducing points to be on a regular grid, so that $\mathbf{K}_{Z,Z}$ is a Toeplitz matrix. In higher dimensions, we need to use a multidimensional grid of points, resulting in $\mathbf{K}_{Z,Z}$ being a Kronecker product of Toeplitz matrices. This enables matrix vector multiplication in $O(N + M \log M)$ time and $O(N + M)$ space. The resulting method is called **KISS-GP** [WN15], which stands for "kernel interpolation for scalable, structured GPs".

Unfortunately, the KISS method can take exponential time in the input dimensions D when exploiting Kronecker structure in $\mathbf{K}_{Z,Z}$, due to the need to create a fully connected multidimensional lattice. In [Gar+18b], they propose a method called **SKIP**, which stands for "SKI for products". The idea is to leverage the fact that many kernels (including ARD) can be written as a product of 1d kernels: $\mathcal{K}(\boldsymbol{x}, \boldsymbol{x}') = \prod_{d=1}^{D} \mathcal{K}^d(\boldsymbol{x}, \boldsymbol{x}')$. This can be combined with the 1d SKI method to enable fast MVMs. The overall running time to compute the log marginal likelihood (which is the bottleneck for kernel learning) using C iterations of CG and a Lanczos decomposition of rank L, becomes $O(DL(N + M \log M) + L^3 N \log D + CL^2 N)$. Typical values are $L \sim 10^1$ and $C \sim 10^2$.

18.5.5.4 Tensor train methods

Consider the Gaussian VFE approach in Section 18.5.4. We have to estimate the covariance $\mathbf{S}$ and the mean $\boldsymbol{m}$. We can represent $\mathbf{S}$ efficiently using Kronecker structure, as used by KISS. Additionally, we can represent $\boldsymbol{m}$ efficiently using the **tensor train decomposition** [Ose11] in combination with **SKI** [WN15]. The resulting **TT-GP** method can scale efficiently to billions of inducing points, as explained in [INK18].

18.5.6 Converting a GP to a SSM

Consider a function defined on a 1d scalar input, such as a time index. For many kernels, the corresponding GP can be modeled using a stochastic differential equation. This induces a block tri-diagonal precision matrix for the posterior $p(\boldsymbol{f}_{1:T}|\boldsymbol{y}_{1:T})$. We can therefore convert this to a linear-Gaussian state space model (Section 29.1), and perform exact inference in $O(T)$ time using Kalman smoothing, as explained in [SSH13; Ada+20]. This conversion can be done exactly for Matern kernels and approximately for Gaussian (RBF) kernels (see [SS19, Ch. 12]). In [SGF21], they describe how to reduce the linear dependence on T to $\log(T)$ time using a **parallel prefix scan** operator, that can be run efficiently on GPUs.

18.6 Learning the kernel

In [Mac98], David MacKay asked: "How can Gaussian processes replace neural networks? Have we thrown the baby out with the bathwater?" This remark was made in the late 1990s, at the end of the second wave of neural networks. Researchers and practitioners had grown weary of the design

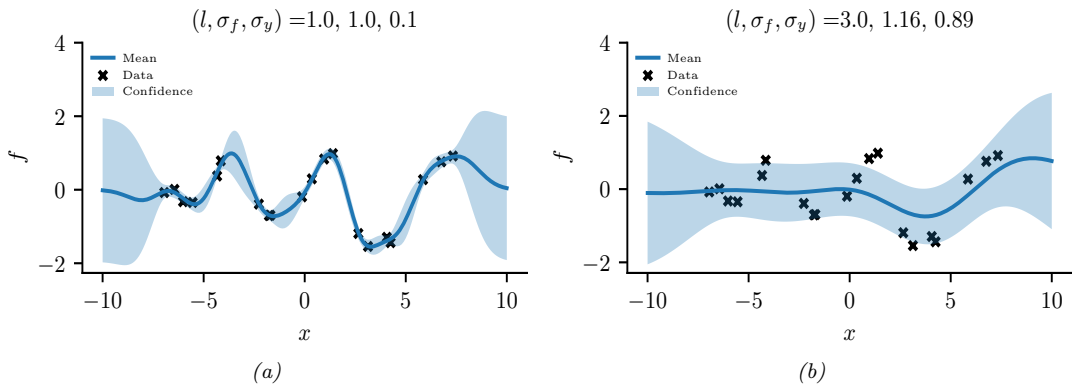


Figure 18.15: Some 1d GPs with RBF kernels but different hyper-parameters fit to 20 noisy observations. The hyper-parameters $(\ell, \sigma_f, \sigma_y)$ are as follows: (a) (1,1,0.1) (b) (3.0, 1.16, 0.89). Adapted from Figure 2.5 of [RW06]. Generated by gpr_demo_change_hparams.ipynb.

decisions associated with neural networks — such as activation functions, optimization procedures, architecture design — and the lack of a principled framework to make these decisions. Gaussian processes, by contrast, were perceived as flexible and principled probabilistic models, which naturally followed from Radford Neal's results on infinite neural networks [Nea96], which we discuss in more depth in Section 18.7.

However, MacKay [Mac98] noted that neural networks could discover rich representations of data through adaptive hidden basis functions, while Gaussian processes with standard kernel functions, such as the RBF kernel, are essentially just smoothing devices. Indeed, the generalization properties of Gaussian processes hinge on the suitability of the kernel function. *Learning* the kernel is how we do representation learning with Gaussian processes, and in many cases will be crucial for good performance — especially when we wish to perform extrapolation, making predictions far away from the data [WA13; Wil+14].

As we will see, learning a kernel is in many ways analogous to training a neural network. Moreover, neural networks and Gaussian processes can be synergistically combined through approaches such as deep kernel learning (see Section 18.6.6) and NN-GPs (Section 18.7.2).

18.6.1 Empirical Bayes for the kernel parameters

Suppose, as in Section 18.3.2, we are performing 1d regression using a GP with an RBF kernel. Since the data has observation noise, the kernel has the following form:

$$\mathcal{K}_y(x_p, x_q) = \sigma_f^2 \exp(-\frac{1}{2\ell^2}(x_p - x_q)^2) + \sigma_y^2 \delta_{pq} \tag{18.162}$$

Here ℓ is the horizontal scale over which the function changes, σ_f^2 controls the vertical scale of the function, and σ_y^2 is the noise variance. Figure 18.15 illustrates the effects of changing these parameters. We sampled 20 noisy data points from the SE kernel using $(\ell, \sigma_f, \sigma_y) = (1, 1, 0.1)$, and then made predictions various parameters, conditional on the data. In Figure 18.15(a), we use

$(\ell, \sigma_f, \sigma_y) = (1, 1, 0.1)$, and the result is a good fit. In Figure 18.15(b), we increase the length scale to $\ell = 3$; now the function looks smoother, but we are arguably underfitting.

To estimate the kernel parameters $\boldsymbol{\theta}$ (sometimes called hyperparameters), we could use exhaustive search over a discrete grid of values, with validation loss as an objective, but this can be quite slow. (This is the approach used by nonprobabilistic methods, such as SVMs, to tune kernels.) Here we consider an empirical Bayes approach, which will allow us to use continuous optimization methods, which are much faster. In particular, we will maximize the marginal likelihood

$$p(\boldsymbol{y}|\mathbf{X}, \boldsymbol{\theta}) = \int p(\boldsymbol{y}|\boldsymbol{f}, \mathbf{X}) p(\boldsymbol{f}|\mathbf{X}, \boldsymbol{\theta}) d\boldsymbol{f} \tag{18.163}$$

(The reason it is called the marginal likelihood, rather than just likelihood, is because we have marginalized out the latent Gaussian vector $\boldsymbol{f}$.) Since $p(\boldsymbol{f}|\mathbf{X}) = \mathcal{N}(\boldsymbol{f}|\mathbf{0}, \mathbf{K})$, and $p(\boldsymbol{y}|\boldsymbol{f}) = \prod_{n=1}^{N} \mathcal{N}(y_n|f_n, \sigma_y^2)$, the marginal likelihood is given by

$$\log p(\boldsymbol{y}|\mathbf{X}, \boldsymbol{\theta}) = \log \mathcal{N}(\boldsymbol{y}|\mathbf{0}, \mathbf{K}_\sigma) = -\frac{1}{2}\boldsymbol{y}\mathbf{K}_\sigma^{-1}\boldsymbol{y} - \frac{1}{2}\log|\mathbf{K}_\sigma| - \frac{N_\mathcal{D}}{2}\log(2\pi) \tag{18.164}$$

where the dependence of $\mathbf{K}_\sigma$ on $\boldsymbol{\theta}$ is implicit. The first term is a data fit term, the second term is a model complexity term, and the third term is just a constant. To understand the tradeoff between the first two terms, consider a SE kernel in 1D, as we vary the length scale ℓ and hold σ_y^2 fixed. Let $J(\ell) = -\log p(\boldsymbol{y}|\mathbf{X}, \ell)$. For short length scales, the fit will be good, so $\boldsymbol{y}^\mathsf{T}\mathbf{K}_\sigma^{-1}\boldsymbol{y}$ will be small. However, the model complexity will be high: $\mathbf{K}$ will be almost diagonal, since most points will not be considered "near" any others, so the $\log|\mathbf{K}_\sigma|$ will be large. For long length scales, the fit will be poor but the model complexity will be low: $\mathbf{K}$ will be almost all 1's, so $\log|\mathbf{K}_\sigma|$ will be small.

We now discuss how to maximize the marginal likelihood. One can show that

$$\frac{\partial}{\partial\theta_j}\log p(\boldsymbol{y}|\mathbf{X}, \boldsymbol{\theta}) = \frac{1}{2}\boldsymbol{y}^\mathsf{T}\mathbf{K}_\sigma^{-1}\frac{\partial\mathbf{K}_\sigma}{\partial\theta_j}\mathbf{K}_\sigma^{-1}\boldsymbol{y} - \frac{1}{2}\mathrm{tr}(\mathbf{K}_\sigma^{-1}\frac{\partial\mathbf{K}_\sigma}{\partial\theta_j}) \tag{18.165}$$

$$= \frac{1}{2}\mathrm{tr}\left((\boldsymbol{\alpha}\boldsymbol{\alpha}^\mathsf{T} - \mathbf{K}_\sigma^{-1})\frac{\partial\mathbf{K}_\sigma}{\partial\theta_j}\right) \tag{18.166}$$

where $\boldsymbol{\alpha} = \mathbf{K}_\sigma^{-1}\boldsymbol{y}$. It takes $O(N^3)$ time to compute $\mathbf{K}_\sigma^{-1}$, and then $O(N^2)$ time per hyper-parameter to compute the gradient.

The form of $\frac{\partial\mathbf{K}_\sigma}{\partial\theta_j}$ depends on the form of the kernel, and which parameter we are taking derivatives with respect to. Often we have constraints on the hyper-parameters, such as $\sigma_y^2 \geq 0$. In this case, we can define $\theta = \log(\sigma_y^2)$, and then use the chain rule.

Given an expression for the log marginal likelihood and its derivative, we can estimate the kernel parameters using any standard gradient-based optimizer. However, since the objective is not convex, local minima can be a problem, as we illustrate below, so we may need to use multiple restarts.

18.6.1.1 Example

Consider Figure 18.16. We use the SE kernel in Equation (18.162) with $\sigma_f^2 = 1$, and plot $\log p(\boldsymbol{y}|\mathbf{X}, \ell, \sigma_y^2)$ (where $\mathbf{X}$ and $\boldsymbol{y}$ are the 7 data points shown in panels b and c as we vary ℓ and σ_y^2. The two local optima are indicated by + in panel (a). The bottom left optimum corresponds

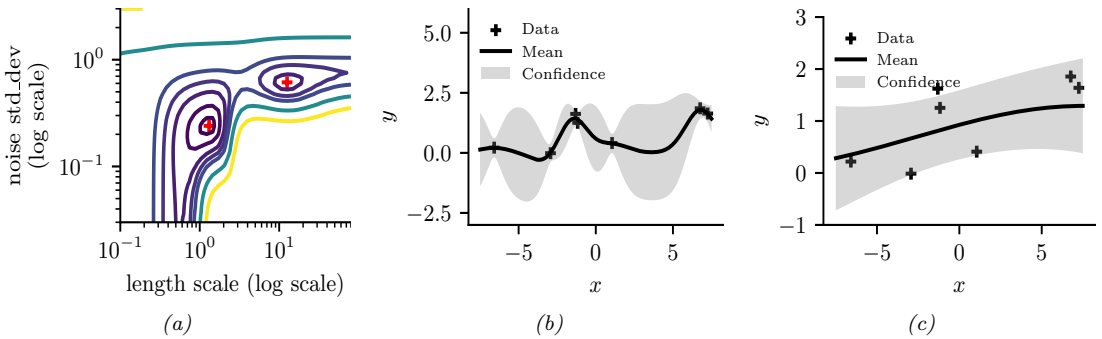


Figure 18.16: Illustration of local minima in the marginal likelihood surface. (a) We plot the log marginal likelihood vs σ_y^2 and ℓ, for fixed $\sigma_f^2 = 1$, using the 7 data points shown in panels b and c. (b) The function corresponding to the lower left local minimum, $(\ell, \sigma_n^2) \approx (1, 0.2)$. This is quite "wiggly" and has low noise. (c) The function corresponding to the top right local minimum, $(\ell, \sigma_n^2) \approx (10, 0.8)$. This is quite smooth and has high noise. The data was generated using $(\ell, \sigma_n^2) = (1, 0.1)$. Adapted from Figure 5.5 of [RW06]. Generated by gpr_demo_marglik.ipynb.

to a low-noise, short-length scale solution (shown in panel b). The top right optimum corresponds to a high-noise, long-length scale solution (shown in panel c). With only 7 data points, there is not enough evidence to confidently decide which is more reasonable, although the more complex model (panel b) has a marginal likelihood that is about 60% higher than the simpler model (panel c). With more data, the more complex model would become even more preferred.

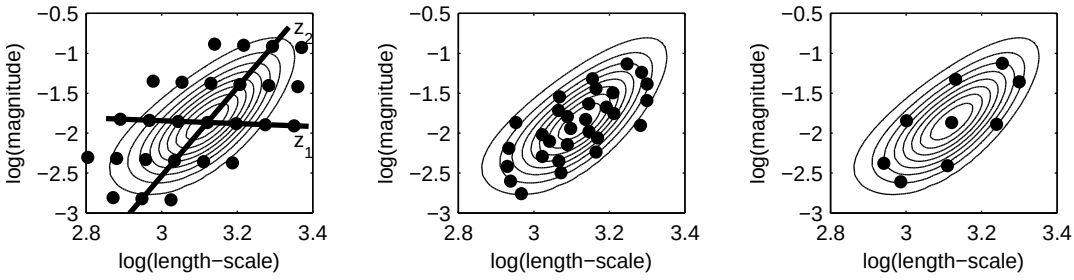
Figure 18.16 illustrates some other interesting (and typical) features. The region where $\sigma_y^2 \approx 1$ (top of panel a) corresponds to the case where the noise is very high; in this regime, the marginal likelihood is insensitive to the length scale (indicated by the horizontal contours), since all the data is explained as noise. The region where $\ell \approx 0.5$ (left hand side of panel a) corresponds to the case where the length scale is very short; in this regime, the marginal likelihood is insensitive to the noise level (indicated by the vertical contours), since the data is perfectly interpolated. Neither of these regions would be chosen by a good optimizer.

18.6.2 Bayesian inference for the kernel parameters

When we have a small number of datapoints (e.g., when using GPs for blackbox optimization, as we discuss in Section 6.8), using a point estimate of the kernel parameters can give poor results [Bul11; WF14]. As a simple example, if the function values that have been observed so far are all very similar, then we may estimate $\hat{\sigma} \approx 0$, which will result in overly confident predictions.[3]

To overcome such overconfidence, we can compute a posterior over the kernel parameters. If the dimensionality of $\boldsymbol{\theta}$ is small, we can compute a discrete grid of possible values, centered on the MAP

3. In [WSN00; BBV11b], they show how we can put a conjugate prior on σ^2 and integrate it out, to generate a Student version of the GP, which is more robust.



(a) (b) (c)

Figure 18.17: Three different approximations to the posterior over hyper-parameters: grid-based, Monte Carlo, and central composite design. From Figure 3.2 of [Van10]. Used with kind permission of Jarno Vanhatalo.

estimate $\hat{\boldsymbol{\theta}}$ (computed as above). We can then approximate the posterior using

$$p(\boldsymbol{f}|\mathcal{D}) = \sum_{s=1}^{S} p(\boldsymbol{f}|\mathcal{D}, \boldsymbol{\theta}_s) p(\boldsymbol{\theta}_s|\mathcal{D}) w_s \tag{18.167}$$

where w_s denotes the weight for grid point s.

In higher dimensions, a regular grid suffers from the curse of dimensionality. One alternative is place grid points at the mode, and at a distance ± 1sd from the mode along each dimension, for a total of $2|\boldsymbol{\theta}| + 1$ points. This is called a **central composite design** [RMC09]. See Figure 18.17 for an illustration.

In higher dimensions, we can use Monte Carlo inference for the kernel parameters when computing Equation (18.167). For example, [MA10] shows how to use slice sampling (Section 12.4.1) for this task, [Hen+15] shows how to use HMC (Section 12.5), and [BBV11a] shows how to use SMC (Chapter 13).

In Figure 18.18, we illustrate the difference between kernel optimization vs kernel inference. We fit a 1d dataset using a kernel of the form

$$\mathcal{K}(r) = \sigma_1^2 \mathcal{K}_{\text{SE}}(r; \tau) \mathcal{K}_{\cos}(r; \rho_1) + \sigma_2^s \mathcal{K}_{32}(r; \rho_2) \tag{18.168}$$

where $\mathcal{K}_{\text{SE}}(r; \ell)$ is the squared exponential kernel (Equation (18.13)), $\mathcal{K}_{\cos}(r; \rho_1)$ is the cosine kernel (Equation (18.20)), and $\mathcal{K}_{32}(r; \rho_2)$ is the Matern $\frac{3}{2}$ kernel (Equation (18.16)). We then compute a point-estimate of the kernel parameters using empirical Bayes, and posterior samples using HMC. We then predicting the posterior mean of f on a 1d test set by plugging in the MLE or averaging over samples. We see that the latter captures more uncertainty (beyond the uncertainty captured by the Gaussian itself).

18.6.3 Multiple kernel learning for additive kernels

A special case of kernel learning arises when the kernel is a sum of B base kernels

$$\mathcal{K}(\boldsymbol{x}, \boldsymbol{x}') = \sum_{b=1}^{B} w_b \mathcal{K}_b(\boldsymbol{x}, \boldsymbol{x}') \tag{18.169}$$

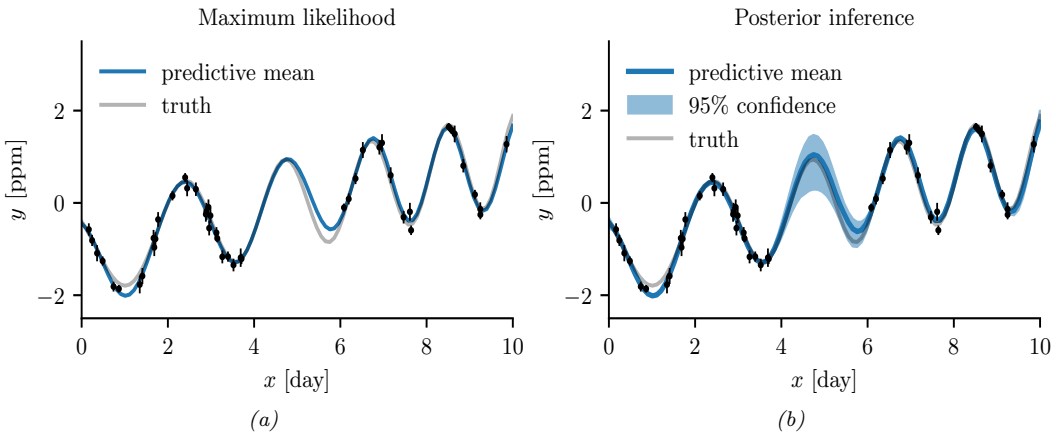


Figure 18.18: Difference between estimation and inference for kernel hyper-parameters. (a) Empirical Bayes approach based on optimization. We plot the posterior predicted mean given a plug-in estimate, $\mathbb{E}\left[f(x)|\mathcal{D}, \hat{\boldsymbol{\theta}}\right]$. (b) Bayesian approach based on HMC. We plot the posterior predicted mean, marginalizing over hyper-parameters, $\mathbb{E}[f(x)|\mathcal{D}]$. Generated by gp_kernel_opt.ipynb.

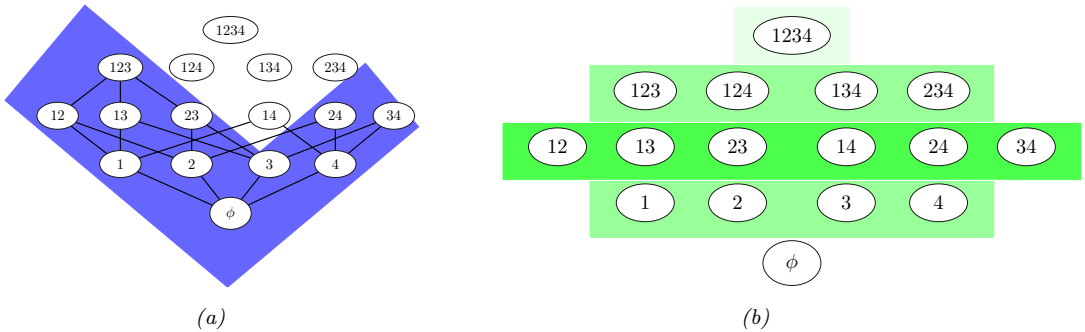


Figure 18.19: Comparison of different additive model classes for a 4d function. Circles represent different interaction terms, ranging from first-order to fourth-order. Left: hierarchical kernel learning uses a nested hierarchy of terms. Right: additive GPs use a weighted sum of additive kernels of different orders. Color shades represent different weighting terms. Adapted from Figure 6.2 of [Duv14].

Optimizing the weights $w_b > 0$ using structural risk minimization is known as **multiple kernel learning**; see e.g., [Rak+08] for details.

Now suppose we constrain the base kernels to depend on a subset of the variables. Furthermore, suppose we enforce a hierarchical inclusion property (e.g., including the kernel k_{123} means we must also include k_{12}, k_{13} and k_{23}), as illustrated in Figure 18.19(left). This is called **hierarchical kernel learning**. We can find a good subset from this model class using convex optimization [Bac09]; however, this requires the use of cross validation to estimate the weights. A more efficient approach is to use the empirical Bayes approach described in [DNR11].

In many cases, it is common to restrict attention to first order additive kernels, i.e., $\mathcal{K}(\boldsymbol{x}, \boldsymbol{x}') =$

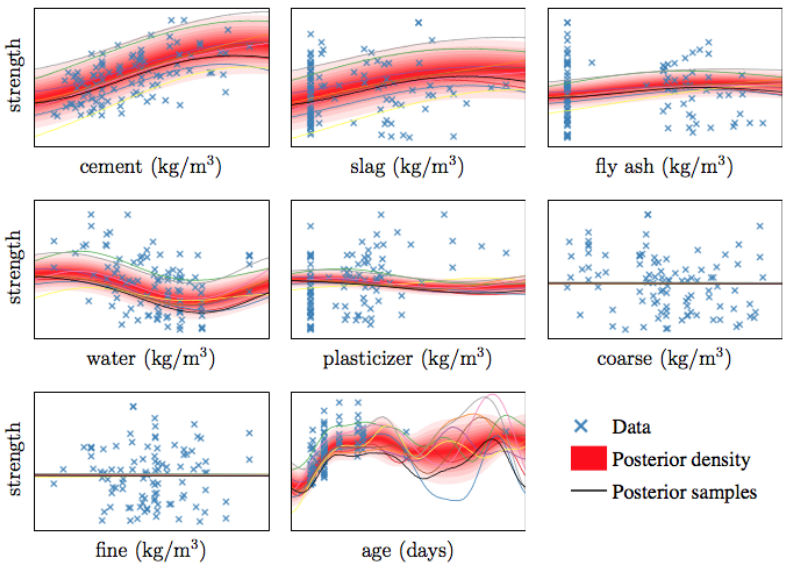


Figure 18.20: Predictive distribution of each term im a GP-GAM model applied to a dataset with 8 continuous inputs and 1 continuous output, representing the strength of some concrete. From Figure 2.7 of [Duv14]. Used with kind permission of David Duvenaud.

$\sum_{d=1}^{D} \mathcal{K}_d(x_d, x'_d)$. The resulting function has the form

$$f(\boldsymbol{x}) = f_1(x_1) + \ldots + f_D(x_D) \tag{18.170}$$

This is called a **generalized additive model** or **GAM**.

Figure 18.20 shows an example of this, where each base kernel has the form $\mathcal{K}_d(x_d, x'_d) = \sigma_d^2 \text{SE}(x_d, x'_d|\ell_d)$, In Figure 18.20, we see that the σ_d^2 terms for the coarse and fine features are set to zero, indicating that these inputs have no impact on the response variable.

[DBW20] considers additive kernels operating on different linear projections of the inputs:

$$\mathcal{K}(\boldsymbol{x}, \boldsymbol{x}') = \sum_{b=1}^{B} w_b \mathcal{K}_b(\mathbf{P}_b \boldsymbol{x}, \mathbf{P}_b \boldsymbol{x}') \tag{18.171}$$

Surprisingly, they show that these models can match or exceed the performance of kernels operating on the original space, even when the projections are into a **single** dimension, and not learned. In other words, it is possible to reduce many regression problems to a single dimension without loss in performance. This finding is particularly promising for scalable inference, such as KISS (see Section 18.5.5.3), and active learning, which are greatly simplified in a low dimensional setting.

More recently, [LBH22] has proposed the **orthogonal additive kernel** (OAK), which imposes an orthogonality constraint on the additive functions. This ensures an identifiable, low-dimensional representation of the functional relationship, and results in improved performance.

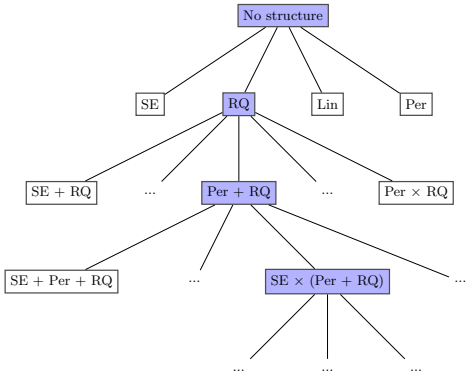


Figure 18.21: Example of a search tree over kernel expressions. Adapted from Figure 3.2 of [Duv14].

18.6.4 Automatic search for compositional kernels

Although the above methods can estimate the hyperparameters of a specified set of kernels, they do not choose the kernels themselves (other than the special case of selecting a subset of kernels from a set). In this section, we describe a method, based on [Duv+13], for sequentially searching through the space of increasingly complex GP models so as to find a parsiminous description of the data.

We start with a simple kernel, such as the white noise kernel, and then consider replacing it with a set of possible alternative kernels, such as an SE kernel, RQ kernel, etc. We use the BIC score (Section 3.9.5.2) to evaluate each candidate model (choice of kernel) m. This has the form $\text{BIC}(m) = \log p(\mathcal{D}|m) - \frac{1}{2}|m| \log N$, where $p(\mathcal{D}|m)$ is the marginal likelihood, and $|m|$ is the number of parameters. The first term measures fit to the data, and the second term is a complexity penalty. We can also consider replacing a kernel by the addition of two kernels, $k \rightarrow (k + k')$, or the multiplication of two kernels, $k \rightarrow (k \times k')$. See Figure 18.21 for an illustration of the search space.

Searching through this space is similar to what a human expert would do. In particular, if we find structure in the residuals, such as periodicity, we can propose a certain "move" through the space. We can also start with some structure that is assumed to hold globally, such as linearity, but if we find this only holds locally, we can multiply the kernel by an SE kernel. We can also add input dimensions incrementally, to capture higher order interactions.

Figure 18.22 shows the output of this process applied to a dataset of monthly totals of international airline passengers. The input to the GP is the set of time stamps, $\boldsymbol{x} = 1 : t$; there are no other features.

The observed data lies in between the dotted vertical lines; curves outside of this region are extrapolations. We see that the system has discovered a fairly interpretable set of patterns in the data. Indeed, it is possible to devise an algorithm to automatically convert the output of this search process to a natural language summary, as shown in [Llo+14]. In this example, it summarizes the data as being generated by the addition of 4 underlying trends: a linearly increasing function; an approximately periodic function with a period of 1.0 years, and with linearly increasing amplitude; a smooth function; and uncorrelated noise with linearly increasing standard deviation.

Recently, [Sun+18] showed how to create a DNN which learns the kernel given two input vectors.

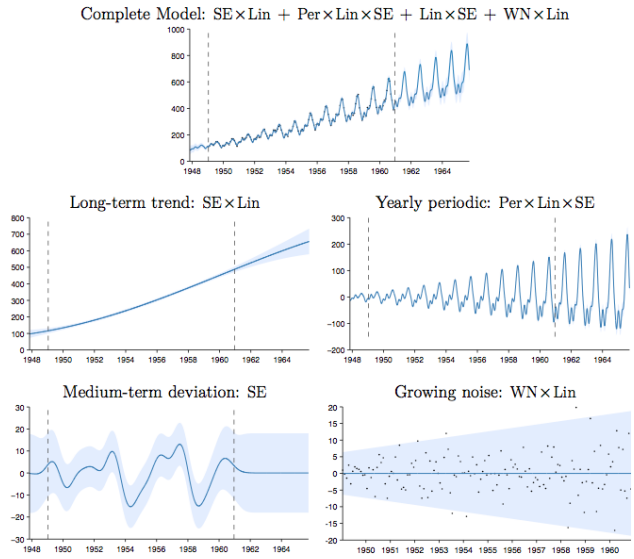


Figure 18.22: Top row: airline dataset and posterior distribution of the model discovered after a search of depth 10. Subsequent rows: predictions of the individual components. From Figure 3.5 of [Duv14], based on [Llo+14]. Used with kind permission of David Duvenaud.

The hidden units are defined as sums and products of elementary kernels, as in the above search based approach. However, the DNN can be trained in a differentiable way, so is much faster.

18.6.5 Spectral mixture kernel learning

Any shift-invariant (stationary) kernel can be converted via the Fourier transform to its dual form, known as its **spectral density**. This means that learning the spectral density is equivalent to learning any shift-invariant kernel. For example, if we take the Fourier transform of an RBF kernel, we get a Gaussian spectral density centered at the origin. If we take the Fourier transform of a Matern kernel, we get a Student-t spectral density centred at the origin. Thus standard approaches to multiple kernel learning, which typically involve additive compositions of RBF and Matern kernels with different length-scale parameters, amount to density estimation with a scale mixture of Gaussian or Student-t distributions at the origin. Such models are very inflexible for density estimation, and thus also very limited in being able to perform kernel learning.

On the other hand, *scale-location* mixture of Gaussians can model any density to arbitrary precision. Moreover, with even a small number of components these mixtures of Gaussians are highly flexible. Thus a spectral density corresponding to a scale-location mixture of Gaussians forms an expressive basis for all shift-invariant kernels. One can evaluate the inverse Fourier transform for a Gaussian mixture analytically, to derive the **spectral mixture kernel** [WA13], which we can express for

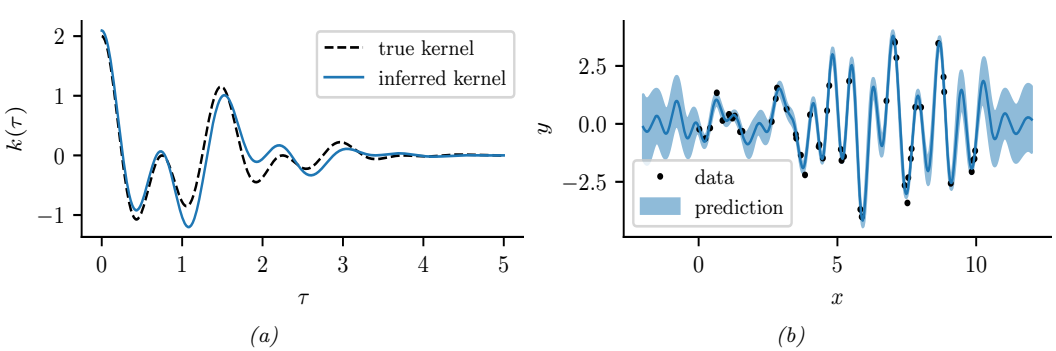


Figure 18.23: Illustration of a GP with a spectral mixture kernel in 1d. (a) Learned vs true kernel. (b) Predictions using learned kernel. Generated by gp_spectral_mixture.ipynb.

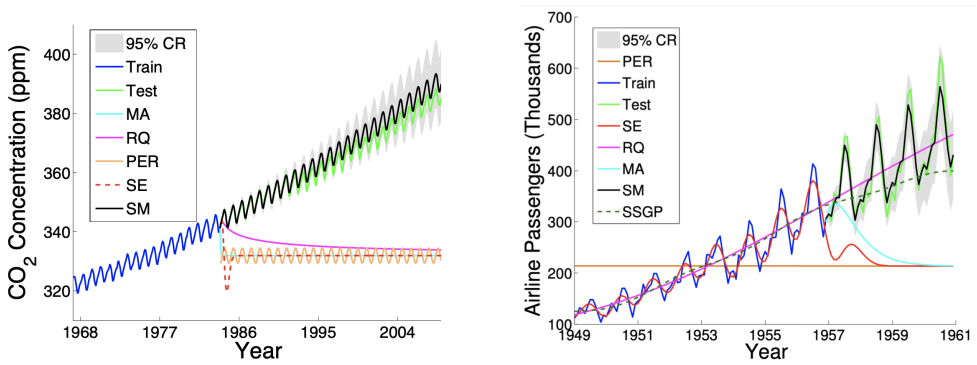


Figure 18.24: Extrapolations (point predictions and 95% credible set) on CO_2 and airline datasets using Gaussian processes with Matern, rational quadratic, periodic, RBF (SE), and spectral mixture kernels, each with hyperparameters learned using empirical Bayes. From [Wil14].

one-dimensional inputs x as:

$$\mathcal{K}(x,x') = \sum_i w_i \cos((x-x')(2\pi\mu_i)) \exp(-2\pi^2(x-x')^2 v_i) \tag{18.172}$$

The mixture weights w_i, as well as the means μ_i and variances v_i of the Gaussians in the spectral density, can be learned by empirical Bayes optimization (Section 18.6.1) or in a fully-Bayesian procedure (Section 18.6.2) [Jan+17]. We illustrate the former approach in Figure 18.23.

By learning the parameters of the spectral mixture kernel, we can discover representations that enable extrapolation — to make reasonable predictions far away from the data. For example, in Section 18.8.1, compositions of kernels are carefully hand-crafted to extrapolate CO_2 concentrations. But in this instance, the human statistician is doing all of the interesting representation learning. Figure Figure 18.24 shows Gaussian processes with learned spectral mixture kernels instead automatically extrapolating on CO_2 and airline passenger problems.

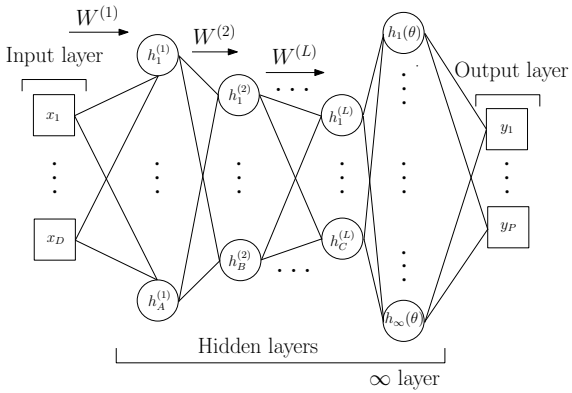


Figure 18.25: Deep Kernel Learning: A Gaussian process with a deep kernel maps D dimensional inputs $\boldsymbol{x}$ through L parametric hidden layers followed by a hidden layer with an infinite number of basis functions, with base kernel hyperparameters $\boldsymbol{\theta}$. Overall, a Gaussian process with a deep kernel produces a probabilistic mapping with an infinite number of adaptive basis functions parametrized by $\boldsymbol{\gamma} = \{\boldsymbol{w}, \boldsymbol{\theta}\}$. All parameters $\boldsymbol{\gamma}$ are learned through the marginal likelihood of the Gaussian process. From Figure 1 of [Wil+16].

These kernels can also be used to extrapolate higher dimensional large-scale spatio-temporal patterns. Large datasets can provide relatively more information for expressive kernel learning. However, scaling an expressive kernel learning approach poses different challenges than scaling a standard Gaussian process model. One faces additional computational constraints, and the need to retain significant model structure for expressing the rich information available in a large dataset. Indeed, in Figure 18.24 we can separately understand the effects of the kernel learning approach and scalable inference procedure, in being able to discover structure necessary to extrapolate textures. An expressive kernel model and a scalable inference approach that preserves a *non-parametric* representation are needed for good performance.

Structure exploiting inference procedures, such as Kronecker methods, as well as KISS-GP and conjugate gradient based approaches, are appropriate for these tasks — since they generally preserve or exploit existing structure, rather than introducing approximations that corrupt the structure. Spectral mixture kernels combined with these scalable inference techniques have been used to great effect for spatiotemporal extrapolation problems, including land-surface temperature forecasting, epidemiological modeling, and policy-relevant applications.

18.6.6 Deep kernel learning

Deep kernel learning [SH07; Wil+16] combines the structural properties of neural networks with the non-parametric flexibility and uncertainty representation provided by Gaussian processes. For example, we can define a "deep RBF kernel" as follows:

$$\mathcal{K}_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{x}') = \exp\left[-\frac{1}{2\sigma^2}||\boldsymbol{h}_{\boldsymbol{\theta}}^L(\boldsymbol{x}) - \boldsymbol{h}_{\boldsymbol{\theta}}^L(\boldsymbol{x}')||^2\right] \tag{18.173}$$

where $\boldsymbol{h}_{\boldsymbol{\theta}}^L(\boldsymbol{x})$ are the outputs of layer L from a DNN. We can then learning the parameters $\boldsymbol{\theta}$ by maximizing the marginal likelihood of the Gaussian processes.

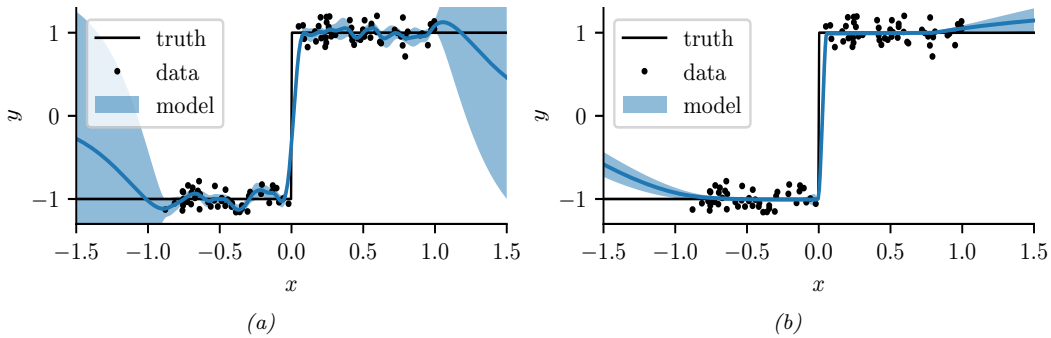


Figure 18.26: Modeling a discontinuous function with (a) a GP with a "shallow" Matern $\frac{3}{2}$ kernel, and (b) a GP with a "deep" MLP + Matern kernel. Generated by gp_deep_kernel_learning.ipynb.

