Notes on Probabilistic Machine Learning

> Probability theory is nothing but common sense reduced to calculations.
>
> — Pierre Laplace, 1812

Contents

I Mathematical Foundations **4**

1 Multivariate Probability Models **4**
1.1 Measures the dependency . . . 4
1.2 The multivariate Gaussian/Normal distribution (MVN) . . . 5
1.3 Marginals and conditionals of an MVN . . . 5
1.4 Linear Gaussian systems and Bayes rule for Gaussians . . . 6

2 Statistics and Estimation **8**
2.1 Maximum Likelihood Estimation . . . 8
2.2 Empirical Risk Minimization . . . 9
2.3 Regularization . . . 9
2.4 Bayesian Statistics . . . 9
2.5 Prior selection . . . 11
2.6 Frequentist Statistics . . . 11

3 Matrix Calculus **14**
3.1 Functions that map scalars to scalars . . . 14
3.2 Functions that map vectors to scalars . . . 14
3.3 Functions that map vectors to vectors . . . 15
3.4 Functions that map matrices to scalars . . . 15

4 Optimization **17**
4.1 Constrained optimization . . . 17
4.1.1 Lagrange multipliers: equality constraints . . . 17
4.1.2 The KKT conditions . . . 18
4.2 Bound optimization . . . 19
4.2.1 Q for concave functions . . . 19
4.2.2 EM algorithm . . . 20

5 Mixture Models **22**
5.1 Gaussian mixture models . . . 22
5.2 EM for a GMM . . . 22
5.3 Bernoulli mixture model . . . 23

II Linear Models and Neural Networks **24**

6 Generalized Linear Models **24**
6.1 Logistic regression . . . 25
6.2 Linear regression . . . 25
6.3 Binomial regression . . . 25
6.4 Poisson regression . . . 26
6.5 Canonical and non-canonical link functions . . . 27
6.6 Maximum likelihood estimation . . . 27

7 Deep Neural Networks **29**
7.1 Multilayer perceptrons . . . 29
7.2 Backpropagation . . . 30
7.2.1 Backpropagation for MLPs . . . 30
7.2.2 Reverse mode differentiation and VJPs with learnable parameters . . . 31
7.2.3 Computation graphs: beyond MLPs . . . 34
7.3 Neural network training . . . 34
7.3.1 Stochastic gradient descent and learning rate tuning . . . 34
7.3.2 Vanishing and exploding gradients . . . 36
7.3.3 Non-saturating activations . . . 36
7.3.4 Residual connections . . . 37
7.3.5 Parameter initialization . . . 38
7.4 Regularization . . . 39

8 Recurrent Neural Networks **41**
8.1 Sequence generation: Vec2Seq . . . 41
8.2 Sequence classification: Seq2Vec . . . 42
8.3 Sequence translation: Seq2Seq . . . 43
8.4 Teacher Forcing . . . 44
8.5 Backpropagation Through Time (BPTT) . . . 45
8.6 Vanishing and Exploding Gradients . . . 46
8.7 RNN Generation . . . 47

III Dimentionality Reduction **48**

9 Principal Component Analysis **48**
9.1 PCA setup . . . 48
9.2 Algorithm derivation . . . 49
9.3 Computational tricks . . . 50

10 Factor Analysis **52**
10.1 Prerequisite: evidence lower bound (ELBO) . . . 52
10.2 EM algorithm for FA . . . 53
10.3 EM algorithm for PPCA . . . 54
10.4 Limiting case: EM algorithm for PCA . . . 54
10.5 Special case: Probabilistic PCA . . . 55
10.6 Summary . . . 57

11 Autoencoders **58**
11.1 Deterministic autoencoders 58
11.2 Variational autoencoders (VAE) 60

IV Kernel Methods **64**

12 Mercer Kernels and Mercer's Theorem **64**
12.1 Mercer's theorem 64
12.2 Mercer kernels examples 65
12.3 Combining kernels 65

13 Gaussian Processes **66**
13.1 Noise-free observations 66
13.2 Noisy observations 67
13.3 Bayesian linear regression 67
13.4 Estimating the kernel 68
13.5 GP for classifications 69
13.6 GP and deep learning 70
13.7 GP for dimensionality reduction 70

Part I
Mathematical Foundations

This part is not aiming to provide a general coverage of all the mathematical foundations, but rather to provide a quick reference for the mathematical concepts that are frequently used and worth noting.

1 Multivariate Probability Models

1.1 Measures the dependency

Covariance

The covariance between two rv's X and Y measures the degree to which X and Y are linearly related.

$$\mathrm{Cov}(X,Y) \triangleq \mathbb{E}[(X-\mathbb{E}[X])(Y-\mathbb{E}[Y])] = \mathbb{E}[XY]-\mathbb{E}[X]\mathbb{E}[Y]$$

If $\boldsymbol{x} \in \mathbb{R}^D$ is a random vector, its covariance matrix is defined to be the following symmetric, positive semi-definite matrix:

$$\begin{aligned}\mathrm{Cov}[\boldsymbol{x}] &\triangleq \mathbb{E}\left[(\boldsymbol{x}-\mathbb{E}[\boldsymbol{x}])(\boldsymbol{x}-\mathbb{E}[\boldsymbol{x}])^{\mathsf{T}}\right] \triangleq \boldsymbol{\Sigma} = \boldsymbol{K}_{xx} \\ &= \begin{pmatrix} \mathbb{V}[X_1] & \mathrm{Cov}[X_1,X_2] & \cdots & \mathrm{Cov}[X_1,X_D] \\ \mathrm{Cov}[X_2,X_1] & \mathbb{V}[X_2] & \cdots & \mathrm{Cov}[X_2,X_D] \\ \vdots & \vdots & \ddots & \vdots \\ \mathrm{Cov}[X_D,X_1] & \mathrm{Cov}[X_D,X_2] & \cdots & \mathbb{V}[X_D] \end{pmatrix}\end{aligned}$$

expanding $\mathbb{E}\left[(\boldsymbol{x}-\mathbb{E}[\boldsymbol{x}])(\boldsymbol{x}-\mathbb{E}[\boldsymbol{x}])^{\mathsf{T}}\right]$ gives the important result

$$\boldsymbol{R}_{xx} = \mathbb{E}[\boldsymbol{x}\boldsymbol{x}^{\mathsf{T}}] = \boldsymbol{\Sigma} + \boldsymbol{\mu}\boldsymbol{\mu}^{\mathsf{T}}$$

Under linear transformations, the covariance matrix transforms as follows:

$$\mathrm{Cov}[\mathbf{A}\boldsymbol{x}+\boldsymbol{b}] = \mathbf{A}\mathrm{Cov}[\boldsymbol{x}]\mathbf{A}^{\mathsf{T}}$$

The cross-covariance between two random vectors

$$\mathrm{Cov}[\boldsymbol{x},\boldsymbol{y}] = \mathbb{E}[(\boldsymbol{x}-\mathbb{E}[\boldsymbol{x}])(\boldsymbol{y}-\mathbb{E}[\boldsymbol{y}])^{\mathsf{T}}] = \mathrm{Cov}[\boldsymbol{x},\boldsymbol{y}]^{\top}$$

where $\mathrm{Cov}[\boldsymbol{x},\boldsymbol{y}] \in \mathbb{R}^{\mathrm{Dim}(\boldsymbol{x})\times\mathrm{Dim}(\boldsymbol{y})}$ and the (i,j)-th entry is $\mathrm{Cov}[\boldsymbol{x},\boldsymbol{y}]_{ij} = \mathrm{Cov}[\boldsymbol{x}_i,\boldsymbol{y}_j]$

Correlation

The correlation coefficient between X and Y is defined as

$$\rho \triangleq \mathrm{corr}[X,Y] = \frac{\mathrm{Cov}[X,Y]}{\sqrt{\mathbb{V}[X]\mathbb{V}[Y]}}$$

which is a normalised measure with $-1 \le \rho \le 1$. This can be thought as a measure of *degree of linearity*. (i.e. $\mathrm{corr}[X,Y]=1$ if and only if $Y=aX+b$ (and $a>0$) for some parameters a and b)

Correlation matrix of a vector $\boldsymbol{x}$ of D related random variables is defined as

$$\text{corr}(\boldsymbol{x}) = \begin{pmatrix} 1 & \rho_{12} & \cdots & \rho_{1D} \\ \rho_{21} & 1 & \cdots & \rho_{2D} \\ \vdots & \vdots & \ddots & \vdots \\ \rho_{D1} & \rho_{D2} & \cdots & 1 \end{pmatrix}$$

or compactly as,

$$\text{corr}(\boldsymbol{x}) = \text{diag}(\boldsymbol{K}_{xx})^{-\frac{1}{2}} \boldsymbol{K}_{xx} \text{diag}(\boldsymbol{K}_{xx})^{-\frac{1}{2}}$$

where $\boldsymbol{K}_{xx}$ is the (auto)covariance matrix of $\boldsymbol{x}$,

$$\begin{aligned} \boldsymbol{K}_{xx} = \boldsymbol{\Sigma} &= \mathbb{E}\left[(\boldsymbol{x} - \mathbb{E}[\boldsymbol{x}])(\boldsymbol{x} - \mathbb{E}[\boldsymbol{x}])^{\mathsf{T}}\right] \\ &= \mathbb{E}[\boldsymbol{x}\boldsymbol{x}^{\mathsf{T}}] - \boldsymbol{\mu}\boldsymbol{\mu}^{\mathsf{T}} = \boldsymbol{R}_{xx} - \boldsymbol{\mu}\boldsymbol{\mu}^{\mathsf{T}} \end{aligned}$$

where $\boldsymbol{R}_{xx}$ is the (auto)correlation matrix of $\boldsymbol{x}$.

Uncorrelated does not imply independent

Independent implies uncorrelated,

$$p(X, Y) = p(X)p(Y) \implies \text{Cov}[X, Y] = 0 \implies \text{corr}[X, Y] = 0$$

However, the converse is not true: uncorrelated does not imply independent (eg. $X \sim U(-1, 1), Y = X^2$ have zero correlation but are not independent). A more general measure of dependence between random variables is mutual information. This is zero only if the variables truly are independent.

1.2 The multivariate Gaussian/Normal distribution (MVN)

The MVN density is defined by the following:

$$\mathcal{N}(\boldsymbol{y}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) \triangleq \frac{1}{(2\pi)^{D/2}|\boldsymbol{\Sigma}|^{1/2}} \exp\left[-\frac{1}{2}(\boldsymbol{y} - \boldsymbol{\mu})^{\mathsf{T}}\boldsymbol{\Sigma}^{-1}(\boldsymbol{y} - \boldsymbol{\mu})\right]$$

where $\boldsymbol{\mu} = \mathbb{E}[\boldsymbol{y}] \in \mathbb{R}^D$ is the mean vector, and $\boldsymbol{\Sigma} = \text{Cov}[\boldsymbol{y}]$ is the $D \times D$ covariance matrix.

1.3 Marginals and conditionals of an MVN

For a Gaussian random vector $\boldsymbol{y} = (\boldsymbol{y}_1, \boldsymbol{y}_2)$, if we are conditioned on a set of noise-free observations $\boldsymbol{y}_2$, then we can infer the hidden parts $\boldsymbol{y}_1$ by the posterior $p(\boldsymbol{y}_1|\boldsymbol{y}_2)$.

Suppose $\boldsymbol{y} = (\boldsymbol{y}_1, \boldsymbol{y}_2)$ is jointly Gaussian with parameters

$$\boldsymbol{\mu} = \begin{pmatrix} \boldsymbol{\mu}_1 \\ \boldsymbol{\mu}_2 \end{pmatrix}, \quad \boldsymbol{\Sigma} = \begin{pmatrix} \boldsymbol{\Sigma}_{11} & \boldsymbol{\Sigma}_{12} \\ \boldsymbol{\Sigma}_{21} & \boldsymbol{\Sigma}_{22} \end{pmatrix}, \quad \boldsymbol{\Lambda} = \boldsymbol{\Sigma}^{-1} = \begin{pmatrix} \boldsymbol{\Lambda}_{11} & \boldsymbol{\Lambda}_{12} \\ \boldsymbol{\Lambda}_{21} & \boldsymbol{\Lambda}_{22} \end{pmatrix}$$

where $\boldsymbol{\Lambda}$ is the **precision matrix** and $\boldsymbol{\Sigma}_{12} = \boldsymbol{\Sigma}_{21}^{\top} = \text{Cov}[\boldsymbol{y}_1, \boldsymbol{y}_2]$.

Then the marginals are given by

$$\boxed{\begin{aligned} p(\boldsymbol{y}_1) &= \mathcal{N}(\boldsymbol{y}_1|\boldsymbol{\mu}_1, \boldsymbol{\Sigma}_{11}) \\ p(\boldsymbol{y}_2) &= \mathcal{N}(\boldsymbol{y}_2|\boldsymbol{\mu}_2, \boldsymbol{\Sigma}_{22}) \end{aligned}}$$

and the posterior conditional is given by

$$\boxed{\begin{aligned} p(\boldsymbol{y}_1|\boldsymbol{y}_2) &= \mathcal{N}(\boldsymbol{y}_1|\boldsymbol{\mu}_{1|2}, \boldsymbol{\Sigma}_{1|2}) \\ \boldsymbol{\mu}_{1|2} &= \boldsymbol{\mu}_1 + \boldsymbol{\Sigma}_{12}\boldsymbol{\Sigma}_{22}^{-1}(\boldsymbol{y}_2 - \boldsymbol{\mu}_2) \\ &= \boldsymbol{\mu}_1 - \boldsymbol{\Lambda}_{11}^{-1}\boldsymbol{\Lambda}_{12}(\boldsymbol{y}_2 - \boldsymbol{\mu}_2) \\ &= \boldsymbol{\Sigma}_{1|2}\left(\boldsymbol{\Lambda}_{11}\boldsymbol{\mu}_1 - \boldsymbol{\Lambda}_{12}(\boldsymbol{y}_2 - \boldsymbol{\mu}_2)\right) \\ \boldsymbol{\Sigma}_{1|2} &= \boldsymbol{\Sigma}_{11} - \boldsymbol{\Sigma}_{12}\boldsymbol{\Sigma}_{22}^{-1}\boldsymbol{\Sigma}_{21} = \boldsymbol{\Lambda}_{11}^{-1} \end{aligned}}$$

The proofs rely on computing the Schur complement $\boldsymbol{\Sigma}/\boldsymbol{\Sigma}_{22} = \boldsymbol{\Sigma}_{11} - \boldsymbol{\Sigma}_{12}\boldsymbol{\Sigma}_{22}^{-1}\boldsymbol{\Sigma}_{21}$, which is not shown here. For the marginals, just extracting the rows and columns corresponding to $\boldsymbol{y}_1$ or $\boldsymbol{y}_2$. For the conditional: the conditional mean is a linear function of $\boldsymbol{y}_2$, and the conditional covariance is a constant matrix that is independent of $\boldsymbol{y}_2$.

1.4 Linear Gaussian systems and Bayes rule for Gaussians

Let $\boldsymbol{z} \in \mathbb{R}^L$ be an unknown vector of values, and $\boldsymbol{y} \in \mathbb{R}^D$ be some noisy measurement of $\boldsymbol{z}$. We assume these variables are related by the following joint distribution:

$$p(\boldsymbol{z}) = \mathcal{N}(\boldsymbol{z}|\boldsymbol{\mu}_z, \boldsymbol{\Sigma}_z)$$

$$p(\boldsymbol{y}|\boldsymbol{z}) = \mathcal{N}(\boldsymbol{y}|\mathbf{W}\boldsymbol{z} + \boldsymbol{b}, \boldsymbol{\Sigma}_y)$$

where $\mathbf{W}$ is a matrix of size $D \times L$. This is an example of a **linear Gaussian system**.