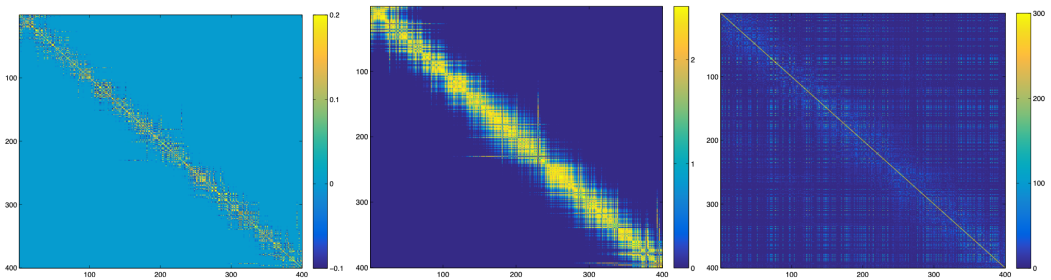


Figure 18.27: **Left***: The learned covariance matrix of a deep kernel with spectral mixture base kernel on a set of test cases for the* **Olivetti faces dataset***, where the test samples are ordered according to the orientations of the input faces.* **Middle***: The respective covariance matrix using a deep kernel with RBF base kernel.* **Right***: The respective covariance matrix using a standard RBF kernel. From Figure 5 of [Wil+16].*

This framework is illustrated in Figure 18.25. We can understand the neural network features as inputs into a base kernel. The neural network can either be (i) pre-trained, (ii) learned jointly with the base kernel parameters, or (iii) pre-trained and then fine-tuned through the marginal likelihood. This approach can be viewed as a "last-layer" Bayesian model, where a Gaussian process is applied to the final layer of a neural network. The base kernel often provides a good measure of distance in feature space, desirably encouraging predictions to have high uncertainty as we move far away from the data.

We can use deep kernel learning to help the GP learn discontinuous functions, as illustrated in Figure 18.26. On the left we show the results of a GP with a standard Matern $\frac{3}{2}$ kernel. It is clear that the out-of-sample predictions are poor. On the right we show the results of the same model where we first transform the input through a learned 2 layer MLP (with 15 and 10 hidden units). It is clear that the model is working much better.

As a more complex example, we consider a regression problem where we wish to map faces (vectors of pixel intensities) to a continuous valued orientation angle. In Figure 18.27, we evaluate the deep kernel matrix (with RBF and spectral mixture base kernels, discussed in Section 18.6.5) on data

ordered by orientation angle. We can see that the learned deep kernels, in the left two panels, have a pronounced diagonal band, meaning that they have *discovered* that faces with similar orientation angles are correlated. On the other hand, in the right panel we see that the entries even for a learned RBF kernel are highly diffuse. Since the RBF kernel essentially uses Euclidean distance as a metric for similarity, it is unable to learn a representation that effectively solves this problem. In this case, one must do highly non-Euclidean metric learning.

However, [ORW21] show that the approach to DKL based on maximizing the marginal likelihood can result in overfitting that is worse than standard DNN learning. They propose a fully Bayesian approach, in which they use SGLD (Section 12.7.1) to sample the DNN weights as well as the GP hyperparameters.

18.7 GPs and DNNs

In Section 18.6.6, we showed how we can combine the structural properties of neural networks with GPs. In Section 18.7.1 we show that, in the limit of infinitely wide networks, a neural network defines a GP with a certain kernel. These kernels are fixed, so the method is not performing representation learning, as a standard neural network would (see e.g., [COB18; Woo+19]). Nonetheless, these kernels are interesting in their own right, for example in modelling non-stationary covariance structure. In Section 18.7.2, we discuss the connection between SGD training of DNNs and GPs. And in Section 18.7.3, we discuss deep GPs, which are similar to DNNs in that they consist of many layers of functions which are composed together, but each layer is a nonparametric function.

18.7.1 Kernels derived from infinitely wide DNNs (NN-GP)

In this section, we show that an MLP with one hidden layer, whose width goes to infinity, and which has a Gaussian prior on all the parameters, converges to a Gaussian process with a well-defined kernel.[4] This result was first shown for in [Nea96; Wil98], and was later extended to deep MLPs in [DFS16; Lee+18], to CNNs in [Nov+19], and to general DNNs in [Yan19]. The resulting kernel is called the **NN-GP** kernel [Lee+18].

We will consider the following model:

$$f_k(\boldsymbol{x}) = b_k + \sum_{j=1}^{H} v_{jk} h_j(\boldsymbol{x}),\ h_j(\boldsymbol{x}) = \varphi(u_{0j} + \boldsymbol{x}^{\mathsf{T}} \boldsymbol{u}_j) \tag{18.174}$$

where H is the number of hidden units, and $\varphi()$ is some nonlinear activation function, such as ReLU. We will assume Gaussian priors on the parameters:

$$b_k \sim \mathcal{N}(0, \sigma_b), v_{jk} \sim \mathcal{N}(0, \sigma_v), u_{0j} \sim \mathcal{N}(0, \sigma_0), \boldsymbol{u}_j \sim \mathcal{N}(0, \boldsymbol{\Sigma}) \tag{18.175}$$

Let $\boldsymbol{\theta} = \{b_k, v_{jk}, u_{0j}, \boldsymbol{u}_j\}$ be all the parameters. The expected output from unit k when applied to one input vector is given by

$$\mathbb{E}_{\boldsymbol{\theta}}\left[f_k(\boldsymbol{x})\right] = \mathbb{E}_{\boldsymbol{\theta}}\left[b_k + \sum_{j=1}^{H} v_{jk} h_j(\boldsymbol{x})\right] = \underbrace{\mathbb{E}_{\boldsymbol{\theta}}\left[b_k\right]}_{=0} + \sum_{j=1}^{H} \underbrace{\mathbb{E}_{\boldsymbol{\theta}}\left[v_{jk}\right]}_{=0} \mathbb{E}_{\boldsymbol{u}}\left[h_j(\boldsymbol{x})\right] = 0 \tag{18.176}$$

4. Our presentation is based on http://cbl.eng.cam.ac.uk/pub/Intranet/MLG/ReadingGroup/presentation_matthias.pdf.

The covariance in the output for unit k when the function is applied to two different inputs is given by the following:[5]

$$\mathbb{E}_{\boldsymbol{\theta}}\left[f_k(\boldsymbol{x})f_k(\boldsymbol{x}')\right] = \mathbb{E}_{\boldsymbol{\theta}}\left[\left(b_k + \sum_{j=1}^{H} v_{jk}h_j(\boldsymbol{x})\right)\left(b_k + \sum_{j=1}^{H} v_{jk}h_j(\boldsymbol{x})\right)\right] \tag{18.177}$$

$$= \sigma_b^2 + \sum_{j=1}^{H} \mathbb{E}_{\boldsymbol{\theta}}\left[v_{jk}^2\right]\mathbb{E}_{\boldsymbol{u}}\left[h_j(\boldsymbol{x})h_j(\boldsymbol{x}')\right] = \sigma_b^2 + \sigma_v^2 H\mathbb{E}_{\boldsymbol{u}}\left[h_j(\boldsymbol{x})h_j(\boldsymbol{x}')\right] \tag{18.178}$$

Now consider the limit $H \to \infty$. We scale the magnitude of the output by defining $\sigma_v^2 = \omega/H$. Since the input to k'th output unit is an infinite sum of random variables (from the hidden units $h_j(\boldsymbol{x})$), we can use the **central limit theorem** to conclude that the output converges to a Gaussian with mean and variance given by

$$\mathbb{E}\left[f_k(\boldsymbol{x})\right] = 0,\ \mathbb{V}\left[f_k(\boldsymbol{x})\right] = \sigma_b^2 + \omega\mathbb{E}_{\boldsymbol{u}}\left[h(\boldsymbol{x})^2\right] \tag{18.179}$$

Furthermore, the joint distribution over $\{f_k(\boldsymbol{x}_n) : n = 1 : N\}$ for any $N \geq 2$ converges to a multivariate Gaussian with covariance given by

$$\mathbb{E}\left[f_k(\boldsymbol{x})f_k(\boldsymbol{x}')\right] = \sigma_b^2 + \omega\mathbb{E}_{\boldsymbol{u}}\left[h(\boldsymbol{x})h(\boldsymbol{x}')\right] \triangleq \mathcal{K}(\boldsymbol{x}, \boldsymbol{x}') \tag{18.180}$$

Thus the MLP converges to a GP. To compute the kernel function, we need to evaluate

$$C(\boldsymbol{x}, \boldsymbol{x}') = \mathbb{E}_{\boldsymbol{u}}\left[h(u_0 + \boldsymbol{u}^{\mathsf{T}}\boldsymbol{x})h(u_0 + \boldsymbol{u}^{\mathsf{T}}\boldsymbol{x}')\right] = \mathbb{E}_{\boldsymbol{u}}\left[h(\tilde{\boldsymbol{u}}^{\mathsf{T}}\tilde{\boldsymbol{x}})h(\tilde{\boldsymbol{u}}^{\mathsf{T}}\tilde{\boldsymbol{x}}')\right] \tag{18.181}$$

where we have defined $\tilde{\boldsymbol{x}} = (1, \boldsymbol{x})$ and $\tilde{\boldsymbol{u}} = (u_0, \boldsymbol{u})$. Let us define

$$\tilde{\boldsymbol{\Sigma}} = \begin{pmatrix} \sigma_0^2 & 0 \\ 0 & \boldsymbol{\Sigma} \end{pmatrix} \tag{18.182}$$

Then we have

$$C(\boldsymbol{x}, \boldsymbol{x}') = \int h(\tilde{\boldsymbol{u}}^{\mathsf{T}}\tilde{\boldsymbol{x}})h(\tilde{\boldsymbol{u}}^{\mathsf{T}}\tilde{\boldsymbol{x}'})\mathcal{N}(\tilde{\boldsymbol{u}}|\boldsymbol{0}, \tilde{\boldsymbol{\Sigma}})d\tilde{\boldsymbol{u}} \tag{18.183}$$

This can be computed in closed form for certain activation functions, as shown in Table 18.4.

This is sometimes called the **neural net kernel**. Note that this is a **nonstationary kernel** (i.e., not a function of $||\boldsymbol{x} - \boldsymbol{x}'||$), and sample paths from it are nearly discontinuous and tend to constant values for large positive or negative inputs, as illustrated in Figure 18.28.

18.7.2 Neural tangent kernel (NTK)

In Section 18.7.1 we derived the NN-GP kernel, under the assumption that all the weights are random. A natural question is: can we derive a kernel from a DNN after it has been trained, or more generally, while it is being trained. It turns out that this can be done, as we show below.

5. We are using the fact that $u \sim \mathcal{N}(0, \sigma^2)$ implies $\mathbb{E}\left[u^2\right] = \mathbb{V}\left[u\right] = \sigma^2$.

$h(\tilde{\boldsymbol{x}})$	$C(\tilde{\boldsymbol{x}}, \tilde{\boldsymbol{x}}')$
$\text{erf}(\tilde{\boldsymbol{x}}^\mathsf{T}\tilde{\boldsymbol{u}})$	$\frac{2}{\pi}\arcsin(f_1(\tilde{\boldsymbol{x}}, \tilde{\boldsymbol{x}}'))$
$\mathbb{I}\left(\tilde{\boldsymbol{x}}^\mathsf{T}\tilde{\boldsymbol{u}} \geq 0\right)$	$\pi - \theta(\tilde{\boldsymbol{x}}, \tilde{\boldsymbol{x}}')$
$\text{ReLU}(\tilde{\boldsymbol{x}}^\mathsf{T}\tilde{\boldsymbol{u}})$	$\frac{f_2(\tilde{\boldsymbol{x}}, \tilde{\boldsymbol{x}}')}{\pi}\sin(\theta(\tilde{\boldsymbol{x}}, \tilde{\boldsymbol{x}}')) + \frac{\pi - \theta(\tilde{\boldsymbol{x}}, \tilde{\boldsymbol{x}}')}{\pi}\tilde{\boldsymbol{x}}^\mathsf{T}\tilde{\boldsymbol{\Sigma}}\tilde{\boldsymbol{x}}'$

Table 18.4: Some neural net GP kernels. Here we define $f_1(\tilde{\boldsymbol{x}}, \tilde{\boldsymbol{x}}') = \frac{2\tilde{\boldsymbol{x}}^\mathsf{T}\tilde{\boldsymbol{\Sigma}}\tilde{\boldsymbol{x}}'}{\sqrt{(1+2\tilde{\boldsymbol{x}}^\mathsf{T}\tilde{\boldsymbol{\Sigma}}\tilde{\boldsymbol{x}})(1+2(\tilde{\boldsymbol{x}}')^\mathsf{T}\tilde{\boldsymbol{\Sigma}}\tilde{\boldsymbol{x}}')}}$, $f_2(\tilde{\boldsymbol{x}}, \tilde{\boldsymbol{x}}') = ||\tilde{\boldsymbol{\Sigma}}^{\frac{1}{2}}\tilde{\boldsymbol{x}}||\,||\tilde{\boldsymbol{\Sigma}}^{\frac{1}{2}}\tilde{\boldsymbol{x}}'||$, $f_3(\tilde{\boldsymbol{x}}, \tilde{\boldsymbol{x}}') = \sqrt{(\tilde{\boldsymbol{x}}^\mathsf{T}\tilde{\boldsymbol{\Sigma}}\tilde{\boldsymbol{x}})((\tilde{\boldsymbol{x}}')^\mathsf{T}\tilde{\boldsymbol{\Sigma}}\tilde{\boldsymbol{x}}')}$, *and* $\theta(\tilde{\boldsymbol{x}}, \tilde{\boldsymbol{x}}') = \arccos(f_3(\tilde{\boldsymbol{x}}, \tilde{\boldsymbol{x}}'))$. *Results are derived in [Wil98; CS09].*

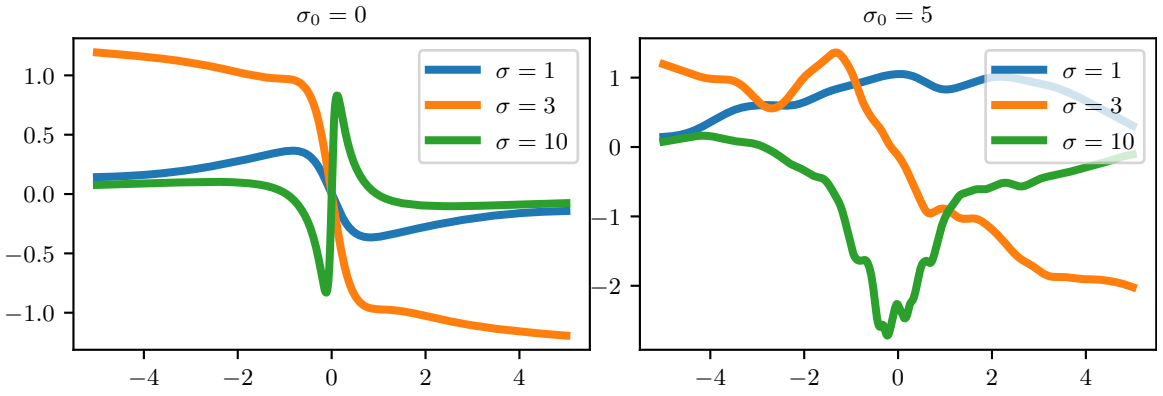


Figure 18.28: Sample output from a GP with an NNGP kernel derived from an infinitely wide one layer MLP with activation function of the form $h(x) = erf(x \cdot u + u_0)$ *where* $u \sim \mathcal{N}(0, \sigma)$ *and* $u_0 \sim \mathcal{N}(0, \sigma_0)$. *Generated by nngp_1d.ipynb. Used with kind permission of Matthias Bauer.*

Let $\boldsymbol{f} = [f(\boldsymbol{x}_n; \theta)]_{n=1}^N$ be the $N \times 1$ prediction vector, let $\nabla_f \mathcal{L} = [\frac{\partial \mathcal{L}}{\partial f(\boldsymbol{x}_n)}]_{n=1}^N$ be the $N \times 1$ loss gradient vector, let $\boldsymbol{\theta} = [\theta_p]_{p=1}^P$ be the $P \times 1$ vector of parameters, and let $\nabla_{\boldsymbol{\theta}}\boldsymbol{f} = [\frac{\partial f(\boldsymbol{x}_n)}{\partial \theta_p}]$ be the $P \times N$ matrix of partials. Suppose we perform continuous time gradient descent with fixed learning rate η. The parameters evolve over time as follows:

$$\partial_t \boldsymbol{\theta}_t = -\eta \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{f}_t) = -\eta \nabla_{\boldsymbol{\theta}} \boldsymbol{f}_t \cdot \nabla_{\boldsymbol{f}} \mathcal{L}(\boldsymbol{f}_t) \tag{18.184}$$

Thus the function evolves over time as follows:

$$\partial_t \boldsymbol{f}_t = \nabla_{\boldsymbol{\theta}} \boldsymbol{f}_t^\mathsf{T} \partial_t \boldsymbol{\theta}_t = -\eta \nabla_{\boldsymbol{\theta}} \boldsymbol{f}_t^\mathsf{T} \nabla_{\boldsymbol{\theta}} \boldsymbol{f}_t \cdot \nabla_{\boldsymbol{f}} \mathcal{L}(\boldsymbol{f}_t) = -\eta \mathcal{T}_t \cdot \nabla_{\boldsymbol{f}} \mathcal{L}(\boldsymbol{f}_t) \tag{18.185}$$

where $\mathcal{T}_t$ is the $N \times N$ kernel matrix

$$\mathcal{T}_t(\boldsymbol{x}, \boldsymbol{x}') \triangleq \nabla_{\boldsymbol{\theta}} f_t(\boldsymbol{x}) \cdot \nabla_{\boldsymbol{\theta}} f_t(\boldsymbol{x}') = \sum_{p=1}^P \frac{\partial f(\boldsymbol{x}; \boldsymbol{\theta})}{\partial \theta_p}\Big|_{\boldsymbol{\theta}_t} \frac{\partial f(\boldsymbol{x}'; \boldsymbol{\theta})}{\partial \theta_p}\Big|_{\boldsymbol{\theta}_t} \tag{18.186}$$

If we let the learning rate η become infinitesimally small, and the widths go to infinity, one can show that this kernel converges to a constant matrix, this is known as the **neural tangent kernel** or

NTK [JGH18]:

$$\mathcal{T}(\boldsymbol{x}, \boldsymbol{x}') \triangleq \nabla_{\boldsymbol{\theta}} f(\boldsymbol{x}; \boldsymbol{\theta}_\infty) \cdot \nabla_{\boldsymbol{\theta}} f(\boldsymbol{x}'; \boldsymbol{\theta}_\infty) \tag{18.187}$$

Details on how to compute this kernel for various models, such as CNNs, graph neural nets, and general neural nets, can be found in [Aro+19; Du+19; Yan19]. A software libary to compute the NN-GP kernel and NTK is available in [Ano19].

The assumptions behind the NTK results in the parameters barely changing from their initial values (which is why a linear approximation around the starting parameters is valid). This can still lead to a change in the final predictions (and zero final training error), because the final layer weights can learn to use the random features just like in kernel regression. However, this phenomenon — which has been called "**lazy training**" [COB18] — is not representative of DNN behavior in practice [Woo+19], where parameters often change a lot. Fortunately it is possible to use a different parameterization which does result in feature learning in the infinite width limit [YH21].

dev

18.7.3 Deep GPs

A **deep Gaussian process** or **DGP** is a composition of GPs [DL13]. More formally, a DGP of L layers is a hierachical model of the form

$$\text{DGP}(\boldsymbol{x}) = f_L \circ \cdots \circ \boldsymbol{f}_1(\boldsymbol{x}), \;\; \boldsymbol{f}_i(\cdot) = [f_i^{(1)}(\cdot), \ldots, f_i^{(H_i)}(\cdot)], \;\; f_i^{(j)} \sim \text{GP}(0, \mathcal{K}_i(\cdot, \cdot)) \tag{18.188}$$

This is similar to a deep neural network, except the hidden nodes are now hidden functions.

A natural question is: what is gained by this approach compared to a standard GP? Although conventional single-layer GPs are nonparametric, and can model any function (assuming the use of a non-degenerate kernel) with enough data, in practice their performance is limited by the choice of kernel. It is tempting to thing that deep kernel learning (Section 18.6.6) can solve this problem, but in theory a GP on top of a DNN is still just a GP. However, one can show that a composition of GPs is strictly more general. Unfortunately, inference in deep GPs is rather complicated. so we leave the details to Supplementary Section 18.1. See also [Jak21] for a recent survey on this topic.

18.8 Gaussian processes for timeseries forecasting

It is possible to use Gaussian processes to perform timeseries forecasting (see e.g., [Rob+13]). The basic idea is to model the unknown output as a function of time, $f(t)$, and to represent a prior about the form of f as a GP; we then update this prior given the observed evidence, and forecast into the future. Naively this would take $O(T^3)$ time. However, for certain stationary kernels, it is possible to reformulate the problem as a linear-Gaussian state space model, and then use the Kalman smoother to perform inference in $O(T)$ time, as explained in [SSH13; SS19; Ada+20]. This conversion can be done exactly for Matern kernels and approximately for Gaussian (RBF) kernels (see [SS19, Ch. 12]). In [SGF21], they describe how to reduce the linear dependence on T to $\log(T)$ time using a parallel prefix scan operator, that can be run efficiently on GPUs (see Section 8.3.3.4).

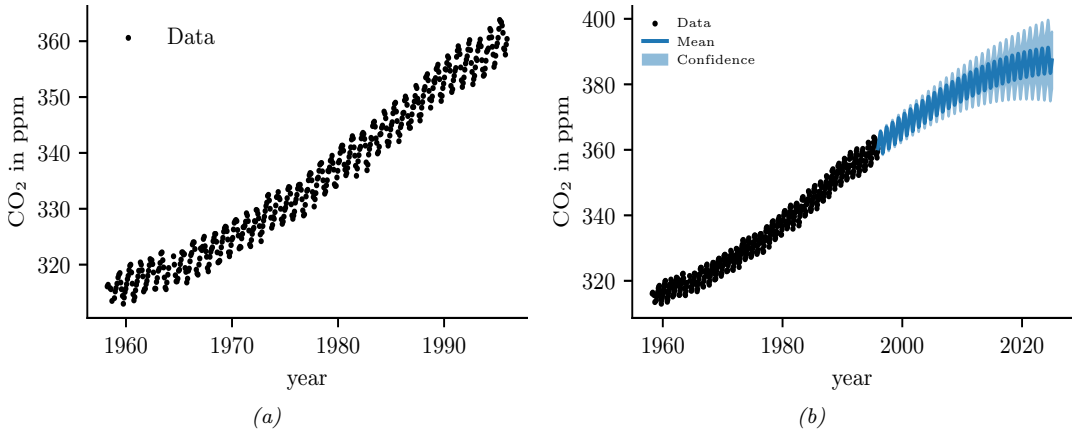


Figure 18.29: (a) The observed Mauna Loa CO_2 time series. (b) Forecasts from a GP. Generated by gp_mauna_loa.ipynb.

18.8.1 Example: Mauna Loa

In this section, we use the Mauna Loa CO_2 dataset from Section 29.12.3. We show the raw data in Figure 18.29(a). We see that there is periodic (or quasi-periodic) signal with a year-long period superimposed on a long term trend. Following [RW06, Sec 5.4.3], we will model this with a composition of kernels:

$$\mathcal{K}(r) = \mathcal{K}_1(r) + \mathcal{K}_2(r) + \mathcal{K}_3(r) + \mathcal{K}_4(r) \tag{18.189}$$

where $\mathcal{K}_i(t, t') = \mathcal{K}_i(t - t')$ for the i'th kernel.

To capture the long term smooth rising trend, we let $\mathcal{K}_1$ be a squared exponential (SE) kernel, where θ_0 is the amplitude and θ_1 is the length scale:

$$\mathcal{K}_1(r) = \theta_0^2 \exp\left(-\frac{r^2}{2\,\theta_1^2}\right) \tag{18.190}$$

To model the periodicity, we can use periodic or exp-sine-squared kernel from Equation (18.19) with a period of 1 year. However, since it is not clear if the seasonal trend is exactly periodic, we multiply this periodic kernel with another SE kernel to allow for a decay away from periodicity; the result is $\mathcal{K}_2$, where θ_2 is the magnitude, θ_3 is the decay time for the periodic component, $\theta_4 = 1$ is the period, and θ_5 is the smoothness of the periodic component.

$$\mathcal{K}_2(r) = \theta_2^2 \exp\left(-\frac{r^2}{2\,\theta_3^2} - \theta_5 \sin^2\left(\frac{\pi\, r}{\theta_4}\right)\right) \tag{18.191}$$

To model the (small) medium term irregularitries, we use a rational quadratic kernel (Equation (18.21)):

$$\mathcal{K}_3(r) = \theta_6^2 \left[1 + \frac{r^2}{2\,\theta_7^2\,\theta_8}\right]^{-\theta_8} \tag{18.192}$$

where θ_6 is the magnitude, θ_7 is the typical length scale, and θ_8 is the shape parameter.

The magnitude of the independent noise can be incorporated into the observation noise of the likelihood function. For the correlated noise, we use another SE kernel:

$$\mathcal{K}_4(r) = \theta_9^2 \exp\left(-\frac{r^2}{2\,\theta_{10}^2}\right) \tag{18.193}$$

where θ_9 is the magnitude of the correlated noise, and θ_{10} is the length scale. (Note that the combination of $\mathcal{K}_1$ and $\mathcal{K}_4$ is non-identifiable, but this does not affect predictions.)

We can fit this model by optimizing the marginal likelihood wrt $\boldsymbol{\theta}$ (see Section 18.6.1). The resulting forecast is shown in Figure 18.29(b).

19 Beyond the iid assumption

19.1 Introduction

The standard approach to supervised ML assumes the training and test sets both contain independent and identically distributed (iid) samples from the same distribution. However, there are many settings in which the test distribution may be different from the training distribution; this is known as **distribution shift**, as we discuss in Section 19.2.

In some cases, we may have data from multiple related distributions, not just train and test, as we discuss in Section 19.6. We may also encounter data in a streaming setting, where the data distribution may be changing continuously, or in a piecewise constant fashion, as we discuss in Section 19.7. Finally, in Section 19.8, we discuss settings in which the test distribution is chosen by an adversary to minimize performance of a prediction system.

19.2 Distribution shift

Suppose we have a labeled training set from a **source distribution** $p(\boldsymbol{x}, \boldsymbol{y})$ which we use to fit a predictive model $p(\boldsymbol{y}|\boldsymbol{x})$. At test time we encounter data from the **target distribution** $q(\boldsymbol{x}, \boldsymbol{y})$. If $p \neq q$, we say that there has been a **distribution shift** or **datatset shift** [QC+08; BD+10]. This can adversely affect the performance of predictive models, as we illustrate in Section 19.2.1. In Section 19.2.2 we give a taxonomy of some kinds of distribution shift using the language of causal graphical models. We then proceed to discuss a variety of strategies that can be adopted to ameliorate the harm caused by distribution shift. In particular, in Section 19.3, we discuss techniques for detecting shifts, so that we can abstain from giving an incorrect prediction if the model is not confident. In Section 19.4, we discuss techniques to improve robustness to shifts; in particular, given labeled data from $p(\boldsymbol{x}, \boldsymbol{y})$, we aim to create a model that approximates $q(\boldsymbol{y}|\boldsymbol{x})$. In Section 19.5, we discuss techniques to adapt the model to the target distribution given some labeled or unlabeled data from the target.

19.2.1 Motivating examples

Figure 19.1 shows how shifting the test distribution slightly, by adding a small amount of Gaussian noise, can hurt performance of an otherwise high accuracy image classifier. Similar effects occur with other kinds of **common corruptions**, such as image blurring [HD19]. Analogous problems can also occur in the text domain [Ryc+19], and the speech domain (c.f. male vs female speakers in

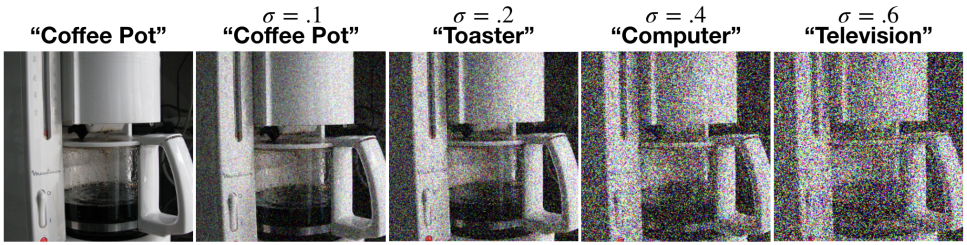


Figure 19.1: Effect of Gaussian noise of increasing magnitude on an image classifier. The model is a ResNet-50 CNN trained on ImageNet. From Figure 23 of [For+19]. Used with kind permission of Justin Gilmer.

(A) **Cow: 0.99**, Pasture: 0.99, Grass: 0.99, No Person: 0.98, Mammal: 0.98

(B) No Person: 0.99, Water: 0.98, Beach: 0.97, Outdoors: 0.97, Seashore: 0.97

(C) No Person: 0.97, **Mammal: 0.96**, Water: 0.94, Beach: 0.94, Two: 0.94

Figure 19.2: Illustration of how image classifiers generalize poorly to new environments. (a) In the training data, most cows ocur on grassy backgrounds. (b-c) In these test image, the cow occurs "out of context", namely on a beach. The background is considered a "spurious correlation". In (b), the cow is not detected. In (c), it is classified with a generic "mammal" label. Top five labels and their confidences are produced by ClarifAI.com, which is a state of the art commerical vision system. From Figure 1 of [BVHP18]. Used with kind permission of Sara Beery.

Figure 34.1). These examples illustrate that high performing predictive models can be very sensitive to small changes in the input distribution.

Performance can also drop on "clean" images, but which exhibit other kinds of shift. Figure 19.2 gives an amusing example of this. In particular, it illustrates how the performance of a CNN image classifier can be very accurate on **in-domain** data, but can be very inaccurate on **out-of-domain** data, such as images with a different background, or taken at a different time or location (see e.g., [Koh+20b]) or from a novel viewing angle (see e.g., [KH22])).

The root cause of many of these problems is the fact that discriminative models often leverage features that are predictive of the output *in the training set*, but which are not reliable in general. For example, in an image classification dataset, we may find that green grass in the background is very predictive of the class label "cow", but this is not a feature that is stable across different distributions; these are called **spurious correlations** or **shortcut features**. Unfortunately, such features are often easier for models to learn, for reasons explained in [Gei+20a; Xia+21; Sha+20;

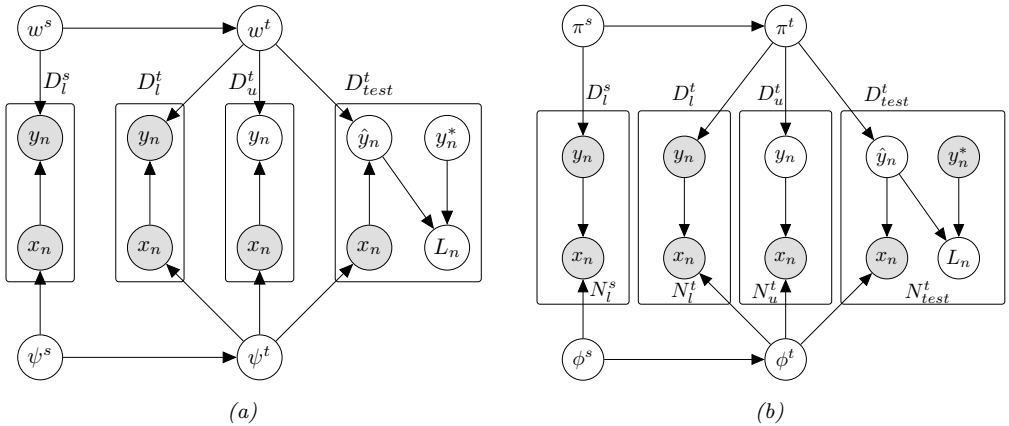


Figure 19.3: Models for distribution shift from source s to target t. Here $\mathcal{D}_L^s$ is the labeled training set from the source, $\mathcal{D}_L^t$ is an optional labeled training set from the target, $\mathcal{D}_U^t$ is an optional unlabeled training set from the target, and $\mathcal{D}_{test}^t$ is a labeled test set from the target. In the latter case, $\hat{\boldsymbol{y}}_n$ is the prediction on the n'th test case (generated by the model), $\boldsymbol{y}_n^*$ is the true value, and $\ell_n = \ell(\boldsymbol{y}_n^*, \hat{\boldsymbol{y}}_n)$ is the corresponding loss. (Note that we don't evaluate the loss on the source distribution.) (a) Discriminative (causal) model. (b) Generative (anti-causal).

Pez+21].

Relying on these shortcuts can have serious real-world consequences. For example, [Zec+18a] found that a CNN trained to recognize pneumonia was relying on hospital-specific metal tokens in the chest X-ray scans, rather than focusing on the lungs themselves, and thus the model did not generalize to new hospitals.

Analogous problems arise with other kinds of ML models, as well as other data types, such as text (e.g., changing "he" to "she" can flip the output of a sentiment analysis system), audio (e.g., adding background noise can easily confuse speech recognition systems), and medical records [Ros22]. Furthermore, the changes to the input needed to change the output can often be imperceptible, as we discuss in the section on adversarial robustness (Section 19.8).

19.2.2 A causal view of distribution shift

In the sections below, we briefly summarize some canonical kinds of distribution shift. We adopt a causal view of the problem, following [Sch+12a; Zha+13b; BP16; Mei18a; CWG20; Bud+21; SCS22]).[1] (See Section 4.7 for a brief discussion of causal DAGs, and Chapter 36 for more details.)

We assume the inputs to the model (the covariates) are X and the outputs to be predicted (the labels) are Y. If we believe that X causes Y, denoted $X \to Y$, we call it **causal prediction**. If we believe that Y causes X, denoted $Y \to X$, we call it **anti-causal prediction** [Sch+12a].

The decision about which model to use depends on our assumptions about the underlying **data**

1. In the causality literature, the question of whether a model can generalize to a new distribution is called the question of **external validity**. If a model is externally valid, we say that it is **transportable** from one distribution to another [BP16].

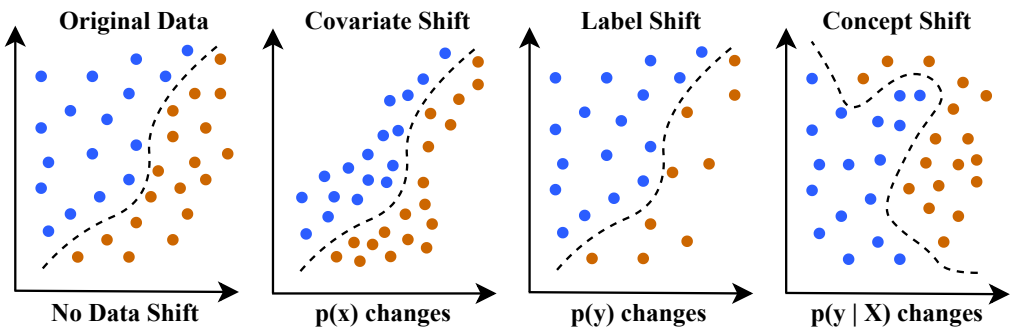


Figure 19.4: Illustration of the 4 main kinds of distribution shift for a 2d binary classification problem. Adapted from Figure 1 of [al21].

generating process. For example, suppose X is a medical image, and Y is an image segmentation created by a human expert or an algorithm. If we change the image, we will change the annotation, and hence $X \to Y$. Now suppose X is a medical image and Y is the ground truth disease state of the patient, as estimated by some other means (e.g., a lab test). In this case, we have $Y \to X$, since changing the disease state will change the appearance of the image. As another example, suppose X is a text review of a movie, and Y is a measure of how informative the review is. Clearly we have $X \to Y$. Now suppose Y is the star rating of the movie, representing the degree to which the user liked it; this will affect the words that they write, and hence $Y \to X$.

Based on the above discussion, we can factor the joint distribution in two possible ways. One way is to define a discriminative (causal) model:

$$p_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{y}) = p_{\boldsymbol{\psi}}(\boldsymbol{x}) p_{\boldsymbol{w}}(\boldsymbol{y}|\boldsymbol{x}) \tag{19.1}$$

See Figure 19.3a. Alternatively we can define a generative (anti-causal) model:

$$p_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{y}) = p_{\boldsymbol{\pi}}(\boldsymbol{y}) p_{\boldsymbol{\phi}}(\boldsymbol{x}|\boldsymbol{y}) \tag{19.2}$$

See Figure 19.3b. For each of these 2 models model types, different parts of the distribution may change from source to target. This gives rise to 4 canonical type of shift, as we discuss in Section 19.2.3.

19.2.3 The four main types of distribution shift

The four main types of distribution shift are summarized in Section 19.2 and are illustrated in Figure 19.4. We give more details below.

19.2.3.1 Covariate shift

In a causal (discriminative) model, if $p_{\boldsymbol{\psi}}(\boldsymbol{x})$ changes (so $\boldsymbol{\psi}^s \neq \boldsymbol{\psi}^t$), we call it **covariate shift**, also called **domain shift**. For example, the training distribution may be clean images of coffee pots, and the test distribution may be images of coffee pots with Gaussian noise, as shown in Figure 19.1; or the

Name	Source	Target	Causal
Covariate / domain shift	$p(X)p(Y\|X)$	$q(X)p(Y\|X)$	Causal
Concept shift	$p(X)p(Y\|X)$	$p(X)q(Y\|X)$	Causal
Label (prior) shift	$p(Y)p(X\|Y)$	$q(Y)p(X\|Y)$	Anti-causal
Manifestation shift	$p(Y)p(X\|Y)$	$p(Y)q(X\|Y)$	Anti-causal

Table 19.1: The 4 main types of distribution shift.

training distribution may be photos of objects in a catalog, with uncluttered white backgrounds, and the test distribution may be photos of the same kinds of objects collected "in the wild"; or the training data may be synthetically generated images, and the test distribution may be real images. Similar shifts can occur in the text domain; for example, the training distribution may be movie reviews written in English, and the test distribution may be translations of these reviews into Spanish.

Some standard strategies to combat covariate shift include importance weighting (Section 19.5.2) and domain adaptation (Section 19.5.3).

19.2.3.2 Concept shift

In a causal (discriminative) model, if $p_{\boldsymbol{w}}(\boldsymbol{y}|\boldsymbol{x})$ changes (so $\boldsymbol{w}^s \neq \boldsymbol{w}^t$), we call it **concept shift**, also called **annotation shift**. For example, consider the medical imaging context: the conventions for annotating images might be different between the training distribution and test distribution. Another example of concept shift occurs when a new label can occur in the target distribution that was not part of the source distribution. This is related to open world recognition, discussed in Section 19.3.4.

Since concept shift is a change in what we "mean" by a label, it is impossible to fix this problem without seeing labeled examples from the target distribution, which defines each label by means of examples.

19.2.3.3 Label / prior shift

In an anti-causal (generative) model, if $p_{\boldsymbol{\pi}}(\boldsymbol{y})$ changes (i.e., $\boldsymbol{\pi}^s \neq \boldsymbol{\pi}^t$), we call it **label shift**, also called **prior shift** or **prevalence shift**. For example, consider the medical imaging context, where $Y = 1$ if the patient has some disease and $Y = 0$ otherwise. If the training distribution is an urban hospital and the test distribution is a rural hospital, then the prevalence of the disease, represented by $p(Y = 1)$, might very well be different.

Some standard strategies to combat label shift are to reweight the output of a discriminative classifier using an estimate of the new label distribution, as we discuss in Section 19.5.4.

19.2.3.4 Manifestation shift

In an anti-causal (generative) model, if $p_{\boldsymbol{\phi}}(\boldsymbol{x}|\boldsymbol{y})$ changes (i.e., $\boldsymbol{\phi}^s \neq \boldsymbol{\phi}^t$), we call **manifestation shift** [CWG20], or **Conditional shift** [Zha+13b]. This is, in some sense, the inverse of concept shift. For example, consider the medical imaging context: the way that the same disease Y manifests itself in the shape of a tumor X might be different. This is usually due to the presence of a hidden confounding factor that has changed between source and target (e.g., different age of the patients).

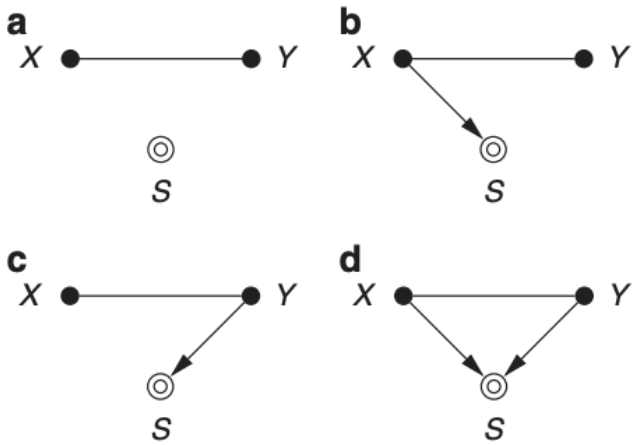


Figure 19.5: Causal diagrams for different sample selection strategies. Undirected edges can be oriented in either direction. The selection variable S is set to 1 its parent nodes match the desired criterion; only these samples are included in the dataset. (a) No selection. (b) Selection on X. (c) Selection on Y. (d) Selection on X and Y. Adapted from Figure 4 of [CWG20].

19.2.4 Selection bias

In some cases, we may induce a shift in the distribution just due to the way the data is collected, without any changes to the underlying distributions. In particular, let $S = 1$ if a sample from the population is included in the training set, and $S = 0$ otherwise. Thus the source distribution is $p(X, Y) = p(X, Y|S = 1)$ but the target distribution is $q(X, Y) = p(X, Y|S \in \{0, 1\}) = p(X, Y)$, so there is no selection.

In Figure 19.5 we visualize the four kinds of selection. For example, suppose we select based on X meeting certain criteria, e.g., images of a certain quality, or exhibiting a certain pattern; this can induce domain shift or covariate shift. Now suppose we select based on Y meeting certain criteria, e.g., we are more likely to select rare examples where $Y = 1$, in order to **balance the dataset** (for reasons of computational efficiency); this can induce label shift. Finally, suppose we select based on both X and Y; this can induce non-causal dependencies between X and Y, a phenomenon known as **selection bias** (see Section 4.2.4.2 for details).

19.3 Detecting distribution shifts

In general it will not be possible to make a model robust to all of the ways a distribution can shift at test time, nor will we always have access to test samples at training time. As an alternative, it may be sufficient for the model to *detect* that a shift has happened, and then to respond in the appropriate way. There are several ways of detecting distribution shift, some of which we summarize below. (See also Section 29.5.6, where we discuss changepoint detection in time series data.) The main distinction between methods is based on whether we have a set of samples from the target

distribution, or just a single sample, and whether the test samples are labeled or unlabeled. We discuss these different scenarios below.

19.3.1 Detecting shifts using two-sample testing

Suppose we collect a set of samples from the source and target distribution. We can then use standard techniques for **two-sample testing** to estimate if the null hypothesis, $p(\boldsymbol{x}, y) = q(\boldsymbol{x}, y)$, is true or not. (If we have unlabeled samples, we just test if $p(\boldsymbol{x}) = q(\boldsymbol{x})$.) For example, we can use MMD (Section 2.7.3) to measure the distance between the set of input samples (see e.g., [Liu+20a]). Or we can measure (Euclidean) distances in the embedding space of a classifier trained on the source (see e.g., [KM22]).

In some cases it may be possible to just test if the distribution of the labels $p(y)$ has changed, which is an easier problem than testing for changes in the distribution of inputs $p(\boldsymbol{x})$. In particular, if the label shift assumption (Section 19.2.3.3) holds (i.e., $q(\boldsymbol{x}|y) = p(\boldsymbol{x}|y)$), plus some other assumptions, then we can use the blackbox shift estimation technique from Section 19.5.4 to estimate $q(y)$. If we find that $q(y) = p(y)$, then we can conclude that $q(\boldsymbol{x}, y) = p(\boldsymbol{x}, y)$. In [RGL19], they showed experimentally that this method worked well for detecting distribution shifts even when the label shift assumption does not hold.

It is also possible to use conformal prediction (Section 14.3) to develop "distribution free" methods for detecting covariate shift, given only acccess to a calibration set and some conformity scoring function [HL20].

19.3.2 Detecting single out-of-distribution (OOD) inputs

Now suppose we just have *one* unlabeled sample from the target distribution, $\boldsymbol{x} \sim q$, and we want to know if $\boldsymbol{x}$ is in-distribution (**ID**) or out-of-distribution (**OOD**). We will call this problem **out-of-distribution detection**, although it is also called **anomaly detection**, and **novelty detection**.[2]

The OOD detection problem requires making a binary decision about whether the test sample is ID or OOD. If it is ID, we may optionally require that we return its class label, as shown in Figure 19.6. In the sections below, we give a brief overview of techniques that have been proposed for tackling this problem, but for more details, see e.g., [Pan+21; Ruf+21; Bul+20; Yan+21; Sal+21; Hen+19b].

19.3.2.1 Supervised ID/OOD methods (outlier exposure)

The simplest method for OOD detection assumes we have access to labeled ID and OOD samples at training time. Then we just fit a binary classifier to distinguish the OOD or background class (called "**known unknowns**") from the ID class (called "**known knowns**") This technique is called **outlier exposure** (see e.g., [HMD19; Thu+21; Bit+21]) and can work well. However, in most cases we will not have enough examples from the OOD distribution, since the OOD set is basically the set of all possible inputs except for the ones of interest.

2. The task of **outlier detection** is somewhat different from anomaly or OOD detection, despite the similar name. In the outlier detection literature, the assumption is that there is a single unlabeled dataset, and the goal is to identify samples which are "untypical" compared to the majority. This is often used for **data cleaning**. (Note that this is a **transductive learning** task, where the model is trained and evaluated on the same data. We focus on inductive tasks, where we train a model on one dataset, and then test it on another.)

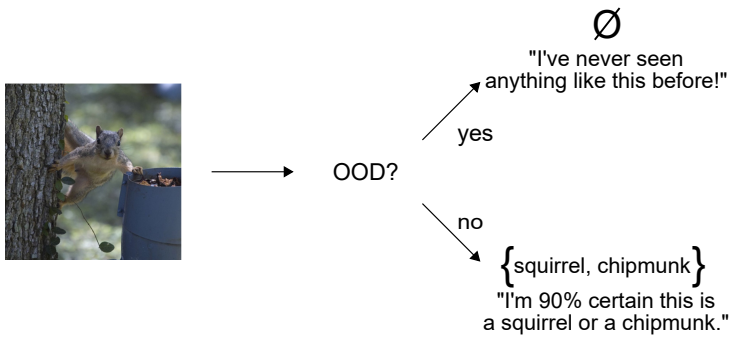


Figure 19.6: Illustration of a two-stage decision problem. First we must decide if the input image is out-of-distribution (OOD) or not. If it is not, we must return the set of class labels that have high probabilitiy. From [AB21]. Used with kind permission of Anastasios Angelopoulos.

19.3.2.2 Classification confidence methods

Instead of trying to solve the binary ID/OOD classification problem, we can directly try to predict the class of the input. Let the probabilities over the C labels be $p_c = p(y = c|\boldsymbol{x})$, and let the logits be $\ell_c = \log p_c$. We can derive a **confidence score** or **uncertainty metric** in a variety of ways from these quantities, e.g., the max probability $s = \max_c p_c$, the margin $s = \max_c \ell_c - \max_c^2 \ell_c$ (where $\max^2$ means the second largest element), the entropy $s = \mathbb{H}(\boldsymbol{p}_{1:C})$[3], the "**energy score**" $s = \sum_c \ell_c$ [Liu+21b], etc. In [Mil+21; Vaz+22] they show that the simple max probability baseline performs very well in practice.

19.3.2.3 Conformal prediction

It is possible to create a method for OOD detection and ID classification that has provably bounded risk using conformal prediction (Section 14.3). The details are in [Ang+21], but we sketch the basic idea here.

We want to solve the two-stage decision problems illustrated in Figure 19.6. We define the prediction set as follows:

$$\mathcal{T}_{\boldsymbol{\lambda}}(\boldsymbol{x}) = \begin{cases} \emptyset & \text{if } \text{OOD}(\boldsymbol{x}) > \lambda_1 \\ \text{APS}(\boldsymbol{x}) & \text{otherwise} \end{cases} \tag{19.3}$$

where $\text{OOD}(\boldsymbol{x})$ is some heuristic OOD score (such as max class probability), and $\text{APS}(\boldsymbol{x})$ is the adaptive prediction set method of Section 14.3.1, which returns the set of the top K class labels, such that the sum of their probabilities exceeds threshold λ_2. (Formally, $\text{APS}(\boldsymbol{x}) = \{\pi_1, \ldots, \pi_K\}$ where π sorts $f(\boldsymbol{x})_{1:C}$ in descending order, and $K = \min\{K' : \sum_{c=1}^{K'} f(\boldsymbol{x})_c > \lambda_2\}$.)

We choose the thresholds λ_1 and λ_2 using a calibration set and a frequentist hypothesis testing

3. [Kir+21] argues against using entropy, since it confuses uncertainty about which of the C labels to use with uncertainty about whether any of the labels is suitable, compared to a "none-of-the-above" option.

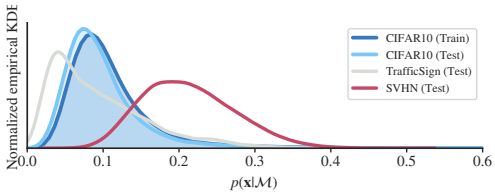


Figure 19.7: Likelihoods from a Glow normalizing flow model (Section 23.2.1) trained on CIFAR10 and evaluated on different test sets. The SVHN street sign dataset has lower visual complexity, and hence higher likelihood. Qualitatively similar results are obtained for other generative models and data set. From Figure 1 of [Ser+20]. Used with kind permission of Joan Serra.

method (see [Ang+21]). The resulting thresholds will jointly minimize the following risks:

$$R_1(\boldsymbol{\lambda}) = p(\mathcal{T}_{\boldsymbol{\lambda}}(\boldsymbol{x}) = \emptyset) \tag{19.4}$$

$$R_2(\boldsymbol{\lambda}) = p(\boldsymbol{y} \notin \mathcal{T}_{\boldsymbol{\lambda}}(\boldsymbol{x}) | \mathcal{T}_{\boldsymbol{\lambda}}(\boldsymbol{x}) \neq \emptyset) \tag{19.5}$$

where $p(\boldsymbol{x}, y)$ is the true but unknown source distribution (of ID samples, no OOD samples required), R_1 is the chance that an ID sample will be incorrectly rejected as OOD (type-I error), and R_2 is the chance (conditional on the decision to classify) that the true label is not in the predicted set. The goal is to set λ_1 as large as possible (so we can detect OOD examples when they arise) while controlling the type-I error (e.g., we may want to ensure that we falsely flag (as OOD) no more than 10% of in-distribution samples). We then set λ_2 in the usual way for the APS method in Section 14.3.1.

19.3.2.4 Unsupervised methods

If we don't have labeled examples, a natural approach to OOD detection is to fit an unconditional density model (such as a VAE) to the ID samples, and then to evaluate the likelihood $p(\boldsymbol{x})$ and compare this to some threshold value. Unfortunately for many kinds of deep model and datasets, we sometimes find that $p(\boldsymbol{x})$ is lower for samples that are from the source distribution than from a novel target distribution. For example, if we train a pixel-CNN model (Section 22.3.2) or a normalizing-flow model (Chapter 23) on Fashion-MNIST and evaluate it on MNIST, we find it gives higher likelihood to the MNIST samples [Nal+19a; Ren+19; KIW20; ZGR21]. This phenomenon occurs for several other models and datasets (see Figure 19.7).

One solution to this is to use log a **likelihood ratio** relative to a baseline density model, $R(\boldsymbol{x}) = \log p(\boldsymbol{x})/q(\boldsymbol{x})$, as opposed to the raw log likelihood, $L(\boldsymbol{x}) = \log p(\boldsymbol{x})$. (This technique was explored in [Ren+19], amongst other papers.) An important advantage of this is that the ratio is invariant to transformations of the data. To see this, let $\boldsymbol{x}' = \boldsymbol{\phi}(\boldsymbol{x})$ be some invertible, but possibly nonlinear, transformation. By the change of variables, we have $p(\boldsymbol{x}') = p(\boldsymbol{x})|\det \text{Jac}(\boldsymbol{\phi}^{-1})(\boldsymbol{x})|$. Thus $L(\boldsymbol{x}')$ will differ from $L(\boldsymbol{x})$ in a way that depends on the transformation. By contrast, we have $R(\boldsymbol{x}) = R(\boldsymbol{x}')$, regardless of $\boldsymbol{\phi}$, since

$$R(\boldsymbol{x}') = \log p(\boldsymbol{x}') - \log q(\boldsymbol{x}') = \log p(\boldsymbol{x}) + \log|\det \text{Jac}(\boldsymbol{\phi}^{-1})(\boldsymbol{x})| - \log q(\boldsymbol{x}) - \log|\det \text{Jac}(\boldsymbol{\phi}^{-1})(\boldsymbol{x})| \tag{19.6}$$

Various other strategies have been proposed, such as computing the log-likelihood adjusted by a measure of the complexity (coding length computed by a lossless compression algorithm) of the input [Ser+20], computing the likelihood of model features instead of inputs [Mor+21a], etc.

A closely related technique relies on **reconstruction error**. The idea is to fit an autoencoder or VAE (Section 21.2) to the ID samples, and then measure the reconstruction error of the input: a sample that is OOD is likely to incur larger error (see e.g. [Pol+19]). However, this suffers from the same problems as density estimation methods.

An alternative to trying to estimate the likelihood, or reconstruct the output, is to use a GAN (Chapter 26) that is trained to discriminate "real" from "fake" data. This has been extended to the open set recognition setting in the **OpenGAN** method of [KR21b].

19.3.3 Selective prediction

Suppose the system has a confidence level of p that an input is OOD (see Section 19.3.4 for a discussion of some ways to compute such confidence scores). If p is below some threshold, the system may choose to **abstain** from classifying it with a specific label. By varying the threshold, we can control the tradeoff between accuracy and abstention rate. This is called **selective prediction** (see e.g., [EW10; GEY19; Ziy+19; JKG18]), and is useful for applications where an error can be more costly than asking a human expert for help (e.g., medical image classification).

19.3.3.1 Example: SGLD vs SGD for MLPs

One way to improve performance of OOD detection is to "be Bayesian" about the parameters of the model, so that the uncertainty in their values is reflected in the posterior predictive distribution. This can result in better performance in selective prediction tasks.

In this section, we give a simple example of this, where we fit a shallow MLP to the MNIST dataset using either standard SGD (specifically RMSprop) or Stochastic Gradient Langevin Dynamics (see Section 12.7.1), which is a form of MCMC inference. We use 6,000 training steps, where each step uses a minibatch of size 1,000. After fitting the model to the training set, we evaluate its predictions on the test set. To assess how well calibrated the model is, we select a subset of predictions whose confidence is above a threshold t. (The confidence value is just the probability assigned to the MAP class.) As we increase the threshold t from 0 to 1, we make predictions on fewer examples, but the accuracy should increase. This is shown in Figure 19.8: the green curve is the fraction of the test set for which we make a prediction, and the blue curve is the accuracy. On the left we show SGD, and on the right we show SGLD. In this case, performance is quite similar, although SGD has slightly higher accuracy. However, the story changes somewhat when there is distribution shift.

To study the effects under distriution shift, we apply both models to FashionMNIST data. We show the results in Figure 19.9. The accuracy of both models is very low (less than the chance level of 10%). but SGD remains quite confident in many more of its predictions than SGLD, which is more conservative. To see this, consider a confidence threshold of 0.5: the SGD approach predicts on about 97% of the examples (recall that the green curve corresponds to the right hand axis), whereas the SGLD only predicts on about 70% of the examples.

More details on the behavior of Bayesian neural networks under distribution shift can be found in Section 17.4.6.2.

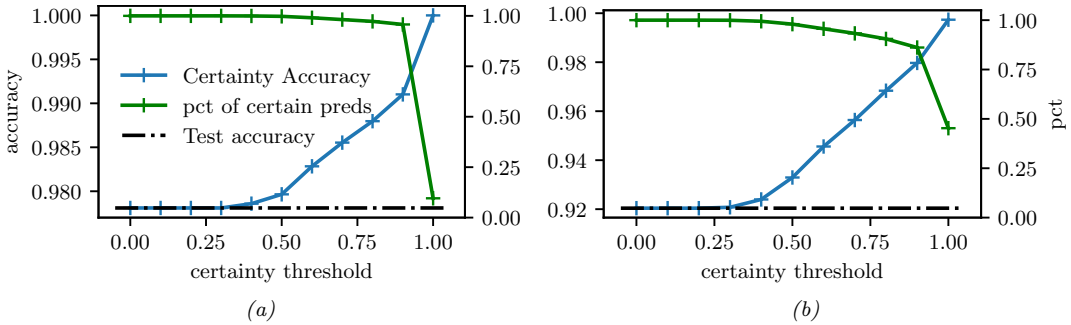


Figure 19.8: Accuracy vs confidence plots for an MLP fit to the MNIST training set, and then evaluated on one batch from the MNIST test set. Scale for blue accuracy curve is on the left, scale for green percentage predicted curve is on the right. (a) Plugin approach, computed using SGD. (b) Bayesian approach, computed using 10 samples from SGLD. Generated by bnn_mnist_sgld.ipynb.

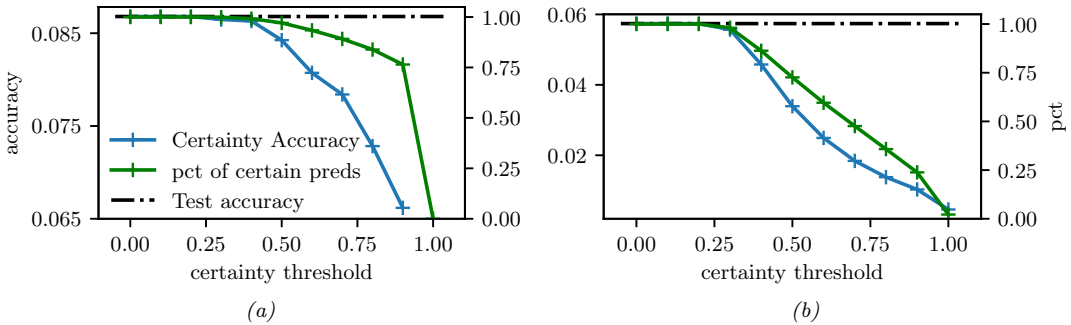


Figure 19.9: Similar to Figure 19.8, except that performance is evaluated on the Fashion MNIST dataset. (a) SGD. (b) SGLD. Generated by bnn_mnist_sgld.ipynb.

19.3.4 Open world recognition

In Section 19.3.3, we discussed methods that "refuse to classify" if the input is suspected to be OOD, i.e., is not one of the existing classes. An alternative approach is to treat the input as an example from a new class. If the set of classes is allowed to grow over time in this way, the problem is called **open world classification** [BB15a].[4] Note that open world classification is most naturally tackled in the context of a continual learning system, which we discuss in Section 19.7.3.

19.4 Robustness to distribution shifts

In this section, we discuss techniques to improve the **robustness** of a model to distribution shifts. In particular, given labeled data from $p(\boldsymbol{x}, \boldsymbol{y})$, we aim to create a model that approximates $q(\boldsymbol{y}|\boldsymbol{x})$.

4. This is not to be confused with **open set recognition**, which refers to classification problems in which the system should label samples from unknown classes as OOD, rather than trying to incrementally learn about new classes. See e.g., [GHC20] for a review of open set recognition methods.

19.4.1 Data augmentation

A simple approach to potentially increasing the robustness of a predictive model to distribution shifts is to simulate samples from the target distribution by modifying the source data. This is called **data augmentation**, and is widely used in the deep learning community. For example, it is standard to apply small perturbations to images (e.g., shifting them or rotating them), while keeping the label the same (assuming that the label should be invariant to such changes); see e.g., [SK19; Hen+20] for details. Similarly, in NLP (natural language processing), it is standard to change words that should not affect the label (e.g., replacing "he" with "she" in a sentiment analysis system), or to use **back translation** (from a source language to a target language and back) to generate paraphrases; see e.g., [Fen+21] for a review of such techniques. For a causal perspective on data augmentation, see e.g., [Kau+21].

19.4.2 Distributionally robust optimization

We can make a discriminative model that is robust to (some forms of) covariate shift by solving the following **distributionally robust optimization** (DRO) problem:

$$\min_{f \in \mathcal{F}} \max_{\boldsymbol{w} \in \mathcal{W}} \frac{1}{N} \sum_{n=1}^{N} w_n \ell(f(\boldsymbol{x}_n), \boldsymbol{y}_n) \tag{19.7}$$

where the samples are from the source distribution, $(\boldsymbol{x}_n, \boldsymbol{y}_n) \sim p$. This is an example of a **min-max optimization problem**, in which we want to minimize the worst case risk. The specification of the robustness set, $\mathcal{W}$, is a key factor that determines how well the method works, and how difficult the optimization problem is. Typically it is specified in terms of an ℓ_2 ball around the inputs, but this could also be defined in a feature (embedding space) It is also possible to define the robustness set in terms of local changes to a structural causal model [Mei18a]. For more details on DRO, see e.g., [WYG14; Hu+18; CP20a; LFG21; Sag+20].

19.5 Adapting to distribution shifts

In this section, we discuss techniques to **adapt** the model to the target distribution. If we have some labeled data from the target distribution, we can use transfer learning, as we discuss in Section 19.5.1. However, getting labeled data from the target distribution is often not an option. Therefore, in the other sections, we discuss techniques that just rely on *unlabeled* data from the target distribution.

19.5.1 Supervised adaptation using transfer learning

Suppose we have labeled training data from a source distribution, $\mathcal{D}^s = \{(\boldsymbol{x}_n, \boldsymbol{y}_n) \sim p : n = 1 : N_s\}$, and also some some labeled data from the target distribution, $\mathcal{D}^t = \{(\boldsymbol{x}_n, \boldsymbol{y}_n) \sim q : n = 1 : N_t\}$. Our goal is to minimize the risk on the target distibution q, which can be computed using

$$R(f, q) = \mathbb{E}_{q(\boldsymbol{x}, \boldsymbol{y})} \left[\ell(\boldsymbol{y}, f(\boldsymbol{x}))\right] \tag{19.8}$$

We can approximate the risk empirically using

$$\hat{R}(f, \mathcal{D}^t) = \frac{1}{|\mathcal{D}^t|} \sum_{(\boldsymbol{x}_n, \boldsymbol{y}_n) \in \mathcal{D}^t} \ell(\boldsymbol{y}_n, f(\boldsymbol{x}_n)) \tag{19.9}$$

If $\mathcal{D}^t$ is large enough, we can directly optimize this using standard empirical risk minimization (ERM). However, if $\mathcal{D}^t$ is small, we might want to use $\mathcal{D}^s$ somehow as a regularizer. This is called **transfer learning**, since we hope to "transfer knowledge" from p to q. There are many approaches to transfer learning (see e.g., [Zhu+21] for a review). We briefly mention a few below.

19.5.1.1 Pre-train and fine-tune

The simplest and most widely used approach to transfer learning is the **pre-train and fine-tune** approach. We first fit a model to the source distribution by computing $f^s = \text{argmin}_f \hat{R}(f, \mathcal{D}^s)$. (Note that the source data may be unlabeled, in which case we can use self-supervised learning methods.) We then adapt the model to work on the target distribution by computing

$$f^t = \underset{f}{\text{argmin}}\, \hat{R}(f, \mathcal{D}^t) + \lambda ||f - f^s|| \tag{19.10}$$

where $||f - f^s||$ is some distance between the functions, and $\lambda \geq 0$ controls the degree of regularization.

Since we assume that we have very few samples from the target distribution, we typically "freeze" most of the parameters of the source model. (This makes an implicit assumption that the features that are useful for the source distribution also work well for the target.) We can then solve Equation (19.10) by "chopping off the head" from f^s and replacing it with a new linear layer, to map to the new set of labels for the target distribution, and then compute a new MAP estimate for the parameters on the target distribution. (We can also compute a prior for the parameters of the source model, and use it to compute a posterior for the parameters of the target model, as discussed in Section 17.2.3.)

This approach is very widely used in practice, since it is simple and effective. In particular, it is common to take a large pre-trained model, such as a transformer, that has been trained (often using self supervised learning, Section 32.3.3) on a lot of data, such as the entire web, and then to use this model as a feature extractor (see e.g., [Kol+20]). The features are fed to the downstream model, which may be a linear classifier or a shallow MLP, which is trained on the target distribution.

19.5.1.2 Prompt tuning (in-context learning)

Recently another approach to transfer learning has been developed, that leverages large models, such as transformers (Section 22.4), which are trained on massive web datasets, usually in an unsupervised way, and then adapted to a small, task-specific target distribution. The interesting thing about this approach is the parameters of the original model are not changed; instead, the model is simply "conditioned" on new training data, usually in the form of a text **prompt** $\boldsymbol{z}$. That is, we compute

$$f^t(\boldsymbol{x}) = f^s(\boldsymbol{x} \cup \boldsymbol{z}) \tag{19.11}$$

where we (manually or automatically) optimize $\boldsymbol{z}$ while keeping f^s frozen. This approach is called **prompt tuning** or **in-context learning** (see e.g., [Liu+21a]), and is an instance of **few-shot learning** (see Figure 22.5 for an example).

Here $\boldsymbol{z}$ acts like a small training dataset, and f^s uses attention (Section 16.2.7) to "look at" all its inputs, comparing $\boldsymbol{x}$ with the examples in $\boldsymbol{z}$, and uses this to make a prediction. This works because the text training data often has a similar hierarchical structure (see [Xie+22] for a Bayesian interpretation).

19.5.2 Weighted ERM for covariate shift

In this section we reconsider the risk minimization objective in Equation (19.8), but leverage unlabeled data from the target distribution to estimate it. If we make the covariate shift assumption (i.e., $q(\boldsymbol{x}, \boldsymbol{y}) = q(\boldsymbol{x})p(\boldsymbol{y}|\boldsymbol{x})$), then we have

$$R(f, q) = \int q(\boldsymbol{x})q(\boldsymbol{y}|\boldsymbol{x})\ell(\boldsymbol{y}, f(\boldsymbol{x}))d\boldsymbol{x}d\boldsymbol{y} \tag{19.12}$$

$$= \int q(\boldsymbol{x})p(\boldsymbol{y}|\boldsymbol{x})\ell(\boldsymbol{y}, f(\boldsymbol{x}))d\boldsymbol{x}d\boldsymbol{y} \tag{19.13}$$

$$= \int \frac{q(\boldsymbol{x})}{p(\boldsymbol{x})}p(\boldsymbol{x})p(\boldsymbol{y}|\boldsymbol{x})\ell(\boldsymbol{y}, f(\boldsymbol{x}))d\boldsymbol{x}d\boldsymbol{y} \tag{19.14}$$

$$\approx \frac{1}{N}\sum_{(\boldsymbol{x}_n, \boldsymbol{y}_n)\in\mathcal{D}_L^s} w_n\ell(\boldsymbol{y}_n, f(\boldsymbol{x}_n)) \tag{19.15}$$

where the weights are given by the ratio

$$w_n = w(\boldsymbol{x}_n) = \frac{q(\boldsymbol{x}_n)}{p(\boldsymbol{x}_n)} \tag{19.16}$$

Thus we can solve the covariate shift problem by using **weighted ERM**.

However, this raises two questions. First, why do we need to use this technique, since a discriminative model $p(\boldsymbol{y}|\boldsymbol{x})$ should work for any input $\boldsymbol{x}$, regardless of which distribution it comes from. Second, given that we do need to use this method, in practice how should we estimate the weights $w_n = w(\boldsymbol{x}_n) = \frac{q(\boldsymbol{x}_n)}{p(\boldsymbol{x}_n)}$. We discuss these issues below.

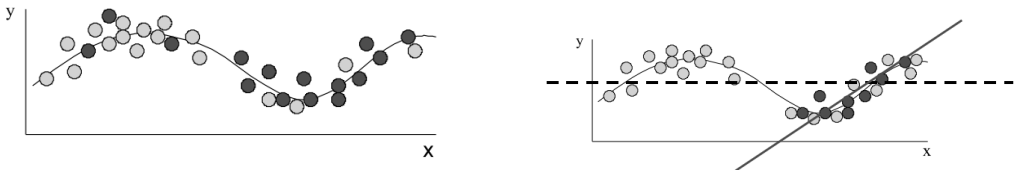
19.5.2.1 Why is covariate shift a problem for discriminative models?

For a discriminative model of the form $p(\boldsymbol{y}|\boldsymbol{x})$, it might seem that such a change in $p(\boldsymbol{x})$ will not affect the predictions. If the predictor $p(y|\boldsymbol{x})$ is the correct model for all parts of the input space $\boldsymbol{x}$, then this conclusion is warranted. However, most models will only be accurate in certain parts of the input space. This is illustrated in Figure 19.10b, where we show that a linear model fit to the source distribution may perform much worse on the target distribution than a model that weights target points more heavily during training.

19.5.2.2 How should we estimating the ERM weights?

One approach to estimating the ERM weights $w_n = w(\boldsymbol{x}_n) = \frac{q(\boldsymbol{x}_n)}{p(\boldsymbol{x}_n)}$ is to learn a density model for the source and target. However, density esimation is difficult for high dimensional features. An alternative approach is to try to approximate the density ratio, by fitting a binary classifier to distinguish the two distributions, as discussed in Section 2.7.5. In particular, suppose we have an equal number of samples from $p(\boldsymbol{x})$ and $q(\boldsymbol{x})$. Let us label the first set with $c = -1$ and the second set with $c = 1$. Then we have

$$p(c = 1|\boldsymbol{x}) = \frac{q(\boldsymbol{x})}{q(\boldsymbol{x}) + p(\boldsymbol{x})} \tag{19.17}$$



(a) (b)

Figure 19.10: (a) Illustration of covariate shift. Light gray represents training distribution, dark gray represents test distribution. We see the test distribution has shifted to the right but the underlying input-output function is constant. (b) Dashed line: fitting a linear model across the full support of X. Solid black line: fitting the same model only on parts of input space that have high likelihood under the test distribution. From Figures 1–2 of [Sto09]. Used with kind permission of Amos Storkey.

and hence $\frac{p(c=1|\boldsymbol{x})}{p(c=-1|\boldsymbol{x})} = \frac{q(\boldsymbol{x})}{p(\boldsymbol{x})}$. If the classifier has the form $f(\boldsymbol{x}) = p(c=1|\boldsymbol{x}) = \sigma(h(\boldsymbol{x})) = \frac{1}{1+\exp(-h(\boldsymbol{x}))}$, where $h(\boldsymbol{x})$ is the prediction function that returns the logits, then the importance weights are given by

$$w_n = \frac{1/(1+\exp(-h(\boldsymbol{x}_n)))}{\exp(-h(\boldsymbol{x}_n))/(1+\exp(-h(\boldsymbol{x}_n)))} = \exp(h(\boldsymbol{x}_n)) \tag{19.18}$$

Of course this method requires that $\boldsymbol{x}$ values that may occur in the test distribution should also be possible in the training distribution, i.e. $q(\boldsymbol{x}) > 0 \implies p(\boldsymbol{x}) > 0$. Hence there are no guarantees about this method being able to interpolate beyond the training distribution.

19.5.3 Unsupervised domain adaptation for covariate shift

We now turn to methods that only need access to unlabeled examples from the target distribution.

The technique of **unsupervised domain adaptation** or **UDA** assumes access to a labeled dataset from the source distribution, $\mathcal{D}_1 = \mathcal{D}_L^s \sim p(\boldsymbol{x}, \boldsymbol{y})$ and an unlabeled dataset from the target distribution, $\mathcal{D}_2 = \mathcal{D}_U^t \sim q(\boldsymbol{x})$. It then uses the unlabeled target data to improve robustness or invariance of the predictor, rather than using a weighted ERM method.

There are many forms of UDA (see e.g., [KL21; CB20] for reviews). Here we just focus on one method, called **domain adversarial learning** [Gan+16a]. Let $f_{\boldsymbol{\alpha}} : \mathcal{X}_1 \cup \mathcal{X}_2 \to \mathcal{H}$ be a feature extractor defined on the two input domains, let $c_{\boldsymbol{\beta}} : \mathcal{H} \to \{1, 2\}$ be a classifier that maps from the feature space to the domain from which the input was taken, either domain 1 or 2 (source or target), and let $g_{\boldsymbol{\gamma}} : \mathcal{H} \to \mathcal{Y}$ be a classifier that maps from the feature space to the label space. We want to train the feature extractor so that it cannot distinguish whether the input is coming from the source or target distribution; in this case, it will only be able to use features that are common to both domains. Hence we optimize

$$\min_{\boldsymbol{\gamma}} \max_{\boldsymbol{\alpha},\boldsymbol{\beta}} \frac{1}{N_1+N_2} \sum_{\boldsymbol{x}_n \in \mathcal{D}_1, \mathcal{D}_2} \ell(d_n, c_{\boldsymbol{\beta}}(f_{\boldsymbol{\alpha}}(\boldsymbol{x}_n))) + \frac{1}{N_1} \sum_{(\boldsymbol{x}_n, \boldsymbol{y}_n) \in \mathcal{D}_1} \ell(\boldsymbol{y}_n, g_{\boldsymbol{\gamma}}(f_{\boldsymbol{\alpha}}(\boldsymbol{x}_n))) \tag{19.19}$$

The objective in Equation (19.19) minimizes the loss on the desired task of classifying y, but *maximizes*

the loss on the auxiliary task of classifying the domain label d. This can be implemented by the **gradient sign reversal** trick, and is related to GANs (Section 26.7.6).

19.5.4 Unsupervised techniques for label shift

In this section, we describe an approach known as **black box shift estimation**, due to [LWS18], which can be used to tackle the label shift problem in an unsupervised way. We assume that the only thing that changes in the target distribution is the label prior, i.e., if the source distribution is denoted by $p(\boldsymbol{x}, \boldsymbol{y})$ and target distribution is denoted by $q(\boldsymbol{x}, \boldsymbol{y})$, we assume $q(\boldsymbol{x}, \boldsymbol{y}) = p(\boldsymbol{x}|\boldsymbol{y})q(\boldsymbol{y})$.

First note that, for any deterministic function $f : \mathcal{X} \to \mathcal{Y}$, we have

$$p(\boldsymbol{x}|y) = q(\boldsymbol{x}|y) \implies p(f(\boldsymbol{x})|y) = q(f(\boldsymbol{x})|y) \implies p(\hat{y}|y) = q(\hat{y}|y) \tag{19.20}$$

where $\hat{y} = f(\boldsymbol{x})$ is the predicted label. Let $\mu_i = q(\hat{y} = i)$ be the empirical fraction of times the model predicts class i on the test set, and let $q(y = i)$ be the true but unknown label distribution on the test set, and let $C_{ij} = p(\hat{y} = i|y = j)$ be the class confusion matrix estimated on the training set. Then we have

$$\mu_{\hat{y}} = \sum_y q(\hat{y}|y)q(y) = \sum_y p(\hat{y}|y)q(y) = \sum_y p(\hat{y}, y)\frac{q(y)}{p(y)} \tag{19.21}$$

We can write this in matrix-vector form as follows:

$$\mu_i = \sum_i C_{ij}q_j, \implies \boldsymbol{\mu} = \mathbf{C}\boldsymbol{q} \tag{19.22}$$

Hence we can solve $\boldsymbol{q} = \mathbf{C}^{-1}\boldsymbol{\mu}$, providing that $\mathbf{C}$ is not singular (this will be the case if $\mathbf{C}$ is strongly diagonal, i.e., the model predicts class y_i correctly more often than any other class y_j). We also require that for every $q(y) > 0$ we have $p(y) > 0$, which means we see every label at training time.

Once we know the new label distribution, $q(\boldsymbol{y})$, we can adjust our discriminative classifier to take the new label prior into account as follows:

$$q(y|\boldsymbol{x}) = \frac{q(\boldsymbol{x}|y)q(y)}{q(\boldsymbol{x})} = \frac{p(\boldsymbol{x}|y)q(y)}{q(\boldsymbol{x})} = \frac{p(y|\boldsymbol{x})p(\boldsymbol{x})}{p(y)}\frac{q(y)}{q(\boldsymbol{x})} = p(y|\boldsymbol{x})\frac{q(y)}{p(y)}\frac{p(\boldsymbol{x})}{q(\boldsymbol{x})} \tag{19.23}$$

We can safely ignore the $\frac{p(\boldsymbol{x})}{q(\boldsymbol{x})}$ term, which is constant wrt y, and we can plug in our estimates of the label distributions to compute the $\frac{q(y)}{p(y)}$.

In summary, there are three requirements for this method: (1) The confusion matrix is invertible; (2) no new labels at test time; (3) the only thing that changes is the label prior. If these three conditions hold, the above approach is a valid estimator. See [LWS18] for the details.

19.5.5 Test-time adaptation

In some settings, it is possible to continuously update the model parameters. This allows the model to adapt to changes in the input distribution. This is called **test time adaptation** or **TTA**. The difference from the unsupervised domain adaptation methods of Section 19.5.3 is that, in the online setting, we just have the model which was trained on the source, and not to the source distribution.

In [Sun+20] they proposed an approach called **TTT** ("test-time training") for adapting a discriminative model. In this approach, a self-supervised proxy task is used to create pseudo-labels, which can then be used to adapt the model at run time. In more detail, suppose we create a Y-structured network, where we first perform feature extraction, $\boldsymbol{x} \to \boldsymbol{h}$, and then use $\boldsymbol{h}$ to predict the output $\boldsymbol{y}$ and some proxy output $\boldsymbol{r}$, such as the angle of rotation of the input image. The rotation angle is known if we use data augmentation. Hence we can apply this technique at test time, even if $\boldsymbol{y}$ is unknown, and update the $\boldsymbol{x} \to \boldsymbol{h} \to \boldsymbol{r}$ part of the network, which influences the prediction for $\boldsymbol{y}$ via the shared bottleneck (feature layer) $\boldsymbol{h}$.