The corresponding joint distribution, $p(\boldsymbol{z}, \boldsymbol{y}) = p(\boldsymbol{z})p(\boldsymbol{y}|\boldsymbol{z})$, is a $L + D$ dimensional Gaussian, with mean and covariance given by

$$\boldsymbol{\mu} = \begin{pmatrix} \boldsymbol{\mu}_z \\ \mathbf{W}\boldsymbol{\mu}_z + \boldsymbol{b} \end{pmatrix}$$

$$\boldsymbol{\Sigma} = \begin{pmatrix} \boldsymbol{\Sigma}_z & \boldsymbol{\Sigma}_z\mathbf{W}^\mathsf{T} \\ \mathbf{W}\boldsymbol{\Sigma}_z & \boldsymbol{\Sigma}_y + \mathbf{W}\boldsymbol{\Sigma}_z\mathbf{W}^\mathsf{T} \end{pmatrix}$$

By applying the previous Gaussian conditioning formula to the joint $p(\boldsymbol{y}, \boldsymbol{z})$ we can compute the posterior $p(\boldsymbol{z}|\boldsymbol{y})$ as

$$\begin{aligned} p(\boldsymbol{z}|\boldsymbol{y}) &= \mathcal{N}(\boldsymbol{z}|\boldsymbol{\mu}_{z|y}, \boldsymbol{\Sigma}_{z|y}) \\ \boldsymbol{\mu}_{z|y} &= \boldsymbol{\mu}_z + \boldsymbol{\Sigma}_z\mathbf{W}^\mathsf{T}(\boldsymbol{\Sigma}_y + \mathbf{W}\boldsymbol{\Sigma}_z\mathbf{W}^\mathsf{T})^{-1}(\mathbf{y} - \mathbf{W}\boldsymbol{\mu}_z - \mathbf{b}) \\ \boldsymbol{\Sigma}_{z|y} &= \boldsymbol{\Sigma}_z - \boldsymbol{\Sigma}_z\mathbf{W}^\mathsf{T}(\boldsymbol{\Sigma}_y + \mathbf{W}\boldsymbol{\Sigma}_z\mathbf{W}^\mathsf{T})^{-1}\mathbf{W}\boldsymbol{\Sigma}_z \end{aligned}$$

However, this requires inverting a $D \times D$ matrix $(\boldsymbol{\Sigma}_y + \mathbf{W}\boldsymbol{\Sigma}_z\mathbf{W}^\mathsf{T})$, which can be expensive if D is large. Instead, we can use the matrix inversion lemma (Woodbury matrix identity) to rewrite the posterior in a more efficient form:

$$\boxed{\begin{aligned} p(\boldsymbol{z}|\boldsymbol{y}) &= \mathcal{N}(\boldsymbol{z}|\boldsymbol{\mu}_{z|y}, \boldsymbol{\Sigma}_{z|y}) \\ \boldsymbol{\Sigma}_{z|y}^{-1} &= \boldsymbol{\Sigma}_z^{-1} + \mathbf{W}^\mathsf{T}\boldsymbol{\Sigma}_y^{-1}\mathbf{W} \\ \boldsymbol{\mu}_{z|y} &= \boldsymbol{\Sigma}_{z|y}\left[\mathbf{W}^\mathsf{T}\boldsymbol{\Sigma}_y^{-1}(\boldsymbol{y} - \boldsymbol{b}) + \boldsymbol{\Sigma}_z^{-1}\boldsymbol{\mu}_z\right] \end{aligned}}$$

This is known as **Bayes rule for Gaussians**. Furthermore, the normalization constant of the posterior is given by

$$p(\boldsymbol{y}) = \int \mathcal{N}(\boldsymbol{z}|\boldsymbol{\mu}_z, \boldsymbol{\Sigma}_z)\mathcal{N}(\boldsymbol{y}|\mathbf{W}\boldsymbol{z} + \boldsymbol{b}, \boldsymbol{\Sigma}_y)\, d\boldsymbol{z} = \mathcal{N}(\boldsymbol{y}|\mathbf{W}\boldsymbol{\mu}_z + \boldsymbol{b}, \boldsymbol{\Sigma}_y + \mathbf{W}\boldsymbol{\Sigma}_z\mathbf{W}^\mathsf{T})$$

The interpretation is as follows: in the generative story ($\boldsymbol{z} \rightarrow \boldsymbol{y}$), we first sample $\boldsymbol{z}$ from the prior $p(\boldsymbol{z})$, then we sample $\boldsymbol{y}$ from the likelihood $p(\boldsymbol{y}|\boldsymbol{z})$; in the inference story ($\boldsymbol{y} \rightarrow \boldsymbol{z}$), we observe $\boldsymbol{y}$ and want to infer the posterior distribution $p(\boldsymbol{z}|\boldsymbol{y})$. While the prior and likelihood are specified by the modeler, the posterior is determined by the data $\boldsymbol{y}$ through the Bayes rule.

2 Statistics and Estimation

The process of estimating model parameters $\boldsymbol{\theta}$ from data $\mathcal{D}$ is called **model fitting**, or training, and is at the heart of machine learning. The key optimization problem is,

$$\hat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta}}{\operatorname{argmax}}\, \mathcal{L}(\boldsymbol{\theta})$$

where $\mathcal{L}(\boldsymbol{\theta})$ is some kind of loss function or objective function.

2.1 Maximum Likelihood Estimation

The most common approach to parameter estimation is to pick the parameters that assign the highest probability to the training data, called **maximum likelihood estimation** or **MLE**.

$$\hat{\boldsymbol{\theta}}_{mle} = \underset{\boldsymbol{\theta}}{\operatorname{argmax}}\, p(\mathcal{D} \mid \boldsymbol{\theta})$$

Often, we assume the training examples are independently sampled from the same distribution (**iid** assumption),

$$p(\mathcal{D} \mid \boldsymbol{\theta}) = \prod_{n=1}^{N} p(\mathbf{y}_n \mid \mathbf{x}_n, \boldsymbol{\theta})$$

working with log likelihood

$$\hat{\boldsymbol{\theta}}_{mle} = \underset{\boldsymbol{\theta}}{\operatorname{argmax}} \log p(\mathcal{D} \mid \boldsymbol{\theta}) = \underset{\boldsymbol{\theta}}{\operatorname{argmax}} \sum_{n=1}^{N} \log p(\mathbf{y}_n \mid \mathbf{x}_n, \boldsymbol{\theta})$$

In some tasks, the optimization algorithm is designed to minimize the loss function, so equivalently,

$$\hat{\boldsymbol{\theta}}_{mle} = \underset{\boldsymbol{\theta}}{\operatorname{argmin}}\, \mathrm{NLL}(\boldsymbol{\theta}) = \underset{\boldsymbol{\theta}}{\operatorname{argmin}} - \sum_{n=1}^{N} \log p(\mathbf{y}_n \mid \mathbf{x}_n, \boldsymbol{\theta})$$

In the special case of unsupervised learning, we have $\mathbf{x}_n = \emptyset$, so the MLE changes $p(\mathbf{y}_n \mid \mathbf{x}_n, \boldsymbol{\theta}) = p(\mathbf{y}_n \mid \boldsymbol{\theta})$

Justification

To justify the use of the MLE is that the resulting predictive distribution $p(\boldsymbol{y}|\hat{\boldsymbol{\theta}}_{\mathrm{mle}})$ is as close as possible to the **empirical distribution** of the data. The empirical distribution is defined by

$$p_{\mathcal{D}}(\boldsymbol{x}, \boldsymbol{y}) = p_{\mathcal{D}}(\boldsymbol{y}|\boldsymbol{x}) p_{\mathcal{D}}(\boldsymbol{x}) = \frac{1}{N} \sum_{n=1}^{N} \delta(\boldsymbol{x} - \boldsymbol{x}_n) \delta(\boldsymbol{y} - \boldsymbol{y}_n)$$

the empirical distribution is a series of delta functions at the observed training points.

If we define $q(Y \mid \boldsymbol{x})$ denotes the model's predictive distribution, then the expected KL then becomes

$$\begin{aligned}
\mathbb{E}_{p_{\mathcal{D}}(\boldsymbol{x})} \left[D_{\mathbb{KL}} \left(p_{\mathcal{D}}(Y|\boldsymbol{x}) \,\|\, q(Y|\boldsymbol{x}) \right) \right] &= \sum_{\boldsymbol{x}} p_{\mathcal{D}}(\boldsymbol{x}) \left[\sum_{\boldsymbol{y}} p_{\mathcal{D}}(\boldsymbol{y}|\boldsymbol{x}) \log \frac{p_{\mathcal{D}}(\boldsymbol{y}|\boldsymbol{x})}{q(\boldsymbol{y}|\boldsymbol{x})} \right] \\
&= \text{const} - \sum_{\boldsymbol{x},\boldsymbol{y}} p_{\mathcal{D}}(\boldsymbol{x}, \boldsymbol{y}) \log q(\boldsymbol{y}|\boldsymbol{x}) \\
&= \text{const} - \frac{1}{N} \sum_{n=1}^{N} \log p(\boldsymbol{y}_n|\boldsymbol{x}_n, \boldsymbol{\theta})
\end{aligned}$$

Minimizing the expected KL divergence is equivalent to minimizing the conditional NLL.

2.2 Empirical Risk Minimization

MLE can be generalized by replacing the (conditional) log loss term, $\ell(\boldsymbol{y}_n, f(\boldsymbol{x}_n; \boldsymbol{\theta})) = -\log p(\boldsymbol{y}_n|\boldsymbol{x}_n, \boldsymbol{\theta})$, with any other loss function, to get

$$\hat{\boldsymbol{\theta}}_{erm} = \underset{\boldsymbol{\theta}}{\operatorname{argmin}}\, \mathcal{L}(\boldsymbol{\theta}) = \underset{\boldsymbol{\theta}}{\operatorname{argmin}}\, \frac{1}{N} \sum_{n=1}^{N} \ell(\boldsymbol{y}_n, f(\boldsymbol{x}_n; \boldsymbol{\theta}))$$

This is **empirical risk minimization** or **ERM**, where the loss ℓ can be set manually either probabilistic or non-probabilistic.

2.3 Regularization

Overfitting may happen with the use of MLE or ERM, as they will try to pick parameters that minimize loss on the training set, but this may not result in a model that has low loss on unseen data. This is called **overfitting**.

The main solution to overfitting is to use **regularization** by adding a penalty term. Thus,

$$\mathcal{L}(\boldsymbol{\theta}; \lambda) = \left[\frac{1}{N} \sum_{n=1}^{N} \ell(\boldsymbol{y}_n, \boldsymbol{\theta}; \boldsymbol{x}_n) \right] + \lambda C(\boldsymbol{\theta})$$

where $\lambda \geq 0$ is the **regularization parameter**, and $C(\boldsymbol{\theta})$ is some form of **complexity penalty**.

A common complexity penalty is to use $C(\boldsymbol{\theta}) = -\log p(\boldsymbol{\theta})$, where $p(\boldsymbol{\theta})$ is the **prior** for $\boldsymbol{\theta}$. If ℓ is the log loss, the regularized objective becomes

$$\mathcal{L}(\boldsymbol{\theta}; \lambda) = -\frac{1}{N} \sum_{n=1}^{N} \log p(\boldsymbol{y}_n|\boldsymbol{x}_n, \boldsymbol{\theta}) - \lambda \log p(\boldsymbol{\theta})$$

By setting $\lambda = 1$ and rescaling $p(\boldsymbol{\theta})$ appropriately, we can equivalently minimize the following:

$$\mathcal{L}(\boldsymbol{\theta}; \lambda) = -\left[\sum_{n=1}^{N} \log p(\boldsymbol{y}_n|\boldsymbol{x}_n, \boldsymbol{\theta}) + \log p(\boldsymbol{\theta}) \right] = -\left[\log p(\mathcal{D}|\boldsymbol{\theta}) + \log p(\boldsymbol{\theta}) \right]$$

Minimizing this is equivalent to maximizing the log posterior:

$$\hat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta}}{\operatorname{argmax}} \log p(\boldsymbol{\theta}|\mathcal{D}) = \underset{\boldsymbol{\theta}}{\operatorname{argmax}} \left[\log p(\mathcal{D}|\boldsymbol{\theta}) + \log p(\boldsymbol{\theta}) - \text{const} \right]$$

This is known as **MAP estimation** or **maximum a posterior estimation**.

2.4 Bayesian Statistics

In the Bayesian approach to statistics, parameters $\boldsymbol{\theta}$ are unknown, and data $\mathcal{D}$ are fixed and known. The uncertainty about the parameters after seeing the data is given by the posterior distribution,

$$p(\boldsymbol{\theta} \mid \mathcal{D}) = \frac{p(\boldsymbol{\theta})\, p(\mathcal{D} \mid \boldsymbol{\theta})}{p(\mathcal{D})} = \frac{p(\boldsymbol{\theta})\, p(\mathcal{D} \mid \boldsymbol{\theta})}{\int p(\boldsymbol{\theta}')\, p(\mathcal{D} \mid \boldsymbol{\theta}')\, d\boldsymbol{\theta}'}$$

where

- likelihood $p(\mathcal{D} \mid \boldsymbol{\theta})$: beliefs about what data we expect to see for each setting of the parameters
- prior $p(\boldsymbol{\theta})$: beliefs about the parameters before seeing the data
- marginal likelihood or evidence $p(\mathcal{D})$: normalization constant

The task of computing this posterior is called Bayesian inference or inference.

In machine learning, we are often very interested in predicting outcomes $\boldsymbol{y}$ given input features $\boldsymbol{x}$. For this, the conditional probability distribution $p(\boldsymbol{y}|\boldsymbol{x}, \boldsymbol{\theta})$ is used, which can be a GLM or a neural net. This also serves as the building block of the objective function during training, as $p(\mathcal{D}|\boldsymbol{\theta}) = \prod_{n=1}^{N} p(\boldsymbol{y}_n|\boldsymbol{x}_n, \boldsymbol{\theta})$.

The main quantity of interest is the **posterior predictive distribution**, given by

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

By **marginalizing out** the unknown parameters, we reduce the chance of overfitting, since we are effectively computing the weighted average of predictions from an infinite number of models. This act of integrating over uncertainty is at the heart of the Bayesian approach.

Comparing the Bayesian approach to **plugin approximation**, in which we compute a point estimate $\hat{\boldsymbol{\theta}}$ of the parameters (such as the MLE), and then plug them into the model to make predictions using $p(\boldsymbol{y}|\boldsymbol{x}, \hat{\boldsymbol{\theta}})$. i.e. $p(\boldsymbol{\theta} \mid \mathcal{D}) \approx \delta(\boldsymbol{\theta} - \hat{\boldsymbol{\theta}})$, since

$$p(\boldsymbol{y} \mid \boldsymbol{x}, \mathcal{D}) \approx \int p(\boldsymbol{y} \mid \boldsymbol{x}, \boldsymbol{\theta})\, \delta(\boldsymbol{\theta} - \hat{\boldsymbol{\theta}})\, d\boldsymbol{\theta} = p(\boldsymbol{y} \mid \boldsymbol{x}, \hat{\boldsymbol{\theta}})$$

The uncertainty captured by plugin approximation is **aleatoric uncertainty** or intrinsic uncertainty, and would persist even if we knew the true model and true parameters. However, it ignores **epistemic uncertainty** due to a lack of knowledge about the truth, which can result in an underestimate of the predictive uncertainty.

Exchangeability

For a sequence of random variables $(\boldsymbol{x}_1, \cdots, \boldsymbol{x}_n)$ is infinitely exchangeable if, for any n, the joint probability $p(\boldsymbol{x}_1, \cdots, \boldsymbol{x}_n)$ is invariant to permutation of the indices. That is, for any permutation π,

$$p(\boldsymbol{x}_1, \cdots, \boldsymbol{x}_n) = p(\boldsymbol{x}_{\pi_1}, \cdots, \boldsymbol{x}_{\pi_n})$$

For iid variables, they are always excanheable as $p(\boldsymbol{x}_1, \cdots, \boldsymbol{x}_n) = \prod_i p(\boldsymbol{x}_i)$. However, exchangeable variables are not always iid, as exchangeablility only requires permutation-invariant and allows dependencies across variables.