Of course, if the proxy output, such as the rotation angle, is not known, we cannot use proxy-supervised learning methods such as TTT. In [Wan+20a], they propose an approach, inspired by semi-supervised learning methods, which they call call **TENT**, which stands for "test-time adaptation by entropy minimization". The idea is to update the classifier parameters to minimize the entropy of the predictive distribution on a batch of test examples. In [Goy+22], they give a justification for this heuristic from the meta-learning perspective. In [ZL21], they present a Bayesian version of TENT, which they call **BACS**, which stands for "Bayesian adaptation under covariate shift", In [ZLF21], they propose a method called **MEMO** ("marginal entropy minimization with one test point") that can be used for any architecture. The idea is, once again, to apply data augmentation at test time to the input $\boldsymbol{x}$, to create a set of inputs, $\tilde{\boldsymbol{x}}_1, \ldots, \tilde{\boldsymbol{x}}_B$. Now we update the parameters so as to minimize the predictive entropy produced by the averaged distribution

$$\overline{p}(\boldsymbol{y}|\boldsymbol{x}, \boldsymbol{w}) = \frac{1}{B} \sum_{b=1}^{B} p(\boldsymbol{y}|\tilde{\boldsymbol{x}}_b, \boldsymbol{w}) \tag{19.24}$$

This ensures that the model gives the same predictions for each perturbation of the input, and that the predictions are confident (low entropy).

An alternative to entropy based methods is to use pseudo-labels (predicted outputs from the source model), and then to self-train on these. This has been shown to work well for unsupervised domain adaptation [KML20]. In the context of TTA, [Che+22] use this technique, combined with contrastive learning, to adapt a classifier to a new distribution.

19.6 Learning from multiple distributions

In Section 19.2, we discussed the setting in which a model is trained on a single source distribution, and then evaluated on a distinct target distribution. In this section, we generalize this to a setting in which the model is trained on data from $J \geq 2$ source distributions, before being tested on data from a target distribution. This includes a variety of different problem settings, depending on the value of J, as we summarize in Figure 19.11.

19.6.1 Multi-task learning

In **multi-task learning** (MTL) [Car97], we have labeled data from J different distributions, $\mathcal{D}^j = \{(\boldsymbol{x}_n^j, \boldsymbol{y}_n^j) : n = 1 : N_j\}$, and the goal is to learn a model that predicts well on all J of them simultaneously, where $f(\boldsymbol{x}, j) : \mathcal{X} \to \mathcal{Y}_j$ is the output for the j'th task. For example, we might want to map a color image of size $H \times W \times 3$ to a set of semantic labels per pixel, $\mathcal{Y}^1 = \{1, \ldots, C\}^{HW}$, as well as a set of predicted depth values per pixel, $\mathcal{Y}^2 = \mathbb{R}^{HW}$. We can do this using ERM where we

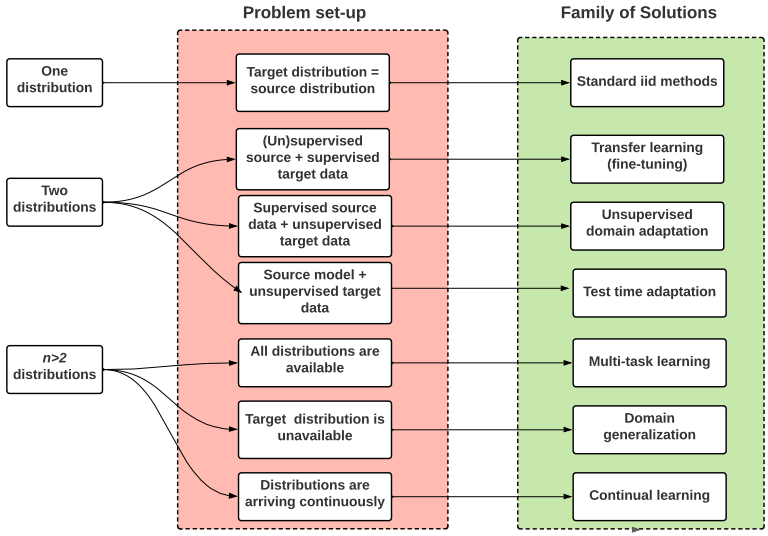


Figure 19.11: Schematic overview of techniques for learning from 1 or more different distributions. Adapted from slide 3 of [Sca21].

have multiple samples for each task:

$$f^* = \operatorname*{argmin}_f \sum_{j=1}^{J} \sum_{n=1}^{N_j} \ell_j(\boldsymbol{y}_n^j, f(\boldsymbol{x}_n^j, j)) \tag{19.25}$$

where ℓ_j is the loss function for task j.

There are many approaches to solving MTL. The simplest is to fit a single model with multiple "output heads", as illustrated in Figure 19.12. The main challenge is defining a suitable architecture (i.e., deciding which parts of the feature extractor to share across tasks), and how to balance the different losses from each task. See [BLS11] for a theoretical analysis of this problem, and [ZY21] for a more detailed review of neural approaches.

Note that multi-task learning does not always help performance on each task because sometimes there can be "**task interference**" or "**negative transfer**" (see e.g., [WZR20]). In such cases, we should use separate networks, rather than using one model with multiple output heads.

19.6.2 Domain generalization

The problem of **domain generalization** assumes we train on J different labeled source distributions or "**environments**" (also called "**domains**"), and then test on a new target distribution (denoted by $t = J + 1$). In some cases each environment is just identified with a meaningless integer id. In more realistic settings, each different distribution has associated **meta-data** or **context variables** that characterizes the environment in which the data was collected, such as the time, location, imaging device, etc.

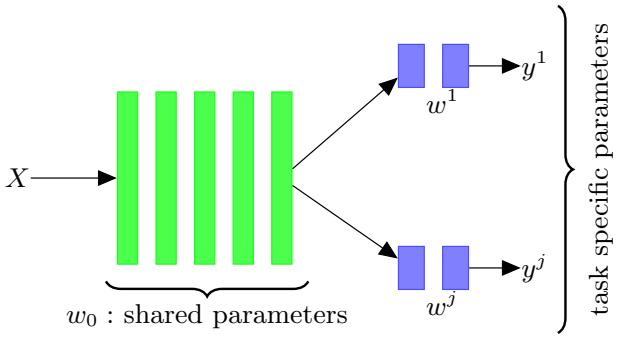


Figure 19.12: Illustration of multi-headed network for multi-task learning.

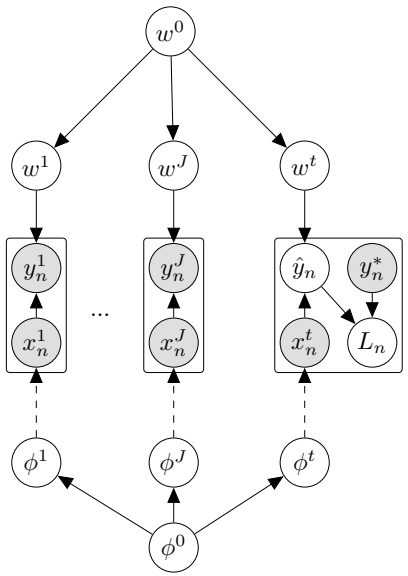


Figure 19.13: Hierarchical Bayesian discriminative model for learning from J different environments (distributions), and then testing on a new target distribution $t = J + 1$. Here $\hat{\boldsymbol{y}}_n$ is the prediction for test example $\boldsymbol{x}_n$, $\boldsymbol{y}_n^*$ is the true output, and $\ell_n = \ell(\boldsymbol{y}_n^t, \boldsymbol{y}_n^*)$ is the associated loss. The parameters of the distribution over input features $p_{\boldsymbol{\phi}}(\boldsymbol{x})$ are shown with dotted edges, since these distributions do not need to be learned in a discriminative model.

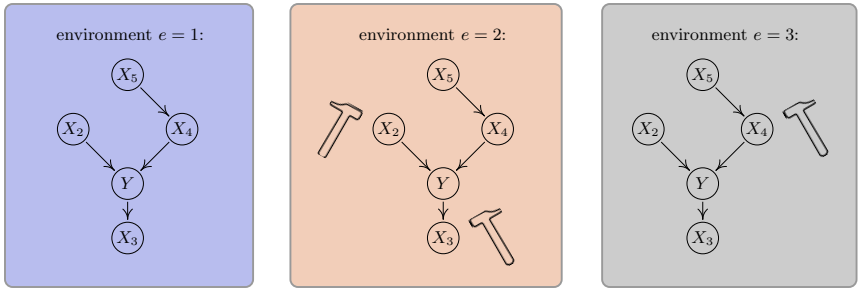


Figure 19.14: Illustration of invariant causal prediction. The hammer symbol represents variables whose distribution is perturbed in the given environment. An invariant predictor must use features $\{X_2, X_4\}$. Considering indirect causes instead of direct ones (e.g. $\{X_2, X_5\}$) or an incomplete set of direct causes (e.g. $\{X_4\}$) may not be sufficient to guarantee invariant prediction. From Figure 1 of [PBM16b]. Used with kind permission of Jonas Peters.

Domain generalization (DG) is similar to multi-task learning, but differs in what we want to predict. In particular, in DG, we only care about prediction accuracy on the target distribution, not the J training distribution. Furthermore, we assume we don't have any labeled data from the target distribution. We therefore have to make some assumptions about how $p^t(\boldsymbol{x}, \boldsymbol{y})$ relates to $p^j(\boldsymbol{x}, \boldsymbol{y})$ for $j = 1 : J$.

One way to formalize this is to create a hierarchical Bayesian model, as proposed in [Bax00], and illustrated in Figure 19.13. This encodes the assumption that $p^t(\boldsymbol{x}, \boldsymbol{y}) = p(\boldsymbol{x}|\boldsymbol{\phi}^t)p(\boldsymbol{y}|\boldsymbol{x}, \boldsymbol{w}^t)$ where $\boldsymbol{w}^t$ is derived from a common "population level" model $\boldsymbol{w}^0$, shared across all distributions, and similarly for $\boldsymbol{\phi}^t$. (Note, however, that in a discriminative model, we don't need to model $p(\boldsymbol{x}|\boldsymbol{\phi}^t)$.) See Section 15.5 for discussion of hierarchical Bayesian GLMs, and Section 17.6 for discussion of hierarchical Bayesian MLPs.

Many other techniques have been proposed for DG. Note, however, that [GLP21] found that none of these methods worked consistently better than the baseline approach of performing empirical risk minimization across all the provided datasets. For more information, see e.g., [GLP21; She+21; Wan+21; Chr+21].

19.6.3 Invariant risk minimization

One approach to domain generalization that has received a lot of attention is called **invariant risk minimization** or **IRM** [Arj+19]. The goal is to learn a predictor that works well across all environments, yet is less prone to depending on the kinds of "spurious features" we discussed in Section 19.2.1.

IRM is an extension of an earlier method called **invariant causal prediction** (ICP) [PBM16b]. This uses hypothesis testing methods to find the set of predictors (features) that directly cause the outcome in each environment, rather than features that are indirect causes, or are just correlated with the outcome. See Figure 19.14 for an illustration.

In [Arj+19], they proposed an extension of ICP to handle the case of high dimensional inputs, where the individual variables do not have any causal meaning (e.g., they correspond to pixels).

Their approach requires finding a predictor that works well on average, across all environments, while also being optimal for each individual environment. That is, we want to find

$$f^* = \operatorname*{argmin}_{f \in \mathcal{F}} \sum_{j=1}^{J} \frac{1}{N_j} \sum_{n=1}^{N_j} \ell(\boldsymbol{y}_n^j, f(\boldsymbol{x}_n^j)) \tag{19.26}$$

$$\text{such that } f \in \arg\min_{g \in \mathcal{F}} \frac{1}{N_j} \sum_{n=1}^{N_j} \ell(\boldsymbol{y}_n^j, g(\boldsymbol{x}_n^j)) \text{ for all } j \in \mathcal{E} \tag{19.27}$$

where $\mathcal{E}$ is the set of environments, and $\mathcal{F}$ is the set of prediction functions. The intuition behind this is as follows: there may be many functions that achieve low empirical loss on any given environment, since the problem may be underspecified, but if we pick the one that also works well on all environments, it is more likely to rely on causal features rather than spurious features.

Unfortunately, more recent work has shown that the IRM principle often does not work well for covariate shift, both in theory [RRR21] and practice [GLP21], although it can work well in some anti-causal (generative) models [Ahu+21].

19.6.4 Meta-learning

The goal of **meta-learning** is to "learn the learning algorithm" [TP97]. A common way to do this is to provide the meta-learner with a set of datasets from different distributions. This is very similar to domain generalization (Section 19.6.2), except that we partition each training distribution into training and test, so we can "practice" learning to generalize from a training set to a test set. A general review of meta-learning can be found in [Hos+20a]. Here we present a unifying summary based on the hierarchical Bayesian framework proposed in [Gor+19].

19.6.4.1 Meta-learning as probabilistic inference for prediction

We assume there are J tasks (distributions), each of which has a training set $\mathcal{D}_{\text{train}}^j = \{(\boldsymbol{x}_n^j, \boldsymbol{y}_n^j) : n = 1 : N^j\}$ and a test set $\mathcal{D}_{\text{test}}^j = \{(\tilde{\boldsymbol{x}}_m^j, \tilde{\boldsymbol{y}}_m^j) : m = 1 : M^j\}$. In addition, $\boldsymbol{w}^j$ are the task specific parameters, and $\boldsymbol{w}^0$ are the shared parameters, as shown in Figure 19.15. This is very similar to the domain generalization model in Figure 19.13, except for two differences: first there is the trivial difference due to the use of plate notation; second, in meta learning, we have both training and test partitions for all distributions, whereas in DG, we only have a test set for the target distribution.

We will learn a point estimate for the global parameters $\boldsymbol{w}^0$, since it is shared across all datasets, and thus has little uncertainty. However, we will compute an approximate posterior for $\boldsymbol{w}^j$, since each task often has little data. We denote this posterior by $p(\boldsymbol{w}^j|\mathcal{D}_{\text{train}}^j, \boldsymbol{w}^0)$. From this, we can compute the posterior predictive distribution for each task:

$$p(\tilde{\boldsymbol{y}}^j|\tilde{\boldsymbol{x}}^j, \mathcal{D}_{\text{train}}^j, \boldsymbol{w}^0) = \int p(\tilde{\boldsymbol{y}}^j|\tilde{\boldsymbol{x}}^j, \boldsymbol{w}^j) p(\boldsymbol{w}^j|\mathcal{D}_{\text{train}}^j, \boldsymbol{w}^0) d\boldsymbol{w}^j \tag{19.28}$$

Since computing the posterior is in general intractable, we will learn an amortized approximation (see Section 10.3.6) to the predictive distribution, denoted by $q_{\boldsymbol{\phi}}(\tilde{\boldsymbol{y}}^j|\tilde{\boldsymbol{x}}^j, \mathcal{D}_{\text{train}}^j, \boldsymbol{w}^0)$. We choose the parameters of the prior $\boldsymbol{w}^0$ and the inference network $\boldsymbol{\phi}$ to make this *predictive posterior* as accurate

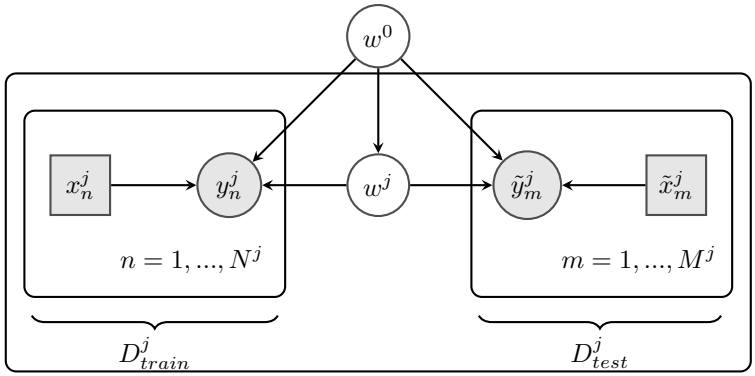


Figure 19.15: Hierarchical Bayesian model for meta-learning. There are J tasks, each of which has a training set $\mathcal{D}^j = \{(\boldsymbol{x}_n^j, \boldsymbol{y}_n^j) : n = 1 : N^j\}$ and a test set $\mathcal{D}_{\text{test}}^j = \{(\tilde{\boldsymbol{x}}_m^j, \tilde{\boldsymbol{y}}_m^j) : m = 1 : M^j\}$. $\boldsymbol{w}^j$ are the task specific parameters, and $\boldsymbol{\theta}$ are the shared parameters. Adapted from Figure 1 of [Gor+19].

as possible for any given input dataset:

$$\boldsymbol{\phi}^* = \operatorname*{argmin}_{\boldsymbol{\phi}} \mathbb{E}_{p(\mathcal{D}_{\text{train}}, \tilde{\boldsymbol{x}})} \left[D_{\mathbb{KL}} \left(p(\tilde{\boldsymbol{y}}|\tilde{\boldsymbol{x}}, \mathcal{D}_{\text{train}}, \boldsymbol{w}^0) \parallel q_{\boldsymbol{\phi}}(\tilde{\boldsymbol{y}}|\tilde{\boldsymbol{x}}, \mathcal{D}_{\text{train}}, \boldsymbol{w}^0) \right) \right] \tag{19.29}$$

$$= \operatorname*{argmin}_{\boldsymbol{\phi}} \mathbb{E}_{p(\mathcal{D}_{\text{train}}, \tilde{\boldsymbol{x}})} \left[\mathbb{E}_{p(\tilde{\boldsymbol{y}}|\tilde{\boldsymbol{x}}, \mathcal{D}_{\text{train}}, \boldsymbol{w}^0)} \left[\log q_{\boldsymbol{\phi}}(\tilde{\boldsymbol{y}}|\tilde{\boldsymbol{x}}, \mathcal{D}_{\text{train}}, \boldsymbol{w}^0) \right] \right] \tag{19.30}$$

$$= \operatorname*{argmin}_{\boldsymbol{\phi}} \mathbb{E}_{p(\mathcal{D}_{\text{train}}, \tilde{\boldsymbol{x}}, \tilde{\boldsymbol{y}})} \left[\log \int p(\tilde{\boldsymbol{y}}|\tilde{\boldsymbol{x}}, \boldsymbol{w}) q_{\boldsymbol{\phi}}(\boldsymbol{w}|\mathcal{D}_{\text{train}}, \boldsymbol{w}^0) d\boldsymbol{w} \right] \tag{19.31}$$

where we made the approximation $p(\tilde{\boldsymbol{y}}|\tilde{\boldsymbol{x}}, \mathcal{D}_{\text{train}}, \boldsymbol{w}^0) \approx p(\tilde{\boldsymbol{y}}|\tilde{\boldsymbol{x}}, \mathcal{D}_{\text{train}})$. We can then make a Monte Carlo approximation to the outer expectation by sampling J tasks (distributions) from $p(\mathcal{D})$, each of which gets partitioned into a train and test set, $\{(\mathcal{D}_{\text{train}}^j, \mathcal{D}_{\text{test}}^j) \sim p(\mathcal{D}) : j = 1 : J\}$, where $\mathcal{D}_{\text{test}}^j = \{(\tilde{\boldsymbol{x}}_m, \tilde{\boldsymbol{y}}_m\}$. We can make an MC approximation to the inner expectation (the integral) by drawing S samples from the task-specific parameter posterior $\boldsymbol{w}_s^j \sim q_{\boldsymbol{\phi}}(\boldsymbol{w}^j|\mathcal{D}^j, \boldsymbol{w}^0)$. The resulting objective has the following form (where we assume each test set has M samples for notational simplicity):

$$\mathcal{L}_{\text{meta}}(\boldsymbol{w}^0, \boldsymbol{\phi}) = \frac{1}{MJ} \sum_{m=1}^{M} \sum_{j=1}^{J} \log \left(\frac{1}{S} \sum_{s=1}^{S} p(\tilde{\boldsymbol{y}}_m^j|\tilde{\boldsymbol{x}}_m^j, \boldsymbol{w}_s^j) \right) \tag{19.32}$$

Note that this is different from standard (amortized) variational inference, that focuses on approximating the expected accuracy of the *parameter posterior* given all of the data for a task, $\mathcal{D}_{\text{all}}^j = \mathcal{D}_{\text{train}}^j \cup \mathcal{D}_{\text{test}}^j$, rather than focusing on predictive accuracy of a test set given a training set. Indeed, the standard objective has the form

$$\mathcal{L}_{\text{VI}}(\boldsymbol{w}^0, \boldsymbol{\phi}) = \frac{1}{J} \sum_{j=1}^{J} \left(\sum_{(\boldsymbol{x}, \boldsymbol{y}) \in \mathcal{D}_{\text{all}}^j} \left[\frac{1}{S} \sum_{s=1}^{S} \log p(\tilde{\boldsymbol{y}}^j|\tilde{\boldsymbol{x}}^j, \boldsymbol{w}_s^j) \right] - D_{\mathbb{KL}} \left(q_{\boldsymbol{\phi}}(\boldsymbol{w}^j|\mathcal{D}_{\text{all}}^j, \boldsymbol{w}^0) \parallel p(\boldsymbol{w}^j|\boldsymbol{w}^0) \right) \right)$$

(19.33)

where $\boldsymbol{w}_s^j \sim q_{\boldsymbol{\phi}}(\boldsymbol{w}^j|\mathcal{D}_{\text{all}}^j)$. We see that the standard formulation takes the average of a log, but the meta-learning formulation takes the log of an average. The latter can give provably better predictive accuracy, as pointed out in [MAD20]. Another difference is that the meta-learning formulation optimizes the forward KL, not reverse KL. Finally, in the meta-learning formulation, we do not have the KL penalty term on the parameter posterior.

Below we show how this framework includes several common approaches to meta-learning.

19.6.4.2 Neural processes

In the special case that the task-specific inference network computes a point estimate, $q(\boldsymbol{w}^j|\mathcal{D}^j, \boldsymbol{w}^0) = \delta(\boldsymbol{w}^j - \mathcal{A}_{\boldsymbol{\phi}}(\mathcal{D}^j, \boldsymbol{w}^0))$, the posterior predictive distribution becomes

$$q(\tilde{\boldsymbol{y}}^j|\tilde{\boldsymbol{x}}^j, \mathcal{D}^j, \boldsymbol{w}^0) = \int p(\tilde{\boldsymbol{y}}^j|\tilde{\boldsymbol{x}}^j, \boldsymbol{w}^j) q(\boldsymbol{w}^j|\mathcal{D}^j, \boldsymbol{w}^0) d\boldsymbol{w}^j = p(\tilde{\boldsymbol{y}}^j|\tilde{\boldsymbol{x}}^j, \mathcal{A}_{\boldsymbol{\phi}}(\mathcal{D}^j, \boldsymbol{w}^0), \boldsymbol{w}^0) \tag{19.34}$$

where $\mathcal{A}_{\boldsymbol{\phi}}(\mathcal{D}^j, \boldsymbol{w}^0)$ is a function that takes in a set, and returns some parameters. We can evaluate this predictive distribution empirically, and directly optimize it (wrt $\boldsymbol{\phi}$ and $\boldsymbol{w}^0$) using standard supervised maximum likelihood methods. This approach is called a **neural process** [Gar+18e; Gar+18d]. (See [DGF20] for a good tutorial.)

However, a more common approach to meta-learning is to specify the form of the estimator $\mathcal{A}_{\boldsymbol{\phi}}(\mathcal{D}^j, \boldsymbol{w}^0)$ using various heuristics, some of which we discuss below.

19.6.4.3 Gradient-based meta-learning (MAML)

In **gradient-based meta-learning**, we define the task specific inference procedure as follows:

$$\hat{\boldsymbol{w}}^j = \mathcal{A}(\mathcal{D}^j, \boldsymbol{w}^0) = \boldsymbol{w}^0 + \eta \nabla_{\boldsymbol{w}} \log \sum_{n=1}^{N^j} p(\boldsymbol{y}_n^j|\boldsymbol{x}_n^j, \boldsymbol{w})|_{\boldsymbol{w}^0} \tag{19.35}$$

That is, we set the task specific parameters to be shared parameters $\boldsymbol{w}^0$, modified by one step along the gradient of the log conditional likelihood. This approach is called **model-agnostic meta-learning** or **MAML** [FAL17]. It is also possible to take multiple gradient steps, by feeding the gradient into an RNN [RL17].

19.6.4.4 Metric-based few-shot learning (prototypical networks)

Now suppose $\boldsymbol{w}^0$ correspond to the parameters of a shared neural feature extractor, $h_{\boldsymbol{w}^0}(\boldsymbol{x})$, and the task specific parameters are the weights and biases of the last linear layer of a classifier, $\boldsymbol{w}^j = \{\boldsymbol{w}_c^j, b_c^j\}_{c=1}^C$. Let us compute the average of the feature vectors for each class in each task's training set:

$$\boldsymbol{\mu}_c^j = \frac{1}{|\mathcal{D}_c^j|} \sum_{\boldsymbol{x}_n^c \in \mathcal{D}_c^j} h_{\boldsymbol{w}^0}(\boldsymbol{x}_n^c) \tag{19.36}$$

Now define the task specific inference procedure as follows. We first compute the vector containing the centroid and norm for each class:

$$\hat{\boldsymbol{w}}^j = \mathcal{A}(\mathcal{D}^j, \boldsymbol{w}^0) = [\boldsymbol{\mu}_c^j, \ -\frac{1}{2}||\boldsymbol{\mu}_c^j||^2]_{c=1}^C \tag{19.37}$$

The predictive distribution becomes

$$q(\tilde{y}^j = c|\tilde{\boldsymbol{x}}^j, \mathcal{D}^j, \boldsymbol{w}^0) \propto \exp\left(-d(h_{\boldsymbol{w}^0}(\tilde{\boldsymbol{x}}), \boldsymbol{\mu}_c^j)\right) = \exp\left(h_{\boldsymbol{w}^0}(\tilde{\boldsymbol{x}})^\mathsf{T}\boldsymbol{\mu}_c^j - \frac{1}{2}||\boldsymbol{\mu}_c^j||^2\right) \tag{19.38}$$

where $d(\boldsymbol{u}, \boldsymbol{v})$ is the Euclidean distance. This is equivalent to the technique known as **prototypical networks** [SSZ17].

19.7 Continual learning

In this section, we discuss **continual learning** (see e.g., [Had+20; Del+21; Qu+21; LCR21; Mai+22; Lin+22]), also called **life-long learning** (see e.g., [Thr98; CL18]), in which the system learns from a sequence of different distributions, $p_1, p_2, \ldots$. In particular, at each time step t, the model receives a batch of labeled data,

$$\mathcal{D}_t = \{(\boldsymbol{x}_n, \boldsymbol{y}_n) : n = 1 : N_t, \boldsymbol{x}_n \sim p_t(\boldsymbol{x}), \boldsymbol{y}_n = f_t(\boldsymbol{x}_n)\} \tag{19.39}$$

where $p_t(\boldsymbol{x})$ is the unknown input distribution, and $f_t : \mathcal{X}_t \to \mathcal{Y}_t$ is the unknown prediction function. (We focus on noise-free outputs, for notational simplicity.) The learner is then expected to update its belief state about the true function f_t, and to use its beliefs to make predictions on an independent test set,

$$\mathcal{D}_t^{\text{test}} = \{(\boldsymbol{x}_n, \boldsymbol{y}_n) : n = 1 : N_t^{\text{test}}, \boldsymbol{x}_n \sim p_t^{\text{test}}(\boldsymbol{x}), \boldsymbol{y}_n = f_t^{\text{test}}(\boldsymbol{x}_n)\} \tag{19.40}$$

Depending on how we assume $p_t(\boldsymbol{x})$ and f_t evolve over time, and how the test set is defined, we can create a variety of different CL scenarios, as we discuss below.

19.7.1 Domain drift

The problem of **domain drift** refers to the setting in which $p_t(\boldsymbol{x})$ changes over time (i.e., covariate shift), but the functional mapping $f_t : \mathcal{X} \to \mathcal{Y}$ is constant. For example, the vision system of a self driving car may have to classify cars vs pedestrians under shifting lighting conditions (see e.g., [Sun+22]).

To evaluate such a model, we assume $f_t^{\text{test}} = f_t$ and define $p_t^{\text{test}}(\boldsymbol{x})$ to be the current input distribution p_t (e.g., if it is currently night time, we want the detector to work well on dark images). Alternatively we can define $p_t^{\text{test}}(\boldsymbol{x})$ to be the union of all the input distributions seen so far, $p_t^{\text{test}} = \cup_{s=1}^T p_s$ (e.g., we want the detector to work well on dark and light images). This latter assumption is illustrated in Figure 19.16.

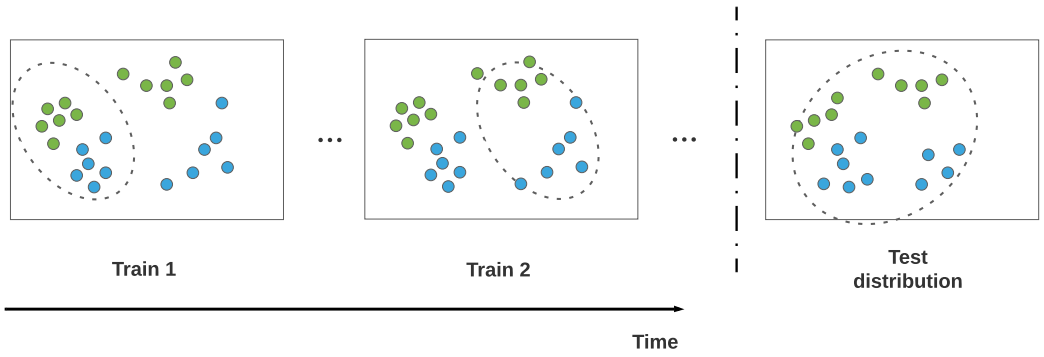


Figure 19.16: An illustration of domain drift.

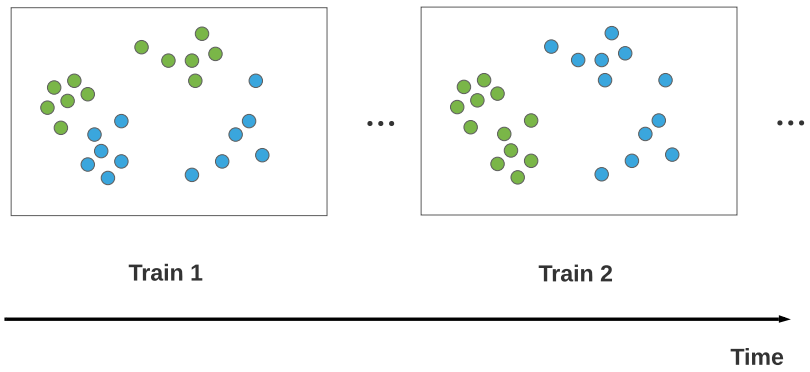


Figure 19.17: An illustration of concept drift.

19.7.2 Concept drift

The problem of **concept drift** refers to the setting where the functional mapping $f_t : \mathcal{X} \to \mathcal{Y}$ changes over time, but the input distribution $p_t(\boldsymbol{x})$ is constant [WK96]. For example, we can imagine a setting in which people engage in certain behaviors, and at step t some of these are classified as illegal, and at step $t' > t$, the definition of what is legal changes, and hence the decision boundary changes. This is illustrated in Figure 19.17.

As another example, we might initially be faced with a sort-by-color task, where red objects go on the left and blue objects on the right, and then a sort-by-shape task, where square objects go on the left and circular objects go on the right.[5] We can think of this as a problem where $p(y|\boldsymbol{x}, \text{task})$ is stationary, but the task is unobserved, so $p(y|\boldsymbol{x})$ changes.

In the concept drift scenario, we see that the prediction for the same underlying input point $\boldsymbol{x} \in \mathcal{X}$ will change depending on when the prediction is performed. This means that the test distribution

5. This example is from Mike Mozer.

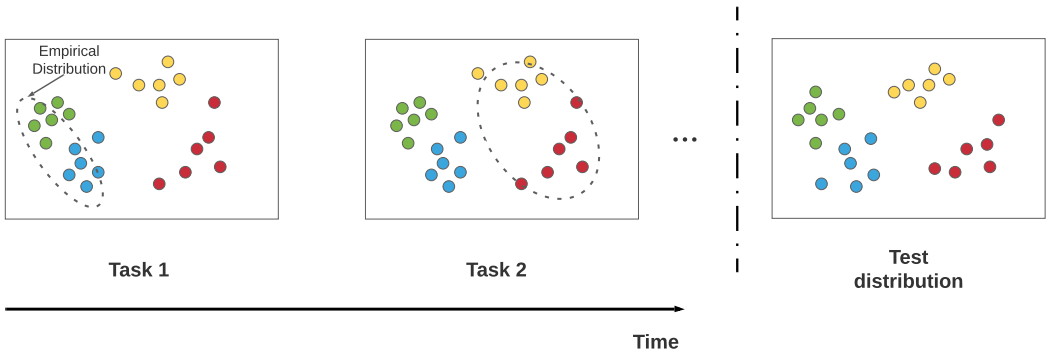


Figure 19.18: An illustration of class incremental learning. Adapted from Figure 1 of [LCR21].

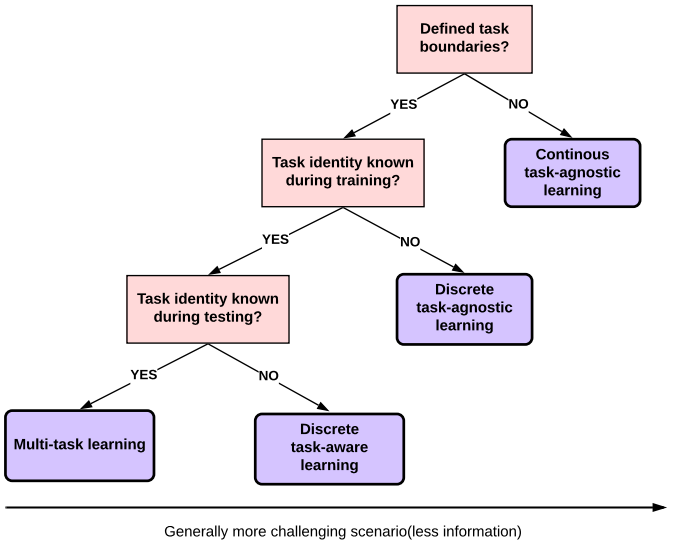


Figure 19.19: Different kinds of incremental learning. Adapted from Figure 1 of [Zen+18].

also needs to change over time for meaningful identification. Alternatively, we can "tag" each input with the corresponding time stamp or task id.

19.7.3 Task incremental learning

A very widely studied form of continual learning focuses on the setting in which new class labels are "revealed" over time. That is, there is assumed to be a true static prediction function $f : \mathcal{X} \to \mathcal{Y}$, but at step t, the learner only sees samples from $(\mathcal{X}, \mathcal{Y}_t)$, where $\mathcal{Y}_t \subset \mathcal{Y}$. For example, $\mathcal{X}$ may be

the space of images, and $\mathcal{Y}_1$ might be $\{\text{cats}, \text{dogs}\}$, and $\mathcal{Y}_2$ might be $\{\text{cars}, \text{bikes}, \text{trucks}\}$. Learning to classify with an increasing number of categories is called **class incremental learning** (see e.g., [Mas+20]). This is also called **task incremental learning**, since each distribution is considered as a different **task**. See Figure 19.18 for an illustration.

The problem of class incremental learning has been studied under a variety of different assumptions, as discussed in [Hsu+18; VT18; FG18; Del+21]. The most common scenarios are shown in Figure 19.19. If we assume there are no well defined boundaries between tasks, we have **continuous task-agnostic learning** (see e.g., [SKM21; Zen+21]). If there are well defined boundaries (i.e., discontinuous changes of the training distribution), then we can distinguish two subcases. If the boundaries are not known during training (similar to detecting distribution shift), we have **discrete task-agnostic learning**. Finally, if the boundaries are given to the training algorithm, we have a **task-aware learning** problem.

A common experimental setup in the task-aware setting is to define each task to be a different version of the MNIST dataset, e.g., with all 10 classes present but with the pixels randomly permuted (this is called **permuted MNIST**) or with a subset of 2 classes present at each step (this is called **split MNIST**).[6] In the task-aware setting, the task label may or may not be known at test time. If it is, the problem is essentially equivalent to multi-task learning (see Section 19.6.1). If it is not, the model must predict the task and corresponding class label within that task (which is a standard supervised problem with a hierarchical label space); this is commonly done by using a multi-headed DNN, with CT outputs, where C is the number of classes, and T is the number of tasks.

In the multi-headed approach, the number of "heads" is usually specified as input to the algorithm, because the softmax imposes a sum-to-one constraint that prevents incremental estimation of the output weights in the open-class setting. An alternative approach is to wait until a new class label is encountered for the first time, and then train the model with an enlarged output head. This requires storing past data from each class, as well as data for the new class (see e.g., [PTD20]). Alternatively, we can use generative classifiers where we do not need to worry about "output heads". If we use a "deep" nearest neighbor classifier, with a shared feature extractor (embedding function), the main challenge is to efficiently update the stored prototypes for past classes as the feature extractor parameters change (see e.g., [DLT21]). If we fit a separate generative model per class (e.g., a VAE, as in [VLT21]), then online learning becomes easier, but the method may be less sample efficient.

At the time of writing, most of the CL literature focuses on the task-aware setting. However, from a practical point of view, the assumption that task boundaries are provided at training or test time is very unrealistic. For example, consider the problem of training a robot to perform various activities: The data just streams in, and the robot must learn what to do, without anyone telling it that it is now being given an example from a new task or distribution (see e.g., [Fon+21; Woł+21]). Thus future research should focus on the task-agnostic setting, with either discrete or continuous changes.

19.7.4 Catastrophic forgetting

In the class incremental learning literature, it is common to train on a sequence of tasks, but to test (at each step) on all tasks. In this scenario, there are two main possible failure modes. The first possible problem is called "**catastrophic forgetting**" (see e.g., [Rob95b; Fre99; Kir+17]). This

6. In the split MNIST setup, for task 1, digits (0,1) get labeled as (0,1), but in task 2, digits (2,3) get labeled as (0,1). So the "meaning" of the output label depends on what task we are solving. Thus the output space is really hierarchical, namely the cross product of task id and class label.

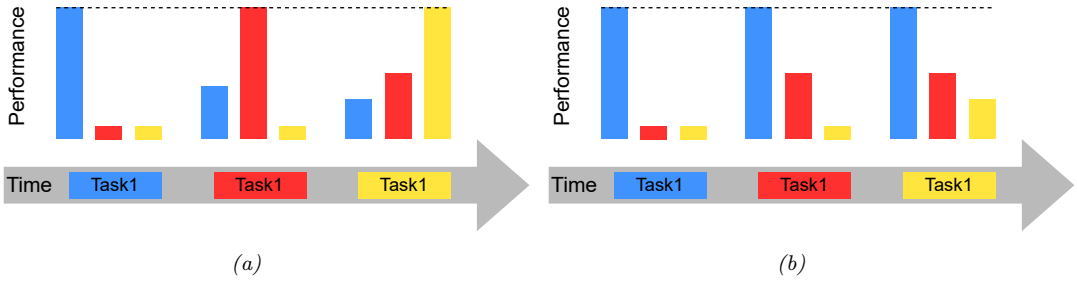


Figure 19.20: Some failure modes in class incremental learning. We train on task 1 (blue) and evaluate on tasks 1–3 (blue, orange, yellow); we then train on task 2 and evaluate on tasks 1–3; etc. (a) Catastrophic forgetting refers to the phenomenon in which performance on a previous task drops when trained on a new task. (b) Too little plasticity (e.g., due to too much regularization) refers to the phenomenon in which only the first task is learned. Adapted from Figure 2 of [Had+20].

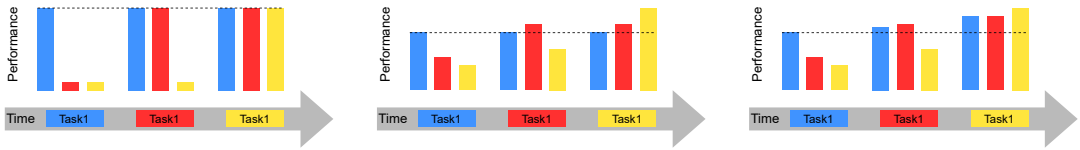


Figure 19.21: What success looks like for class incremental learning. We train on task 1 (blue) and evaluate on tasks 1–3 (blue, orange, yellow); we then train on task 2 and evaluate on tasks 1–3; etc. (a) No forgetting refers to the phenomenon in which performance on previous tasks does not degrade over time. (b) Forward transfer refers to the phenomenon in which training on past tasks improves performance on future tasks beyond what would have been obtained by training from scratch. (c) Backwards transfer refers to the phenomenon in which training on future tasks improves performance on past tasks beyond what would have been obtained by training from scratch. Adapted from Figure 2 of [Had+20].

refers refers to the phenomenon in which performance on a previous task drops when trained on a new task (see Figure 19.20(a)). Another possible problem is that only the first task is learned, and the model does not adapt to new tasks (see Figure 19.20(b)).

If we avoid these problems, we should expect to see the performance profile in Figure 19.21(a), where performance of incremental training is equal to training on each task separately. However, we might hope to do better by virtue of the fact that we are training on multiple tasks, which are often assumed to be related. In particular, we might hope to see **forward transfer**, in which training on past tasks improves performance on future tasks beyond what would have been obtained by training from scratch (see Figure 19.21(b)). Additionally, we might hope to see **backwards transfer**, in which training on future tasks improves performance on past tasks (see Figure 19.21(c)).

We can quantify the degree of transfer as follows, following [LPR17]. If R_{ij} is the performance on task j after it was trained on task i, R_j^{ind} is the performance on task j when trained just on j, and

there are T tasks, then the amount of forward transfer is

$$\text{FWT} = \frac{1}{T}\sum_{j=1}^{T} R_{j,j} - R_j^{\text{ind}} \tag{19.41}$$

and the amount of backwards transfer is

$$\text{BWT} = \frac{1}{T}\sum_{j=1}^{T} R_{T,j} - R_{j,j} \tag{19.42}$$

There are many methods that have been devised to overcome the problem of catastrophic forgetting, but we can group them into three main types. The first is **regularization methods**, which add a loss to preserve information that is relevant to old tasks. (For example, online Bayesian inference is of this type, since the posterior for the parameters is derived from the new data and the past prior; see e.g., the **elastic weight consolidation** method discussed in Section 17.5.1, or the **variational continual learning** method discussed in Supplementary Section 10.2). The second is **memory methods**, which rely on some kind of **experience replay** or **rehearsal** of past data (see e.g.,[Hen+21]), or some kind of generative model of past data. The third is **architectural methods**, that add capacity to the network whenever a task boundary is encountered, such as a new class label (see e.g., [Rus+16]).

Of course, these techniques can be combined. For example, we can create a semi-parametric model, in which we store some past data (exemplars) while also learning parameters online in a Bayesian (regularized) way (see e.g., [Kur+20]). The "right" method depends, as usual, on what inductive bias you want to use, and want your computational budget is in terms of time and memory.

19.7.5 Online learning

The problem of **online learning** is similar to continual learning, except the loss metric is different, and we usually assume that learning and evaluation occur at each step. More precisely, we assume the data generating distribution, $p_t^*(\boldsymbol{x}, \boldsymbol{y}) = p(\boldsymbol{x}|\boldsymbol{\phi}_t)p(\boldsymbol{y}|\boldsymbol{x}, \boldsymbol{w}_t)$, evolves over time, as shown in Figure 19.22. At each step t nature generates a data sample, $(\boldsymbol{x}_t, \boldsymbol{y}_t) \sim p_t^*$. The agent sees $\boldsymbol{x}_t$ and is asked to predict $\boldsymbol{y}_t$ by computing the posterior predictive distribution

$$\hat{p}_{t|t-1} = p(\boldsymbol{y}|\boldsymbol{x}_t, \mathcal{D}_{1:t-1}) \tag{19.43}$$

where $\mathcal{D}_{1:t-1} = \{(\boldsymbol{x}_s, \boldsymbol{y}_s) : s = 1 : t-1\}$ is all past data. It then incurs a loss of

$$\mathcal{L}_t = \ell(\hat{p}_{t|t-1}, \boldsymbol{y}_t) \tag{19.44}$$

For classification problems, we often use 0-1 loss, $\mathcal{L}_t = \mathbb{I}(\hat{y}_t \neq y_t)$. In this case the optimal action is to predict the most probable label, $\hat{y}_t = \text{argmax}_c\, p(y = c|\boldsymbol{x}_t, \mathcal{D}_{1:t-1})$..

In contrast to the continual learning scenarios studied above, the loss incurred at each step is what matters, rather than loss on a fixed test set. That is, we want to minimize

$$\mathcal{L} = \sum_{t=1}^{T} \mathcal{L}_t \tag{19.45}$$

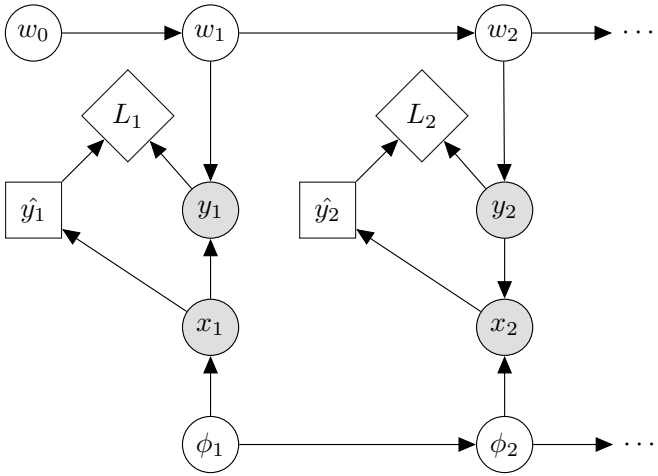


Figure 19.22: Online learning illustrated as an influence diagram (Section 34.2). Here $\hat{y}_t = \text{argmax}_y\, p(y|\boldsymbol{x}_t, \mathcal{D}_{1:t-1})$ *is the action (MAP predicted output) at time* t*, and* $L_t = \ell(y_t, \hat{y}_t)$ *is the corresponding loss (utility) function. The data at time* t *is assumed to come from a model of the form* $p(\boldsymbol{x}_t, y_t) = p(\boldsymbol{x}_t|\boldsymbol{\phi}_t)p(y_t|\boldsymbol{x}_t, \boldsymbol{w}_t)$*. The parameters of the world model can change arbitrarily over time.*

Since it is hard to interpret this number, it is common to compare it to the optimal value one could have obtained in hindsight. This yields a quantity called the **regret**:

$$\text{regret} = \sum_{t=1}^{T} \left[\ell(\hat{p}_{t|t-1}, \boldsymbol{y}_t) - \ell(\hat{p}_{t|T}, \boldsymbol{y}_t)\right] \tag{19.46}$$

where $\hat{p}_{t|t-1} = p(\boldsymbol{y}|\boldsymbol{x}_t, \mathcal{D}_{1:t-1})$ is the online prediction, and $\hat{p}_{t|T} = p(\boldsymbol{y}|\boldsymbol{x}_t, \mathcal{D}_{1:T})$ is the optimal estimate at the end of training. It is possible to convert bounds on regret, which are backwards looking, into bounds on risk (i.e., expected future loss), which is forwards looking. See [HT15] for details.

Online learning is very useful for decision and control problems, such as multi-armed bandits (Section 34.4) and reinforcement learning (see Chapter 35), where the agent "lives forever", and where there is no fixed training phase followed by a test phase. (See e.g., Section 17.5 where we discuss online Bayesian inference for neural networks.)

The previous continual learning scenarios can be derived as special cases of online learning, by defining a suitable sequence of distributions, and by requiring the agent to either train or test at each step on a suitable minibatch of data. (We leave the details of this mapping as an exercise to the reader.)

19.8 Adversarial examples

This section is coauthored with Justin Gilmer.

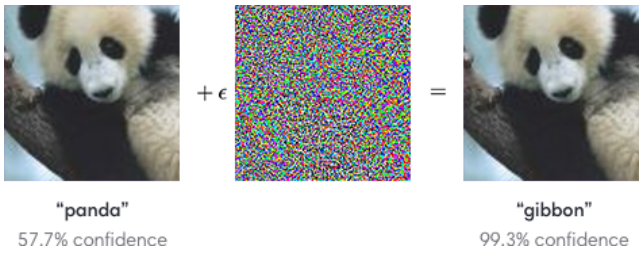


Figure 19.23: Example of an adversarial attack on an image classifier. Left column: original image which is correctly classified. Middle column: small amount of structured noise which is added to the input (magnitude of noise is magnified by 10×). Right column: new image, which is confidently misclassified as a "gibbon", even though it looks just like the original "panda" image. Here $\epsilon = 0.007$ From Figure 1 of [GSS15]. Used with kind permission of Ian Goodfellow.

In Section 19.2, we discussed what happens to a predictive model when the input distribution shifts for some reason. In this section, we consider the case where an adversary deliberately chooses inputs to minimize the performance of a predictive model. That is, suppose an input $\boldsymbol{x}$ is classified as belonging to class c. We then choose a new input $\boldsymbol{x}_{\text{adv}}$ which minimizes the probability of this label, subject to the constraint that $\boldsymbol{x}_{\text{adv}}$ is "perceptually similar" to the original input $\boldsymbol{x}$. This gives rise to the following objective:

$$\boldsymbol{x}_{\text{adv}} = \underset{\boldsymbol{x}' \in \Delta(\boldsymbol{x})}{\operatorname{argmin}} \log p(y = c|\boldsymbol{x}') \tag{19.47}$$

where $\Delta(\boldsymbol{x})$ is the set of images that are "similar" to $\boldsymbol{x}$ (we discuss different notions of similarity below).

Equation (19.47) is an example of an **adversarial attack**. We illustrate this in Figure 19.23. The input image $\boldsymbol{x}$ is on the left, and is predicted to be a panda with probability 57%. By adding a tiny amount of carefully chosen noise (shown in the middle) to the input, we generate the **adversarial image** $\boldsymbol{x}_{\text{adv}}$ on the right: this "looks like" the input, but is now classified as a gibbon with probability 99%.

The ability to create adversarial images was first noted in [Sze+14]. It is suprisingly easy to create such examples, which seems paradoxical, given the fact that modern classifiers seem to work so well on normal inputs, and the perturbed images "look" the same to humans. We explain this paradox in Section 19.8.5.

The existence of adversarial images also raises security concerns. For example, [Sha+16] showed they could force a face recognition system to misclassify person A as person B, merely by asking person A to wear a pair of sunglasses with a special pattern on them, and [Eyk+18] show that is possible to attach small "**adverarial stickers**" to traffic signs to classify stop signs as speed limit signs.

Below we briefly discuss how to create adversarial attacks, why they occur, and how we can try to defend against them. We focus on the case of deep neural nets for images, although it is important to note that many other kinds of models (including logistic regression and generative models) can also suffer from adversarial attacks. Furthermore, this is not restricted to the image domain, but occurs with many kinds of high dimensional inputs. For example, [Li+19] contains an audio attack

and [Dal+04; Jia+19] contains a text attack. More details on adversarial examples can be found in e.g., [Wiy+19; Yua+19].

19.8.1 Whitebox (gradient-based) attacks

To create an adversarial example, we must find a "small" perturbation $\boldsymbol{\delta}$ to add to the input $\boldsymbol{x}$ to create $\boldsymbol{x}_{\text{adv}} = \boldsymbol{x} + \boldsymbol{\delta}$ so that $f(\boldsymbol{x}_{\text{adv}}) = y'$, where $f()$ is the classifier, and y' is the label we want to force the system to output. This is known as a **targeted attack**. Alternatively, we may just want to find a perturbation that causes the current predicted label to change from its current value to any other value, so that $f(\boldsymbol{x} + \boldsymbol{\delta}) \neq f(\boldsymbol{x})$, which is known as **untargeted attack**.

In general, we define the objective for the adversary as *maximizing* the following loss:

$$\boldsymbol{x}_{\text{adv}} = \underset{\boldsymbol{x}' \in \Delta(\boldsymbol{x})}{\text{argmax}}\, \mathcal{L}(\boldsymbol{x}', y; \boldsymbol{\theta}) \tag{19.48}$$

where y is the true label. For the untargeted case, we can define $\mathcal{L}(\boldsymbol{x}', y; \boldsymbol{\theta}) = -\log p(y|\boldsymbol{x}')$, so we minimize the probability of the true label; and for the targeted case, we can define $\mathcal{L}(\boldsymbol{x}', y; \boldsymbol{\theta}) = \log p(y'|\boldsymbol{x}')$, where we maximize the probability of the desired label $y' \neq y$.

To define what we mean by "small" perturbation, we impose the constraint that $\boldsymbol{x}_{\text{adv}} \in \Delta(\boldsymbol{x})$, which is the set of "perceptually similar" images to the input $\boldsymbol{x}$. Most of the literature has focused on a simplistic setting in which the adversary is restricted to making bounded l_p perturbations of a clean input $\boldsymbol{x}$, that is

$$\Delta(\boldsymbol{x}) = \{\boldsymbol{x}' : ||\boldsymbol{x}' - \boldsymbol{x}||_p < \epsilon\} \tag{19.49}$$

Typically people assume $p = 1$ or $p = 0$. We will discuss more realistic threat models in Section 19.8.3.

In this section, we assume that the attacker knows the model parameters $\boldsymbol{\theta}$; this is called a **whitebox attack**, and lets us use gradient based optimization methods. We relax this assumption in Section 19.8.2.)

To solve the optimization problem in Equation (19.48), we can use any kind of constrained optimization method. In [Sze+14] they used bound-constrained BFGS. [GSS15] proposed the more efficient **fast gradient sign** (**FGS**) method, which performs iterative updates of the form

$$\boldsymbol{x}_{t+1} = \boldsymbol{x}_t + \boldsymbol{\delta}_t \tag{19.50}$$

$$\boldsymbol{\delta}_t = \epsilon\, \text{sign}(\nabla_{\boldsymbol{x}} \log p(y'|\boldsymbol{x}, \boldsymbol{\theta})|_{\boldsymbol{x}_t}) \tag{19.51}$$

where $\epsilon > 0$ is a small learning rate. (Note that this gradient is with respect to the input pixels, not the model parameters.) Figure 19.23 gives an example of this process.

More recently, [Mad+18] proposed the more powerful **projected gradient descent** (**PGD**) attack; this can be thought of as an iterated version of FGS. There is no "best" variant of PGD for solving 19.48. Instead, what matters more is the implementation details, e.g. how many steps are used, the step size, and the exact form of the loss. To avoid local minima, we may use random restarts, choosing random points in the constraint space Δ to initialize the optimization. The algorithm should be carefully tuned to the specific problem, and the loss should be monitored to check for optimization issues. For best practices, see [Car+19].

Figure 19.24: Images that look like random noise but which cause the CNN to confidently predict a specific class. From Figure 1 of [NYC15]. Used with kind permission of Jeff Clune.

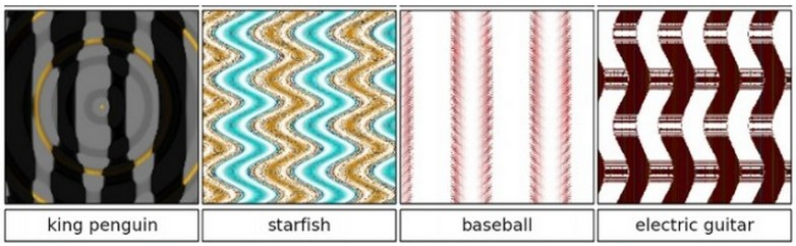


Figure 19.25: Synthetic images that cause the CNN to confidently predict a specific class. From Figure 1 of [NYC15]. Used with kind permission of Jeff Clune.

19.8.2 Blackbox (gradient-free) attacks

In this section, we no longer assume that the adversary knows the parameters $\boldsymbol{\theta}$ of the predictive model f. This is known as a **black box attack**. In such cases, we must use derivative-free optimization (DFO) methods (see Section 6.9).

Evolutionary algorithms (EA) are one class of DFO solvers. These were used in [NYC15] to create blackbox attacks. Figure 19.24 shows some images that were generated by applying an EA to a random noise image. These are known as **fooling images**, as opposed to adversarial images, since they are not visually realistic. Figure 19.25 shows some fooling images that were generated by applying EA to the parameters of a compositional pattern-producing network (CPPN) [Sta07].[7] By suitably perturbing the CPPN parameters, it is possible to generate structured images with high fitness (classifier score), but which do not look like natural images [Aue12].

In [SVK19], they used differential evolution to attack images by modifying a single pixel. This is equivalent to bounding the ℓ_0 norm of the perturbation, so that $||\boldsymbol{x}_{\text{adv}} - \boldsymbol{x}||_0 = 1$.

In [Pap+17], they learned a differentiable surrogate model of the blackbox, by just querying its predictions y for different inputs $\boldsymbol{x}$. They then used gradient-based methods to generate adversarial attacks on their surrogate model, and then showed that these attacks transferred to the real model. In this way, they were able to attack various the image classification APIs of various cloud service

7. A CPPN is a set of elementary functions (such as linear, sine, sigmoid, and Gaussian) which can be composed in order to specify the mapping from each coordinate to the desired color value. CPPN was originally developed as a way to encode abstract properties such as symmetry and repetition, which are often seen during biological development.

Figure 19.26: An adversarially modified image to evade spam detectors. The image is constructed from scratch, and does not involve applying a small perturbation to any given image. This is an illustrative example of how large the space of possible adversarial inputs Δ can be when the attacker has full control over the input. From [Big+11]. Used with kind permission of Battista Biggio.

providers, including Google, Amazon and MetaMind.

19.8.3 Real world adversarial attacks

Typically, the space of possible adversarial inputs Δ can be quite large, and will be difficult to exactly define mathematically as it will depend on semantics of the input based on the attacker's goals [BR18]. (The set of variations Δ that we want the model to be invariant to is called the **threat model**.)

Consider for example of the content constrained threat model discussed in [Gil+18a]. One instance of this threat model involves image spam, where the attacker wishes to upload an image attachment in an email that will not be classified as spam by a detection model. In this case Δ is incredibly large as it consists of all possible images which contain some semantic concept the attacker wishes to upload (in this case an advertisement). To explore Δ, spammers can utilize different fonts, word orientations or add random objects to the background as is the case of the adversarial example in Figure 19.26 (see [Big+11] for more examples). Of course, optimization based methods may still be used here to explore parts of Δ. However, in practice it may be preferable to design an adversarial input by hand as this can be significantly easier to execute with only limited-query black-box access to the underlying classifier.

19.8.4 Defenses based on robust optimization

As discussed in Section 19.8.3, securing a system against adversarial inputs in more general threat models seems extraordinarily difficult, due to the vast space of possible adversarial inputs Δ. However, there is a line of research focused on producing models which are invariant to perturbations within a small constraint set $\Delta(\boldsymbol{x})$, with a focus on l_p-robustness where $\Delta(\boldsymbol{x}) = \{\boldsymbol{x}' : ||\boldsymbol{x} - \boldsymbol{x}'||_p < \epsilon\}$. Although solving this toy threat model has little application to security settings, enforcing smoothness

priors have in some cases improved robustness to random image corruptions [SHS], lead to models which transfer better [Sal+20], and can bias models towards different features in the data [Yin+19a].

Perhaps the most straightforward method for improving l_p-robustness is to directly optimize for it through **robust optimization** [BTEGN09], also known as **adversarial training** [GSS15]. We define the **adversarial risk** to be

$$\min_{\theta} \mathbb{E}_{(\boldsymbol{x},\boldsymbol{y})\sim p(\boldsymbol{x},\boldsymbol{y})} \left[\max_{\boldsymbol{x}' \in \Delta(\boldsymbol{x})} L(\boldsymbol{x}', \boldsymbol{y}; \theta) \right] \tag{19.52}$$

The min max formulation in equation 19.52 poses unique challenges from an optimization perspective—it requires solving both the non-concave inner maximazation and the non-convex outer minimization problems. Even worse, the inner max is NP-hard to solve in general [Kat+17]. However, in practice it may be sufficient to compute the gradient of the outer objective $\nabla_\theta L(\boldsymbol{x}_{\text{adv}}, \boldsymbol{y}, ; \theta)$ at an approximately maximal point in the inner problem $\boldsymbol{x}_{\text{adv}} \approx \text{argmax}_{\boldsymbol{x}} L(\boldsymbol{x}_{\text{adv}}, \boldsymbol{y}; \theta)$ [Mad+18]. Currently, best practice is to approximate the inner problem using a few steps of PGD.

Other methods seek to **certify** that a model is robust within a given region $\Delta(x)$. One method for certification uses randomized smoothing [CRK19]—a technique for converting a model robust to random noise into a model which is provably robust to bounded worst-case perturbations in the l_2-metric. Another class of methods applies specifically for networks with ReLU activations, leveraging the property that the model is locally linear, and that certifying in region defined by linear constraints reduces to solving a series of linear programs, for which standard solvers can be applied [WK18].

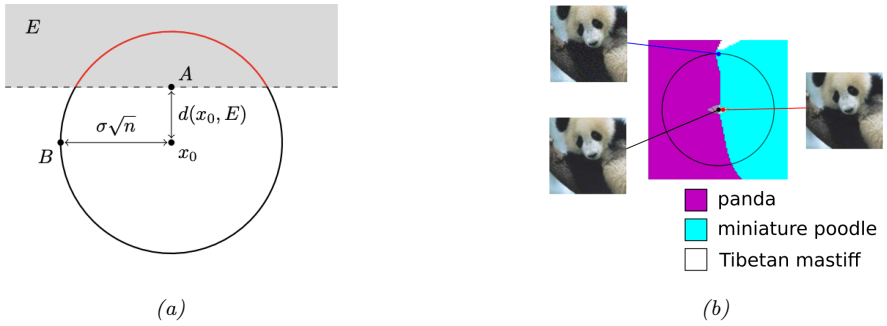
19.8.5 Why models have adversarial examples

The existence of adversarial inputs is paradoxical, since modern classifiers seem to do so well on normal inputs. However, the existence of adversarial examples is a natural consequence of the general lack of robustness to distribution shift discussed in Section 19.2. To see this, suppose a model's accuracy drops on some shifted distribution of inputs $p_{\text{te}}(\boldsymbol{x})$ that differs from the training distribution $p_{\text{tr}}(\boldsymbol{x})$; in this case, the model will necessarily be vunerable to an adversarial attack: if errors exist, there must be a nearest such error. Furthermore, if the input distribution is high dimensional, then we should expect the nearest error to be significantly closer than errors which are sampled randomly from some out-of-distribution $p_{\text{te}}(\boldsymbol{x})$.

A cartoon illustration of what is going on is shown in Figure 19.27a, where $\boldsymbol{x}_0$ is the clean input image, B is an image corrupted by Gaussian noise, and A is an adversarial image. If we assume a linear decision boundary, then the error set E is a half space a certain distance from $\boldsymbol{x}_0$. We can relate the distance to the decision boundary $d(\boldsymbol{x}_0, E)$ with the error rate in noise at some input $\boldsymbol{x}_0$, denoted by $\mu = \mathbb{P}_{\delta\sim N(0,\sigma I)}[\boldsymbol{x}_0 + \delta \in E]$. With a linear decision boundary the relationship between these two quantities is determined by

$$d(\boldsymbol{x}_0, E) = -\sigma \Phi^{-1}(\mu) \tag{19.53}$$

where Φ^{-1} denotes the inverse cdf of the gaussian distribution. When the input dimension is large, this distance will be significantly smaller than the distance to a randomly sampled noisy image $\boldsymbol{x}_0 + \delta$ for $\delta \sim N(0, \sigma I)$, as the noise term will with high propbability have norm $||\delta||_2 \approx \sigma\sqrt{d}$. As a concrete example consider the ImageNet dataset, where $d = 224 \times 224 \times 3$ and suppose we set $\sigma = .2$.



(a) *(b)*

Figure 19.27: (a) When the input dimension n is large and the decision boundary is locally linear, even a small error rate in random noise will imply the existence of small adversarial perturbations. Here, $d(\boldsymbol{x}_0, E)$ denotes the distance from a clean input $\boldsymbol{x}_0$ to an adversarial example (A) while the distance from $\boldsymbol{x}_0$ to a random sample $N(0; \sigma^2 I)$ (B) will be approximately $\sigma\sqrt{n}$. As $n \to \infty$ the ratio of $d(x_0, A)$ to $d(\boldsymbol{x}_0, B)$ goes to 0. (b) A 2d slice of the InceptionV3 decision boundary through three points: a clean image (black), an adversarial example (red), and an error in random noise (blue). The adversarial example and the error in noise lie in the same region of the error set which is misclassified as "miniature poodle", which closely resembles a halfspace as in Figure (a). Used with kind permission of Justin Gilmer.

Then if the error rate in noise is just $\mu = .01$, equation 19.53 will imply that $d(\boldsymbol{x}_0, E) = .5$. Thus the distance to an adversarial example will be more than 100 times closer than the distance to a typical noisy images, which will be $\sigma\sqrt{d} \approx 77.6$. This phenomenon of small volume error sets being close to most points in a data distribution $p(\boldsymbol{x})$ is called **concentration of measure**, and is a property common among many high dimensional data distributions [MDM19; Gil+18b].

In summary, although the existence of adversarial examples is often discussed as an unexpected phenomenon, there is nothing special about the existence of worst-case errors for ML classifiers—they will always exist as long as errors exist.

Part IV

Generation

20 Generative models: an overview

20.1 Introduction

A **generative model** is a joint probability distribution $p(\boldsymbol{x})$, for $\boldsymbol{x} \in \mathcal{X}$. In some cases, the model may be conditioned on inputs or covariates $\boldsymbol{c} \in \mathcal{C}$, which gives rise to a **conditional generative model** of the form $p(\boldsymbol{x}|\boldsymbol{c})$.

There are many kinds of generative model. We give a brief summary in Section 20.2, and go into more detail in subsequent chapters. See also [Tom22] for a recent book on this topic that goes into more depth.

20.2 Types of generative model

There are many kinds of generative model, some of which we list in Table 20.1. At a high level, we can distinguish between **deep generative models** (DGM) — which use deep neural network to learn a complex mapping from a single latent vector $\boldsymbol{z}$ to the observed data $\boldsymbol{x}$ — and more "classical" **probabilistic graphical models** (PGM), that map a set of interconnected latent variables $\boldsymbol{z}_1, \ldots, \boldsymbol{z}_L$ to the observed variables $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_D$ using simpler, often linear, mappings. Of course, many hybrids are possible. For example, PGMs can use neural networks, and DGMs can use structured state spaces. We discuss PGMs in general terms in Chapter 4, and give examples in Chapter 28, Chapter 29, Chapter 30. In this part of the book, we mostly focus on DGMs.

The main kinds of DGM are: **variational autoencoders** (**VAE**), **autoregressive** (**AR**) models, **normalizing flows**, **diffusion models**, **energy based models** (**EBM**), and **generative adversarial networks** (**GAN**). We can categorize these models in terms of the following criteria (see Figure 20.1 for a visual summary):

- Density: does the model support pointwise evaluation of the probability density function $p(\boldsymbol{x})$, and if so, is this fast or slow, exact, approximate or a bound, etc? For **implicit models**, such as GANs, there is no well-defined density $p(\boldsymbol{x})$. For other models, we can only compute a lower bound on the density (VAEs), or an approximation to the density (EBMs, UPGMs).
- Sampling: does the model support generating new samples, $\boldsymbol{x} \sim p(\boldsymbol{x})$, and if so, is this fast or slow, exact or approximate? Directed PGMs, VAEs and GANs all support fast sampling. However, undirected PGMs, EBMs, AR, diffusion and flows are slow for sampling.
- Training: what kind of method is used for parameter estimation? For some models (such as AR, flows and directed PGMs), we can perform exact maximum likelihood estimation (MLE), although

Model	Chapter	Density	Sampling	Training	Latents	Architecture
PGM-D	Section 4.2	Exact, fast	Fast	MLE	Optional	Sparse DAG
PGM-U	Section 4.3	Approx, slow	Slow	MLE-A	Optional	Sparse graph
VAE	Chapter 21	LB, fast	Fast	MLE-LB	$\mathbb{R}^L$	Encoder-Decoder
AR	Chapter 22	Exact, fast	Slow	MLE	None	Sequential
Flows	Chapter 23	Exact, slow/fast	Slow	MLE	$\mathbb{R}^D$	Invertible
EBM	Chapter 24	Approx, slow	Slow	MLE-A	Optional	Discriminative
Diffusion	Chapter 25	LB	Slow	MLE-LB	$\mathbb{R}^D$	Encoder-Decoder
GAN	Chapter 26	NA	Fast	Min-max	$\mathbb{R}^L$	Generator-Discriminator

Table 20.1: Characteristics of common kinds of generative model. Here D is the dimensionality of the observed $\boldsymbol{x}$, and L is the dimensionality of the latent $\boldsymbol{z}$, if present. (We usually assume $L \ll D$, although overcomplete representations can have $L \gg D$.) Abbreviations: Approx = approximate, AR = autoregressive, EBM = Energy Based Model, GAN = generative adversarial network, MLE = maximum likelihood estimation, MLE-A = MLE (Approximate), MLE-LB = MLE (Lower Bound), NA = not available, PGM = probabilistic graphical model, PGM-D = directed PGM, PGM-U = undirected PGM, VAE = variational autoencoder.

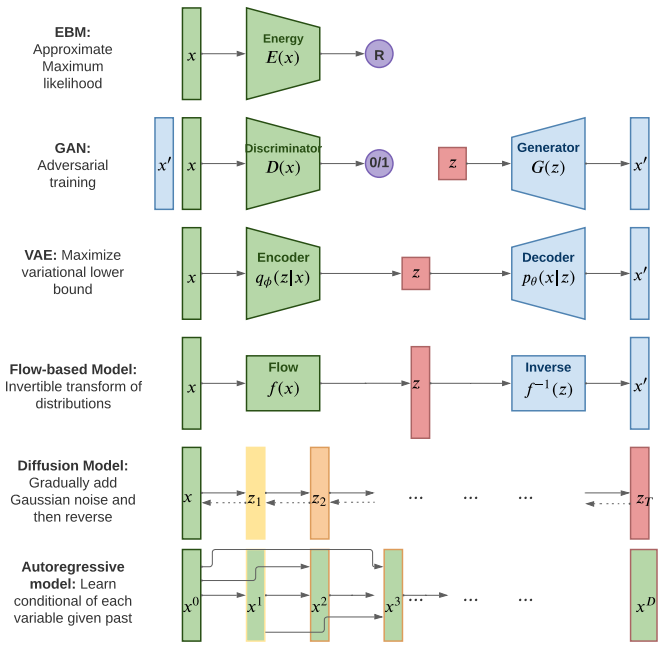


Figure 20.1: Summary of various kinds of deep generative model. Here $\boldsymbol{x}$ is the observed data, $\boldsymbol{z}$ is the latent code, and $\boldsymbol{x}'$ is a sample from the model. AR models do not have a latent code $\boldsymbol{z}$. For diffusion models and flow models, the size of $\boldsymbol{z}$ is the same as $\boldsymbol{x}$. For AR models, x^d is the d'th dimension of $\boldsymbol{x}$. R represents real-valued output, 0/1 represents binary output. Adapted from Figure 1 of [Wen21].

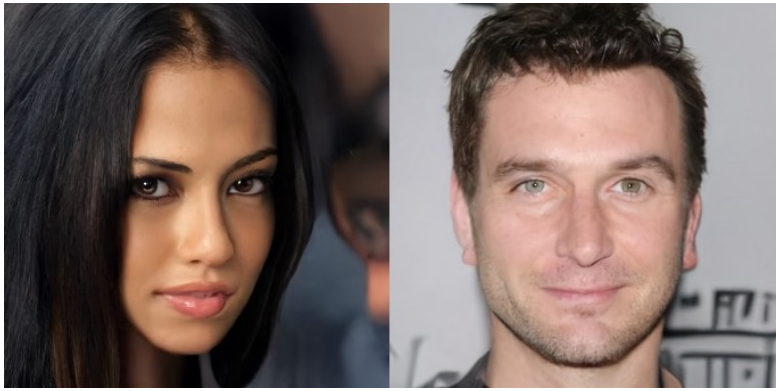


Figure 20.2: Synthetic faces from a score-based generative model (Section 24.3.5). From Figure 12 of [Son+21]. Used with kind permission of Yang Song.

the objective is usually non-convex, so we can only reach a local optimum. For other models, we cannot tractably compute the likelihood. In the case of VAEs, we maximize a lower bound on the likelihood; in the case of EBMs and UGMs, we maximize an approximation to the likelihood. For GANs we have to use min-max training, which can be unstable, and there is no clear objective function to monitor.

- Latents: does the model use use a latent vector $\boldsymbol{z}$ to generate $\boldsymbol{x}$ or not, and if so, is it the same size as $\boldsymbol{x}$ or is it a potentially compressed representation? For example, AR models do not use latents; flows and diffusion use latents, but they are not compressed.[1] Graphical models, including EBMs, may or may not use latents.

- Architecture: what kind of neural network should we use, and are there restrictions? For flows, we are restricted to using invertible neural networks where each layer has a tractable Jacobian. For EBMs, we can use any model we like. The other models have different restrictions.

20.3 Goals of generative modeling

There are several different kinds of tasks that we can use generative models for, as we discuss below.

20.3.1 Generating data

One of the main goals of generative models is to generate (create) new data samples. For example, if we fit a model $p(\boldsymbol{x})$ to images of faces, we can sample new faces from it, as illustrated in Figure 20.2.[2]

1. Flow models define a latent vector $\boldsymbol{z}$ that has the same size as $\boldsymbol{x}$, although the internal deterministic computation may use vectors that are larger or smaller than the input (see e.g., the DenseFlow paper [GGS21]).
2. These images were made with a technique called score-based generative modeling (Section 24.3.5), although similar results can be obtained using many other techniques. See for example https://this-person-does-not-exist.com/en which shows results from a GAN model (Chapter 26).

(a) Teddy bears swimming at the Olympics 400m Butterfly event.

(b) A cute corgi lives in a house made out of sushi.

(c) A cute sloth holding a small treasure chest. A bright golden glow is coming from the chest.

Figure 20.3: Some 1024×1024 *images generated from text prompts by the Imagen diffusion model (Section 25.2.3). From Figure 1 of [Sah+22]. Used with kind permission of William Chan.*

(a)

(b)

(c)

Figure 20.4: Converting a gray-scale image (left) to a colorized one (middle) using a conditional GAN (Chapter 26). Right column shows the original image. (Photo depicts author's wife, Margaret Murphy, photo taken July 2022.) Generated by `https://github.com/jantic/DeOldify`.

Similar methods can be used to create samples of text, audio, etc. When this technology is abused to make fake content, they are called **deep fakes**. For a review of this topic, see e.g., [Ngu+19].

To control what is generated, it is useful to use a **conditional generative model** of the form $p(\boldsymbol{x}|\boldsymbol{c})$. Here are some examples:

- $\boldsymbol{c}$ = text prompt, $\boldsymbol{x}$ = image. This is a **text-to-image** model (see Figure 20.3 and Figure 22.8 for examples).
- $\boldsymbol{c}$ = image, $\boldsymbol{x}$ = text. This is an **image-to-text** model, which is useful for **image captioning**.
- $\boldsymbol{c}$ = image, $\boldsymbol{x}$ = image. This is an **image-to-image** model (see Figure 20.4 and Figure 25.4 for examples).
- $\boldsymbol{c}$ = sequence of sounds, $\boldsymbol{x}$ = sequence of words. This is a **speech-to-text** model, which is useful for **automatic speech recognition** (**ASR**).

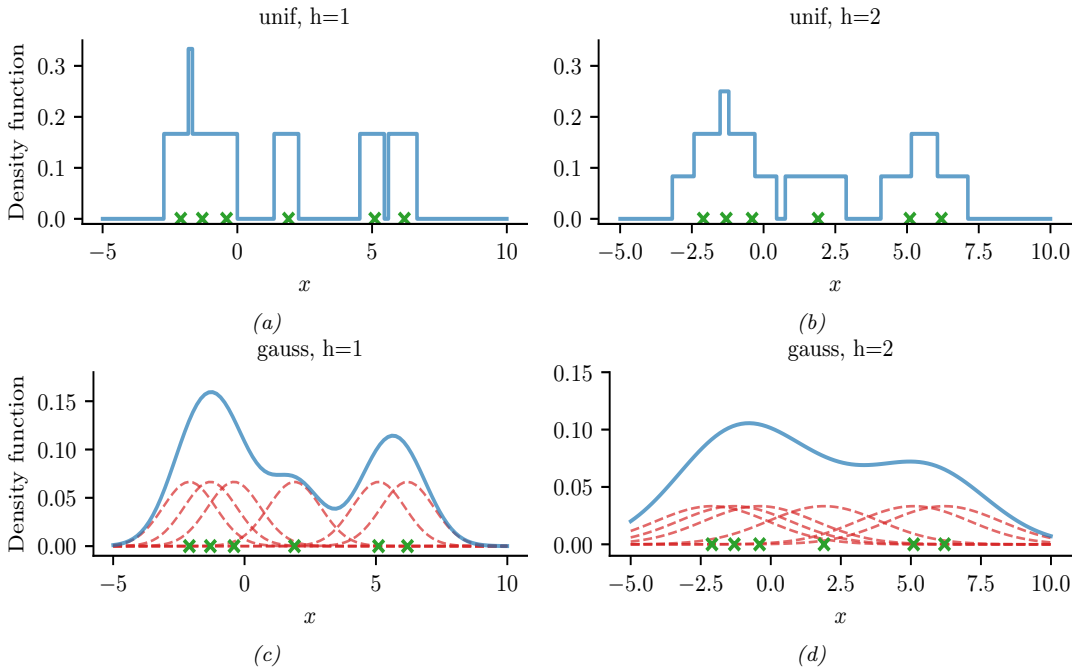


Figure 20.5: A nonparametric (Parzen) density estimator in 1d estimated from 6 data points, denoted by x. Top row: uniform kernel. Bottom row: Gaussian kernel. Left column: bandwidth parameter $h = 1$. Right column: bandwidth parameter $h = 2$. Adapted from http://en.wikipedia.org/wiki/Kernel_density_estimation*. Generated by parzen_window_demo.ipynb.*

- $\boldsymbol{c}$ = sequence of English words, $\boldsymbol{x}$ = sequence of French words. This is a **sequence-to-sequence** model, which is useful for **machine translation**.
- $\boldsymbol{c}$ = initial prompt, $\boldsymbol{x}$ = continuation of the text. This is another sequence-to-sequence model, which is useful for automatic **text generation** (see Figure 22.4 for an example).