Generalizing this idea, an exchangeable sequence can be viewed as coming from a hidden common cause (an unknown random variable $\boldsymbol{\theta}$). If $\boldsymbol{\theta}$ are parameters, the following theorem tells that if we believe the data are infinitely exchangeable, then there must exist a $\boldsymbol{\theta}$, a likelihood, and a prior.

Theorem 1 (**de Finetti's theorem**). *A sequence of random variables $(\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots)$ is infinitely exchangeable iff, for all n, we have*

$$p(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n) = \int \prod_{i=1}^{n} p(\boldsymbol{x}_i \mid \boldsymbol{\theta})\, p(\boldsymbol{\theta})\, d\boldsymbol{\theta}$$

where $\boldsymbol{\theta}$ is some hidden common random variable (possibly infinite dimensional). That is, $\boldsymbol{x}_i$ are iid conditional on $\boldsymbol{\theta}$.

2.5 Prior selection

To compute the posterior distribution, the likelihood or the data-generating model $p(\mathcal{D}|\boldsymbol{\theta})$ is known, so the knowledge about the prior distribution $p(\boldsymbol{\theta})$ is essential. This can be chosen (conjugate or noninformative prior) or learned (hierarchical priors or Empirical Bayes).

1. **Conjugate priors**. A prior $p(\boldsymbol{\theta}) \in \mathcal{F}$ is conjugate prior for a likelihood function $p(\mathcal{D}|\boldsymbol{\theta})$ if the posterior is in the same parameterized family as the prior $p(\boldsymbol{\theta}|\mathcal{D}) \in \mathcal{F}$. Thus, $\mathcal{F}$ is closed during Bayesian updating. For various likelihood distributions, there are tables listing the conjugate priors.

2. **Noninformative priors**. When we have little or no domain specific knowledge, it is desirable to use an noninformative priors, to "let the data speak for itself", eg. flat prior $p(\theta) \propto 1$.

3. **Hierarchical priors**. Let the parameters of the prior be called hyperparameters, and will be denoted by $\boldsymbol{\xi}$. Then put a prior on hyperparameters $p(\boldsymbol{\xi})$, which defines a hierarchical Bayesian model, i.e. $\boldsymbol{\xi} \to \boldsymbol{\theta} \to \mathcal{D}$. Assuming the prior on the hyperparameters is fixed (e.g., some kind of minimally informative prior), so the joint distribution:

$$p(\boldsymbol{\xi}, \boldsymbol{\theta}, \mathcal{D}) = p(\boldsymbol{\xi})\, p(\boldsymbol{\theta} \mid \boldsymbol{\xi})\, p(\mathcal{D} \mid \boldsymbol{\theta})$$

   The idea is to learn $\boldsymbol{\xi}$ by treating the parameters $\boldsymbol{\theta}$ themselves as latent datapoints being generated by $p(\boldsymbol{\theta}|\boldsymbol{\xi})$. For a group of related datasets $\mathcal{D} = \left(\mathcal{D}_1, \mathcal{D}_2, \cdots, \mathcal{D}_J\right)$, each group of dataset has its own $\boldsymbol{\theta}_j$ and shares the same $\boldsymbol{\xi}$. The full hierarchical Bayes infers them jointly by computing the joint posterior $p(\boldsymbol{\theta} = \{\boldsymbol{\theta}_j\}_{j=1}^J, \boldsymbol{\xi}|\mathcal{D})$ using MCMC or variational inference, which can be expensive.

4. **Empirical priors**. Instead of calculating the joint posterior $p(\boldsymbol{\theta}, \boldsymbol{\xi}|\mathcal{D})$, Empirical Bayes calculates the cheaper conditional posterior $p(\boldsymbol{\theta}|\hat{\boldsymbol{\xi}}, \mathcal{D})$, with a point estimate of $\boldsymbol{\xi}$ via

$$\hat{\boldsymbol{\xi}} = \operatorname*{argmax}_{\boldsymbol{\xi}} p(\mathcal{D} \mid \boldsymbol{\xi}) = \operatorname*{argmax}_{\boldsymbol{\xi}} \prod_{j=1}^{J} \int p(\mathcal{D}_j \mid \boldsymbol{\theta}_j)\, p(\boldsymbol{\theta}_j \mid \boldsymbol{\xi})\, d\boldsymbol{\theta}_j$$

   Effectively, we are fitting $\boldsymbol{\xi}$ by computing how likely $\mathcal{D}$ is for a given $\boldsymbol{\xi}$ averaging over every $\boldsymbol{\theta}$, without estimating $\boldsymbol{\theta}$ explicitly. Once $\hat{\boldsymbol{\xi}}$ is known, the prior on $\boldsymbol{\theta}$ can be computed, then the conditional posterior is computable $p(\boldsymbol{\theta}|\hat{\boldsymbol{\xi}}, \mathcal{D}) \propto p(\mathcal{D}|\boldsymbol{\theta})p(\boldsymbol{\theta}|\hat{\boldsymbol{\xi}})$.

2.6 Frequentist Statistics

In frequentist statistics, uncertainty is not represented by the posterior distribution of a random variable, but instead by the sampling distribution of an estimator.

An **estimator** is a decision procedure that specifies what action to take given some observed data. In the context of parameter estimation, where the action space is to return a parameter vector,

$$\hat{\boldsymbol{\theta}} = \hat{\Theta}(\mathcal{D})$$

where $\hat{\boldsymbol{\theta}}$ can be MLE, MAP etc.

The **sampling distribution** of an estimator is the distribution of results we would see if we applied the estimator multiple times to different datasets sampled from some distribution. In the context of parameter estimation, it is the distribution of $\hat{\boldsymbol{\theta}}$, viewed as a random variable that depends on the random sample $\mathcal{D}$.

Imagine sampling S different datasets, each of size N, from some true model $p(\boldsymbol{x}|\boldsymbol{\theta}^*)$ to generate

$$\tilde{\mathcal{D}}^{(s)} = \{\boldsymbol{x}_n \sim p(\boldsymbol{x}_n \mid \boldsymbol{\theta}^*) : n = 1 : N\} \sim \boldsymbol{\theta}^*$$

Now apply the estimator to each $\tilde{\mathcal{D}}^{(s)}$ to get a set of estimates, $\{\hat{\Theta}(\tilde{\mathcal{D}}^{(s)})\}_{s=1}^S$. As we let $S \to \infty$, the distribution induced by this set is the sampling distribution of the estimator, i.e.

$$p(\hat{\Theta}(\tilde{\mathcal{D}}) = \boldsymbol{\theta} \mid \tilde{\mathcal{D}} \sim \boldsymbol{\theta}^*) \approx \frac{1}{S}\sum_{s=1}^{S} \delta(\boldsymbol{\theta} - \hat{\Theta}(\tilde{\mathcal{D}}^{(s)}))$$

Since $\boldsymbol{\theta}^*$ is unknown, $\tilde{\mathcal{D}}^{(s)}$ cannot be generated using $\boldsymbol{\theta}^*$, so this is usually approximated by a Monte Carlo technique called **bootstrap**.

- Parametric bootstrap. Generate $\tilde{\mathcal{D}}^{(s)}$ using $\hat{\boldsymbol{\theta}} = \hat{\Theta}(\mathcal{D})$, i.e. $\tilde{\mathcal{D}}^{(s)} = \{\boldsymbol{x}_n \sim p(\boldsymbol{x}_n \mid \hat{\boldsymbol{\theta}}) : n = 1 : N\}$. Then apply the estimator Θ to each $\tilde{\mathcal{D}}^{(s)}$, yielding the approximated sampling distribution.
- Non-parametric bootstrap. Assuming no parametric generative model $p(\boldsymbol{x}|\boldsymbol{\theta})$ for the data. Then, sampling N data points from the original dataset $\mathcal{D}$ with replacement, i.e. for $s = 1, \cdots, S$, form $\tilde{\mathcal{D}}^{(s)} = \{\boldsymbol{x}_j\}_{j=1}^N$ where each $\boldsymbol{x}_j$ is drawn uniformly from the original size N dataset $\mathcal{D}$. Then apply the estimator Θ to each $\tilde{\mathcal{D}}^{(s)}$, yielding the approximated sampling distribution.

Although bootstrap approximation is generally applicable, when the estimator is MLE and the sample size is large, there is a behaviour called asymptotic normality.

Theorem 2 (**Asymptotic normality of the sampling distribution of the MLE**). *With the MLE estimator, when the sample size becomes large, the sampling distribution of the MLE for certain models becomes Gaussian.*

$$\sqrt{N}(\hat{\boldsymbol{\theta}} - \boldsymbol{\theta}^*) \to \mathcal{N}(\mathbf{0}, \mathbf{F}(\boldsymbol{\theta}^*)^{-1})$$

where $\mathbf{F}(\boldsymbol{\theta}^*)$ is the Fisher information matrix, $\boldsymbol{\theta}^*$ are the parameters of the data generating process to which the estimator will be applied. $\to$ means convergence in distribution.

Therefore, as the sample size goes to infinity, the sampling distribution of the MLE will converge to a Gaussian centered on the true parameter, with a precision equal to the Fisher information.

Fisher information matrix

The **score function** is defined to be the gradient of the log likelihood wrt the parameter vector:

$$\boldsymbol{s}(\boldsymbol{\theta}) \triangleq \nabla_{\boldsymbol{\theta}} \log p(\boldsymbol{x} \mid \boldsymbol{\theta})$$

Consider a scalar θ, $\int p(x|\theta)dx = 1$ gives $\frac{\partial}{\partial\theta}\int p(x|\theta)dx = 0$. Thus, it can be shown that the expectation of the score function is zero.

$$\begin{aligned}\mathbb{E}\left[s(\theta)\right] &= \int \Big[\frac{\partial}{\partial\theta}\log p(x|\theta)\Big] p(x|\theta)dx = \int \frac{\partial}{\partial\theta} p(x|\theta)dx = 0 \\ &\Rightarrow \mathbb{E}_{p(\boldsymbol{x}|\boldsymbol{\theta})}\left[\nabla_{\boldsymbol{\theta}} \log p(\boldsymbol{x} \mid \boldsymbol{\theta})\right] = \mathbf{0}\end{aligned}$$

The **Fisher information matrix (FIM)** is defined to be the covariance of the score function:

$$\mathbf{F}(\boldsymbol{\theta}) \triangleq \mathbb{E}_{\boldsymbol{x}\sim p(\boldsymbol{x}|\boldsymbol{\theta})}\left[\nabla_{\boldsymbol{\theta}} \log p(\boldsymbol{x} \mid \boldsymbol{\theta}) \nabla_{\boldsymbol{\theta}} \log p(\boldsymbol{x} \mid \boldsymbol{\theta})^{\mathsf{T}}\right]$$

so the (i, j)'th entry has the form

$$\begin{aligned}\mathbf{F}(\boldsymbol{\theta})_{ij} &= \mathbb{E}_{\boldsymbol{x}\sim p(\boldsymbol{x}|\boldsymbol{\theta})}\left[\left(\frac{\partial}{\partial \theta_i} \log p(\boldsymbol{x} \mid \boldsymbol{\theta})\right)\left(\frac{\partial}{\partial \theta_j} \log p(\boldsymbol{x} \mid \boldsymbol{\theta})\right)\right] \\ &= -\mathbb{E}_{\boldsymbol{x}\sim p(\boldsymbol{x}|\boldsymbol{\theta})}\left[\frac{\partial^2}{\partial \theta_i \theta_j} \log p(\boldsymbol{x} \mid \boldsymbol{\theta})\right]\end{aligned}$$

where the second line states that the Fisher information matrix equals the expected Hessian of the negative log likelihood (NLL), but this requires $\log p(\boldsymbol{x}|\boldsymbol{\theta})$ to be twice differentiable. The Hessian measures the curvature of the likelihood, the FIM tells how well the likelihood function can identify the best set of parameters (if a likelihood function is flat $F \approx 0$, we cannot infer anything about the parameters).

3 Matrix Calculus

3.1 Functions that map scalars to scalars

A scalar-argument function $f : \mathbb{R} \to \mathbb{R}$, its derivative is defined as

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

It can be interpreted as the slope of the tangent line at $f(x)$, and hence

$$f(x+h) \approx f(x) + f'(x)h$$

3.2 Functions that map vectors to scalars

A vector-argument functions, $f : \mathbb{R}^n \to \mathbb{R}$, by defining the **partial derivative** of f with respect to x_i to be

$$\frac{\partial f}{\partial x_i} = \lim_{h \to 0} \frac{f(\boldsymbol{x} + h\boldsymbol{e}_i) - f(\boldsymbol{x})}{h}$$

where $\boldsymbol{e}_i$ is the i'th unit vector.

The **gradient** of a function at a point $\boldsymbol{x}$ is the vector of its partial derivatives:

$$\boldsymbol{g} = \frac{\partial f}{\partial \boldsymbol{x}} = \nabla f = \begin{pmatrix} \frac{\partial f}{\partial x_1} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{pmatrix}$$

The **directional derivative** measures how much the function $f : \mathbb{R}^n \to \mathbb{R}$ changes along a direction $\boldsymbol{v}$ in space,

$$D_{\boldsymbol{v}} f(\boldsymbol{x}) = \lim_{h \to 0} \frac{f(\boldsymbol{x} + h\boldsymbol{v}) - f(\boldsymbol{x})}{h} = \nabla f^\top \boldsymbol{v}$$

Suppose that some of the arguments to the function depend on each other $f(t, x(t), y(t))$. The **total derivative** of f wrt t is:

$$\frac{df}{dt} = \frac{\partial f}{\partial t} + \frac{\partial f}{\partial x}\frac{dx}{dt} + \frac{\partial f}{\partial y}\frac{dy}{dt}$$

then **total differential**,

$$df = \frac{\partial f}{\partial t}dt + \frac{\partial f}{\partial x}dx + \frac{\partial f}{\partial y}dy$$

This measures how much f changes when we change t, both via the direct effect of t on f, but also indirectly, via the effects of t on x and y.

The **Hessian matrix** is the (symmetric) $n \times n$ matrix of second partial derivatives:

$$\mathbf{H}_f = \frac{\partial^2 f}{\partial \boldsymbol{x}^2} = \nabla^2 f = \begin{pmatrix} \frac{\partial^2 f}{\partial x_1^2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ & \vdots & \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{pmatrix}$$

the Hessian is the Jacobian of the gradient.