Note that, in the conditional case, we sometimes denote the inputs by $\boldsymbol{x}$ and the outputs by $\boldsymbol{y}$. In this case the model has the familiar form $p(\boldsymbol{y}|\boldsymbol{x})$. In the special case that $\boldsymbol{y}$ denotes a low dimensional quantity, such as a integer class label, $y \in \{1, \ldots, C\}$, we get a predictive (discriminative) model. The main difference beween a discriminative model and a conditional generative model is this: in a discriminative model, we assume there is one correct output, whereas in a conditional generative model, we assume there may be multiple correct outputs. This makes it harder to evaluate generative models, as we discuss in Section 20.4.

20.3.2 Density estimation

The task of **density estimation** refers to evaluating the probablity of an observed data vector, i.e., computing $p(\boldsymbol{x})$. This can be useful for outlier detection (Section 19.3.2), data compression (Section 5.4), generative classifiers, model comparison, etc.

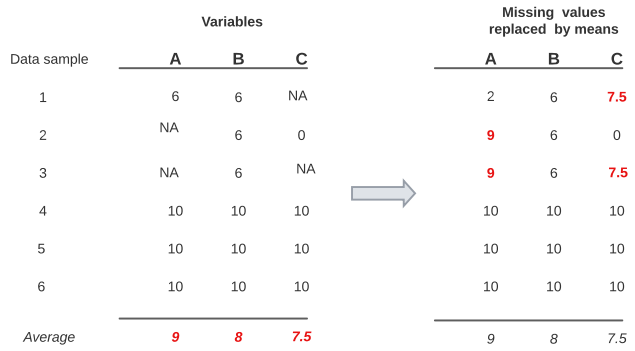


Figure 20.6: Missing data imputation using the mean of each column.

A simple approach to this problem, which works in low dimensions, is to use **kernel density estimation** or **KDE**, which has the form

$$p(\boldsymbol{x}|\mathcal{D}) = \frac{1}{N}\sum_{n=1}^{N} \mathcal{K}_h\left(\boldsymbol{x} - \boldsymbol{x}_n\right) \tag{20.1}$$

Here $\mathcal{D} = \{vx_1, \ldots, \boldsymbol{x}_N\}$ is the data, and $\mathcal{K}_h$ is a density kernel with **bandwidth** h, which is a function $\mathcal{K} : \mathbb{R} \to \mathbb{R}_+$ such that $\int \mathcal{K}(x)dx = 1$ and $\int x\mathcal{K}(x)dx = 0$. We give a 1d example of this in Figure 20.5: in the top row, we use a uniform (boxcar) kernel, and in the bottomr row we use a Gaussian kernel.

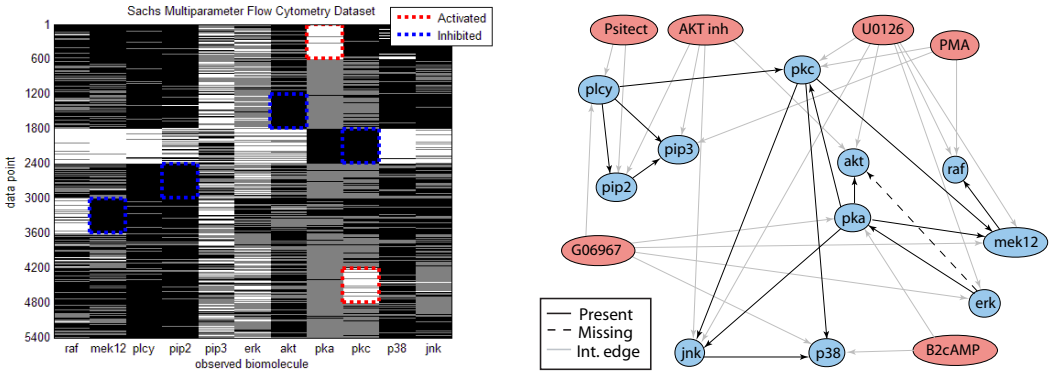
In higher dimensions, KDE suffers from the **curse of dimensionality** (see e.g., [AHK01]), and we need to use parametric density models $p_{\boldsymbol{\theta}}(\boldsymbol{x})$ of some kind.

20.3.3 Imputation

The task of **imputation** refers to "filling in" missing values of a data vector or data matrix. For example, suppose $\mathbf{X}$ is an $N \times D$ matrix of data (think of a spreadsheet) in which some entries, call them $\mathbf{X}_m$, may be missing, while the rest, $\mathbf{X}_o$, are observed. A simple way to fill in the missing data is to use the mean value of each feature, $\mathbb{E}\left[x_d\right]$; this is called **mean value imputation**, and is illustrated in Figure 20.6. However, this ignores dependencies between the variables within each row, and does not return any measure of uncertainty.

We can generalize this by fitting a generative model to the observed data, $p(\mathbf{X}_o)$, and then computing samples from $p(\mathbf{X}_m|\mathbf{X}_o)$. This is called **multiple imputation**. A generative model can be used to fill in more complex data types, such as **in-painting** occluded pixels in an image (see Figure 25.4).

See Section 21.3.4 for a more general discussion of missing data.



(a) *(b)*

Figure 20.7: (a) A design matrix consisting of 5400 data points (rows) measuring the state (using flow cytometry) of 11 proteins (columns) under different experimental conditions. The data has been discretized into 3 states: low (black), medium (grey) and high (white). Some proteins were explicitly controlled using activating or inhibiting chemicals. (b) A directed graphical model representing dependencies between various proteins (blue circles) and various experimental interventions (pink ovals), which was inferred from this data. We plot all edges for which $p(G_{ij} = 1|\mathcal{D}) > 0.5$. Dotted edges are believed to exist in nature but were not discovered by the algorithm (1 false negative). Solid edges are true positives. The light colored edges represent the effects of intervention. From Figure 6d of [EM07].

20.3.4 Structure discovery

Some kinds of generative models have latent variables $\boldsymbol{z}$, which are assumed to be the "causes" that generated the observed data $\boldsymbol{x}$. We can use Bayes rule to invert the model to compute $p(\boldsymbol{z}|\boldsymbol{x}) \propto p(\boldsymbol{z})p(\boldsymbol{x}|\boldsymbol{z})$. This can be useful for discovering latent, low-dimensional patterns in the data.

For example, suppose we perturb various proteins in a cell and measure the resulting phosphorylation state using a technique known as flow cytometry, as in [Sac+05]. An example of such a dataset is shown in Figure 20.7(a). Each row represents a data sample $\boldsymbol{x}_n \sim p(\cdot|\boldsymbol{a}_n, \boldsymbol{z})$, where $\boldsymbol{x} \in \mathbb{R}^{11}$ is a vector of outputs (phosphorylations), $\boldsymbol{a} \in \{0, 1\}^6$ is a vector of input actions (perturbations) and $\boldsymbol{z}$ is the unknown cellular signaling network structure. We can infer the graph structure $p(\boldsymbol{z}|\mathcal{D})$ using graphical model structure learning techhniques (see Section 30.3). In particular, we can use the dynamic programming method described in [EM07] to get the result is shown in Figure 20.7(b). Here we plot the median graph, which includes all edges for which $p(z_{ij} = 1|\mathcal{D}) > 0.5$. (For a more recent approach to this problem, see e.g., [Bro+20b].)

20.3.5 Latent space interpolation

One of the most interesting abilities of certain latent variable models is the ability to generate samples that have certain desired properties by interpolating between existing data points in latent space. To explain how this works, let $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ be two inputs (e.g. images), and let $\boldsymbol{z}_1 = e(\boldsymbol{x}_1)$ and $\boldsymbol{z}_2 = e(\boldsymbol{x}_2)$ be their latent encodings. (The method used for computing these will depend on the type of model; we discuss the details in later chapters.) We can regard $\boldsymbol{z}_1$ and $\boldsymbol{z}_2$ as two "anchors" in

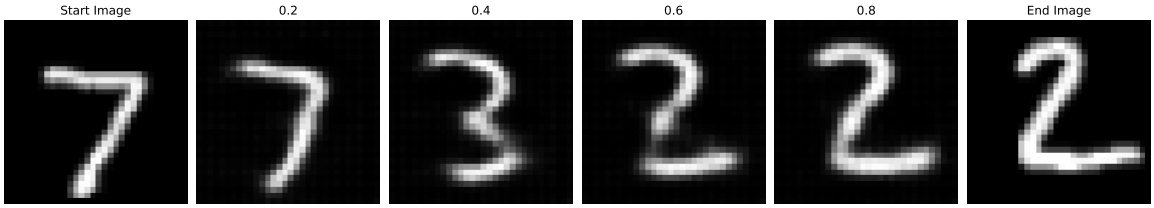


Figure 20.8: Interpolation between two MNIST images in the latent space of a β-VAE (with $\beta = 0.5$). Generated by mnist_vae_ae_comparison.ipynb.

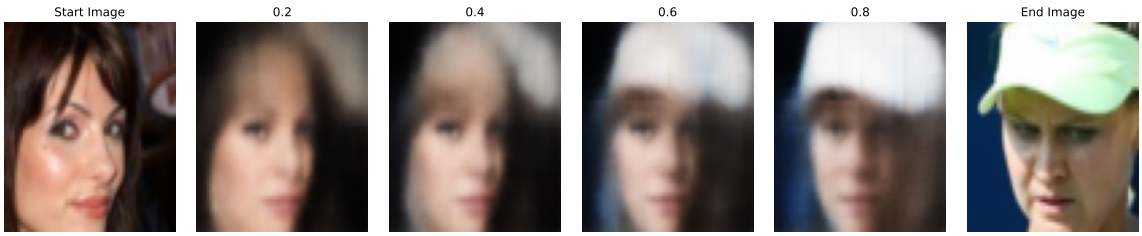


Figure 20.9: Interpolation between two CelebA images in the latent space of a β-VAE (with $\beta = 0.5$). Generated by celeba_vae_ae_comparison.ipynb.

latent space. We can now generate new images that interpolate between these points by computing $\boldsymbol{z} = \lambda \boldsymbol{z}_1 + (1-\lambda)\boldsymbol{z}_2$, where $0 \leq \lambda \leq 1$, and then decoding by computing $\boldsymbol{x}' = d(\boldsymbol{z})$, where $d()$ is the decoder. This is called **latent space interpolation**, and will generate data that combines semantic features from both $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$. (The justification for taking a linear interpolation is that the learned manifold often has approximately zero curvature, as shown in [SKTF18]. However, sometimes it is better to use nonlinear interpolation [Whi16; MB21; Fad+20].)

We can an example of this process in Figure 20.8, where we use a β-VAE model (Section 21.3.1) fit to the MNIST dataset. We see that the model is able to produce plausible interpolations between the digit 7 and the digit 2. As a more interesting example, we can fit a β-VAE to the the **CelebA** dataset [Liu+15].[3] The results are shown in Figure 20.9, and look reasonable. (We can get much better quality if we use a larger model trained on more data for a longer amount of time.)

It is also possible to perform interpolation in the latent space of text models, as illustrated in Figure 21.10.

20.3.6 Latent space arithmetic

In some cases, we can go beyond interpolation, and can perform **latent space arithmetic**, in which we can increase or decrease the amount of a desired "semantic factor of variation". This was first shown in the **word2vec** model [Mik+13], but it also is possible in other latent variable models. For example, consider our VAE model fit to CelebA dataset, which has faces of celebrities and some

3. CelebA contains about 200k images of famous celebrities. The images are also annotated with 40 attributes. We reduce the resolution of the images to 64x64, as is conventional. See Section 21.2.6 for the details of the model.

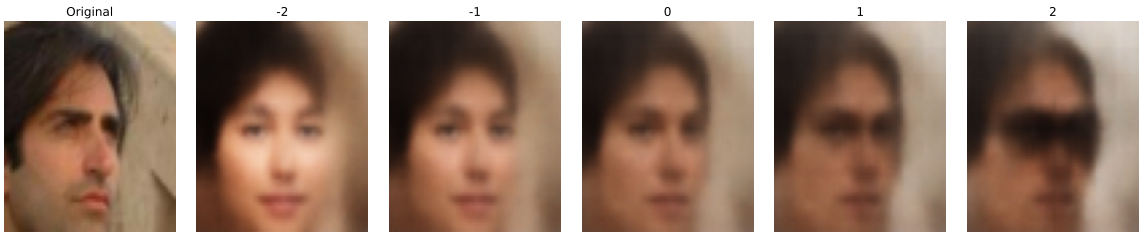


Figure 20.10: Arithmetic in the latent space of a β-VAE (with $\beta = 0.5$). The first column is an input image, with embedding $\boldsymbol{z}$. Subsequent columns show the decoding of $\boldsymbol{z} + s\boldsymbol{\Delta}$, where $s \in \{-2, -1, 0, 1, 2\}$ and $\boldsymbol{\Delta} = \overline{\boldsymbol{z}}^+ - \overline{\boldsymbol{z}}^-$ is the difference in the average embeddings of images with or without a certain attribute (here, wearing sunglasses). Generated by celeba_vae_ae_comparison.ipynb.

corresponding attributes. Let $\mathbf{X}_i^+$ be a set of images which have attribute i, and $\mathbf{X}_i^-$ be a set of images which do not have this attribute. Let $\mathbf{Z}_i^+$ and $\mathbf{Z}_i^-$ be the corresponding embeddings, and $\overline{\boldsymbol{z}}_i^+$ and $\overline{\boldsymbol{z}}_i^-$ be the average of these embeddings. We define the offset vector as $\boldsymbol{\Delta}_i = \overline{\boldsymbol{z}}_i^+ - \overline{\boldsymbol{z}}_i^-$. If we add some positive multiple of $\boldsymbol{\Delta}_i$ to a new point $\boldsymbol{z}$, we increase the amount of the attribute i; if we subtract some multiple of $\boldsymbol{\Delta}_i$, we decrease the amount of the attribute i. [Whi16].

We give an example of this in Figure 20.10. We consider the attribute of wearing sunglasses. The j'th reconstruction is computed using $\hat{\boldsymbol{x}}_j = d(\boldsymbol{z} + s_j\boldsymbol{\Delta})$, where $\boldsymbol{z} = e(\boldsymbol{x})$ is the encoding of the original image, and s_j is a scale factor. When $s_j > 0$ we add sunglasses to the face. When $s_j < 0$ we remove sunglasses; but this also has the side effect of making the face look younger and more female, possibly a result of dataset bias.

20.3.7 Generative design

Another interesting use case for (deep) generative models is **generative design**, in which we use the model to generate candidate objects, such as molecules, which have desired properties (see e.g., [RNA22]). One approach is to fit a VAE to unlabeled samples, and then to perform Bayesian optimization (Section 6.8) in its latent space, as discussed in Section 21.3.6.2.

20.3.8 Model-based reinforcement learning

We discuss reinforcement learning (RL) in Chapter 35. The main success stories of RL to date have been in computer games, where simulators exist and data is abundant. However, in other areas, such as robotics, data is expensive to acquire. In this case, it can be useful to learn a generative "**world model**", so the agent can do planning and learning "in it's head". See Section 35.4 for more details.

20.3.9 Representation learning

Representation learning refers to learning (possibly uninterpretable) latent factors $\boldsymbol{z}$ that generate the observed data $\boldsymbol{x}$. The primary goal is for these features to be used in "**downstream**" supervised tasks. This is discussed in Chapter 32.

20.3.10 Data compression

Models which can assign high probability to frequently occuring data vectors (e.g., images, sentences), and low probability to rare vectors, can be used for **data compression**, since we can assign shorter codes to the more common items. Indeed, the optimal coding length for a vector $\boldsymbol{x}$ from some stochastic source $p(\boldsymbol{x})$ is $l(\boldsymbol{x}) = -\log p(\boldsymbol{x})$, as proved by Shannon. See Section 5.4 for details.

20.4 Evaluating generative models

This section is coauthored with Mihaela Rosca, Shakir Mohamed and Balaji Lakshminarayanan.

Evaluating generative models requires metrics which capture

- **sample quality** - are samples generated by the model a part of the data distribution?
- **sample diversity** - are samples from the model distribution capturing all modes of the data distribution?
- **generalization** - is the model generalizing beyond the training data?

There is no known metric which meets all these requirements, but various metrics have been proposed to capture different aspects of the learned distribution, some of which we discuss below.

20.4.1 Likelihood-based evaluation

A standard way to measure how close a model q is to a true distribution p is in terms of the KL divergence (Section 5.1):

$$D_{\mathbb{KL}}\left(p \parallel q\right) = \int p(\boldsymbol{x}) \log \frac{p(\boldsymbol{x})}{q(\boldsymbol{x})} = -\mathbb{H}\left(p\right) + \mathbb{H}_{ce}\left(p, q\right) \tag{20.2}$$

where $\mathbb{H}\left(p\right)$ is a constant, and $\mathbb{H}_{ce}\left(p, q\right)$ is the cross entropy. If we approximate $p(\boldsymbol{x})$ by the empirical distribution, we can evaluate the cross entropy in terms of the empirical **negative log likelihood** on the dataset:

$$\text{NLL} = -\frac{1}{N} \sum_{n=1}^{N} \log q(\boldsymbol{x}_n) \tag{20.3}$$

Usually we care about negative log likelihood on a held-out test set.[4]

20.4.1.1 Computing log-likelihood

For models of discrete data, such as language models, it is easy to compute the (negative) log likelihood. However, it is common to measure performance using a quantity called **perplexity**, which is defined as 2^H, where $H = \text{NLL}$ is the cross entropy or negative log likelihood.

4. In some applications, we report **bits per dimension**, which is the NLL using log base 2, divided by the dimensionality of $\boldsymbol{x}$. (To compute this metric, recall that $\log_2 L = \frac{\log_e L}{\log_e 2}$.)

For image and audio models, one complication is that the model is usually a continuous distribution $p(\boldsymbol{x}) \geq 0$ but the data is usually discrete (e.g., $\boldsymbol{x} \in \{0, \ldots, 255\}^D$ if we use one byte per pixel). Consequently the average log likelihood can be arbitrary large, since the pdf can be bigger than 1. To avoid this it is standard pratice to use **uniform dequantization** [TOB16], in which we add uniform random noise to the discrete data, and then treat it as continuous-valued data. This gives a lower bound on the average log likelihood of the discrete model on the original data.

To see this, let $\boldsymbol{z}$ be a continuous latent variable, and $\boldsymbol{x}$ be a vector of binary observations computed by rounding, so $p(\boldsymbol{x}|\boldsymbol{z}) = \delta(\boldsymbol{x} - \text{round}(\boldsymbol{z}))$, computed elementwise. We have $p(\boldsymbol{x}) = \int p(\boldsymbol{x}|\boldsymbol{z})p(\boldsymbol{z})d\boldsymbol{z}$. Let $q(\boldsymbol{z}|\boldsymbol{x})$ be a probabilistic inverse of $\boldsymbol{x}$, that is, it has support only on values where $p(\boldsymbol{x}|\boldsymbol{z}) = 1$. In this case, Jensen's inequality gives

$$\log p(\boldsymbol{x}) \geq \mathbb{E}_{q(\boldsymbol{z}|\boldsymbol{x})} \left[\log p(\boldsymbol{x}|\boldsymbol{z}) + \log p(\boldsymbol{z}) - \log q(\boldsymbol{z}|\boldsymbol{x})\right] \tag{20.4}$$

$$= \mathbb{E}_{q(\boldsymbol{z}|\boldsymbol{x})} \left[\log p(\boldsymbol{z}) - \log q(\boldsymbol{z}|\boldsymbol{x})\right] \tag{20.5}$$

Thus if we model the density of $\boldsymbol{z} \sim q(\boldsymbol{z}|\boldsymbol{x})$, which is a dequantized version of $\boldsymbol{x}$, we will get a lower bound on $p(\boldsymbol{x})$.

20.4.1.2 Challenges with using the likelihood

Unfortunately, there are several challenges with using the likelihood to evaluate generative models, some of which we discuss below.

20.4.1.3 Likelihood can be hard to compute

For many models, computing the likelihood can be computationally expensive, since it requires knowing the normalization constant of the probability model. One solution is to use variational inference (Chapter 10), which provides a way to efficiently compute lower (and sometimes upper) bounds on the log likelihood. Another solution is to use annealed importance sampling (Section 11.5.4.1), which provides a way to estimate the log likelihood using Monte Carlo sampling. However, in the case of implicit generative models, such as GANs (Chapter 26), the likelihood is not even defined, so we need to find evaluation metrics that do not rely on likelihood.

20.4.1.4 Likelihood is not related to sample quality

A more subtle concern with likelihood is that it is often uncorrelated with the perceptual quality of the samples, at least for real-valued data, such as images and sound. In particular, a model can have great log-likelihood but create poor samples and vice versa.

To see why a model can have good likelihoods but create bad samples, consider the following argument from [TOB16]. Suppose q_0 is a density model for D-dimensional data $\boldsymbol{x}$ which performs arbitrarily well as judged by average log-likelihood, and suppose q_1 is a bad model, such as white noise. Now consider samples generated from the mixture model

$$q_2(\boldsymbol{x}) = 0.01 q_0(\boldsymbol{x}) + 0.99 q_1(\boldsymbol{x}) \tag{20.6}$$

Clearly 99% of the samples will be poor. However, the log-likelihood per pixel will hardly change between q_2 and q_0 if D is large, since

$$\log q_2(\boldsymbol{x}) = \log[0.01 q_0(\boldsymbol{x}) + 0.99 q_1(\boldsymbol{x})] \geq \log[0.01 q_0(\boldsymbol{x})] = \log q_0(\boldsymbol{x}) - 100 \tag{20.7}$$

For high-dimensional data, $|\log q_0(\boldsymbol{x})| \sim D \gg 100$, so $\log q_2(\boldsymbol{x}) \approx \log q_0(\boldsymbol{x})$, and hence mixing in the poor sampler does not significantly impact the log likelihood.

Now consider a case where the model has good samples but bad likelihoods. To achieve this, suppose q is a GMM centered on the training images:

$$q(\boldsymbol{x}) = \frac{1}{N}\sum_{n=1}^{N} \mathcal{N}(\boldsymbol{x}|\boldsymbol{x}_n, \epsilon^2 \mathbf{I}) \tag{20.8}$$

If ϵ is small enough that the Gaussian noise is imperceptible, then samples from this model will look good, since they correspond to the training set of real images. But this model will almost certainly have poor likelihood on the test set due to overfitting. (In this case we say the model has effectively just memorized the training set.)

20.4.2 Distances and divergences in feature space

Due to the challenges associated with comparing distributions in high dimensional spaces, and the desire to compare distributions in a semantically meaningful way, it is common to use domain-specific **perceptual distance metrics**, that measure how similar data vectors are to each other or to the training data. However, most metrics used to evaluate generative models do not directly compare raw data (e.g. pixels) but use a neural network to obtain features from the raw data and compare the feature distribution obtained from model samples with the feature distribution obtained from the dataset. The neural network used to obtain features can be trained solely for the purpose of evaluation, or can be pretrained; a common choice is to use a pretrained classifier (see e.g., [Sal+16; Heu+17b; Bin+18; Kyn+19; SSG18a]).

The **Inception Score** [Sal+16] measures the average KL divergence between the marginal distribution of class labels obtained from the samples $p_{\boldsymbol{\theta}}(y) = \int p(y|\boldsymbol{x}) p_{\boldsymbol{\theta}}(\boldsymbol{x})$ and the distribution $p(y|\boldsymbol{x})$ obtained from a sample $\boldsymbol{x} \sim p_{\boldsymbol{\theta}}(\boldsymbol{x})$. This leads to the following score:

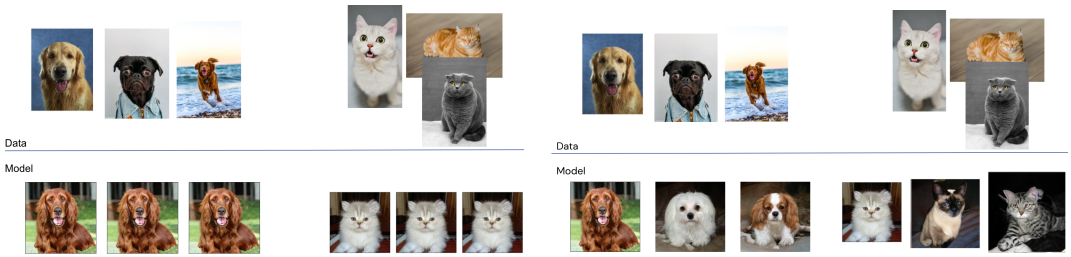
$$\text{IS} = \exp\left[\mathbf{E}_{p_{\boldsymbol{\theta}}(\boldsymbol{x})} D_{\mathbb{KL}}\left(p(y|\boldsymbol{x}) \parallel p_{\boldsymbol{\theta}}(y)\right)\right] \tag{20.9}$$

If a model produces high quality samples from all classes in the dataset, then $p_{\boldsymbol{\theta}}(y)$ will often be close to uniform, while $p(y|\boldsymbol{x})$ should be a sharp distribution corresponding to the class associated with $\boldsymbol{x}$; this leads to a high $D_{\mathbb{KL}}\left(p(y|\boldsymbol{x}) \parallel p_{\boldsymbol{\theta}}(y)\right)$ and thus a high Inception Score score.

The Inception Score solely relies on class labels, and thus does not measure overfitting or sample diversity outside the predifined dataset classes. For example, a model which generates one perfect example per class would get a perfect Inception Score, despite not capturing the variety of examples inside a class, as shown in Figure 20.11a. To address this drawback, the **Fréchet Inception Distance** or **FID** score [Heu+17b] measures the Fréchet distance between two Gaussian distributions on sets of features of a pre-trained classifier. One Gaussian is obtained by passing model samples through a pretrained classifier, and the other by passing samples the dataset through the same classifier. If we assume that the mean and covariance obtained from model features are $\boldsymbol{\mu}_m$ and $\boldsymbol{\Sigma}_m$ and those from the data are $\boldsymbol{\mu}_d$ and $\boldsymbol{\Sigma}_d$, then the FID is

$$\text{FID} = \|\boldsymbol{\mu}_m - \boldsymbol{\mu}_d\|_2^2 + \text{trace}\Big(\boldsymbol{\Sigma}_d + \boldsymbol{\Sigma}_m - 2(\boldsymbol{\Sigma}_d \boldsymbol{\Sigma}_m)^{1/2}\Big) \tag{20.10}$$

Since it uses features instead of class logits, the Fréchet distance captures more than modes captured by class labels, as shown in Figure 20.11b. Unlike the Inception score, a lower score is better since we want the two distributions to be as close as possible.



(a) (b)

Figure 20.11: (a) Model samples with good (high) inception score are visually realistic. (b) Model samples with good (low) FID score are visually realistic and diverse.

Unfortunately, the Fréchet distance has been shown to have a high bias, with results varying widely based on the number of samples used to compute the score. To mitigate this issue, the **Kernel Inception Distance** has been introduced [Bin+18], which measures the squared MMD (Section 2.7.3) between the features obtained from the data and features obtained from model samples.

20.4.3 Precision and recall metrics

Since the FID only measures the distance between the data and model distributions, it is difficult to use it as a diagnostic tool: a bad (high) FID can indicate that the model is not able to generate high quality data, or that it puts too much mass around the data distribution, or that the model only captures a subset of the data (e.g. in Figure 26.7). Trying to disentangle between these two failure modes has been the motivation to seek individual precision (sample quality) and recall (sample diversity) metrics in the context of generative models [LPO17; Kyn+19]. (The diversity question is especially important in the context of GANs, where mode collapse (Section 26.3.3) can be an issue.)

A common approach is to use nearest neighbors in the feature space of a pretrained classifier to define precision and recall [Kyn+19]. To formalize this, let us define

$$f_k(\phi, \Phi) = \begin{cases} 1 & \text{if } \exists \phi' \in \Phi s.t. \|\phi - \phi'\|_2^2 \leq \|\phi' - \text{NN}_k(\phi', \Phi)\|_2^2 \\ 0 & \text{otherwise} \end{cases} \tag{20.11}$$

where Φ is a set of feature vectors and $\text{NN}_k(\phi', \Phi)$ is a function returning the k-th nearest neighbor of ϕ' in Φ. We now define precision and recall as follows:

$$\text{precision}(\Phi_{model}, \Phi_{data}) = \frac{1}{|\Phi_{model}|} \sum_{\phi \in \Phi_{model}} f_k(\phi, \Phi_{data}); \tag{20.12}$$

$$\text{recall}(\Phi_{model}, \Phi_{data}) = \frac{1}{|\Phi_{data}|} \sum_{\phi \in \Phi_{data}} f_k(\phi, \Phi_{model}); \tag{20.13}$$

Precision and recall are always between 0 and 1. Intuitively, the precision metric measures whether samples are as close to data as data is to other data examples, while recall measures whether data is as close to model samples as model samples are to other samples. The parameter k controls

how lenient the metrics will be – the higher k, the higher both precision and recall will be. As in classification, precision and recall in generative models can be used to construct a trade-off curve between different models which allows practitioners to make an informed decision regarding which model they want to use.

20.4.4 Statistical tests

Statistical tests have long been used to determine whether two sets of samples have been generated from the same distribution; these types of statistical tests are called **two sample tests**. Let us define the null hypothesis as the statement that both set of samples are from the same distribution. We then compute a statistic from the data and compare it to a threshold, and based on this we decide whether to reject the null hypothesis. In the context of evaluating implicit generative models such as GANs, statistics based on classifiers [Saj+18] and the MMD [Liu+20b] have been used. For use in scenarios with high dimensional input spaces, which are ubiquitous in the era of deep learning, two sample tests have been adapted to use learned features instead of raw data.

Like all other evaluation metrics for generative models, statistical tests have their own advantages and disadvantages: while users can specify Type 1 error – the chance they allow that the null hypothesis is wrongly rejected – statistical tests tend to be computationally expensive and thus cannot be used to monitor progress in training; hence they are best used to compare fully trained models.

20.4.5 Challenges with using pretrained classifiers

While popular and convenient, evaluation metrics that rely on pretrained classifiers (such as IS, FID, nearest neighbors in feature space, and statistical tests in feature space) have significant drawbacks. One might not have a pretrained classifier available for the dataset at hand, so classifiers trained on other datasets are used. Given the well known challenges with neural network generalization (see Section 17.4), the features of a classifier trained on images from one dataset might not be reliable enough to provide a fine grained signal of quality for samples obtained from a model trained on a different dataset. If the generative model is trained on the same dataset as the pre-trained classifier but the model is not capturing the data distribution perfectly, we are presenting the pre-trained classifier with out-of-distribution data and relying on its features to obtain score to evaluate our models. Far from being purely theoretical concerns, these issues have been studied extensively and have been shown to affect evaluation in practice [RV19; BS18].

20.4.6 Using model samples to train classifiers

Instead of using pretrained classifiers to evaluate samples, one can train a classifier on samples from conditional generative models, and then see how good these classifiers are at classifying data. For example, does adding synthetic (sampled) data to the real data help? This is closer to a reliable evaluation of generative model samples, since ultimately, the performance of generative models is dependent on the downstream task they are trained for. If used for semi supervised learning, one should assess how much adding samples to a classifier dataset helps with test accuracy. If used for model based reinforcement learning, one should assess how much the generative model helps with agent performance. For examples of this approach, see e.g., [SSM18; SSA18; RV19; SS20b].

Figure 20.12: Illustration of nearest neighbors in feature space: in the top left we have the query sample generated using BigGAN, and the rest of the images are its nearest neighbors from the dataset. The nearest neighbors search is done in the feature space of a pretrained classifier. From Figure 13 of [BDS18]. Used with kind permission of Andy Brock.

20.4.7 Assessing overfitting

Many of the metrics discussed so far capture the sample quality and diversity, but do not capture overfitting to the training data. To capture overfitting, often a visual inspection is performed: a set of samples is generated from the model and for each sample its closest K nearest neighbors in the feature space of a pretrained classifier are obtained from the dataset. While this approach requires manually assessing samples, it is a simple way to test whether a model is simply memorizing the data. We show an example in Figure 20.12: since the model sample in the top left is quite different than its neighbors from the dataset (remaining images), we can conclude the sample is not simply memorised from the dataset. Similarly, sample diversity can be measured by approximating the support of the learned distribution by looking for similar samples in a large sample pool — as in the pigeonhole principle — but it is expensive and often requires manual human assessment[AZ17].

For likelihood-based models — such as variational autoencoders Chapter 21, autoregressive models Chapter 22, and normalising flows Chapter 23 — we can assess memorisation by seeing how much the log-likelihood of a model changes when a sample is included in the model's training set or not [BW21].

20.4.8 Human evaluation

One approach to evaluate generative models is to use human evaluation, by presenting samples from the model alongside samples from the data distribution, and ask human raters to compare the quality

of the samples [Zho+19b]. Human evaluation is a suitable metric if the model is used to create art or other data for human display, or if reliable automated metrics are hard to obtain. However, human evaluation can be difficult to standardize, hard to automate and can be expensive or cumbersome to set up.

21 Variational autoencoders

21.1 Introduction

In this chapter, we discuss generative models of the form

$$\boldsymbol{z} \sim p_{\boldsymbol{\theta}}(\boldsymbol{z}) \tag{21.1}$$

$$\boldsymbol{x}|\boldsymbol{z} \sim \text{Expfam}(\boldsymbol{x}|d_{\boldsymbol{\theta}}(\boldsymbol{z})) \tag{21.2}$$

where $p(\boldsymbol{z})$ is some kind of prior on the latent code $\boldsymbol{z}$, $d_{\boldsymbol{\theta}}(\boldsymbol{z})$ is a deep neural network, known as the **decoder**, and $\text{Expfam}(\boldsymbol{x}|\boldsymbol{\eta})$ is an exponential family distribution, such as a Gaussian or product of Bernoullis. This is called a **deep latent variable model** or **DLVM**. When the prior is Gaussian (as is often the case), this model is called a **deep latent Gaussian model** or **DLGM**.

Posterior inference (i.e., computing $p_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x})$) is computationally intractable, as is computing the marginal likelihood

$$p_{\boldsymbol{\theta}}(\boldsymbol{x}) = \int p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z}) p_{\boldsymbol{\theta}}(\boldsymbol{z})\, d\boldsymbol{z} \tag{21.3}$$

Hence we need to resort to approximate inference. For most of this chapter, we will use **amortized inference**, which we discussed in Section 10.3.6. This trains another model, $q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})$, called the **recognition network** or **inference network**, simultaneously with the generative model to do approximate posterior inference. This combination is called a **variational autoencoder** or **VAE** [KW14; RMW14b; KW19a], since it can be thought of as a probabilistic version of a deterministic autoencoder, discussed in Section 16.3.3.

In this chapter, we introduce the basic VAE, as well as some extensions. Note that the literature on VAE-like methods is vast[1], so we will only discuss a small subset of the ideas that have been explored.

21.2 VAE basics

In this section, we discuss the basics of variational autoencoders.

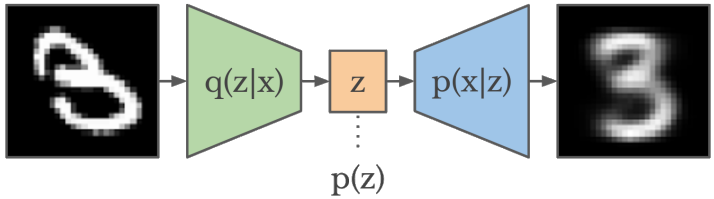


Figure 21.1: Schematic illustration of a VAE. From a figure in [Haf18]. Used with kind permission of Danijar Hafner.

21.2.1 Modeling assumptions

In the simplest setting, a VAE defines a generative model of the form

$$p_{\boldsymbol{\theta}}(\boldsymbol{z}, \boldsymbol{x}) = p_{\boldsymbol{\theta}}(\boldsymbol{z}) p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z}) \tag{21.4}$$

where $p_{\boldsymbol{\theta}}(\boldsymbol{z})$ is usually a Gaussian, and $p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})$ is usually a product of exponential family distributions (e.g., Gaussians or Bernoullis), with parameters computed by a neural network decoder, $d_{\boldsymbol{\theta}}(\boldsymbol{z})$. For example, for binary observations, we can use

$$p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z}) = \prod_{d=1}^{D} \text{Ber}(x_d|\sigma(d_{\boldsymbol{\theta}}(\boldsymbol{z})) \tag{21.5}$$

In addition, a VAE fits a recognition model

$$q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}) = q(\boldsymbol{z}|e_{\boldsymbol{\phi}}(\boldsymbol{x})) \approx p_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x}) \tag{21.6}$$

to perform approximate posterior inference. Here $q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})$ is usually a Gaussian, with parameters computed by a neural network encoder $e_{\boldsymbol{\phi}}(\boldsymbol{x})$:

$$q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}) = \mathcal{N}(\boldsymbol{z}|\boldsymbol{\mu}, \text{diag}(\exp(\boldsymbol{\ell}))) \tag{21.7}$$

$$(\boldsymbol{\mu}, \boldsymbol{\ell}) = e_{\boldsymbol{\phi}}(\boldsymbol{x}) \tag{21.8}$$

where $\boldsymbol{\ell} = \log \boldsymbol{\sigma}$. The model can be thought of as encoding the input $\boldsymbol{x}$ into a stochastic latent bottleneck $\boldsymbol{z}$ and then decoding it to approximately reconstruct the input, as shown in Figure 21.1.

The idea of training an inference network to "invert" a generative network, rather than running an optimization algorithm to infer the latent code, is called amortized inference, and is discussed in Section 10.3.6. This idea was first proposed in the **Helmholtz machine** [Day+95]. However, that paper did not present a single unified objective function for inference and generation, but instead used the wake-sleep (Section 10.6) method for training. By contrast, the VAE optimizes a variational lower bound on the log-likelihood, which means that convergence to a locally optimal MLE of the parameters is guaranteed.

We can use other approaches to fitting the DLGM (see e.g., [Hof17; DF19]). However, learning an inference network to fit the DLGM is often faster and can have some regularization benefits (see e.g., [KP20]).[2]

1. For example, the website https://github.com/matthewvowels1/Awesome-VAEs lists over 900 papers.
2. Combining a generative model with an inference model in this way results in what has been called a "**monference**"

21.2.2 Evidence lower bound (ELBO)

When fitting the model, our goal is to maximize the marginal likelihood

$$p_{\boldsymbol{\theta}}(\boldsymbol{x}) = \int p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z}) p_{\boldsymbol{\theta}}(\boldsymbol{z}) \, d\boldsymbol{z} \tag{21.9}$$

Unfortunately, computing this quantity is intractable. However, we can use an inference network to compute an approximate posterior, $q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})$, and hence a lower bound to the marginal likelihood. This idea is discussed in Section 10.1.2, but we repeat the details here, using slightly different notation.