Consider a differentiable function $f : \mathbb{R}^n \to \mathbb{R}$. Here are some useful identities:

$$\frac{\partial(\boldsymbol{a}^\mathsf{T}\boldsymbol{x})}{\partial \boldsymbol{x}} = \boldsymbol{a}$$

$$\frac{\partial(\boldsymbol{b}^\mathsf{T}\mathbf{A}\boldsymbol{x})}{\partial \boldsymbol{x}} = \mathbf{A}^\mathsf{T}\boldsymbol{b}$$

$$\frac{\partial(\boldsymbol{x}^\mathsf{T}\mathbf{A}\boldsymbol{x})}{\partial \boldsymbol{x}} = (\mathbf{A} + \mathbf{A}^\mathsf{T})\boldsymbol{x}$$

3.3 Functions that map vectors to vectors

A function that maps a vector to another vector, $f : \mathbb{R}^n \to \mathbb{R}^m$, can be written as

$$f(\boldsymbol{x}) = \begin{pmatrix} f_1(\boldsymbol{x}) \\ \vdots \\ f_m(\boldsymbol{x}) \end{pmatrix}$$

where each component $f_i : \mathbb{R}^n \to \mathbb{R}$ is a scalar-valued function. The **Jacobian matrix** of this function is an $m \times n$ matrix of partial derivatives:

$$\mathbf{J}_{\boldsymbol{f}}(\boldsymbol{x}) = \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}^\mathsf{T}} \triangleq \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{pmatrix} = \begin{pmatrix} \nabla f_1(\boldsymbol{x})^\mathsf{T} \\ \vdots \\ \nabla f_m(\boldsymbol{x})^\mathsf{T} \end{pmatrix} \in \mathbb{R}^{m \times n}$$

There are two ways to multiply the Jacobian with a vector:

- Jacobian vector product (JVP): $\mathbf{J}_{\boldsymbol{f}}(\boldsymbol{x})\boldsymbol{v} = \begin{pmatrix} \nabla f_1(\boldsymbol{x})^\mathsf{T} \\ \vdots \\ \nabla f_m(\boldsymbol{x})^\mathsf{T} \end{pmatrix} \boldsymbol{v} = \begin{pmatrix} \nabla f_1(\boldsymbol{x})^\mathsf{T}\boldsymbol{v} \\ \vdots \\ \nabla f_m(\boldsymbol{x})^\mathsf{T}\boldsymbol{v} \end{pmatrix}$.
- Vector Jacobian product (VJP): $\boldsymbol{u}^\mathsf{T}\mathbf{J}_{\boldsymbol{f}}(\boldsymbol{x}) = \boldsymbol{u}^\mathsf{T}\left(\frac{\partial \boldsymbol{f}}{\partial x_1}, \cdots, \frac{\partial \boldsymbol{f}}{\partial x_n}\right) = \left(\boldsymbol{u} \cdot \frac{\partial \boldsymbol{f}}{\partial x_1}, \cdots, \boldsymbol{u} \cdot \frac{\partial \boldsymbol{f}}{\partial x_n}\right)$.

Let $h(\boldsymbol{x}) = g(f(\boldsymbol{x}))$, by the chain rule of calculus,

$$\mathbf{J}_h(\boldsymbol{x}) = \mathbf{J}_g(f(\boldsymbol{x}))\mathbf{J}_f(\boldsymbol{x})$$

3.4 Functions that map matrices to scalars

A function $f : \mathbb{R}^{m \times n} \to \mathbb{R}$,

$$\frac{\partial f}{\partial \mathbf{X}} = \begin{pmatrix} \frac{\partial f}{\partial x_{11}} & \cdots & \frac{\partial f}{\partial x_{1n}} \\ & \vdots & \\ \frac{\partial f}{\partial x_{m1}} & \cdots & \frac{\partial f}{\partial x_{mn}} \end{pmatrix}$$

Here are some useful identities:

$$\frac{\partial}{\partial \mathbf{X}}(\boldsymbol{a}^\mathsf{T}\mathbf{X}\boldsymbol{b}) = \boldsymbol{a}\boldsymbol{b}^\mathsf{T}, \quad \frac{\partial}{\partial \mathbf{X}}(\boldsymbol{a}^\mathsf{T}\mathbf{X}^\mathsf{T}\boldsymbol{b}) = \boldsymbol{b}\boldsymbol{a}^\mathsf{T}$$

$$\frac{\partial}{\partial \mathbf{X}} \mathrm{tr}(\mathbf{AXB}) = \mathbf{A}^\mathsf{T}\mathbf{B}^\mathsf{T}, \quad \frac{\partial}{\partial \mathbf{X}} \mathrm{tr}(\mathbf{X}^\mathsf{T}\mathbf{A}) = \mathbf{A}$$

$$\frac{\partial}{\partial \mathbf{X}} \mathrm{tr}(\mathbf{X}^{-1}\mathbf{A}) = -\mathbf{X}^{-\mathsf{T}}\mathbf{A}^\mathsf{T}\mathbf{X}^{-\mathsf{T}}, \quad \frac{\partial}{\partial \mathbf{X}} \mathrm{tr}(\mathbf{X}^\mathsf{T}\mathbf{AX}) = (\mathbf{A} + \mathbf{A}^\mathsf{T})\mathbf{X}$$

$$\frac{\partial}{\partial \mathbf{X}} \det(\mathbf{AXB}) = \det(\mathbf{AXB})\mathbf{X}^{-\mathsf{T}}, \quad \frac{\partial}{\partial \mathbf{X}} \log(\det(\mathbf{X})) = \mathbf{X}^{-\mathsf{T}}$$

4 Optimization

An optimization problem means we are trying to find the values for a set of variables $\boldsymbol{\theta} \in \Theta$, that minimize a scalar-valued function $\mathcal{L} : \Theta \to \mathbb{R}$:

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta} \in \Theta} \mathcal{L}(\boldsymbol{\theta}) = \arg\max_{\boldsymbol{\theta} \in \Theta} -\mathcal{L}(\boldsymbol{\theta})$$

where the parameter space is given by $\Theta \subseteq \mathbb{R}^D$ and D is the number of variables being optimized over.

4.1 Constrained optimization

A constrained optimization problem means

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta} \in \mathcal{C}} \mathcal{L}(\boldsymbol{\theta})$$

where the constraint set is

$$\mathcal{C} = \{\boldsymbol{\theta} \in \mathbb{R}^D : h_i(\boldsymbol{\theta}) = 0,\, i \in \mathcal{E},\, g_j(\boldsymbol{\theta}) \leq 0,\, j \in \mathcal{I}\}$$

where $\mathcal{E}$ is the set of **equality constraints**, and $\mathcal{I}$ is the set of **inequality constraints**.

4.1.1 Lagrange multipliers: equality constraints

If the constraint set only contains equality constraints, the method of Lagrange multipliers can be used.

Consider a single equality constaint $h_i(\boldsymbol{\theta}) = 0$, $\nabla h(\boldsymbol{\theta})$ will be orthogonal to the constraint surface since the Taylor expansion

$$h(\boldsymbol{\theta} + \boldsymbol{\epsilon}) \approx h(\boldsymbol{\theta}) + \boldsymbol{\epsilon}^\top \nabla h(\boldsymbol{\theta})$$

where both $\boldsymbol{\theta}$ and $\boldsymbol{\theta} + \boldsymbol{\epsilon}$ are close on the constraint surface, then $h(\boldsymbol{\theta}) = h(\boldsymbol{\theta} + \boldsymbol{\epsilon})$ and hence $\boldsymbol{\epsilon}^\mathsf{T} \nabla h(\boldsymbol{\theta}) \approx 0$. Since $\boldsymbol{\epsilon}$ is parallel to the constraint surface, $\nabla h(\boldsymbol{\theta})$ must be perpendicular to it.

For a point $\boldsymbol{\theta}^*$ on the constraint surface such that $\mathcal{L}(\boldsymbol{\theta})$ is minimized, $\nabla h(\boldsymbol{\theta}^*)$ is orthogonal to the constraint surface. Also, $\nabla \mathcal{L}(\boldsymbol{\theta}^*)$ is orthogonal to the constraint surface, by the definition of a min/max point. Therefore,

$$\nabla \mathcal{L}(\boldsymbol{\theta}^*) = \lambda^* \nabla h(\boldsymbol{\theta}^*) \quad \text{for } \lambda^* \in \mathbb{R}$$

which can be converted into an objective, called the **Lagrangian**,

$$L(\boldsymbol{\theta}, \lambda) \triangleq \mathcal{L}(\boldsymbol{\theta}) + \lambda h(\boldsymbol{\theta})$$

At a stationary point of the Lagrangian,

$$\nabla_{\boldsymbol{\theta}, \lambda} L(\boldsymbol{\theta}, \lambda) = \mathbf{0} \iff \lambda \nabla_{\boldsymbol{\theta}} h(\boldsymbol{\theta}) = -\nabla \mathcal{L}(\boldsymbol{\theta}),\ h(\boldsymbol{\theta}) = 0$$

Combining all the constraints, the Lagrangian becomes:

$$L(\boldsymbol{\theta}, \boldsymbol{\lambda}) = \mathcal{L}(\boldsymbol{\theta}) + \sum_{j=1}^{m} \lambda_j h_j(\boldsymbol{\theta})$$

where λ is the Lagrangian multiplier.

4.1.2 The KKT conditions

Generalizing the concept of Lagrange multipliers to additionally handle inequality constraints.

Introduction

Consider a single inequality constaint $g_j(\boldsymbol{\theta}) \leq 0$, to find the optimum, one approach is to construct an objective which penalizes $g_j(\boldsymbol{\theta}) > 0$ by going to infinity. Adding an extra term to the original objective $\mu g(\boldsymbol{\theta})$ where $\mu \geq 0$, then taking the maximum of this Lagrangian

$$\max_{\mu \geq 0} L(\boldsymbol{\theta}, \mu) = \max_{\mu \geq 0} \mathcal{L}(\boldsymbol{\theta}) + \mu g(\boldsymbol{\theta}) \begin{cases} \infty & \text{if } g(\boldsymbol{\theta}) > 0, \\ \mathcal{L}(\boldsymbol{\theta}) & \text{otherwise} \end{cases}$$

Thus, the optimization problem becomes

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \left\{ \max_{\mu \geq 0} L(\boldsymbol{\theta}, \mu) \right\}$$

Generalized Lagrangian

Consider the general case with multiple constraints, $\mathcal{C} = \{\boldsymbol{\theta} \in \mathbb{R}^D : h_i(\boldsymbol{\theta}) = 0,\, i \in \mathcal{E},\, g_j(\boldsymbol{\theta}) \leq 0,\, j \in \mathcal{I}\}$, the generalized Lagrangian is,

$$L(\boldsymbol{\theta}, \boldsymbol{\mu} = \{\mu_j\}_{j \in \mathcal{I}}, \boldsymbol{\lambda} = \{\lambda_i\}_{i \in \mathcal{E}}) = \mathcal{L}(\boldsymbol{\theta}) + \sum_j \mu_j g_j(\boldsymbol{\theta}) + \sum_i \lambda_i h_i(\boldsymbol{\theta})$$

where the optimization problem is

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \left\{ \max_{\boldsymbol{\mu} \geq 0, \boldsymbol{\lambda}} L(\boldsymbol{\theta}, \boldsymbol{\mu}, \boldsymbol{\lambda}) \right\}$$

Note that $\max_{\boldsymbol{\mu} \geq 0, \boldsymbol{\lambda}}$ penalizes every $g_j(\boldsymbol{\theta}) > 0$ and $h_i(\boldsymbol{\theta}) \neq 0$ by setting L to infinity.

Karush-Kuhn-Tucker (KKT) conditions

In the generalized Lagrangian, the KKT conditions are

- All constraints are satisfied

$$g_j(\boldsymbol{\theta}) \leq 0,\, h_i(\boldsymbol{\theta}) = 0 \quad \text{for } \forall i, j$$

- The solution is a stationary point:

$$\nabla \mathcal{L}(\boldsymbol{\theta}^*) + \sum_j \mu_j \nabla g_j(\boldsymbol{\theta}^*) + \sum_i \lambda_i \nabla h_i(\boldsymbol{\theta}^*) = \mathbf{0}$$

- The penalty for the inequality constraint points in the right direction $\boldsymbol{\mu} \geq \mathbf{0}$.

- The Lagrange multipliers pick up any slack in the inactive constraints, i.e., either $\mu_j = 0$ or $g_j(\boldsymbol{\theta}^*) = 0$, so $\boldsymbol{\mu} \odot \boldsymbol{g} = \mathbf{0}$. This is due to the fact that, for a single inequality constraint $g_j(\boldsymbol{\theta}^*) \leq 0$: when $g_j(\boldsymbol{\theta}^*) = 0$ (active), this turns out to be an equality constraint, thus $\nabla \mathcal{L} = -\mu \nabla g_j$; when $g_j(\boldsymbol{\theta}^*) < 0$, we have $\nabla \mathcal{L} = -\mu \nabla g_j$ and $\nabla \mathcal{L} = 0$, thus $\mu = 0$.

If $\mathcal{L}$ and $g_{j \in \mathcal{I}}$ are convex and $h_{i \in \mathcal{E}}$ are affine, KKT conditions are sufficient for global optimality.

4.2 Bound optimization

If the optimization goal is to maximize a function $\ell(\boldsymbol{\theta})$ wrt its parameters $\boldsymbol{\theta}$. The idea of bound optimization is to construct a surrogate function $Q(\boldsymbol{\theta}; \boldsymbol{\theta}^t)$ which is a tight lowerbound to $\ell(\boldsymbol{\theta})$ such that:

$$Q(\boldsymbol{\theta}; \boldsymbol{\theta}^t) \leq \ell(\boldsymbol{\theta}) \quad \forall \boldsymbol{\theta}, \boldsymbol{\theta}^t \quad \text{and} \quad Q(\boldsymbol{\theta}^t; \boldsymbol{\theta}^t) = \ell(\boldsymbol{\theta}^t)$$

Then, perform the following iterative update:

$$\boldsymbol{\theta}^{t+1} = \operatorname*{argmax}_{\boldsymbol{\theta}} Q(\boldsymbol{\theta}; \boldsymbol{\theta}^t)$$

which guarantees monotonic increase of the objective function $\ell(\boldsymbol{\theta})$:

$$\ell(\boldsymbol{\theta}^{t+1}) \geq Q(\boldsymbol{\theta}^{t+1}; \boldsymbol{\theta}^t) \geq Q(\boldsymbol{\theta}^t; \boldsymbol{\theta}^t) = \ell(\boldsymbol{\theta}^t)$$

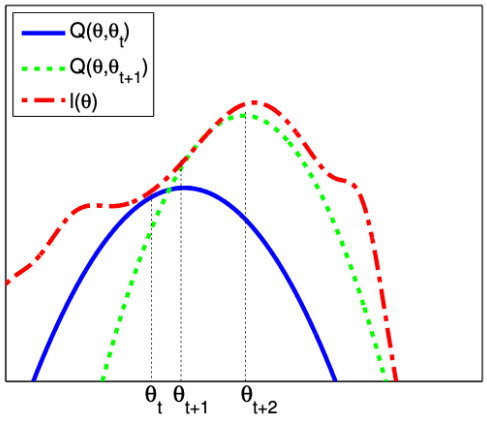


Figure 1: The solid blue curve is the lower bound, evaluated at $\boldsymbol{\theta}^t$; this touches the objective function at $\boldsymbol{\theta}^t$. We then set $\boldsymbol{\theta}^{t+1}$ to the maximum of the lower bound (blue curve), and fit a new bound at that point (dotted green curve). The maximum of this new bound becomes $\boldsymbol{\theta}^{t+2}$, etc

4.2.1 Q for concave functions

If $\ell(\boldsymbol{\theta})$ is a concave function we want to maximize, then one way to obtain a surrogate function is through its Hessian. Consider the second-order Taylor expansion of $\ell(\boldsymbol{\theta})$,

$$\begin{aligned}\ell(\boldsymbol{\theta}) &\approx \ell(\boldsymbol{\theta}^t) + (\boldsymbol{\theta} - \boldsymbol{\theta}^t)^\mathsf{T} \nabla \ell(\boldsymbol{\theta}^t) + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}^t)^\mathsf{T} \mathbf{H} (\boldsymbol{\theta} - \boldsymbol{\theta}^t) \\ &\geq \ell(\boldsymbol{\theta}^t) + (\boldsymbol{\theta} - \boldsymbol{\theta}^t)^\mathsf{T} \nabla \ell(\boldsymbol{\theta}^t) + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}^t)^\mathsf{T} \mathbf{B} (\boldsymbol{\theta} - \boldsymbol{\theta}^t)\end{aligned}$$

where $\mathbf{H}(\boldsymbol{\theta})$ is negative definite due to concavity, and $\mathbf{B}$ is a negative definite matrix such that $\mathbf{0} \succ \mathbf{H}(\boldsymbol{\theta}) \succ \mathbf{B}$. Then, expanding the brackets and dropping terms constant in $\boldsymbol{\theta}$,

$$Q(\boldsymbol{\theta}, \boldsymbol{\theta}^t) = \boldsymbol{\theta}^\mathsf{T}(\nabla \ell(\boldsymbol{\theta}^t) - \mathbf{B}\boldsymbol{\theta}^t) + \frac{1}{2}\boldsymbol{\theta}^\mathsf{T}\mathbf{B}\boldsymbol{\theta}$$

Solving $\nabla_{\boldsymbol{\theta}} Q(\boldsymbol{\theta}, \boldsymbol{\theta}^t) = \mathbf{0}$ gives a closed-form update,

$$\boldsymbol{\theta}^{t+1} = \boldsymbol{\theta}^t - \mathbf{B}^{-1} \nabla \ell(\boldsymbol{\theta}^t)$$

4.2.2 EM algorithm

Expectation maximization (EM) algorithm is a special case of bound optimization designed to compute the MLE or MAP parameter estimates for probability models that have **missing data** and/or **hidden variables**.

EM algorithm alternate between:

- E-step: estimating the hidden variables $\boldsymbol{z}_n$
- M-step: using the fully observed data $\boldsymbol{y}_n$ to compute the MLE

Lower bound: ELBO

The goal of EM is to maximize the log likelihood of the observed data:

$$\ell(\boldsymbol{\theta}) = \sum_{n=1}^{N} \log p(\boldsymbol{y}_n \mid \boldsymbol{\theta}) = \sum_{n=1}^{N} \log \left[\sum_{\boldsymbol{z}_n} p(\boldsymbol{y}_n, \boldsymbol{z}_n \mid \boldsymbol{\theta}) \right]$$

which is hard to optimize since the log cannot be pushed inside the sum.

Consider a set of arbitrary distribution $q(\boldsymbol{z}_n)$ over each hidden variable $\boldsymbol{z}_n$, then apply Jensen's inequality,

$$\begin{aligned} \ell(\boldsymbol{\theta}) = \sum_{n=1}^{N} \log \left[\sum_{\boldsymbol{z}_n} q(\boldsymbol{z}_n) \frac{p(\boldsymbol{y}_n, \boldsymbol{z}_n \mid \boldsymbol{\theta})}{q(\boldsymbol{z}_n)} \right] &\geq \sum_{n=1}^{N} \sum_{\boldsymbol{z}_n} q(\boldsymbol{z}_n) \log \frac{p(\boldsymbol{y}_n, \boldsymbol{z}_n \mid \boldsymbol{\theta})}{q(\boldsymbol{z}_n)} \\ &= \sum_{n} \mathbb{E}_{q_n} \left[\log p(\boldsymbol{y}_n, \boldsymbol{z}_n | \boldsymbol{\theta}) \right] + \mathbb{H}(q_n) \\ &= \sum_{n} \mathrm{L}(\boldsymbol{\theta}, q_n) \triangleq \mathrm{L}(\boldsymbol{\theta}, \{q_n\}) \end{aligned}$$

where $\mathrm{L}(\boldsymbol{\theta}, \{q_n\})$ is the evidence lower bound (ELBO).

E step: fix $\boldsymbol{\theta} = \boldsymbol{\theta}^t$, and update $\{q_n\}$

The lower bound is a sum of N terms, each of which has the form:

$$\begin{aligned} \mathrm{L}(\boldsymbol{\theta}, q_n) &= \sum_{\boldsymbol{z}_n} q_n(\boldsymbol{z}_n) \log \frac{p(\boldsymbol{y}_n, \boldsymbol{z}_n | \boldsymbol{\theta})}{q_n(\boldsymbol{z}_n)} \\ &= \sum_{\boldsymbol{z}_n} q_n(\boldsymbol{z}_n) \log \frac{p(\boldsymbol{z}_n | \boldsymbol{y}_n, \boldsymbol{\theta}) p(\boldsymbol{y}_n | \boldsymbol{\theta})}{q_n(\boldsymbol{z}_n)} \\ &= \sum_{\boldsymbol{z}_n} q_n(\boldsymbol{z}_n) \log \frac{p(\boldsymbol{z}_n | \boldsymbol{y}_n, \boldsymbol{\theta})}{q_n(\boldsymbol{z}_n)} + \sum_{\boldsymbol{z}_n} q_n(\boldsymbol{z}_n) \log p(\boldsymbol{y}_n | \boldsymbol{\theta}) \\ &= -D_{\mathbb{KL}} \left(q_n(\boldsymbol{z}_n) \,\|\, p(\boldsymbol{z}_n | \boldsymbol{y}_n, \boldsymbol{\theta}) \right) + \log p(\boldsymbol{y}_n | \boldsymbol{\theta}) \end{aligned}$$

where $D_{\mathbb{KL}}(q \| p) \geq 0$ and $D_{\mathbb{KL}}(q \| p) = 0$ iff $q = p$. Hence, the maximization of $\mathrm{L}(\boldsymbol{\theta}, \{q_n\})$ wrt $\{q_n\}$ can be done by setting each one to $q_n^* = p(\boldsymbol{z}_n | \boldsymbol{y}_n, \boldsymbol{\theta})$. Note that, $p(\boldsymbol{z}_n | \boldsymbol{y}_n, \boldsymbol{\theta}) = \frac{p(\boldsymbol{y}_n, \boldsymbol{z}_n | \boldsymbol{\theta})}{\sum_{\boldsymbol{z}'_n} p(\boldsymbol{y}_n, \boldsymbol{z}'_n | \boldsymbol{\theta})}$ and $p(\boldsymbol{y}_n, \boldsymbol{z}_n \mid \boldsymbol{\theta}) = p(\boldsymbol{y}_n \mid \boldsymbol{z}_n, \boldsymbol{\theta}) p(\boldsymbol{z}_n)$ are computable since the prior and conditional distribution on $\boldsymbol{z}_n$ is defined when defining the probabilistic latent model.

This can be shown that fits into the $Q(\boldsymbol{\theta}, \boldsymbol{\theta}^t)$ formulation.

M step: fix $\{q_n\} = \{q_n^*\}$, and update $\boldsymbol{\theta}$

In the M step, we need to maximize $\mathrm{L}(\boldsymbol{\theta}, \{q_n^t\})$ wrt $\boldsymbol{\theta}$, where the q_n^t are the distributions computed in the E step at iteration t. Since the entropy terms $\mathbb{H}(q_n)$ are constant wrt $\boldsymbol{\theta}$, so dropping them in the M step. Then, left with

$$\ell^t(\boldsymbol{\theta}) = \sum_n \mathbb{E}_{q_n^t}\left[\log p(\boldsymbol{y}_n, \boldsymbol{z}_n|\boldsymbol{\theta})\right]$$

This is the **expected complete data log likelihood**. Then, the optimization task is,

$$\boldsymbol{\theta}^{t+1} = \arg\max_{\boldsymbol{\theta}} \sum_n \mathbb{E}_{q_n^t}\left[\log p(\boldsymbol{y}_n, \boldsymbol{z}_n|\boldsymbol{\theta})\right]$$

5 Mixture Models

To create more complex probability models is to take a convex combination of simple distributions. This is called a mixture model.

$$p(\boldsymbol{y}|\boldsymbol{\theta}) = \sum_{k=1}^{K} p(z=k|\boldsymbol{\theta}) p(\boldsymbol{y}|z=k,\boldsymbol{\theta}) = \sum_{k=1}^{K} \pi_k p(\boldsymbol{y}|z=k,\boldsymbol{\theta})$$

π_k is the mixture weight for component k, and $\sum_{k=1}^{K} \pi_k = 1$, and the **latent variable** $z \in \{1,\ldots,K\}$ which specifies which distribution to use for generating the output $\boldsymbol{y}$.

5.1 Gaussian mixture models

A **Gaussian mixture models (GMM)** is a mixture model where each component is a Gaussian distribution,

$$p(\boldsymbol{y}|\boldsymbol{\theta}) = \sum_{k=1}^{K} \pi_k \mathcal{N}(\boldsymbol{y}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$$

GMMs are often used for unsupervised clustering of real-valued data samples $\boldsymbol{y}_n \in \mathbb{R}^D$. First, the parameters $\boldsymbol{\theta}$ of the GMM can be estimated using MLE, often with the Expectation-Maximization (EM) algorithm.

$$\hat{\boldsymbol{\theta}} = \operatorname*{argmax}_{\boldsymbol{\theta}} \log p(\{\boldsymbol{y}_n : n = 1,\ldots,N\} \mid \boldsymbol{\theta})$$

Then, we associate each data point $\boldsymbol{y}_n$ with a discrete latent variable $z_n \in \{1,\ldots,K\}$, which indicates which mixture component or cluster generated the data point. The posterior distribution over the latent variable is given by Bayes' rule:

$$p(z_n = k|\boldsymbol{y}_n,\boldsymbol{\theta}) = \frac{p(\boldsymbol{y}_n|z_n=k,\boldsymbol{\theta})p(z_n=k|\boldsymbol{\theta})}{\sum_{j=1}^{K} p(\boldsymbol{y}_n|z_n=j,\boldsymbol{\theta})p(z_n=j|\boldsymbol{\theta})}$$

This quantity is known as the **responsibility** of cluster k for data point $\boldsymbol{y}_n$. Then, we can compute the most probable cluster assignment as:

$$\hat{z}_n = \operatorname*{argmax}_k p(z_n = k|\boldsymbol{y}_n,\boldsymbol{\theta})$$

This forms the basis of the clustering alorithms like the K-means clustering.

5.2 EM for a GMM

E step

The E step computes the responsibility of cluster k for generating data point n, as estimated using the current parameter estimates $\boldsymbol{\theta}^t$:

$$r_{nk}^t = p^*(z_n = k|\boldsymbol{y}_n,\boldsymbol{\theta}^t) = \frac{\pi_k^t p(\boldsymbol{y}_n|z_n=k,\boldsymbol{\theta}^t)}{\sum_{k'} \pi_{k'}^t p(\boldsymbol{y}_n|z_n=k',\boldsymbol{\theta}^t)}$$

M step

The M step maximizes the expected complete data log likelihood, given by

$$\begin{aligned}
\ell^t(\boldsymbol{\theta}) &= \mathbb{E}\left[\sum_n \log p(z_n|\boldsymbol{\pi}) + \sum_n \log p(\boldsymbol{y}_n|z_n,\boldsymbol{\theta})\right] \\
&= \mathbb{E}\left[\sum_n \log\left(\prod_k \pi_k^{z_{nk}}\right) + \sum_n \log\left(\prod_k \mathcal{N}(\boldsymbol{y}_n|\boldsymbol{\mu}_k,\boldsymbol{\Sigma}_k)^{z_{nk}}\right)\right] \\
&= \sum_n \sum_k \mathbb{E}[z_{nk}] \log \pi_k + \sum_n \sum_k \mathbb{E}[z_{nk}] \log \mathcal{N}(\boldsymbol{y}_n|\boldsymbol{\mu}_k,\boldsymbol{\Sigma}_k) \\
&= \sum_n \sum_k r_{nk}^{(t)} \log(\pi_k) - \frac{1}{2}\sum_n \sum_k r_{nk}^{(t)} \left[\log|\boldsymbol{\Sigma}_k| + (\boldsymbol{y}_n - \boldsymbol{\mu}_k)^\mathsf{T} \boldsymbol{\Sigma}_k^{-1}(\boldsymbol{y}_n - \boldsymbol{\mu}_k)\right] + \text{const}
\end{aligned}$$

where $z_{nk} = \mathbb{I}(z_n = k)$ is a one-hot encoding of the categorical value z_n. Then, the parameter estimates are

$$\boldsymbol{\mu}_k^{(t+1)} = \frac{\sum_n r_{nk}^{(t)} \boldsymbol{y}_n}{r_k^{(t)}}$$

$$\begin{aligned}
\boldsymbol{\Sigma}_k^{(t+1)} &= \frac{\sum_n r_{nk}^{(t)} (\boldsymbol{y}_n - \boldsymbol{\mu}_k^{(t+1)})(\boldsymbol{y}_n - \boldsymbol{\mu}_k^{(t+1)})^\mathsf{T}}{r_k^{(t)}} \\
&= \frac{\sum_n r_{nk}^{(t)} \boldsymbol{y}_n \boldsymbol{y}_n^\mathsf{T}}{r_k^{(t)}} - \boldsymbol{\mu}_k^{(t+1)} (\boldsymbol{\mu}_k^{(t+1)})^\mathsf{T}
\end{aligned}$$

where $r_k^{(t)} \triangleq \sum_n r_{nk}^{(t)}$ is the weighted number of points assigned to cluster k. The mean of cluster k is just the weighted average of all points assigned to cluster k, and the covariance is proportional to the weighted empirical scatter matrix.

The M step for the mixture weights is a weighted form of the usual MLE:

$$\pi_k^{(t+1)} = \frac{1}{N}\sum_n r_{nk}^{(t)} = \frac{r_k^{(t)}}{N}$$

5.3 Bernoulli mixture model

A **Bernoulli mixture model (BMM)** can be used for binary-valued data, where each mixture component has the following form:

$$p(\boldsymbol{y}|z = k, \boldsymbol{\theta}) = \prod_{d=1}^D \text{Ber}(y_d|\mu_{dk}) = \prod_{d=1}^D \mu_{dk}^{y_d}(1 - \mu_{dk})^{1-y_d}$$

Part II
Linear Models and Neural Networks

6 Generalized Linear Models

A GLM is a conditional version of an exponential family distribution $p(y \mid \theta) = h(y) \exp\left(\theta^\top T(y) - A(\theta)\right)$, in which the natural parameters are a linear function of the input. More precisely, the model has the following form:

$$p(y_n \mid \boldsymbol{x}_n, \boldsymbol{w}, \sigma^2) = \exp\left[\frac{y_n \eta_n - A(\eta_n)}{\sigma^2} + \log h(y_n, \sigma^2)\right]$$

where

- y_n is the output/response variable, $\boldsymbol{x_n}$ is the input/feature vector, $\boldsymbol{w}$ and σ is the learnable model parameter
- $\eta_n \triangleq \boldsymbol{w}^\mathsf{T} \boldsymbol{x}_n$ is the (input dependent) natural parameter or canonical parameter
- $A(\eta_n)$ is the log normalizer
- $\mathcal{T}(y_n) = y_n$ is the sufficient statistic, which is the smallest summary of the output data y that retains everything relevant for estimating the parameter θ – while it may still lose other information about y itself.
- σ^2 is the dispersion term
- $h(y_n, \sigma^2)$ is the scaling constant or base measure

We can show that the mean and variance of the response variable are as follows:

$$\int p(y_n \mid \eta_n, \sigma^2)\, dy_n = 1 \quad \text{for all } \eta_n$$

To derive the mean, differentiate the equation w.r.t to η_n

$$\int \frac{\partial p(y_n \mid \eta_n, \sigma^2)}{\partial \eta_n}\, dy_n = 0$$

$$\int p(y_n \mid \eta_n, \sigma^2) \cdot \frac{y_n - A'(\eta_n)}{\sigma^2}\, dy_n = 0$$

$$\frac{1}{\sigma^2}\left[\int y_n\, p(y_n \mid \eta_n, \sigma^2)\, dy_n - A'(\eta_n)\right] = 0$$

thus, the **mean** is:

$$\mathbb{E}\left[y_n \mid \boldsymbol{x}_n, \boldsymbol{w}, \sigma^2\right] = A'(\eta_n) \triangleq \ell^{-1}(\eta_n)$$

where the mapping from the linear inputs to the mean of the output using $\mu_n = \ell^{-1}(\eta_n)$, and the function ℓ is known as the link function, and ℓ^{-1} is known as the mean function.