First note that we have the following decomposition:

$$\log p_{\boldsymbol{\theta}}(\boldsymbol{x}) = \mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\left[\log p_{\boldsymbol{\theta}}(\boldsymbol{x})\right] \tag{21.10}$$

$$= \mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\left[\log\left(\frac{p_{\boldsymbol{\theta}}(\boldsymbol{x},\boldsymbol{z})}{p_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x})}\right)\right] \tag{21.11}$$

$$= \mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\left[\log\left(\frac{p_{\boldsymbol{\theta}}(\boldsymbol{x},\boldsymbol{z})}{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\frac{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}{p_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x})}\right)\right] \tag{21.12}$$

$$= \underbrace{\mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\left[\log\left(\frac{p_{\boldsymbol{\theta}}(\boldsymbol{x},\boldsymbol{z})}{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\right)\right]}_{Ł_{\boldsymbol{\theta},\boldsymbol{\phi}}(\boldsymbol{x})} + \underbrace{\mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\left[\log\left(\frac{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}{p_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x})}\right)\right]}_{D_{\mathbb{KL}}(q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}) \| p_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x}))} \tag{21.13}$$

The second term in Equation (21.13) is non-negative, and hence

$$Ł_{\boldsymbol{\theta},\boldsymbol{\phi}}(\boldsymbol{x}) \leq \log p_{\boldsymbol{\theta}}(\boldsymbol{x}) \tag{21.14}$$

The quantity $\log p_{\boldsymbol{\theta}}(\boldsymbol{x})$ is the log marginal likelihood, also called the **evidence**. Hence $Ł_{\boldsymbol{\theta},\boldsymbol{\phi}}(\boldsymbol{x})$ is called the **evidence lower bound** or **ELBO**.

We can rewrite the ELBO in 3 equivalent ways as follows:

$$Ł_{\boldsymbol{\theta},\boldsymbol{\phi}}(\boldsymbol{x}) = \mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\left[\log p_{\boldsymbol{\theta}}(\boldsymbol{x},\boldsymbol{z}) - \log q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})\right] \tag{21.15}$$

$$= \mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\left[\log p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z}) + \log p_{\boldsymbol{\theta}}(\boldsymbol{z})\right] + \mathbb{H}(q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})) \tag{21.16}$$

$$= \mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\left[\log p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})\right] - D_{\mathbb{KL}}\left(q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}) \parallel p_{\boldsymbol{\theta}}(\boldsymbol{z})\right) \tag{21.17}$$

We can interpret this last objective as the expected log likelihood plus a regularization term, that ensures the (per-sample) posterior is "well behaved" (does not deviate too far from the prior in terms of KL divergence).

The tightness of this lower bound is controlled by the **variational gap**, which is given by $D_{\mathbb{KL}}\left(q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}) \parallel p_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x})\right)$. A better approximate posterior results in a tighter bound. When the KL goes to zero, the posterior is exact, so any improvements to the ELBO directly translate to improvements in the likelihood of the data, as in the EM algorithm (see Section 6.6.3).

An alternative to maximizing the ELBO is to minimize the **negative ELBO**, also called the **variational free energy**, given by

$$\mathcal{L}(\boldsymbol{\theta},\boldsymbol{\phi};\boldsymbol{x}) = \mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\left[-\log p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})\right] + D_{\mathbb{KL}}\left(q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}) \parallel p_{\boldsymbol{\theta}}(\boldsymbol{z})\right) \tag{21.18}$$

i.e., model-inference hybrid. See the blog by Jacob Andreas, http://blog.jacobandreas.net/monference.html, for further discussion.

21.2.3 Evaluating the ELBO

We can approximate the ELBO by sampling from the posterior, $\boldsymbol{z}_s \sim q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})$, and then evaluating the expectation term using a Monte Carlo estimate,

$$\mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\left[\log p_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{z})\right] \approx \frac{1}{S}\sum_{s=1}^{S} \log p_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{z}_s) \tag{21.19}$$

We can often compute the (differential) entropy term $\mathbb{H}(q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}))$ analytically, depending on the choice of variational distribution. For example, if we use a Gaussian posterior, $q(\boldsymbol{z}|\boldsymbol{x}) = \mathcal{N}(\boldsymbol{z}|\boldsymbol{\mu}, \boldsymbol{\Sigma})$, we can use Equation (5.95) to compute the entropy:

$$\mathbb{H}(q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})) = \frac{1}{2}\ln|\boldsymbol{\Sigma}| + \text{const} \tag{21.20}$$

Similarly we can often compute the KL term $D_{\mathbb{KL}}\left(q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}) \parallel p_{\boldsymbol{\theta}}(\boldsymbol{z})\right)$ analytically. For example, if we assume a diagonal Gaussian prior, $p(\boldsymbol{z}) = \mathcal{N}(\boldsymbol{z}|\boldsymbol{0}, \mathbf{I})$, and diagonal gaussian posterior, $q(\boldsymbol{z}|\boldsymbol{x}) = \mathcal{N}(\boldsymbol{z}|\boldsymbol{\mu}, \text{diag}(\boldsymbol{\sigma}))$, we can use Equation (5.81) to compute the KL in closed form:

$$D_{\mathbb{KL}}\left(q \parallel p\right) = -\frac{1}{2}\sum_{k=1}^{K}\left[\log\sigma_k^2 - \sigma_k^2 - \mu_k^2 + 1\right] \tag{21.21}$$

where K is the number of latent dimensions.

In cases where we cannot evaluate the entropy or KL in closed form, we can use just use Monte Carlo to approximate all terms:

$$Ł_{\boldsymbol{\theta},\boldsymbol{\phi}}(\boldsymbol{x}) \approx \frac{1}{S}\sum_{s=1}^{S}\left[\log p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z}_s) + \log p_{\boldsymbol{\theta}}(\boldsymbol{z}_s) - \log q_{\boldsymbol{\phi}}(\boldsymbol{z}_s|\boldsymbol{x})\right] \tag{21.22}$$

We often use a single MC sample, so $S = 1$.

21.2.4 Optimizing the ELBO

The ELBO for a single datapoint $\boldsymbol{x}$ is given in Equation (21.17). The ELBO for the whole dataset, scaled by $N = |\mathcal{D}|$, the number of examples, is given by

$$Ł_{\boldsymbol{\theta},\boldsymbol{\phi}}(\mathcal{D}) = \frac{1}{N}\sum_{\boldsymbol{x}_n\in\mathcal{D}} Ł_{\boldsymbol{\theta},\boldsymbol{\phi}}(\boldsymbol{x}_n) = \frac{1}{N}\sum_{\boldsymbol{x}_n\in\mathcal{D}}\left[\mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}_n)}\left[\log p_{\boldsymbol{\theta}}(\boldsymbol{x}_n|\boldsymbol{z}) + \log p_{\boldsymbol{\theta}}(\boldsymbol{z}) - \log q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}_n)\right]\right] \tag{21.23}$$

Our goal is to maximize this wrt $\boldsymbol{\theta}$ and $\boldsymbol{\phi}$: the former fits the model to the data, the latter reduces the KL gap between the approximate and true posterior.

We can create an unbiased minibatch approximation of this objective by sampling examples $\boldsymbol{x}$, and then computing the objective for a given $\boldsymbol{x}$. So now we focus on a fixed $\boldsymbol{x}$, for brevity, and drop the sum over $\boldsymbol{x}_n$.

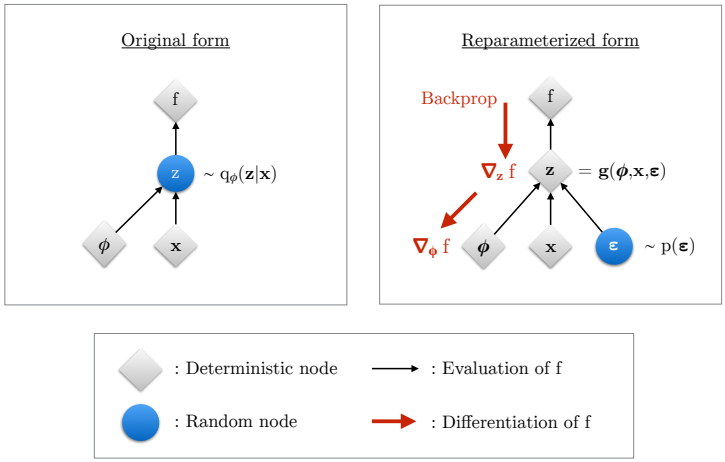


Figure 21.2: Illustration of the reparameterization trick. The objective f depends on the variational parameters $\boldsymbol{\phi}$, the observed data $\boldsymbol{x}$, and the latent random variable $\boldsymbol{z} \sim q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})$. On the left, we show the standard form of the computation graph. On the right, we show a reparameterized form, in which we move the stochasticity into the noise source $\boldsymbol{\epsilon}$, and compute $\boldsymbol{z}$ deterministically, $\boldsymbol{z} = g(\boldsymbol{\phi}, \boldsymbol{x}, \boldsymbol{\epsilon})$. The rest of the graph is deterministic, so we can backpropagate the gradient of the scalar f wrt $\boldsymbol{\phi}$ through $\boldsymbol{z}$ and into $\boldsymbol{\phi}$. From Figure 2.3 of [KW19a]. Used with kind permission of Durk Kingma.

The gradient wrt the generative parameters $\boldsymbol{\theta}$ is easy to compute, since we can push gradients inside the expectation, and use a single Monte Carlo sample:

$$\nabla_{\boldsymbol{\theta}} \text{Ł}_{\boldsymbol{\theta},\boldsymbol{\phi}}(\boldsymbol{x}) = \nabla_{\boldsymbol{\theta}} \mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})} \left[\log p_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{z}) - \log q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})\right] \tag{21.24}$$

$$= \mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})} \left[\nabla_{\boldsymbol{\theta}} \left\{\log p_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{z}) - \log q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})\right\}\right] \tag{21.25}$$

$$\approx \nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{z}^s) \tag{21.26}$$

where $\boldsymbol{z}^s \sim q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})$. This is an unbiased estimate of the gradient, so can be used with SGD.

The gradient wrt the inference parameters $\boldsymbol{\phi}$ is harder to compute since

$$\nabla_{\boldsymbol{\phi}} \text{Ł}_{\boldsymbol{\theta},\boldsymbol{\phi}}(\boldsymbol{x}) = \nabla_{\boldsymbol{\phi}} \mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})} \left[\log p_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{z}) - \log q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})\right] \tag{21.27}$$

$$\neq \mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})} \left[\nabla_{\boldsymbol{\phi}} \left\{\log p_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{z}) - \log q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})\right\}\right] \tag{21.28}$$

However, we can often use the reparameterization trick, which we discuss in Section 21.2.5. (If not, we can use black box VI, which we discuss in Section 10.3.2.)

21.2.5 Using the reparameterization trick to compute ELBO gradients

In this section, we discuss the **reparameterization trick** for taking gradients wrt distributions over continuous latent variables $\boldsymbol{z} \sim q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})$. We explain this in detail in Section 6.5.4, but we summarize the basic idea here.

The key trick is to rewrite the random variable $\boldsymbol{z} \sim q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})$ as some differentiable (and invertible) transformation r of another random variable $\boldsymbol{\epsilon} \sim p(\boldsymbol{\epsilon})$, which does not depend on $\boldsymbol{\phi}$, i.e., we assume we can write

$$\boldsymbol{z} = r(\boldsymbol{\epsilon}, \boldsymbol{\phi}, \boldsymbol{x}) \tag{21.29}$$

For example,

$$\boldsymbol{z} \sim \mathcal{N}(\boldsymbol{\mu}, \text{diag}(\boldsymbol{\sigma})) \iff \boldsymbol{z} = \boldsymbol{\mu} + \boldsymbol{\epsilon} \odot \boldsymbol{\sigma}, \;\; \boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}) \tag{21.30}$$

Using this, we we have

$$\mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\left[f(\boldsymbol{z})\right] = \mathbb{E}_{p(\boldsymbol{\epsilon})}\left[f(\boldsymbol{z})\right] \quad \text{s.t.} \quad \boldsymbol{z} = r(\boldsymbol{\epsilon}, \boldsymbol{\phi}, \boldsymbol{x}) \tag{21.31}$$

where we define

$$f(\boldsymbol{z}) = \log p_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{z}) - \log q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}) \tag{21.32}$$

Hence

$$\nabla_{\boldsymbol{\phi}} \mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\left[f(\boldsymbol{z})\right] = \nabla_{\boldsymbol{\phi}} \mathbb{E}_{p(\boldsymbol{\epsilon})}\left[f(\boldsymbol{z})\right] = \mathbb{E}_{p(\boldsymbol{\epsilon})}\left[\nabla_{\boldsymbol{\phi}} f(\boldsymbol{z})\right] \tag{21.33}$$

which we can approximate with a single Monte Carlo sample. This lets us propagate gradients back through the f function and then into the DNN transformation function r that is used to compute $\boldsymbol{z} = r(\boldsymbol{\epsilon}, \boldsymbol{\phi}, \boldsymbol{x})$. See Figure 21.2 for an illustration.

Since we are now working with the random variable $\boldsymbol{\epsilon}$, we need to use the change of variables formula to compute

$$\log q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}) = \log p(\boldsymbol{\epsilon}) - \log \left| \det \left(\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{\epsilon}} \right) \right| \tag{21.34}$$

where $\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{\epsilon}}$ is the Jacobian:

$$\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{\epsilon}} = \begin{pmatrix} \frac{\partial z_1}{\partial \epsilon_1} & \cdots & \frac{\partial z_1}{\partial \epsilon_k} \\ \vdots & \ddots & \vdots \\ \frac{\partial z_k}{\partial \epsilon_1} & \cdots & \frac{\partial z_k}{\partial \epsilon_k} \end{pmatrix} \tag{21.35}$$

We design the transformation $\boldsymbol{z} = r(\boldsymbol{\epsilon})$ such that this Jacobian is tractable to compute. We give some examples below.

21.2.5.1 Fully factorized Gaussian

Suppose we have a fully factorized Gaussian posterior:

$$\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}) \tag{21.36}$$

$$\boldsymbol{z} = \boldsymbol{\mu} + \boldsymbol{\sigma} \odot \boldsymbol{\epsilon} \tag{21.37}$$

$$(\boldsymbol{\mu}, \log \boldsymbol{\sigma}) = e_{\boldsymbol{\phi}}(\boldsymbol{x}) \tag{21.38}$$

Then the Jacobian is $\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{\epsilon}} = \text{diag}(\boldsymbol{\sigma})$, so

$$\log q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}) = \sum_{k=1}^{K} \log \mathcal{N}(\epsilon_k|0,1) - \log \sigma_k = \sum_{k=1}^{K} -\frac{1}{2}\log(2\pi) - \frac{1}{2}\epsilon_k^2 - \log \sigma_k \tag{21.39}$$

21.2.5.2 Full covariance Gaussian

Now consider a full covariance Gaussian posterior:

$$\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}) \tag{21.40}$$

$$\boldsymbol{z} = \boldsymbol{\mu} + \mathbf{L}\boldsymbol{\epsilon} \tag{21.41}$$

where $\mathbf{L}$ is a lower triangular matrix with non-zero entries on the diagonal, which satisfies $\boldsymbol{\Sigma} = \mathbf{L}\mathbf{L}^{\mathsf{T}}$. The Jacobian of this affine transformation is $\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{\epsilon}} = \mathbf{L}$. Since $\mathbf{L}$ is a triangular matrix, its determinant is the product of its main diagonal, so

$$\log \left| \det \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{\epsilon}} \right| = \sum_{k=1}^{K} \log |L_{kk}| \tag{21.42}$$

We need to make the parameters of the transformation r be a function of the inputs $\boldsymbol{x}$. One way to do this is to define

$$(\boldsymbol{\mu}, \log \boldsymbol{\sigma}, \mathbf{L}') = e_{\boldsymbol{\phi}}(\boldsymbol{x}) \tag{21.43}$$

$$\mathbf{L} = \mathbf{M} \odot \mathbf{L}' + \text{diag}(\boldsymbol{\sigma}) \tag{21.44}$$

where $\mathbf{M}$ is a masking matrix with 0s on and above the diagonal, and 1s below the diagonal. With this construction, the diagonal entries of $\mathbf{L}$ are given by $\boldsymbol{\sigma}$, so

$$\log \left| \det \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{\epsilon}} \right| = \sum_{k=1}^{K} \log |L_{kk}| = \sum_{k=1}^{K} \log \sigma_k \tag{21.45}$$

See Algorithm 36 for the corresponding pseudo code for computing the reparameterized ELBO.

Algorithm 36: Computing a single sample unbiased estimate of the reparameterized ELBO for a VAE with full covariance Gaussian posterior, and factorized Bernoulli likelihood. Based on Algorithm 2 of [KW19a].

$(\boldsymbol{\mu}, \log \boldsymbol{\sigma}, \mathbf{L}') = e_{\boldsymbol{\phi}}(\boldsymbol{x})$
$\mathbf{M} = \text{np.triu}(\text{np.ones}(K), -1)$
$\mathbf{L} = \mathbf{M} \odot \mathbf{L}' + \text{diag}(\boldsymbol{\sigma})$
$\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
$\boldsymbol{z} = \mathbf{L}\boldsymbol{\epsilon} + \boldsymbol{\mu}$
$\boldsymbol{p} = d_{\boldsymbol{\theta}}(\boldsymbol{z})$
$\mathcal{L}_{\text{logqz}} = -\sum_{k=1}^{K} \left[\frac{1}{2}\epsilon_k^2 + \frac{1}{2}\log(2\pi) + \log \sigma_k\right]$ // from $q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})$
$\mathcal{L}_{\text{logpz}} = -\sum_{k=1}^{K} \left[\frac{1}{2}z_k^2 + \frac{1}{2}\log(2\pi)\right]$ // from $p_{\boldsymbol{\theta}}(\boldsymbol{z})$
$\mathcal{L}_{\text{logpx}} = -\sum_{d=1}^{D} \left[x_d \log p_d + (1 - x_d)\log(1 - p_d)\right]$ // from $p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})$
$\mathcal{L} = \mathcal{L}_{\text{logpx}} + \mathcal{L}_{\text{logpz}} - \mathcal{L}_{\text{logqz}}$

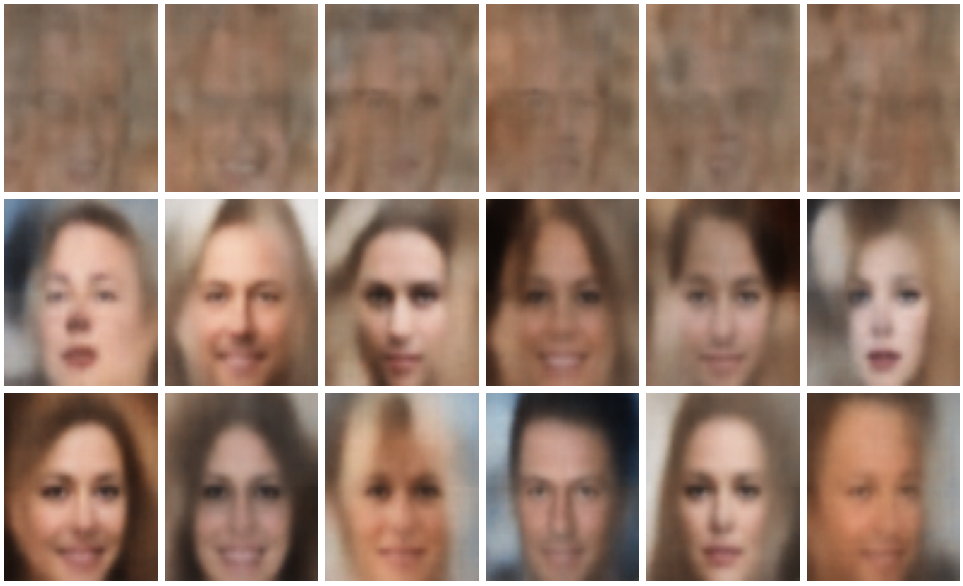


Figure 21.3: Illustration of unconditional image generation using (V)AEs trained on CelebA. Row 1: Deterministic autoencoder. Row 2: β-VAE with $\beta = 0.5$. Row 3: VAE (with $\beta = 1$). Generated by celeba_vae_ae_comparison.ipynb.

21.2.5.3 Inverse autoregressive flows

In Section 10.4.3, we discuss how to use inverse autoregressive flows to learn more expressive posteriors $q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})$, leveraging the tractability of the Jacobian of this nonlinear transformation.

21.2.6 Comparison of VAEs and autoencoders

VAEs are very similar to deterministic autoencoders (AE). There are 2 main differences: in the AE, the objective is the log likelihood of the reconstruction without any KL term; and in addition, the encoding is deterministic, so the encoder network just needs to compute $\mathbb{E}[\boldsymbol{z}|\boldsymbol{x}]$ and not $\mathbb{V}[\boldsymbol{z}|\boldsymbol{x}]$. In view of these similarities, one can use the same codebase to implement both methods. However, it is natural to wonder what the benefits and potential drawbacks of the VAE are compared to the deterministic AE.

We shall answer this question by fitting both models to the CelebA dataset. Both models have the same convolutional structure with the following number of hidden channels per convolutional layer in the encoder: (32, 64, 128, 256, 512). The spatial size of each layer is as follows: (32, 16, 8, 4, 2). The final $2 \times 2 \times 512$ convolutional layer then gets reshaped and passed through a linear layer to generate the mean and (marginal) variance of the stochastic latent vector, which has size 256. The structure of the decoder is the mirror image of the decoder. Each model is trained for 5 epochs with a batch size of 256, which takes about 20 minutes on a GPU.

The main advantage of a VAE over a deterministic autoencoder is that it defines a proper generative model, that can create sensible-looking novel images by decoding prior samples $\boldsymbol{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$. By

Figure 21.4: Illustration of image reconstruction using (V)AEs trained and applied to CelebA. Row 1: Original images. Row 2: Deterministic autoencoder. Row 3: β-VAE with $\beta = 0.5$. Row 4: VAE (with $\beta = 1$). Generated by celeba_vae_ae_comparison.ipynb.

contrast, an autoencoder only knows how to decode latent codes derived from the training set, so does poorly when fed random inputs. This is illustrated in Figure 21.3.

We can also use both models to reconstruct a given input image. In Figure 21.4, we see that both AE and VAE can reconstruct the input images reasonably well, although the VAE reconstructions are somewhat blurry, for reasons we discuss in Section 21.3.1. We can reduce the amount of blurriness by scaling down the KL penalty term by a factor of β; this is known as the β-VAE, and is discussed in more detail in Section 21.3.1.

21.2.7 VAEs optimize in an augmented space

In this section, we derive several alternative expressions for the ELBO which shed light on how VAEs work.

First, let us define the joint generative distribution

$$p_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{z}) = p_{\boldsymbol{\theta}}(\boldsymbol{z}) p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z}) \tag{21.46}$$

from which we can derive the generative data marginal

$$p_{\boldsymbol{\theta}}(\boldsymbol{x}) = \int_{\boldsymbol{z}} p_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{z}) d\boldsymbol{z} \tag{21.47}$$

and the generative posterior

$$p_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x}) = p_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{z}) / p_{\boldsymbol{\theta}}(\boldsymbol{x}) \tag{21.48}$$

Let us also define the joint *inference* distribution

$$q_{\mathcal{D},\boldsymbol{\phi}}(\boldsymbol{z}, \boldsymbol{x}) = p_{\mathcal{D}}(\boldsymbol{x}) q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}) \tag{21.49}$$

where

$$p_{\mathcal{D}}(\boldsymbol{x}) = \frac{1}{N} \sum_{n=1}^{N} \delta(\boldsymbol{x}_n - \boldsymbol{x}) \tag{21.50}$$

is the empirical distribution. From this we can derive the inference latent marginal, also called the **aggregated posterior**:

$$q_{\mathcal{D},\boldsymbol{\phi}}(\boldsymbol{z}) = \int_{\boldsymbol{x}} q_{\mathcal{D},\boldsymbol{\phi}}(\boldsymbol{x}, \boldsymbol{z}) d\boldsymbol{x} \tag{21.51}$$

and the inference likelihood

$$q_{\mathcal{D},\boldsymbol{\phi}}(\boldsymbol{x}|\boldsymbol{z}) = q_{\mathcal{D},\boldsymbol{\phi}}(\boldsymbol{x}, \boldsymbol{z}) / q_{\mathcal{D},\boldsymbol{\phi}}(\boldsymbol{z}) \tag{21.52}$$

See Figure 21.5 for a visual illustration.

Having defined our terms, we can now derive various alternative versions of the ELBO, following [ZSE19]. First note that the ELBO averaged over all the data is given by

$$\begin{aligned}
Ł_{\boldsymbol{\theta},\boldsymbol{\phi}} &= \mathbb{E}_{p_{\mathcal{D}}(\boldsymbol{x})} \left[\mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})} \left[\log p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z}) \right] \right] - \mathbb{E}_{p_{\mathcal{D}}(\boldsymbol{x})} \left[D_{\mathbb{KL}} \left(q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}) \parallel p_{\boldsymbol{\theta}}(\boldsymbol{z}) \right) \right] && (21.53) \\
&= \mathbb{E}_{q_{\mathcal{D},\boldsymbol{\phi}}(\boldsymbol{x},\boldsymbol{z})} \left[\log p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z}) + \log p_{\boldsymbol{\theta}}(\boldsymbol{z}) - \log q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}) \right] && (21.54) \\
&= \mathbb{E}_{q_{\mathcal{D},\boldsymbol{\phi}}(\boldsymbol{x},\boldsymbol{z})} \left[\log \frac{p_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{z})}{q_{\mathcal{D},\boldsymbol{\phi}}(\boldsymbol{x}, \boldsymbol{z})} + \log p_{\mathcal{D}}(\boldsymbol{x}) \right] && (21.55) \\
&= -D_{\mathbb{KL}} \left(q_{\mathcal{D},\boldsymbol{\phi}}(\boldsymbol{x}, \boldsymbol{z}) \parallel p_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{z}) \right) + \mathbb{E}_{p_{\mathcal{D}}(\boldsymbol{x})} \left[\log p_{\mathcal{D}}(\boldsymbol{x}) \right] && (21.56)
\end{aligned}$$

If we define $\stackrel{c}{=}$ to mean equal up to additive constants, we can rewrite the above as

$$\begin{aligned}
Ł_{\boldsymbol{\theta},\boldsymbol{\phi}} &\stackrel{c}{=} -D_{\mathbb{KL}} \left(q_{\boldsymbol{\phi}}(\boldsymbol{x}, \boldsymbol{z}) \parallel p_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{z}) \right) && (21.57) \\
&\stackrel{c}{=} -D_{\mathbb{KL}} \left(p_{\mathcal{D}}(\boldsymbol{x}) \parallel p_{\boldsymbol{\theta}}(\boldsymbol{x}) \right) - \mathbb{E}_{p_{\mathcal{D}}(\boldsymbol{x})} \left[D_{\mathbb{KL}} \left(q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}) \parallel p_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x}) \right) \right] && (21.58)
\end{aligned}$$

Thus maximizing the ELBO requires minimizing the two KL terms. The first KL term is minimized by MLE, and the second KL term is minimized by fitting the true posterior. Thus if the posterior family is limited, there may be a conflict between these objectives.

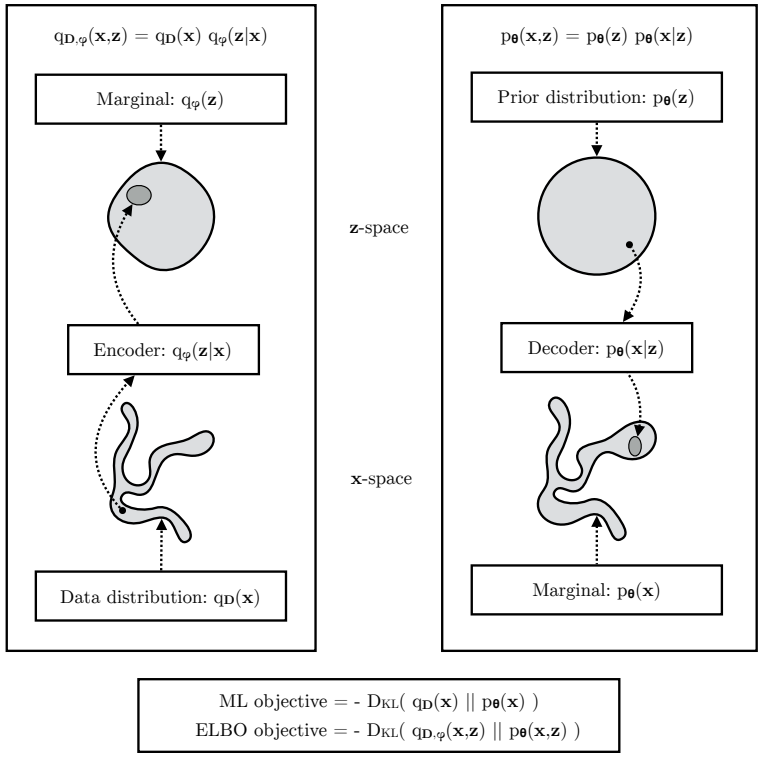


Figure 21.5: The maximum likelihood (ML) objective can be viewed as the minimization of $D_{\mathbb{KL}}\left(p_{\mathcal{D}}(\boldsymbol{x}) \parallel p_{\boldsymbol{\theta}}(\boldsymbol{x})\right)$. (Note: in the figure, $p_{\mathcal{D}}(\boldsymbol{x})$ is denoted by $q_D(\boldsymbol{x})$.) The ELBO objective is minimization of $D_{\mathbb{KL}}\left(q_{\mathcal{D},\boldsymbol{\phi}}(\boldsymbol{x},\boldsymbol{z}) \parallel p_{\boldsymbol{\theta}}(\boldsymbol{x},\boldsymbol{z})\right)$, which upper bounds $D_{\mathbb{KL}}\left(q_{\mathcal{D}}(\boldsymbol{x}) \parallel p_{\boldsymbol{\theta}}(\boldsymbol{x})\right)$. From Figure 2.4 of [KW19a]. Used with kind permission of Durk Kingma.

Finally, we note that the ELBO can also be written as

$$Ł_{\boldsymbol{\theta},\boldsymbol{\phi}} \stackrel{c}{=} -D_{\mathbb{KL}}\left(q_{\mathcal{D},\boldsymbol{\phi}}(\boldsymbol{z}) \parallel p_{\boldsymbol{\theta}}(\boldsymbol{z})\right) - \mathbb{E}_{q_{\mathcal{D},\boldsymbol{\phi}}(\boldsymbol{z})}\left[D_{\mathbb{KL}}\left(q_{\boldsymbol{\phi}}(\boldsymbol{x}|\boldsymbol{z}) \parallel p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})\right)\right] \tag{21.59}$$

We see from Equation (21.59) that VAEs are trying to minimize the difference between the inference marginal and generative prior, $D_{\mathbb{KL}}\left(q_{\boldsymbol{\phi}}(\boldsymbol{z}) \parallel p_{\boldsymbol{\theta}}(\boldsymbol{z})\right)$, while simultaneously minimizing reconstruction error, $D_{\mathbb{KL}}\left(q_{\boldsymbol{\phi}}(\boldsymbol{x}|\boldsymbol{z}) \parallel p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})\right)$ Since $\boldsymbol{x}$ is typically of much higher dimensionality than $\boldsymbol{z}$, the latter term usually dominates. Consequently, if there is a conflict between these two objectives (e.g., due to limited modeling power), the VAE will favor reconstruction accuracy over posterior inference. Thus the learned posterior may not be a very good approximation to the true posterior (see [ZSE19] for further discussion).

21.3 VAE generalizations

In this section, we discuss some variants of the basic VAE model.

21.3.1 β-VAE

It is often the case that VAEs generate somewhat blurry images, as illustrated in Figure 21.4, Figure 21.3 and Figure 20.9. This is not the case for models that optimize the exact likelihood, such as pixelCNNs (Section 22.3.2) and flow models (Chapter 23). To see why VAEs are different, consider the common case where the decoder is a Gaussian with fixed variance, so

$$\log p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z}) = -\frac{1}{2\sigma^2}||\boldsymbol{x} - d_{\boldsymbol{\theta}}(\boldsymbol{z})||_2^2 + \text{const} \tag{21.60}$$

Let $e_{\boldsymbol{\phi}}(\boldsymbol{x}) = \mathbb{E}\left[q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})\right]$ be the encoding of $\boldsymbol{x}$, and $\mathcal{X}(\boldsymbol{z}) = \{\boldsymbol{x} : e_{\boldsymbol{\phi}}(\boldsymbol{x}) = \boldsymbol{z}\}$ be the set of inputs that get mapped to $\boldsymbol{z}$. For a fixed inference network, the optimal setting of the generator parameters, when using squared reconstruction loss, is to ensure $d_{\boldsymbol{\theta}}(\boldsymbol{z}) = \mathbb{E}\left[\boldsymbol{x} : \boldsymbol{x} \in \mathcal{X}(\boldsymbol{z})\right]$. Thus the decoder should predict the average of all inputs $\boldsymbol{x}$ that map to that $\boldsymbol{z}$, resulting in blurry images.

We can solve this problem by increasing the expressive power of the posterior approximation (avoiding the merging of distinct inputs into the same latent code), or of the generator (by adding back information that is missing from the latent code), or both. However, an even simpler solution is to reduce the penalty on the KL term, making the model closer to a deterministic autoencoder:

$$\mathcal{L}_\beta(\boldsymbol{\theta}, \boldsymbol{\phi}|\boldsymbol{x}) = \underbrace{-\mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\left[\log p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})\right]}_{\mathcal{L}_E} + \beta \underbrace{D_{\mathbb{KL}}\left(q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}) \parallel p_{\boldsymbol{\theta}}(\boldsymbol{z})\right)}_{\mathcal{L}_R} \tag{21.61}$$

where $\mathcal{L}_E$ is the reconstruction error (negative log likelihood), and $\mathcal{L}_R$ is the KL regularizer. This is called the β-**VAE** objective [Hig+17a]. If we set $\beta = 1$, we recover the objective used in standard VAEs; if we set $\beta = 0$, we recover the objective used in standard autoencoders.

By varying β from 0 to infinity, we can reach different points on the **rate distortion curve**, as discussed in Section 5.4.2. These points make different tradeoffs between reconstruction error (distortion) and how much information is stored in the latents about the input (rate of the corresponding code). By using $\beta < 1$, we store more bits about each input, and hence can reconstruct images in a less blurry way. If we use $\beta > 1$, we get a more compressed representation.

21.3.1.1 Disentangled representations

One advantage of using $\beta > 1$ is that it encourages the learning of a latent representation that is "**disentangled**". Intuitively this means that each latent dimension represents a different **factor of variation** in the input. This is often formalized in terms of the total correlation (Section 5.3.5.1), which is defined as follows:

$$\text{TC}(\boldsymbol{z}) = \sum_k \mathbb{H}\left(z_k\right) - \mathbb{H}\left(\boldsymbol{z}\right) = D_{\mathbb{KL}}\left(p(\boldsymbol{z}) \parallel \prod_k p_k(z_k)\right) \tag{21.62}$$

This is zero iff the components of $\boldsymbol{z}$ are all mutually independent, and hence disentangled. In [AS18], they prove that using $\beta > 1$ will decrease the TC.

Unfortunately, in [Loc+18] they prove that nonlinear latent variable models are unidentifiable, and therefore for any disentangled representation, there is an equivalent fully entangled representation with exactly the same likelihood. Thus it is not possible to recover the correct latent representation without choosing the appropriate inductive bias, via the encoder, decoder, prior, dataset, or learning algorithm, i.e., merely adjusting β is not sufficient. See Section 32.4.1 for more discussion.

21.3.1.2 Connection with information bottleneck

In this section, we show that the β-VAE is an unsupervised version of the information bottleneck (IB) objective from Section 5.6. If the input is $\boldsymbol{x}$, the hidden bottleneck is $\boldsymbol{z}$, and the target outputs are $\tilde{\boldsymbol{x}}$, then the unsupervised IB objective becomes

$$\mathcal{L}_{\text{UIB}} = \beta\, \mathbb{I}(\boldsymbol{z};\boldsymbol{x}) - \mathbb{I}(\boldsymbol{z};\tilde{\boldsymbol{x}}) \tag{21.63}$$

$$= \beta \mathbb{E}_{p(\boldsymbol{x},\boldsymbol{z})}\left[\log \frac{p(\boldsymbol{x},\boldsymbol{z})}{p(\boldsymbol{x})p(\boldsymbol{z})}\right] - \mathbb{E}_{p(\boldsymbol{z},\tilde{\boldsymbol{x}})}\left[\log \frac{p(\boldsymbol{z},\tilde{\boldsymbol{x}})}{p(\boldsymbol{z})p(\tilde{\boldsymbol{x}})}\right] \tag{21.64}$$

where

$$p(\boldsymbol{x},\boldsymbol{z}) = p_{\mathcal{D}}(\boldsymbol{x})p(\boldsymbol{z}|\boldsymbol{x}) \tag{21.65}$$

$$p(\boldsymbol{z},\tilde{\boldsymbol{x}}) = \int p_{\mathcal{D}}(\boldsymbol{x})p(\boldsymbol{z}|\boldsymbol{x})p(\tilde{\boldsymbol{x}}|\boldsymbol{z})d\boldsymbol{x} \tag{21.66}$$

Intuitively, the objective in Equation (21.63) means we should pick a representation $\boldsymbol{z}$ that can predict $\tilde{\boldsymbol{x}}$ reliably, while not memorizing too much information about the input $\boldsymbol{x}$. The tradeoff parameter is controlled by β.

From Equation (5.180), we have the following variational upper bound on this unsupervised objective:

$$\mathcal{L}_{\text{UVIB}} = -\mathbb{E}_{q_{\mathcal{D},\boldsymbol{\phi}}(\boldsymbol{z},\boldsymbol{x})}\left[\log p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})\right] + \beta \mathbb{E}_{p_{\mathcal{D}}(\boldsymbol{x})}\left[D_{\mathbb{KL}}\left(q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}) \parallel p_{\boldsymbol{\theta}}(\boldsymbol{z})\right)\right] \tag{21.67}$$

which matches Equation (21.61) when averaged over $\boldsymbol{x}$.

21.3.2 InfoVAE

In Section 21.2.7, we discussed some drawbacks of the standard ELBO objective for training VAEs, namely the tendency to ignore the latent code when the decoder is powerful (Section 21.4), and the tendency to learn a poor posterior approximation due to the mismatch between the KL terms in data space and latent space (Section 21.2.7). We can fix these problems to some degree by using a generalized objective of the following form:

$$Ł(\boldsymbol{\theta},\boldsymbol{\phi}|\boldsymbol{x}) = -\lambda D_{\mathbb{KL}}\left(q_{\boldsymbol{\phi}}(\boldsymbol{z}) \parallel p_{\boldsymbol{\theta}}(\boldsymbol{z})\right) - \mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z})}\left[D_{\mathbb{KL}}\left(q_{\boldsymbol{\phi}}(\boldsymbol{x}|\boldsymbol{z}) \parallel p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})\right)\right] + \alpha\, \mathbb{I}_q(\boldsymbol{x};\boldsymbol{z}) \tag{21.68}$$

where $\alpha \geq 0$ controls how much we weight the mutual information $\mathbb{I}_q(\boldsymbol{x};\boldsymbol{z})$ between $\boldsymbol{x}$ and $\boldsymbol{z}$, and $\lambda \geq 0$ controls the tradeoff between $\boldsymbol{z}$-space KL and $\boldsymbol{x}$-space KL. This is called the **InfoVAE** objective [ZSE19]. If we set $\alpha = 0$ and $\lambda = 1$, we recover the standard ELBO, as shown in Equation (21.59).