To derive the variance, differentiate the above equation w.r.t η_n again,

$$\int \left[-A''(\eta_n) \cdot p + (y_n - A'(\eta_n)) \cdot p \cdot \frac{y_n - A'(\eta_n)}{\sigma^2}\right] dy_n = 0$$

$$-A''(\eta_n) \int p\, dy_n + \frac{1}{\sigma^2} \int (y_n - A'(\eta_n))^2\, p\, dy_n = 0$$

thus, the **variance** is:

$$\mathbb{V}\left[y_n \mid \boldsymbol{x}_n, \boldsymbol{w}, \sigma^2\right] = A''(\eta_n)\sigma^2$$

6.1 Logistic regression

Binary logistic regression is used when $y_n \in \{0, 1\}$. It has the form

$$p(y_n \mid \boldsymbol{x}_n, \boldsymbol{w}) = \mu_n^{y_n}(1 - \mu_n)^{1-y_n}$$

where

$$\mu_n = \sigma(\eta_n) = \frac{1}{1 + \exp(-\eta_n)}, \quad \eta_n = \boldsymbol{w}^\mathsf{T}\boldsymbol{x}_n.$$

Hence

$$\log p(y_n \mid \boldsymbol{x}_n, \boldsymbol{w}) = y_n \log \mu_n + (1 - y_n) \log(1 - \mu_n)$$

Using $\mu_n = \exp(\eta_n)/(1 + \exp(\eta_n))$, in GLM form:

$$\log p(y_n \mid \boldsymbol{x}_n, \boldsymbol{w}) = y_n\eta_n - \log(1 + \exp(\eta_n))$$

We see that $A(\eta_n) = \log(1 + \exp(\eta_n))$ and hence

$$\mathbb{E}[y_n] = A'(\eta_n) = \frac{1}{1 + \exp(-\eta_n)}, \quad \mathbb{V}[y_n] = A''(\eta_n) = \mu_n(1 - \mu_n)$$

Thus the link function is the logit function, $\ell(\mu_n) = \log\left(\frac{\mu_n}{1-\mu_n}\right)$, and the mean function is the sigmoid function, $\ell^{-1}(\eta_n) = \sigma(\eta_n)$.

6.2 Linear regression

Linear regression has the form with $y_n \in \mathbb{R}$

$$p(y_n \mid \boldsymbol{x}_n, \boldsymbol{w}, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(y_n - \boldsymbol{w}^\mathsf{T}\boldsymbol{x}_n)^2\right)$$

Hence

$$\log p(y_n \mid \boldsymbol{x}_n, \boldsymbol{w}, \sigma^2) = -\frac{1}{2\sigma^2}(y_n - \eta_n)^2 - \frac{1}{2}\log(2\pi\sigma^2)$$

where $\eta_n = \boldsymbol{w}^\mathsf{T}\boldsymbol{x}_n$. In GLM form:

$$\log p(y_n \mid \boldsymbol{x}_n, \boldsymbol{w}, \sigma^2) = \frac{y_n\eta_n - \frac{\eta_n^2}{2}}{\sigma^2} - \frac{1}{2}\left(\frac{y_n^2}{\sigma^2} + \log(2\pi\sigma^2)\right)$$

We see that $A(\eta_n) = \eta_n^2/2$ and hence

$$\mathbb{E}[y_n] = \eta_n = \boldsymbol{w}^\mathsf{T}\boldsymbol{x}_n, \quad \mathbb{V}[y_n] = \sigma^2$$

6.3 Binomial regression

If the response variable is the number of successes in N_n trials, $y_n \in \{0, \ldots, N_n\}$, we can use binomial regression, which is defined by

$$p(y_n \mid \boldsymbol{x}_n, N_n, \boldsymbol{w}) = \mathrm{Bin}(y_n \mid \sigma(\boldsymbol{w}^\mathsf{T}\boldsymbol{x}_n), N_n)$$

note that binary logistic regression is the special case when $N_n = 1$.

The log pdf is given by

$$\log p(y_n \mid \boldsymbol{x}_n, N_n, \boldsymbol{w}) = y_n \log \mu_n + (N_n - y_n) \log(1 - \mu_n) + \log \binom{N_n}{y_n}$$

$$= y_n \log\left(\frac{\mu_n}{1-\mu_n}\right) + N_n \log(1-\mu_n) + \log \binom{N_n}{y_n}$$

where $\mu_n = \sigma(\eta_n)$. To rewrite this in GLM form, let us define

$$\eta_n \triangleq \log\left(\frac{\mu_n}{1-\mu_n}\right) = \log\left[\frac{1}{1+e^{-\boldsymbol{w}^\mathsf{T}\boldsymbol{x}_n}} \frac{1+e^{-\boldsymbol{w}^\mathsf{T}\boldsymbol{x}_n}}{e^{-\boldsymbol{w}^\mathsf{T}\boldsymbol{x}_n}}\right] = \log\left(\frac{1}{e^{-\boldsymbol{w}^\mathsf{T}\boldsymbol{x}_n}}\right) = \boldsymbol{w}^\mathsf{T}\boldsymbol{x}_n$$

Hence we can write binomial regression in GLM form as follows:

$$\log p(y_n \mid \boldsymbol{x}_n, N_n, \boldsymbol{w}) = y_n \eta_n - A(\eta_n) + h(y_n)$$

where

$$h(y_n) = \log \binom{N_n}{y_n}, \quad A(\eta_n) = -N_n \log(1-\mu_n) = N_n \log(1+e^{\eta_n})$$

Hence,

$$\mathbb{E}[y_n] = \frac{dA}{d\eta_n} = \frac{N_n e^{\eta_n}}{1+e^{\eta_n}} = N_n \mu_n, \quad \mathbb{V}[y_n] = \frac{d^2 A}{d\eta_n^2} = N_n \mu_n (1-\mu_n)$$

6.4 Poisson regression

If the response variable is an integer count, $y_n \in \{0, 1, \ldots\}$, we can use Poisson regression, which is defined by

$$p(y_n \mid \boldsymbol{x}_n, \boldsymbol{w}) = \text{Poi}(y_n \mid \exp(\boldsymbol{w}^\mathsf{T}\boldsymbol{x}_n))$$

where

$$\text{Poi}(y \mid \mu) = e^{-\mu} \frac{\mu^y}{y!}$$

is the Poisson distribution.

The log pdf is given by

$$\log p(y_n \mid \boldsymbol{x}_n, \boldsymbol{w}) = y_n \log \mu_n - \mu_n - \log(y_n!)$$

where $\mu_n = \exp(\boldsymbol{w}^\mathsf{T}\boldsymbol{x}_n)$. Hence in GLM form we have

$$\log p(y_n \mid \boldsymbol{x}_n, \boldsymbol{w}) = y_n \eta_n - A(\eta_n) + h(y_n)$$

where $\eta_n = \log(\mu_n) = \boldsymbol{w}^\mathsf{T}\boldsymbol{x}_n$, $A(\eta_n) = \mu_n = e^{\eta_n}$, and $h(y_n) = -\log(y_n!)$. Hence

$$\mathbb{E}[y_n] = \frac{dA}{d\eta_n} = e^{\eta_n} = \mu_n, \quad \mathbb{V}[y_n] = \frac{d^2 A}{d\eta_n^2} = e^{\eta_n} = \mu_n$$

6.5 Canonical and non-canonical link functions

Link functions connect three quantities:

- θ_n — the **natural parameter** of the *exponential family distribution*.
- $\mu_n = \mathbb{E}[y_n]$ — the **mean** of the output distribution.
- $\eta_n = \boldsymbol{w}^\top \boldsymbol{x}_n$ — the **linear predictor**, built from the *inputs* and weights.

The two connecting functions:

$$\theta_n \xrightarrow{A'} \mu_n \xrightarrow{\ell} \eta_n$$

- A' (derivative of the log-partition function) always maps $\theta_n \to \mu_n$. This relationship is fixed by the choice of exponential-family distribution (Bernoulli, Gaussian, Poisson, …) and never changes.
- ℓ, the **link function**, maps $\mu_n \to \eta_n$. This is the function *you choose* when designing the GLM. Its inverse, ℓ^{-1}, is the **mean function**: $\mu_n = \ell^{-1}(\eta_n)$ — the direction actually used to generate predictions from η_n.

Canonical link. Choosing $\ell = (A')^{-1}$ makes the link exactly undo A':

$$\eta_n = g(\mu_n) = (A')^{-1}(A'(\theta_n)) = \theta_n$$

so the natural parameter and linear predictor coincide: $\theta_n = \eta_n$.

Non-canonical link. Choosing any $g \neq (A')^{-1}$ gives

$$\eta_n = g(A'(\theta_n)) \neq \theta_n \quad \text{in general.}$$

The mean is still recovered correctly via $\mu_n = g^{-1}(\eta_n)$, but the natural parameter θ_n is no longer equal to the linear predictor η_n—recovering it requires the separate map $(A')^{-1}(\mu_n)$.

Examples of links:

- Logit: canonical link $\theta_n = \ell(\mu_n) = \log \frac{\mu_n}{1-\mu_n}$.
- Complementary log-log: non-canonical link $\eta_n = \ell(\mu_n) = \log(-\log(1-\mu_n))$

6.6 Maximum likelihood estimation

GLMs can be fit the negative log-likelihood as the objective function (ignoring constant term h):

$$\text{NLL}(\boldsymbol{w}) = -\log p(\mathcal{D} \mid \boldsymbol{w}) = -\frac{1}{\sigma^2}\sum_{n=1}^{N} \ell_n = -\frac{1}{\sigma^2}\sum_{n=1}^{N}[\eta_n y_n - A(\eta_n)]$$

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

To find the MLE, we must solve $\nabla \text{NNL}(\boldsymbol{w}) = \boldsymbol{g}(\boldsymbol{w}) = 0$. For notational simplicity, we will assume $\sigma^2 = 1$. The gradient for a single term is:

$$\begin{aligned}\boldsymbol{g}_n &= -\frac{1}{\sigma^2}\frac{\partial \ell_n}{\partial \boldsymbol{w}} = -\frac{1}{\sigma^2}\frac{\partial \ell_n}{\partial \eta_n}\frac{\partial \eta_n}{\partial \boldsymbol{w}} \\ &= -\frac{1}{\sigma^2}\left(y_n - A'(\eta_n)\right)\boldsymbol{x}_n = -\frac{1}{\sigma^2}(y_n - \mu_n)\boldsymbol{x}_n\end{aligned}$$

where $\mu_n = f(\boldsymbol{w}^\mathsf{T}\boldsymbol{x})$, and f is the inverse link function that maps from canonical parameters to mean parameters. If we interpret $(y_n - \mu_n)$ as an error signal, then the gradient $-\frac{1}{\sigma^2}\sum_{n=1}^{N}(y_n - \mu_n)\boldsymbol{x}_n$ weights each input $\boldsymbol{x}_n$ by its error, and averages the result.

Gradient-based optimization will find a stationary points where $\boldsymbol{g(w)} = 0$, we need to examine the Hessian matrix to determine whether it is the global optimum. The Hessian is given by

$$\begin{aligned}\boldsymbol{H} &= \frac{\partial^2}{\partial \boldsymbol{w}\partial \boldsymbol{w}^\top}\mathrm{NLL}(\boldsymbol{w}) \\ &= -\sum_{n=1}^{N}\frac{\partial \boldsymbol{g}_n}{\partial \boldsymbol{w}^\top} = -\sum_{n=1}^{N}\frac{\partial \boldsymbol{g}_n}{\partial \mu_n}\frac{\partial \mu_n}{\partial \boldsymbol{w}^\top} = -\sum_{n=1}^{N}\boldsymbol{x}_n f'(\boldsymbol{w}^\top \boldsymbol{x}_n)\boldsymbol{x}_n^\top\end{aligned}$$

In general, the Hessian is positive definite, since $f'(\boldsymbol{w}^\top \boldsymbol{x}_n) > 0$, hence the negative log-likelihood is convex, so the MLE for a GLM is unique given $f(\boldsymbol{w}^\top \boldsymbol{x}_n) > 0$ for all n.

Based on these results, gradient-based optimizers can be used to estimate the model weights.

Stochastic gradient descent

The goal is to solve

$$\hat{\boldsymbol{w}} \triangleq \operatorname*{argmin}_{\boldsymbol{w}} \mathcal{L}(\boldsymbol{w}) = \operatorname*{argmin}_{\boldsymbol{w}} \mathrm{NLL}(\boldsymbol{w})$$

One of the methods is stochastic gradient descent: we use a minibatch from the whole dataset of size N, then we get the following update equation:

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \alpha_t \nabla_{\boldsymbol{w}} \mathrm{NLL}(\boldsymbol{w}_t) = \boldsymbol{w}_t - \alpha_t \frac{N}{B}\sum_{i=1}^{B}(\mu_i - y_i)\boldsymbol{x}_i$$

where we replaced the average over all N samples with a stochastically chosen minibatch of size B at each step t.

Since the objective is convex, then it can be shown that this will converge to a global optimum given the learning rate α_t is decayed appropriately at each step t.

7 Deep Neural Networks

Recall some linear models for classification and regression:

- Binary logistic regression: $p(y|\boldsymbol{x}, \boldsymbol{w}) = \text{Ber}(y|\sigma(\boldsymbol{w}^\top \boldsymbol{x}))$
- Multi-class logistic regression: $p(y|\boldsymbol{x}, \boldsymbol{W}) = \text{Cat}(y|\text{softmax}(\mathbf{W}^\top \boldsymbol{x}))$
- Linear regression: $p(y|\boldsymbol{x}, \boldsymbol{w}, \sigma^2) = \mathcal{N}(y \mid \boldsymbol{w}^\top \boldsymbol{x}, \sigma^2)$
- Generalized linear models: generalized to various output distributions in the exponential family.

To increase the flexibility of such models is to perform a feature transformation, i.e. replacing $\boldsymbol{x}$ with $\boldsymbol{\phi}(\boldsymbol{x})$. Thus, the model becomes $f(\boldsymbol{x}; \boldsymbol{\theta}) = \mathbf{W}\boldsymbol{\phi}(\boldsymbol{x}) + \boldsymbol{b}$ with $\boldsymbol{\theta} = (\mathbf{W}, \boldsymbol{b})$. Then, we can give the feature extractor $\boldsymbol{\phi}$ with its own parameter $\boldsymbol{\theta}_2$, to get $f(\boldsymbol{x}; \boldsymbol{\theta}) = \mathbf{W}\boldsymbol{\phi}(\boldsymbol{x}; \boldsymbol{\theta}_2) + \boldsymbol{b}$ with $\boldsymbol{\theta} = (\mathbf{W}, \boldsymbol{b}, \boldsymbol{\theta}_2)$. This can be done recursively, to create more and more complex functions:

$$f(\boldsymbol{x}; \boldsymbol{\theta}) = f_L(f_{L-1}(\dots f_2(f_1(\boldsymbol{x}; \boldsymbol{\theta}_1); \boldsymbol{\theta}_2) \dots; \boldsymbol{\theta}_{L-1}); \boldsymbol{\theta}_L)$$

where f_ℓ is the ℓ-th layer of the network, and $\boldsymbol{\theta}_\ell$ are the parameters of that layer. This is known as a **multilayer perceptron (MLP)** or a **feedforward neural network**.

An MLP assumes that the input layer is a fixed dimension vector, say $\boldsymbol{x} \in \mathbb{R}^D$, and the data is often stored in an $N \times D$ design matrix (i.e. structured data). There are other types of neural networks that can handle "unstructured" data, such as convolutional neural networks (CNNs) for images, recurrent neural networks (RNNs) for sequences, and graph neural networks (GNNs) for graphs.

7.1 Multilayer perceptrons

A traditional definition of a perceptron is a deterministic version of logistic regression,

$$f(\boldsymbol{x}; \boldsymbol{\theta}) = \mathbb{I}(\boldsymbol{w}^\top \boldsymbol{x} + b > 0)$$

which is essentially a heaviside step function or a linear threshold function. However, the decision boundaries represented by perceptrons are linear, so they are very limited in what they can represent. Also, the heaviside step function is not differentiable, so it is hard to train.

These problems can be solved by,

1. Stacking multiple perceptrons on top of each other, forming a multilayer perceptrons (MLPs). Therefore, we are using a hidden layer of perceptrons to produce a nonlinear decision boundary.

2. Replacing the heaviside step function $\mathbb{R} \to \{0, 1\}$ with a differentiable **activation function** $\varphi : \mathbb{R} \to \mathbb{R}$. Defining the hidden units $\boldsymbol{z}_l$ at each layer l to be a linear transformation of $\boldsymbol{z}_{l-1}$ passed elementwise through this activation function,

$$\boldsymbol{z}_l = f_l(\boldsymbol{z}_{l-1}) = \varphi(\mathbf{W}_l \boldsymbol{z}_{l-1} + \boldsymbol{b}_l)$$

   where the quantity that is passed to the activation function is called the pre-activations.

The term "MLP" almost always refers to this differentiable model, rather than the historical version with non-differentiable heaviside.

Activation functions

Any differentiable function can be used as an activation function. However, if a linear function $\varphi(a) = c_l a$ is used, the whole model reduces to a linear model.

Three types of common activation functions are:

- Sigmoid, which is a smooth approximation to the heaviside step function:

$$\sigma(a) = \frac{1}{1 + e^{-a}}$$

  $\sigma(a)$ saturates at 1 for large positive inputs, and at 0 for large negative inputs.

- Tanh:

$$\tanh(a) = \frac{e^{-a} - e^{a}}{e^{-a} + e^{a}}$$

  $\tanh(a)$ saturates at 1 for large positive inputs, and at -1 for large negative inputs.

- ReLU:

$$\text{ReLU}(a) = \max(0, a) = a\mathbb{I}(a > 0)$$

  ReLU is a non-saturating activation function, which are able to propagate gradients from higher layers to lower layers more effectively.

Network depth

It can be shown that an MLP with one hidden layer is a universal function approximator, meaning it can model any suitably smooth function, given enough hidden units, to any desired level of accuracy. However, deep networks work better than shallow ones. The reason is that later layers can leverage the features that are learned by earlier layers; that is, the function is defined in a compositional or hierarchical way.

7.2 Backpropagation

Backpropagation is an algorithm to compute the gradient of the loss function applied to the output of the network wrt the parameters in each layer. This gradient can then be passed to a gradient-based optimization algorithm to train the network.

7.2.1 Backpropagation for MLPs

MLPs are a simple kind of DNN in which each layer feeds directly into the next, forming a chain structure. Consider a mapping $\boldsymbol{o} = \boldsymbol{f}(\boldsymbol{x})$ where $\boldsymbol{o} \in \mathbb{R}^m$ and $\boldsymbol{x} \in \mathbb{R}^n$. Assuming $\boldsymbol{f}$ is defined as a composition of functions,

$$\boldsymbol{f} = \boldsymbol{f}_4 \circ \boldsymbol{f}_3 \circ \boldsymbol{f}_2 \circ \boldsymbol{f}_1$$

where $\boldsymbol{f}_1 : \mathbb{R}^n \to \mathbb{R}^{m_1}$, $\boldsymbol{f}_2 : \mathbb{R}^{m_1} \to \mathbb{R}^{m_2}$, $\boldsymbol{f}_3 : \mathbb{R}^{m_2} \to \mathbb{R}^{m_3}$, and $\boldsymbol{f}_4 : \mathbb{R}^{m_3} \to \mathbb{R}^m$. The intermediate steps to compute $\boldsymbol{o} = \boldsymbol{f}(\boldsymbol{x})$ are $\boldsymbol{x}_2 = \boldsymbol{f}_1(\boldsymbol{x})$, $\boldsymbol{x}_3 = \boldsymbol{f}_2(\boldsymbol{x}_2)$, $\boldsymbol{x}_4 = \boldsymbol{f}_3(\boldsymbol{x}_3)$, and $\boldsymbol{o} = \boldsymbol{f}_4(\boldsymbol{x}_4)$.

To compute the Jacobian $\mathbf{J}_{\boldsymbol{f}}(\boldsymbol{x}) = \frac{\partial \boldsymbol{o}}{\partial \boldsymbol{x}} \in \mathbb{R}^{m \times n}$ using the chain rule:

$$\frac{\partial \boldsymbol{o}}{\partial \boldsymbol{x}} = \frac{\partial \boldsymbol{o}}{\partial \boldsymbol{x}_4}\frac{\partial \boldsymbol{x}_4}{\partial \boldsymbol{x}_3}\frac{\partial \boldsymbol{x}_3}{\partial \boldsymbol{x}_2}\frac{\partial \boldsymbol{x}_2}{\partial \boldsymbol{x}} = \frac{\partial \boldsymbol{f}_4(\boldsymbol{x}_4)}{\partial \boldsymbol{x}_4}\frac{\partial \boldsymbol{f}_3(\boldsymbol{x}_3)}{\partial \boldsymbol{x}_3}\frac{\partial \boldsymbol{f}_2(\boldsymbol{x}_2)}{\partial \boldsymbol{x}_2}\frac{\partial \boldsymbol{f}_1(\boldsymbol{x})}{\partial \boldsymbol{x}}$$

$$= \mathbf{J}_{\boldsymbol{f}_4}(\boldsymbol{x}_4)\mathbf{J}_{\boldsymbol{f}_3}(\boldsymbol{x}_3)\mathbf{J}_{\boldsymbol{f}_2}(\boldsymbol{x}_2)\mathbf{J}_{\boldsymbol{f}_1}(\boldsymbol{x})$$

To compute the Jacobian $\mathbf{J}_{\boldsymbol{f}}(\boldsymbol{x})$ efficiently, recall that

$$\mathbf{J}_{\boldsymbol{f}}(\boldsymbol{x}) = \frac{\partial \boldsymbol{f}(\boldsymbol{x})}{\partial \boldsymbol{x}} = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{pmatrix} = \begin{pmatrix} \nabla f_1(\boldsymbol{x})^\mathsf{T} \\ \vdots \\ \nabla f_m(\boldsymbol{x})^\mathsf{T} \end{pmatrix} = \left(\frac{\partial \boldsymbol{f}}{\partial x_1}, \cdots, \frac{\partial \boldsymbol{f}}{\partial x_n} \right) \in \mathbb{R}^{m \times n}$$

where $\nabla f_i(\boldsymbol{x})^\mathsf{T} \in \mathbb{R}^{1 \times n}$ is the i'th row (for $i = 1 : m$) and $\frac{\partial \boldsymbol{f}}{\partial x_j} \in \mathbb{R}^m$ is the j'th column (for $j = 1 : n$). Note that, in our notation, when $m = 1$, the gradient, denoted $\nabla \boldsymbol{f}(\boldsymbol{x})$, has the same shape as $\boldsymbol{x}$. It is therefore a column vector, while $\mathbf{J}_{\boldsymbol{f}}(\boldsymbol{x})$ is a row vector. In this case, we therefore technically have $\nabla \boldsymbol{f}(\boldsymbol{x}) = \mathbf{J}_{\boldsymbol{f}}(\boldsymbol{x})^\mathsf{T}$.

To extract individual columns or rows of the Jacobian,

- Extracting the i'th row of $\mathbf{J}_{\boldsymbol{f}}(\boldsymbol{x})$ by using a vector Jacobian Product (VJP),

  $$\boldsymbol{e}_i^\top \mathbf{J}_{\boldsymbol{f}}(\boldsymbol{x}), \text{ where } \boldsymbol{e}_i \in \mathbb{R}^m \text{ is the } i\text{'th unit basis vector}$$

  If $n > m$, it is more efficient to compute $\mathbf{J}_{\boldsymbol{f}}(\boldsymbol{x})$ for each row $i = 1 : m$ by using VJPs in a left-to-right manner,

  $$\boldsymbol{u}^\mathsf{T} \mathbf{J}_{\boldsymbol{f}}(\boldsymbol{x}) = \underbrace{\boldsymbol{u}^\mathsf{T}}_{1 \times m} \underbrace{\mathbf{J}_{\boldsymbol{f}_4}(\boldsymbol{x}_4)}_{m \times m_3} \underbrace{\mathbf{J}_{\boldsymbol{f}_3}(\boldsymbol{x}_3)}_{m_3 \times m_2} \underbrace{\mathbf{J}_{\boldsymbol{f}_2}(\boldsymbol{x}_2)}_{m_2 \times m_1} \underbrace{\mathbf{J}_{\boldsymbol{f}_1}(\boldsymbol{x}_1)}_{m_1 \times n}$$

  This can be done using reverse mode differentiation.

- Extracting the j'th column of $\mathbf{J}_{\boldsymbol{f}}(\boldsymbol{x})$ by using a Jacobian vector Product (JVP),

  $$\mathbf{J}_{\boldsymbol{f}}(\boldsymbol{x}) \boldsymbol{e}_j, \text{ where } \boldsymbol{e}_j \in \mathbb{R}^n \text{ is the } j\text{'th unit basis vector}$$

  If $m > n$, it is more efficient to compute $\mathbf{J}_{\boldsymbol{f}}(\boldsymbol{x})$ for each column $j = 1 : n$ by using JVPs in a right-to-left manner,

  $$\mathbf{J}_{\boldsymbol{f}}(\boldsymbol{x}) \boldsymbol{v} = \underbrace{\mathbf{J}_{\boldsymbol{f}_4}(\boldsymbol{x}_4)}_{m \times m_3} \underbrace{\mathbf{J}_{\boldsymbol{f}_3}(\boldsymbol{x}_3)}_{m_3 \times m_2} \underbrace{\mathbf{J}_{\boldsymbol{f}_2}(\boldsymbol{x}_2)}_{m_2 \times m_1} \underbrace{\mathbf{J}_{\boldsymbol{f}_1}(\boldsymbol{x}_1)}_{m_1 \times n} \underbrace{\boldsymbol{v}}_{n \times 1}$$

  This can be done using forward mode differentiation.

7.2.2 Reverse mode differentiation and VJPs with learnable parameters

Assuming the four layer MLP structure for analysis (without loss of generality), but each layer can have (optional) learnable parameters $\boldsymbol{\theta}_\ell$ for $\ell = 1 : 4$. Assuming the case where the mapping has the form $\mathcal{L} : \mathbb{R}^n \to \mathbb{R}$ (i.e. output is a scalar). For example, consider the loss for a MLP with one hidden layer:

$$\mathcal{L}((\boldsymbol{x}, \boldsymbol{y}), \boldsymbol{\theta}) = \frac{1}{2} \|\boldsymbol{y} - \mathbf{W}_2 \varphi(\mathbf{W}_1 \boldsymbol{x})\|^2$$

which can be represented as a feedforward model,

$$\begin{aligned} \mathcal{L} &= \boldsymbol{f}_4 \circ \boldsymbol{f}_3 \circ \boldsymbol{f}_2 \circ \boldsymbol{f}_1 \\ \boldsymbol{x}_2 &= \boldsymbol{f}_1(\boldsymbol{x}, \boldsymbol{\theta}_1) = \mathbf{W}_1 \boldsymbol{x} \\ \boldsymbol{x}_3 &= \boldsymbol{f}_2(\boldsymbol{x}_2, \emptyset) = \varphi(\boldsymbol{x}_2) \\ \boldsymbol{x}_4 &= \boldsymbol{f}_3(\boldsymbol{x}_3, \boldsymbol{\theta}_3) = \mathbf{W}_2 \boldsymbol{x}_3 \\ \mathcal{L} &= \boldsymbol{f}_4(\boldsymbol{x}_4, \boldsymbol{y}) = \frac{1}{2} \|\boldsymbol{y} - \boldsymbol{x}_4\|^2 \end{aligned}$$

the notation $\boldsymbol{f}_k(\boldsymbol{x}_k, \boldsymbol{\theta}_k)$ denotes the function at layer k with input $\boldsymbol{x}_k$ and learnable parameters $\boldsymbol{\theta}_k$. The loss function $\mathcal{L}$ is a function of the input $\boldsymbol{x}$, the target $\boldsymbol{y}$, and the learnable parameters $\boldsymbol{\theta} = (\boldsymbol{\theta}_1, \boldsymbol{\theta}_2 = \emptyset, \boldsymbol{\theta}_3)$.

Chain rule can be used to compute the gradient of the scalar output wrt the parameters in each layer, once $\frac{\partial \mathcal{L}}{\partial \boldsymbol{x}_4}$ is computed. The gradient of the loss wrt the parameters in each layer can be computed as follows:

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}_3} &= \frac{\partial \mathcal{L}}{\partial \boldsymbol{x}_4}\frac{\partial \boldsymbol{x}_4}{\partial \boldsymbol{\theta}_3} \\
\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}_2} &= \frac{\partial \mathcal{L}}{\partial \boldsymbol{x}_4}\frac{\partial \boldsymbol{x}_4}{\partial \boldsymbol{x}_3}\frac{\partial \boldsymbol{x}_3}{\partial \boldsymbol{\theta}_2} \\
\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}_1} &= \frac{\partial \mathcal{L}}{\partial \boldsymbol{x}_4}\frac{\partial \boldsymbol{x}_4}{\partial \boldsymbol{x}_3}\frac{\partial \boldsymbol{x}_3}{\partial \boldsymbol{x}_2}\frac{\partial \boldsymbol{x}_2}{\partial \boldsymbol{\theta}_1}
\end{aligned}$$

where each $\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}_k} = (\nabla_{\boldsymbol{\theta}_k} \mathcal{L})^\mathsf{T}$ is a d_k-dimensional gradient row vector, where d_k is the number of parameters in layer k. These can be computed recursively, by multiplying the gradient row vector at layer k by the Jacobian $\frac{\partial \boldsymbol{x}_k}{\partial \boldsymbol{x}_{k-1}}$ which is an $n_k \times n_{k-1}$ matrix, where n_k is the number of hidden units in layer k.

Algorithm 1 Backpropagation for an MLP with K layers and scalar output

// Forward pass
$\boldsymbol{x}_1 := \boldsymbol{x}$
for $k = 1 : K$ **do**
    $\boldsymbol{x}_{k+1} = \boldsymbol{f}_k(\boldsymbol{x}_k, \boldsymbol{\theta}_k)$
end for
// Backward pass
$\boldsymbol{u}_{K+1} := 1$
for $k = K : 1$ **do**
    $\boldsymbol{g}_k := \boldsymbol{u}_{k+1}^\mathsf{T} \frac{\partial \boldsymbol{f}_k(\boldsymbol{x}_k, \boldsymbol{\theta}_k)}{\partial \boldsymbol{\theta}_k}$
    $\boldsymbol{u}_k^\mathsf{T} := \boldsymbol{u}_{k+1}^\mathsf{T} \frac{\partial \boldsymbol{f}_k(\boldsymbol{x}_k, \boldsymbol{\theta}_k)}{\partial \boldsymbol{x}_k}$
end for
// Output
Return $\mathcal{L} = \boldsymbol{x}_{K+1}$, $\nabla_{\boldsymbol{x}} \mathcal{L} = \boldsymbol{u}_1$, $\{\nabla_{\boldsymbol{\theta}_k} \mathcal{L} = \boldsymbol{g}_k : k = 1 : K\}$

Since the the final layer returns a scalar $m < n$, it is more efficient to use reverse mode differentation to compute the gradient vectors. This algorithm computes both $\{\nabla_{\boldsymbol{\theta}_k} \mathcal{L} \in \mathbb{R}^{d_k}\}_{k=1}^K$ and $\nabla_{\boldsymbol{x}} \mathcal{L} \in \mathbb{R}^n$, but only the former is used for parameter learning.