Unfortunately, the objective in Equation (21.68) cannot be computed as written, because of the intractable MI term:

$$\mathbb{I}_q(\boldsymbol{x};\boldsymbol{z}) = \mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{x},\boldsymbol{z})}\left[\log \frac{q_{\boldsymbol{\phi}}(\boldsymbol{x},\boldsymbol{z})}{q_{\boldsymbol{\phi}}(\boldsymbol{x})q_{\boldsymbol{\phi}}(\boldsymbol{z})}\right] = -\mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{x},\boldsymbol{z})}\left[\log \frac{q_{\boldsymbol{\phi}}(\boldsymbol{z})}{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\right] \tag{21.69}$$

However, using the fact that $q_{\boldsymbol{\phi}}(\boldsymbol{x}|\boldsymbol{z}) = p_{\mathcal{D}}(\boldsymbol{x})q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})/q_{\boldsymbol{\phi}}(\boldsymbol{z})$, we can rewrite the objective as follows:

$$\text{Ł} = \mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{x},\boldsymbol{z})}\left[-\lambda \log \frac{q_{\boldsymbol{\phi}}(\boldsymbol{z})}{p_{\boldsymbol{\theta}}(\boldsymbol{z})} - \log \frac{q_{\boldsymbol{\phi}}(\boldsymbol{x}|\boldsymbol{z})}{p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})} - \alpha \log \frac{q_{\boldsymbol{\phi}}(\boldsymbol{z})}{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\right] \tag{21.70}$$

$$= \mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{x},\boldsymbol{z})}\left[\log p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z}) - \log \frac{q_{\boldsymbol{\phi}}(\boldsymbol{z})^{\lambda+\alpha-1} p_{\mathcal{D}}(\boldsymbol{x})}{p_{\boldsymbol{\theta}}(\boldsymbol{z})^{\lambda} q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})^{\alpha-1}}\right] \tag{21.71}$$

$$= \mathbb{E}_{p_{\mathcal{D}}(\boldsymbol{x})}\left[\mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\left[\log p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})\right]\right] - (1-\alpha)\mathbb{E}_{p_{\mathcal{D}}(\boldsymbol{x})}\left[D_{\mathbb{KL}}\left(q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}) \parallel p_{\boldsymbol{\theta}}(\boldsymbol{z})\right)\right]$$
$$- (\alpha+\lambda-1)D_{\mathbb{KL}}\left(q_{\boldsymbol{\phi}}(\boldsymbol{z}) \parallel p_{\boldsymbol{\theta}}(\boldsymbol{z})\right) - \mathbb{E}_{p_{\mathcal{D}}(\boldsymbol{x})}\left[\log p_{\mathcal{D}}(\boldsymbol{x})\right] \tag{21.72}$$

where the last term is a constant we can ignore. The first two terms can be optimized using the reparameterization trick. Unfortunately, the last term requires computing $q_{\boldsymbol{\phi}}(\boldsymbol{z}) = \int_{\boldsymbol{x}} q_{\boldsymbol{\phi}}(\boldsymbol{x},\boldsymbol{z})d\boldsymbol{x}$, which is intractable. Fortunately, we can easily sample from this distribution, by sampling $\boldsymbol{x} \sim p_{\mathcal{D}}(\boldsymbol{x})$ and $\boldsymbol{z} \sim q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})$. Thus $q_{\boldsymbol{\phi}}(\boldsymbol{z})$ is an **implicit probability model**, similar to a GAN (see Chapter 26).

As long as we use a strict divergence, meaning $D(q,p) = 0$ iff $q = p$, then one can show that this does not affect the optimality of the procedure. In particular, proposition 2 of [ZSE19] tells us the following:

Theorem 1. *Let $\mathcal{X}$ and $\mathcal{Z}$ be continuous spaces, and $\alpha < 1$ (to bound the MI) and $\lambda > 0$. For any fixed value of $\mathbb{I}_q(\boldsymbol{x};\boldsymbol{z})$, the approximate InfoVAE loss, with any strict divergence $D(q_{\boldsymbol{\phi}}(\boldsymbol{z}), p_{\boldsymbol{\theta}}(\boldsymbol{z}))$, is globally optimized if $p_{\boldsymbol{\theta}}(\boldsymbol{x}) = p_{\mathcal{D}}(\boldsymbol{x})$ and $q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}) = p_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x})$.*

21.3.2.1 Connection with MMD VAE

If we set $\alpha = 1$, the InfoVAE objective simplifies to

$$\text{Ł} \stackrel{c}{=} \mathbb{E}_{p_{\mathcal{D}}(\boldsymbol{x})}\left[\mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\left[\log p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})\right]\right] - \lambda D_{\mathbb{KL}}\left(q_{\boldsymbol{\phi}}(\boldsymbol{z}) \parallel p_{\boldsymbol{\theta}}(\boldsymbol{z})\right) \tag{21.73}$$

The **MMD VAE**[3] replaces the KL divergence in the above term with the (squared) maximum mean discrepancy or **MMD** divergence defined in Section 2.7.3. (This is valid based on the above theorem.) The advantage of this approach over standard InfoVAE is that the resulting objective is tractable. In particular, if we set $\lambda = 1$ and swap the sign we get

$$\mathcal{L} = \mathbb{E}_{p_{\mathcal{D}}(\boldsymbol{x})}\left[\mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\left[-\log p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})\right]\right] + \text{MMD}(q_{\boldsymbol{\phi}}(\boldsymbol{z}), p_{\boldsymbol{\theta}}(\boldsymbol{z})) \tag{21.74}$$

As we discuss in Section 2.7.3, we can compute the MMD as follows:

$$\text{MMD}(p,q) = \mathbb{E}_{p(\boldsymbol{z}),p(\boldsymbol{z}')}\left[\mathcal{K}(\boldsymbol{z},\boldsymbol{z}')\right] + \mathbb{E}_{q(\boldsymbol{z}),q(\boldsymbol{z}')}\left[\mathcal{K}(\boldsymbol{z},\boldsymbol{z}')\right] - 2\mathbb{E}_{p(\boldsymbol{z}),q(\boldsymbol{z}')}\left[\mathcal{K}(\boldsymbol{z},\boldsymbol{z}')\right] \tag{21.75}$$

where $\mathcal{K}()$ is some kernel function, such as the RBF kernel, $\mathcal{K}(\boldsymbol{z},\boldsymbol{z}') = \exp(-\frac{1}{2\sigma^2}||\boldsymbol{z}-\boldsymbol{z}'||_2^2)$. Intuitively the MMD measures the similarity (in latent space) between samples from the prior and samples from the aggregated posterior.

In practice, we can implement the MMD objective by using the posterior predicted mean $\boldsymbol{z}_n = e_{\boldsymbol{\phi}}(\boldsymbol{x}_n)$ for all B samples in the current minibatch, and comparing this to B random samples from the $\mathcal{N}(\mathbf{0},\mathbf{I})$ prior.

3. Proposed in https://ermongroup.github.io/blog/a-tutorial-on-mmd-variational-autoencoders/.

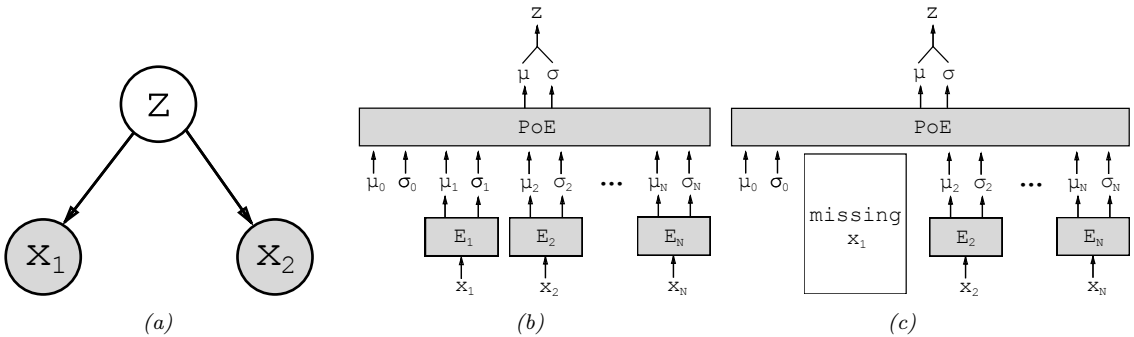


Figure 21.6: Illustration of multi-modal VAE. (a) The generative model with $N = 2$ modalities. (b) The product of experts (PoE) inference network is derived from N individual Gaussian experts E_i. μ_0 and σ_0 are parameters of the prior. (c) If a modality is missing, we omit its contribution to the posterior. From Figure 1 of [WG18]. Used with kind permission of Mike Wu.

If we use a Gaussian decoder with fixed variance, the negative log likelihood is just a squared error term:

$$-\log p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z}) = ||\boldsymbol{x} - d_{\boldsymbol{\theta}}(\boldsymbol{z})||_2^2 \tag{21.76}$$

Thus the entire model is deterministic, and just predicts the means in latent space and visible space.

21.3.2.2 Connection with β-VAEs

If we set $\alpha = 0$ and $\lambda = 1$, we get back the original ELBO. If $\lambda > 0$ is freely chosen, but we use $\alpha = 1 - \lambda$, we get the β-VAE.

21.3.2.3 Connection with adversarial autoencoders

If we set $\alpha = 1$ and $\lambda = 1$, and D is chosen to be the Jensen Shannon divergence (which can be minimized by training a binary discriminator, as explained in Section 26.2.2), then we get a model known as an **adversarial autoencoder** [Mak+15a].

21.3.3 Multi-modal VAEs

It is possible to extend VAEs to create joint distributions over different kinds of variables, such as images and text. This is sometimes called a **multimodal VAE** or **MVAE**. Let us assume there are M modalities. We assume they are conditionally independent given the latent code, and hence the generative model has the form

$$p_{\boldsymbol{\theta}}(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_M, \boldsymbol{z}) = p(\boldsymbol{z}) \prod_{m=1}^{M} p_{\boldsymbol{\theta}}(\boldsymbol{x}_m|\boldsymbol{z}) \tag{21.77}$$

where we treat $p(\boldsymbol{z})$ as a fixed prior. See Figure 21.6(a) for an illustration.

The standard ELBO is given by

$$\text{Ł}_{\boldsymbol{\theta},\boldsymbol{\phi}}(\mathbf{X}) = \mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\mathbf{X})}\left[\sum_m \log p_{\boldsymbol{\theta}}(\boldsymbol{x}_m|\boldsymbol{z})\right] - D_{\mathbb{KL}}\left(q_{\boldsymbol{\phi}}(\boldsymbol{z}|\mathbf{X}) \parallel p(\boldsymbol{z})\right) \tag{21.78}$$

where $\mathbf{X} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_M)$ is the observed data. However, the different likelihood terms $p(\boldsymbol{x}_m|\boldsymbol{z})$ may have different dynamic ranges (e.g., Gaussian pdf for pixels, and categorical pmf for text), so we introduce weight terms $\lambda_m \geq 0$ for each likelihood. In addition, let $\beta \geq 0$ control the amount of KL regularization. This gives us a weighted version of the ELBO, as follows:

$$\text{Ł}_{\boldsymbol{\theta},\boldsymbol{\phi}}(\mathbf{X}) = \mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\mathbf{X})}\left[\sum_m \lambda_m \log p_{\boldsymbol{\theta}}(\boldsymbol{x}_m|\boldsymbol{z})\right] - \beta D_{\mathbb{KL}}\left(q_{\boldsymbol{\phi}}(\boldsymbol{z}|\mathbf{X}) \parallel p(\boldsymbol{z})\right) \tag{21.79}$$

Often we don't have a lot of paired (aligned) data from all M modalities. For example, we may have a lot of images (modality 1), and a lot of text (modality 2), but very few (image, text) pairs. So it is useful to generalize the loss so it fits the marginal distributions of subsets of the features. Let $O_m = 1$ if modality m is observed (i.e., $\boldsymbol{x}_m$ is known), and let $O_m = 0$ if it is missing or unobserved. Let $\mathbf{X} = \{\boldsymbol{x}_m : O_m = 1\}$ be the visible features. We now use the following objective:

$$\text{Ł}_{\boldsymbol{\theta},\boldsymbol{\phi}}(\mathbf{X}) = \mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\mathbf{X})}\left[\sum_{m:O_m=1} \lambda_m \log p_{\boldsymbol{\theta}}(\boldsymbol{x}_m|\boldsymbol{z})\right] - \beta D_{\mathbb{KL}}\left(q_{\boldsymbol{\phi}}(\boldsymbol{z}|\mathbf{X}) \parallel p(\boldsymbol{z})\right) \tag{21.80}$$

The key problem is how to compute the posterior $q_{\boldsymbol{\phi}}(\boldsymbol{z}|\mathbf{X})$ given different subsets of features. In general this can be hard, since the inference network is a discriminative model that assumes all inputs are available. For example, if it is trained on (image, text) pairs, $q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}_1, \boldsymbol{x}_2)$, how can we compute the posterior just given an image, $q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}_1)$, or just given text, $q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}_2)$? (This issue arises in general with VAE when we have missing inputs; we discuss the general case in Section 21.3.4.)

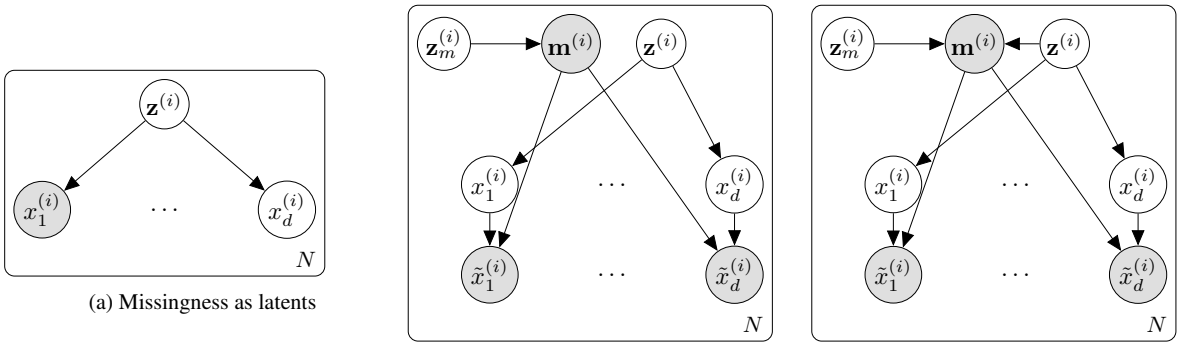
Fortunately, based on our conditional independence assumption between the modalities, we can compute the optimal form for $q_{\boldsymbol{\phi}}(\boldsymbol{z}|\mathbf{X})$ given set of inputs by computing the exact posterior under the model, which is given by

$$p(\boldsymbol{z}|\mathbf{X}) = \frac{p(\boldsymbol{z})p(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_M|\boldsymbol{z})}{p(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_M)} = \frac{p(\boldsymbol{z})}{p(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_M)} \prod_{m=1}^{M} p(\boldsymbol{x}_m|\boldsymbol{z}) \tag{21.81}$$

$$= \frac{p(\boldsymbol{z})}{p(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_M)} \prod_{m=1}^{M} \frac{p(\boldsymbol{z}|\boldsymbol{x}_m)p(\boldsymbol{x}_m)}{p(\boldsymbol{z})} \tag{21.82}$$

$$\propto p(\boldsymbol{z}) \prod_{m=1}^{M} \frac{p(\boldsymbol{z}|\boldsymbol{x}_m)}{p(\boldsymbol{z})} \approx p(\boldsymbol{z}) \prod_{m=1}^{M} \tilde{q}(\boldsymbol{z}|\boldsymbol{x}_m) \tag{21.83}$$

This can be viewed as a product of experts (Section 24.1.1), where each $\tilde{q}(\boldsymbol{z}|\boldsymbol{x}_m)$ is an "expert" for the m'th modality, and $p(\boldsymbol{z})$ is the prior. We can compute the above posterior for any subset of modalities for which we have data by modifying the product over m. If we use Gaussian distributions for the prior $p(\boldsymbol{z}) = \mathcal{N}(\boldsymbol{z}|\boldsymbol{\mu}_0, \boldsymbol{\Lambda}_0^{-1})$ and marginal posterior ratio $\tilde{q}(\boldsymbol{z}|\boldsymbol{x}_m) = \mathcal{N}(\boldsymbol{z}|\boldsymbol{\mu}_m, \boldsymbol{\Lambda}_m^{-1})$, then we



(a) Missingness as latents

(b) MCAR corruption process

(c) MNAR corruption process

Figure 21.7: Illustration of different VAE variants for handling missing data. From Figure 1 of [CNW20]. Used with kind permission of Mark Collier.

can compute the product of Gaussians using the result from Equation (2.115):

$$\prod_{m=0}^{M} \mathcal{N}(\boldsymbol{z}|\boldsymbol{\mu}_m, \boldsymbol{\Lambda}_m^{-1}) \propto \mathcal{N}(\boldsymbol{z}|\boldsymbol{\mu}, \boldsymbol{\Sigma}), \quad \boldsymbol{\Sigma} = (\sum_m \boldsymbol{\Lambda}_m)^{-1}, \quad \boldsymbol{\mu} = \boldsymbol{\Sigma}(\sum_m \boldsymbol{\Lambda}_m \boldsymbol{\mu}_m) \tag{21.84}$$

Thus the overall posterior precision is the sum of individual expert posterior precisions, and the overall posterior mean is the precision weighted average of the individual expert posterior means. See Figure 21.6(b) for an illustration. For a linear Gaussian (factor analysis) model, we can ensure $q(\boldsymbol{z}|\boldsymbol{x}_m) = p(\boldsymbol{z}|\boldsymbol{x}_m)$, in which case the above solution is the exact posterior [WN18], but in general it will be an approximation.

We need to train the individual expert recognition models $q(\boldsymbol{z}|\boldsymbol{x}_m)$ as well as the joint model $q(\boldsymbol{z}|\mathbf{X})$, so the model knows what to do with fully observed as well as partially observed inputs at test time. In [Ved+18], they propose a somewhat complex "triple ELBO" objective. In [WG18], they propose the simpler approach of optimizing the ELBO for the fully observed feature vector, all the marginals, and a set of $\mathcal{J}$ randomly chosen joint modalities:

$$Ł_{\boldsymbol{\theta},\boldsymbol{\phi}}(\mathbf{X}) = Ł_{\boldsymbol{\theta},\boldsymbol{\phi}}(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_M) + \sum_{m=1}^{M} Ł_{\boldsymbol{\theta},\boldsymbol{\phi}}(\boldsymbol{x}_m) + \sum_{j \in \mathcal{J}} Ł_{\boldsymbol{\theta},\boldsymbol{\phi}}(\mathbf{X}_j) \tag{21.85}$$

This generalizes nicely to the semi-supervised setting, in which we only have a few aligned ("labeled") examples from the joint, but have many unaligned ("unlabeled") examples from the individual marginals. See Figure 21.6(c) for an illustration.

Note that the above scheme can only handle the case of a fixed number of missingness patterns; we generalize to allow for arbitrary missingess in Section 21.3.4.

21.3.4 VAEs with missing data

Sometimes we may have **missing data**, in which parts of the data vector $\boldsymbol{x} \in \mathbb{R}^D$ may be unknown. In Section 21.3.3 we saw a special case of this when we discussed multimodal VAEs. In this section

we allow for arbitrary patterns of missingness.

To model the missing data, let $\boldsymbol{m} \in \{0,1\}^D$ be a binary vector where $m_j = 1$ if x_j is missing, and $m_j = 0$ otherwise. Let $\mathbf{X} = \{\boldsymbol{x}^{(n)}\}$ and $\mathbf{M} = \{\boldsymbol{m}^{(n)}\}$ be $N \times D$ matrices. Furthermore, let $\mathbf{X}_o$ be the observed parts of $\mathbf{X}$ and $\mathbf{X}_h$ be the hidden parts. If we assume $p(\mathbf{M}|\mathbf{X}_o, \mathbf{X}_h) = p(\mathbf{M})$, we say the data is **missing completely at random** or **MCAR**, since the missingness does not depend on the hidden or observed features. If we assume $p(\mathbf{M}|\mathbf{X}_o, \mathbf{X}_h) = p(\mathbf{M}|\mathbf{X}_o)$, we say the data is **missing at random** or **MAR**, since the missingness does not depend on the hidden features, but may depend on the visible features. If neither of these assumptions hold, we say the data is **not missing at random** or **NMAR**.

In the MCAR and MAR cases, we can ignore the missingness mechanism, since it tells us nothing about the hidden features. However, in the NMAR case, we need to model the **missing data mechanism**, since the lack of information may be informative. For example, the fact that someone did not fill out an answer to a sensitive question on a survey (e.g., "Do you have COVID?") could be informative about the underlying value. See e.g., [LR87; Mar08] for more information on missing data models.

In the context of VAEs, we can model the MCAR scenario by treating the missing values as latent variables. This is illustrated in Figure 21.7(a). Since missing leaf nodes in a directed graphical model do not affect their parents, we can simply ignore them when computing the posterior $p(\boldsymbol{z}^{(i)}|\boldsymbol{x}_o^{(i)})$, where $\boldsymbol{x}_o^{(i)}$ are the observed parts of example i. However, when using an amortized inference network, it can be difficult to handle missing inputs, since the model is usually trained to compute $p(\boldsymbol{z}^{(i)}|\boldsymbol{x}_{1:d}^{(i)})$. One solution to this is to use the product of experts approach discussed in the context of multi-modal VAEs in Section 21.3.3. However, this is designed for the case where whole blocks (corresponding to different modalities) are missing, and will not work well if there are arbitrary missing patterns (e.g., pixels that get dropped out due to occlusion or scratches on the lens). In addition, this method will not work for the NMAR case.

An alternative approach, proposed in [CNW20], is to explicitly include the missingness indicators into the model, as shown in Figure 21.7(b). We assume the model always generates each $\boldsymbol{x}_j$ for $j = 1 : d$, but we only get to see the "corrupted" versions $\tilde{\boldsymbol{x}}_j$. If $m_j = 0$ then $\tilde{\boldsymbol{x}}_j = \boldsymbol{x}_j$, but if $m_j = 1$, then $\tilde{\boldsymbol{x}}_j$ is a special value, such as 0, unrelated to $\boldsymbol{x}_j$. We can model any correlation between the missingness elements (components of $\boldsymbol{m}$) by using another latent variable $\boldsymbol{z}_m$. This model can easily be extended to the NMAR case by letting $\boldsymbol{m}$ depend on the latent factors for the observed data, $\boldsymbol{z}$, as well as the usual missingess latent factors $\boldsymbol{z}_m$, as shown in Figure 21.7(c).

We modify the VAE to be conditional on the missingness pattern, so the VAE decoder has the form $p(\boldsymbol{x}_o|\boldsymbol{z}, \boldsymbol{m})$, and the encoder has the form $q(\boldsymbol{z}|\boldsymbol{x}_o, \boldsymbol{m})$. However, we assume the prior is $p(\boldsymbol{z})$ as usual, independent of $\boldsymbol{m}$. We can compute a lower bound on the log marginal likelihood of the

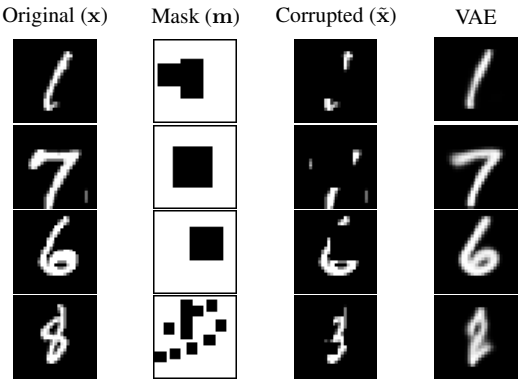


Figure 21.8: Imputing missing pixels given a masked out image using a VAE using a MCAR assumption. From Figure 2 of [CNW20]. Used with kind permission of Mark Collier.

observed data, given the missingness, as follows:

$$\log p(\boldsymbol{x}_o|\boldsymbol{m}) = \log \int\int p(\boldsymbol{x}_o, \boldsymbol{x}_m|\boldsymbol{z}, \boldsymbol{m}) p(\boldsymbol{z}) d\boldsymbol{x}_m d\boldsymbol{z} \tag{21.86}$$

$$= \log \int p(\boldsymbol{x}_o|\boldsymbol{z}, \boldsymbol{m}) p(\boldsymbol{z}) d\boldsymbol{z} \tag{21.87}$$

$$= \log \int p(\boldsymbol{x}_o|\boldsymbol{z}, \boldsymbol{m}) p(\boldsymbol{z}) \frac{q(\boldsymbol{z}|\tilde{\boldsymbol{x}}, \boldsymbol{m})}{q(\boldsymbol{z}|\tilde{\boldsymbol{x}}, \boldsymbol{m})} d\boldsymbol{z} \tag{21.88}$$

$$= \log \mathbb{E}_{q(\boldsymbol{z}|\tilde{\boldsymbol{x}}, \boldsymbol{m})} \left[p(\boldsymbol{x}_o|\boldsymbol{z}, \boldsymbol{m}) \frac{p(\boldsymbol{z})}{q(\boldsymbol{z}|\tilde{\boldsymbol{x}}, \boldsymbol{m})} \right] \tag{21.89}$$

$$\geq \mathbb{E}_{q(\boldsymbol{z}|\tilde{\boldsymbol{x}}, \boldsymbol{m})} \left[\log p(\boldsymbol{x}_o|\boldsymbol{z}, \boldsymbol{m}) \right] - D_{\mathbb{KL}} \left(q(\boldsymbol{z}|\tilde{\boldsymbol{x}}, \boldsymbol{m}) \parallel p(\boldsymbol{z}) \right) \tag{21.90}$$

We can fit this model in the usual way. See Figure 21.8 for an example.

21.3.5 Semi-supervised VAEs

In this section, we discuss how to extend VAEs to the **semi-supervised learning** setting in which we have both labeled data, $\mathcal{D}_L = \{(\boldsymbol{x}_n, y_n)\}$, and unlabeled data, $\mathcal{D}_U = \{(\boldsymbol{x}_n)\}$. We focus on the **M2** model, proposed in [Kin+14a].

The generative model has the following form:

$$p_{\boldsymbol{\theta}}(\boldsymbol{x}, y) = p_{\boldsymbol{\theta}}(y) p_{\boldsymbol{\theta}}(\boldsymbol{x}|y) = p_{\boldsymbol{\theta}}(y) \int p_{\boldsymbol{\theta}}(\boldsymbol{x}|y, \boldsymbol{z}) p_{\boldsymbol{\theta}}(\boldsymbol{z}) d\boldsymbol{z} \tag{21.91}$$

where $\boldsymbol{z}$ is a latent variable, $p_{\boldsymbol{\theta}}(\boldsymbol{z}) = \mathcal{N}(\boldsymbol{z}|\mathbf{0}, \mathbf{I})$ is the latent prior, $p_{\boldsymbol{\theta}}(y) = \text{Cat}(y|\boldsymbol{\pi})$ the label prior, and $p_{\boldsymbol{\theta}}(\boldsymbol{x}|y, \boldsymbol{z}) = p(\boldsymbol{x}|f_{\boldsymbol{\theta}}(y, \boldsymbol{z}))$ is the likelihood, such as a Gaussian, with parameters computed by f (a deep neural network). The main innovation of this approach is to assume that data is generated according to both a latent class variable y as well as the continuous latent variable $\boldsymbol{z}$. The class variable y is observed for labeled data and unobserved for unlabled data.

To compute the likelihood for the *labeled data*, $p_{\boldsymbol{\theta}}(\boldsymbol{x}, y)$, we need to marginalize over $\boldsymbol{z}$, which we can do by using an inference network of the form

$$q_{\boldsymbol{\phi}}(\boldsymbol{z}|y, \boldsymbol{x}) = \mathcal{N}(\boldsymbol{z}|\boldsymbol{\mu}_{\boldsymbol{\phi}}(y, \boldsymbol{x}), \text{diag}(\boldsymbol{\sigma}_{\boldsymbol{\phi}}(y, \boldsymbol{x})) \tag{21.92}$$

We then use the following variational lower bound

$$\log p_{\boldsymbol{\theta}}(\boldsymbol{x}, y) \geq \mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x},y)} \left[\log p_{\boldsymbol{\theta}}(\boldsymbol{x}|y, \boldsymbol{z}) + \log p_{\boldsymbol{\theta}}(y) + \log p_{\boldsymbol{\theta}}(\boldsymbol{z}) - \log q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}, y)\right] = -\mathcal{L}(\boldsymbol{x}, y) \tag{21.93}$$

as is standard for VAEs (see Section 21.2). The only difference is that we observe two kinds of data: $\boldsymbol{x}$ and y.

To compute the likelihood for the *unlabeled* data, $p_{\boldsymbol{\theta}}(\boldsymbol{x})$, we need to marginalize over $\boldsymbol{z}$ *and* y, which we can do by using an inference network of the form

$$q_{\boldsymbol{\phi}}(\boldsymbol{z}, y|\boldsymbol{x}) = q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}) q_{\boldsymbol{\phi}}(y|\boldsymbol{x}) \tag{21.94}$$

$$q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}) = \mathcal{N}(\boldsymbol{z}|\boldsymbol{\mu}_{\boldsymbol{\phi}}(\boldsymbol{x}), \text{diag}(\boldsymbol{\sigma}_{\boldsymbol{\phi}}(\boldsymbol{x})) \tag{21.95}$$

$$q_{\boldsymbol{\phi}}(y|\boldsymbol{x}) = \text{Cat}(y|\boldsymbol{\pi}_{\boldsymbol{\phi}}(\boldsymbol{x})) \tag{21.96}$$

Note that $q_{\boldsymbol{\phi}}(y|\boldsymbol{x})$ acts like a discriminative classifier, that imputes the missing labels. We then use the following variational lower bound:

$$\log p_{\boldsymbol{\theta}}(\boldsymbol{x}) \geq \mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z},y|\boldsymbol{x})} \left[\log p_{\boldsymbol{\theta}}(\boldsymbol{x}|y, \boldsymbol{z}) + \log p_{\boldsymbol{\theta}}(y) + \log p_{\boldsymbol{\theta}}(\boldsymbol{z}) - \log q_{\boldsymbol{\phi}}(\boldsymbol{z}, y|\boldsymbol{x})\right] \tag{21.97}$$

$$= -\sum_y q_{\boldsymbol{\phi}}(y|\boldsymbol{x}) \mathcal{L}(\boldsymbol{x}, y) + \mathbb{H}\left(q_{\boldsymbol{\phi}}(y|\boldsymbol{x})\right) = -\mathcal{U}(\boldsymbol{x}) \tag{21.98}$$

Note that the discriminative classifier $q_{\boldsymbol{\phi}}(y|\boldsymbol{x})$ is only used to compute the log-likelihood of the unlabeled data, which is undesirable. We can therefore add an extra classification loss on the supervised data, to get the following overall objective function:

$$\mathcal{L}(\boldsymbol{\theta}) = \mathbb{E}_{(\boldsymbol{x},y)\sim\mathcal{D}_L} \left[\mathcal{L}(\boldsymbol{x}, y)\right] + \mathbb{E}_{\boldsymbol{x}\sim\mathcal{D}_U} \left[\mathcal{U}(\boldsymbol{x})\right] + \alpha \mathbb{E}_{(\boldsymbol{x},y)\sim\mathcal{D}_L} \left[-\log q_{\boldsymbol{\phi}}(y|\boldsymbol{x})\right] \tag{21.99}$$

where $\mathcal{D}_L$ is the labeled data, $\mathcal{D}_U$ is the unlabeled data, and α is a hyperparameter that controls the relative weight of generative and discriminative learning.

21.3.6 VAEs with sequential encoders/decoders

In this section, we discuss VAEs for sequential data, such as text and biosequences, in which the data $\boldsymbol{x}$ is a variable-length sequence, but we have a fixed-sized latent variable $\boldsymbol{z} \in \mathbb{R}^K$. (We consider the more general case in which $\boldsymbol{z}$ is a variable-length sequence of latents — known as **sequential VAE** or **dynamic VAE** — in Section 29.13.) All we have to do is modify the decoder $p(\boldsymbol{x}|\boldsymbol{z})$ and encoder $q(\boldsymbol{z}|\boldsymbol{x})$ to work with sequences.

21.3.6.1 Models

If we use an RNN for the encoder and decoder of a VAE, we get a model which is called a **VAE-RNN**, as proposed in [Bow+16a]. In more detail, the generative model is $p(\boldsymbol{z}, \boldsymbol{x}_{1:T}) = p(\boldsymbol{z})\text{RNN}(\boldsymbol{x}_{1:T}|\boldsymbol{z})$, where $\boldsymbol{z}$ can be injected as the initial state of the RNN, or as an input to every time step. The

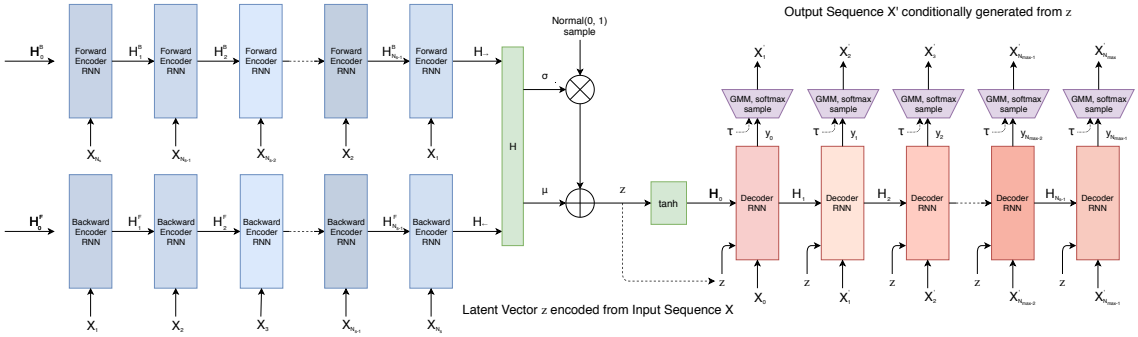


Figure 21.9: Illustration of a VAE with a bidirectional RNN encoder and a unidirectional RNN decoder. The output generator can use a GMM and/or softmax distribution. From Figure 2 of [HE18]. Used with kind permission of David Ha.

inference model is $q(\boldsymbol{z}|\boldsymbol{x}_{1:T}) = \mathcal{N}(\boldsymbol{z}|\boldsymbol{\mu}(\boldsymbol{h}), \boldsymbol{\Sigma}(\boldsymbol{h}))$, where $\boldsymbol{h} = [\boldsymbol{h}_T^{\rightarrow}, \boldsymbol{h}_1^{\leftarrow}]$ is the output of a bidirectional RNN applied to $\boldsymbol{x}_{1:T}$. See Figure 21.9 for an illustration.

More recently, people have tried to combine transformers with VAEs. For example, in the **Optimus** model of [Li+20], they use a BERT model for the encoder. In more detail, the encoder $q(\boldsymbol{z}|\boldsymbol{x})$ is derived from the embedding vector associated with a dummy token corresponding to the "class label" which is appended to the input sequence $\boldsymbol{x}$. The decoder is a standard autoregressive model (similar to GPT), with one additional input, namely the latent vector $\boldsymbol{z}$. They consider two ways of injecting the latent vector. The simplest approach is to add $\boldsymbol{z}$ to the embedding layer of every token in the decoding step, by defining $\boldsymbol{h}'_i = \boldsymbol{h}_i + \mathbf{W}\boldsymbol{z}$, where $\boldsymbol{h}_i \in \mathbb{R}^H$ is the original embedding for the i'th token, and $\mathbf{W} \in \mathbb{R}^{H \times K}$ is a decoding matrix, where K is the size of the latent vector. However, they get better results in their experiments by letting all the layers of the decoder attend to the latent code $\boldsymbol{z}$. An easy way to do this is to define the memory vector $\boldsymbol{h}_m = \mathbf{W}\boldsymbol{z}$, where $\mathbf{W} \in \mathbb{R}^{LH \times K}$, where L is the number of layers in the decoder, and then to append $\boldsymbol{h}_m \in \mathbb{R}^{L \times H}$ to all the other embeddings at each layer.

An alternative approach, known as **transformer VAE**, was proposed in [Gre20]. This model uses a **funnel transformer** [Dai+20b] as the encoder, and the **T5** [Raf+20a] conditional transformer for the decoder. In addition, it uses an MMD VAE (Section 21.3.2.1) to avoid posterior collapse.

21.3.6.2 Applications

In this section, we discuss some applications of VAEs to sequence data.

Text

In [Bow+16b], they apply the VAE-RNN model to natural language sentences. (See also [MB16; SSB17] for related work.) Although this does does not improve performance in terms of the standard perplexity measures (predicting the next word given the previous words), it does provide a way to infer a semantic representation of the sentence. This can then be used for latent space interpolation, as discussed in Section 20.3.5. The results of doing this with the VAE-RNN are illustrated in

```
he was silent for a long moment .
he was silent for a moment .
it was quiet for a moment .
it was dark and cold .
there was a pause .
it was my turn .
```

(a)

```
i went to the store to buy some groceries .
i store to buy some groceries .
i were to buy any groceries .
horses are to buy any groceries .
horses are to buy any animal .
horses the favorite any animal .
horses the favorite favorite animal .
horses are my favorite animal .
```

(b)

Figure 21.10: (a) Samples from the latent space of a VAE text model, as we interpolate between two sentences (on first and last line). Note that the intermediate sentences are grammatical, and semantically related to their neighbors. From Table 8 of [Bow+16b]. (b) Same as (a), but now using a deterministic autoencoder (with the same RNN encoder and decoder). From Table 1 of [Bow+16b]. Used with kind permission of Sam Bowman.

Figure 21.10a. (Similar results are shown in [Li+20], using a VAE-transformer.) By contrast, if we use a standard deterministic autoencoder, with the same RNN encoder and decoder networks, we learn a much less meaningful space, as illustrated in Figure 21.10b. The reason is that the deterministic autoencoder has "holes" in its latent space, which get decoded to nonsensical outputs.

However, because RNNs (and transformers) are powerful decoders, we need to address the problem of posterior collapse, which we discuss in Section 21.4. One common way to avoid this problem is to use KL annealing, but a more effective method is to use the InfoVAE method of Section 21.3.2, which includes adversarial autoencoders (used in [She+20] with an RNN decoder) and MMD autoencoders (used in [Gre20] with a transformer decoder).

Sketches

In [HE18], they apply the VAE-RNN model to generate sketches (line drawings) of various animals and hand-written characters. They call their model **sketch-rnn**. The training data records the sequence of (x, y) pen positions, as well as whether the pen was touching the paper or not. The emission model used a GMM for the real-valued location offsets, and a categorical softmax distribution for the discrete state.

Figure 21.11 shows some samples from various class-conditional models. We vary the temperature parameter τ of the emission model to control the stochasticity of the generator. (More precisely, we multiply the GMM variances by τ, and divide the discrete probabilities by τ before renormalizing.) When the temperature is low, the model tries to reconstruct the input as closely as possible. However, when the input is untypical of the training set (e.g., a cat with three eyes, or a toothbrush), the reconstruction is "regularized" towards a canonical cat with two eyes, while still keeping some features of the input.

Molecular design

In [GB+18], they use VAE-RNNs to model molecular graph structure, represented as a string using the SMILES representation.[4] It is also possible to learn a mapping from the latent space to some

4. See https://en.wikipedia.org/wiki/Simplified_molecular-input_line-entry_system.