Recall that the Jacobian for a layer of the form $\boldsymbol{f} : \mathbb{R}^n \to \mathbb{R}^m$ is defined by,

$$\boldsymbol{J}_{\boldsymbol{f}}(\boldsymbol{x}) = \frac{\partial \boldsymbol{f}(\boldsymbol{x})}{\partial \boldsymbol{x}} = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{pmatrix} = \begin{pmatrix} \nabla f_1(\boldsymbol{x})^\mathsf{T} \\ \vdots \\ \nabla f_m(\boldsymbol{x})^\mathsf{T} \end{pmatrix} = \left(\frac{\partial \boldsymbol{f}}{\partial x_1}, \cdots, \frac{\partial \boldsymbol{f}}{\partial x_n} \right) \in \mathbb{R}^{m \times n}$$

It is essential to be able to compute VJP $\boldsymbol{u}^\top \boldsymbol{J}_{\boldsymbol{f}}(\boldsymbol{x})$ (i.e. rows) for common layers.

- Cross entropy layer. A cross entropy layer takes logits $\boldsymbol{x}$ and target $\boldsymbol{y}$ as inputs, and returning a scalar:

$$z = f(\boldsymbol{x}) = -\sum_c y_c \log(\text{softmax}(\boldsymbol{x})_c) = -\sum_{c=1}^{C} y_c \log p_c$$

where $\boldsymbol{p} = \text{softmax}(\boldsymbol{x}) = \frac{e^{x_c}}{\sum_{c'=1}^{C} e^{x_{c'}}}$ are the predicted class probabilites and $\boldsymbol{y}$ is the true distribution (usually a one-hot vector). Thus, differentiate z wrt an element of $\boldsymbol{x}$:

$$\frac{\partial z}{\partial x_i} = -\sum_{c=1}^{C} y_c \frac{\partial}{\partial x_i} \log p_c = -\sum_{c=1}^{C} y_c \frac{1}{p_c} \frac{\partial p_c}{\partial x_i} = -\sum_{c=1}^{C} y_c (\mathbb{I}(i=c) - p_i)$$

where $\frac{\partial p_c}{\partial x_i} = p_c(\mathbb{I}(i=c) - p_i)$. If the true target is c, then we define $\boldsymbol{y} = \mathbb{I}(i=c)$, the Jacobian wrt the input is

$$\boldsymbol{J} = \frac{\partial z}{\partial \boldsymbol{x}} = (\boldsymbol{p} - \boldsymbol{y})^\top \in \mathbb{R}^{1 \times C}$$

- Elementwise nonlinearity layer. An elementwise nonlinearity layer follows,

$$\boldsymbol{z} = \boldsymbol{f}(\boldsymbol{x}) = \varphi(\boldsymbol{x}), \text{ with } z_i = \varphi(x_i)$$

The Jacobian wrt the input is,

$$\boldsymbol{J} = \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}} = \text{diag}(\varphi'(\boldsymbol{x})) \in \mathbb{R}^{n \times n}$$

since $\frac{\partial z_i}{\partial x_j} = \varphi'(x_i)\mathbb{I}(i=j)$.

- Linear layer. A linear layer follows,

$$\boldsymbol{z} = \boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$$

where $\boldsymbol{W} \in \mathbb{R}^{m \times n}$, $\boldsymbol{z} \in \mathbb{R}^m$, and $\boldsymbol{x} \in \mathbb{R}^n$. The Jacobian wrt the input is,

$$\boldsymbol{J} = \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}} = \boldsymbol{W} \in \mathbb{R}^{m \times n}$$

The Jacobian wrt the weight matrix, $\mathbf{J} = \frac{\partial \boldsymbol{z}}{\partial \mathbf{W}}$ is a $m \times (m \times n)$ matrix, which is effectively m matrices $\{\frac{\partial z_k}{\partial \mathbf{W}}\}_{k=1}^m$ stacking together. For each (i,j) entry of $\frac{\partial z_k}{\partial \mathbf{W}}$, consider

$$z_k = \sum_{l=1}^{n} W_{kl} x_l$$

$$\left[\frac{\partial z_k}{\partial \mathbf{W}}\right]_{ij} = \frac{\partial z_k}{\partial W_{ij}} = \sum_{l=1}^{n} x_l \frac{\partial}{\partial W_{ij}} W_{kl} = \sum_{l=1}^{n} x_l \mathbb{I}(i=k \text{ and } j=l) = x_j \mathbb{I}(i=k)$$

Compactly, the VJP between $\boldsymbol{e}_k^\mathsf{T} \in \mathbb{R}^{1 \times m}$ and $\frac{\partial \boldsymbol{z}}{\partial \mathbf{W}} \in \mathbb{R}^{m \times (m \times n)}$ can be represented as a matrix of shape $1 \times (m \times n)$. Note that

$$\boldsymbol{e}_k^\mathsf{T} \frac{\partial \boldsymbol{z}}{\partial \mathbf{W}} = \frac{\partial z_k}{\partial \mathbf{W}} = \begin{pmatrix} 0 & 0 & \cdots & 0 \\ \vdots & & & \vdots \\ x_1 & x_2 & \cdots & x_n \\ \vdots & & & \vdots \\ 0 & 0 & \cdots & 0 \end{pmatrix}, \text{ where row } k \text{ is } \boldsymbol{x}^\top, \text{ all other rows are zero}$$

$$= \boldsymbol{e}_k \boldsymbol{x}^\top \in \mathbb{R}^{m \times n}$$

Therefore, for a general upstream gradient $\mathbf{g} = \frac{\partial \ell}{\partial \mathbf{z}} \in \mathbb{R}^{m \times 1}$,

$$\mathbf{g}^\top \frac{\partial \mathbf{z}}{\partial \mathbf{W}} = \sum_k g_k \, \mathbf{e}_k \mathbf{x}^\top = \mathbf{g} \, \mathbf{x}^\top$$

since $\sum_k g_k \mathbf{e}_k = \mathbf{g}$.

7.2.3 Computation graphs: beyond MLPs

MLPs are a simple kind of DNN in which each layer feeds directly into the next, forming a chain structure. However, modern DNNs can combine differentiable components in much more complex ways, creating a computation graph. The only restriction is that the computation graph forms a directed acyclic graph (DAG), where each node is a differentiable function of all its inputs.

Note that during the backward pass, since the computation graph is no longer a chain, we may need to sum gradients along multiple paths. In general,

$$\frac{\partial \boldsymbol{o}}{\partial \boldsymbol{x}_j} = \sum_{k \in \mathrm{Ch}(j)} \frac{\partial \boldsymbol{o}}{\partial \boldsymbol{x}_k} \frac{\partial \boldsymbol{x}_k}{\partial \boldsymbol{x}_j}$$

where the sum is over all children of node j in the computation graph. This is multivariate chain rule.

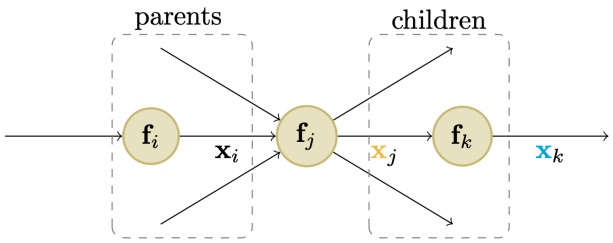


Figure 2: Automatic differentiation at a node j in a computation graph.

7.3 Neural network training

The standard approach to training a neural network is to use maximum likelihood estimation (MLE) to find the parameters $\boldsymbol{\theta}$ that maximize the likelihood of the training data. This is equivalent to minimizing the negative log-likelihood (NLL) loss function:

$$\mathcal{L}(\boldsymbol{\theta}) = -\log p(\mathcal{D}|\boldsymbol{\theta}) = -\sum_{n=1}^{N} \log p(\boldsymbol{y}_n|\boldsymbol{x}_n; \boldsymbol{\theta})$$

Then, use backprop algorithm to compute the gradient of the loss function wrt the parameters $\nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta})$, and use a gradient-based optimization algorithm (eg. SGD) to update the parameters.

Note that the DNN loss is not a convex objective, so in general we will not be able to find the global optimum. However, SGD can often find suprisingly good solutions.

7.3.1 Stochastic gradient descent and learning rate tuning

Stochastic optimization aims to minimize the average value of a function:

$$\mathcal{L}(\boldsymbol{\theta}) = \mathbb{E}_{q(\boldsymbol{z})}[\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{z})]$$

where $\boldsymbol{z}$ is a random input to the objective function, which could be a "noise" term, or it could be a training example drawn randomly from the training set.

Stochastic gradient descent (SGD) assumes a way to compute an unbiased estimate of the gradient of $\mathcal{L}$. Then, at each iteration t, $\mathcal{L}_t(\boldsymbol{\theta}) = \mathcal{L}(\boldsymbol{\theta}, \boldsymbol{z}_t)$ where $\boldsymbol{z}_t \sim q$. If the distribution q is independent from the parameters $\boldsymbol{\theta}$, then denote $\boldsymbol{g}_t = \nabla_{\boldsymbol{\theta}} \mathcal{L}_t(\boldsymbol{\theta}_t)$. In this case, the algorithm is,

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta_t \boldsymbol{g}_t = \boldsymbol{\theta}_t - \eta_t \nabla_{\boldsymbol{\theta}} \mathcal{L}_t(\boldsymbol{\theta}_t, \boldsymbol{z}_t)$$

As long as the gradient estimate is unbiased, then this method will converge to a stationary point, providing the learning rate η_t is decayed properly.

An application of SGD is minimizing the empirical risk.

$$\mathcal{L}(\boldsymbol{\theta}_t) = \frac{1}{N} \sum_{n=1}^{N} \ell(\boldsymbol{y}_n, f(\boldsymbol{x}_n; \boldsymbol{\theta}_t)) = \frac{1}{N} \sum_{n=1}^{N} \mathcal{L}_n(\boldsymbol{\theta}_t)$$

The gradient of this objective has the form

$$\boldsymbol{g}_t = \frac{1}{N} \sum_{n=1}^{N} \nabla_{\boldsymbol{\theta}} \mathcal{L}_n(\boldsymbol{\theta}_t) = \frac{1}{N} \sum_{n=1}^{N} \nabla_{\boldsymbol{\theta}} \ell(\boldsymbol{y}_n, f(\boldsymbol{x}_n; \boldsymbol{\theta}_t))$$

This requires summing over all N training examples, and thus can be slow if N is large. Fortunately we can approximate this by sampling a **minibatch** of $B \ll N$ samples to get

$$\boldsymbol{g}_t \approx \frac{1}{|\mathcal{B}_t|} \sum_{n \in \mathcal{B}_t} \nabla_{\boldsymbol{\theta}} \mathcal{L}_n(\boldsymbol{\theta}_t) = \frac{1}{|\mathcal{B}_t|} \sum_{n \in \mathcal{B}_t} \nabla_{\boldsymbol{\theta}} \ell(\boldsymbol{y}_n, f(\boldsymbol{x}_n; \boldsymbol{\theta}_t))$$

where $\mathcal{B}_t$ is a set of randomly chosen examples to use at iteration t.

Learning rate tuning

Rather than choosing a single constant learning rate, we can use a **learning rate schedule**, in which we adjust the step size over time.

Theoretically, a sufficient condition for SGD to achieve convergence is if the learning rate schedule satisfies the Robbins-Monro conditions:

$$\eta_t \to 0, \quad \sum_{t=1}^{\infty} \eta_t = \infty, \quad \sum_{t=1}^{\infty} \eta_t^2 < \infty$$

$$\Rightarrow \frac{\sum_{t=1}^{\infty} \eta_t^2}{\sum_{t=1}^{\infty} \eta_t} \to 0$$

Intuitively, $\sum_{t=1}^{\infty} \eta_t = \infty$ ensures that the learning rate does not decay too fast; $\sum_{t=1}^{\infty} \eta_t^2 < \infty$ ensures that the learning rate must decay quickly enough that the variance introduced by noisy gradient estimates eventually vanishes.

Some common examples of learning rate schedules:

- Piecewise constant: $\eta_t = \eta_i$ if $t_i < t < t_{i+1}$.
- Exponential decay: $\eta_t = \eta_0 e^{-\lambda t}$.
- Polynomial decay: $\eta_t = \eta_0 (\beta t + 1)^{-\alpha}$.
- Cyclic learning rate: increase and decrease the learning rate multiple times, in a cyclical fashion, to escape local minima.

7.3.2 Vanishing and exploding gradients

When training a very deep neural network, the gradients can either vanish (become very small) or explode (become very large) as they are backpropagated through the layers, because the error signal is being passed through a series of layers which either amplify or diminish it.

Mathematically, consider the gradient of the loss wrt a node at layer l:

$$\underbrace{\frac{\partial \mathcal{L}}{\partial \boldsymbol{z}_l}}_{1\times n_l} = \underbrace{\frac{\partial \mathcal{L}}{\partial \boldsymbol{z}_{l+1}}}_{1\times n_{l+1}} \underbrace{\frac{\partial \boldsymbol{z}_{l+1}}{\partial \boldsymbol{z}_l}}_{n_{l+1}\times n_l} = \left(\mathbf{J}_l \boldsymbol{g}_{l+1}\right)^\top$$

where $\mathbf{J}_l = \left(\frac{\partial \boldsymbol{z}_{l+1}}{\partial \boldsymbol{z}_l}\right)^\top$ is the Jacobian matrix and $\boldsymbol{g}_{l+1} = \left(\frac{\partial \mathcal{L}}{\partial \boldsymbol{z}_{l+1}}\right)^\top$ is the gradient at the next layer.

If $\mathbf{J}_l$ is constant across layer, then the gradient of loss wrt to layer l will be $\mathbf{J}^{L-l}\boldsymbol{g}_L$. Thus the behavior of the system depends on the eigenvectors of $\mathbf{J}$. Although $\mathbf{J}$ is a real-valued matrix, it is not (in general) symmetric, so its eigenvalues and eigenvectors can be complex-valued, with the imaginary components corresponding to oscillatory behavior. Let λ be the spectral radius of $\mathbf{J}$, which is the maximum of the absolute values of the eigenvalues. If this is greater than 1, the gradient can explode; if this is less than 1, the gradient can vanish.

The exploding gradient can be mitigated by gradient clipping, in which capping the magitude of the gradient if it becomes too large,

$$\boldsymbol{g}' = \max(1, \frac{c}{\|g\|})\boldsymbol{g}$$

the norm of $\boldsymbol{g}'$ can never exceed c.

The vanishing gradient problem is harder to solve, the following sections demonstrate several solutions: non-saturating activations, residual connections, parameter initialization.

7.3.3 Non-saturating activations

The sigmoid activation function saturates at 0 for large negative inputs, and at 1 for large positive inputs. It turns out that the gradient signal in these regimes is 0, preventing backpropagation from working

Consider a layer which computes $\boldsymbol{z} = \sigma(\boldsymbol{a}) = \sigma(\mathbf{W}\boldsymbol{x})$, where $\sigma(a) = \frac{1}{1+\exp(-a)}$. The derivative of the activation function (elementwise) is given by $\sigma'(a) = \sigma(a)(1-\sigma(a))$. Thus, the derivative of the output of the layer $\boldsymbol{z}$ wrt the input $\boldsymbol{x}$ is,

$$\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}} = \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{a}} \cdot \frac{\partial \boldsymbol{a}}{\partial \boldsymbol{x}} = \text{diag}(\boldsymbol{z} \odot (1-\boldsymbol{z})) \cdot \mathbf{W}$$

The derivative of the output of the layer $\boldsymbol{z}$ wrt the input $\mathbf{W}$ is,

$$\begin{aligned}\frac{\partial z_k}{\partial \mathbf{W}} = \frac{\partial z_k}{\partial a_k} \cdot \frac{\partial a_k}{\partial \mathbf{W}} &= \boldsymbol{e}_k^\top \text{diag}(\boldsymbol{z} \odot (1-\boldsymbol{z}))\boldsymbol{e}_k \cdot \boldsymbol{e}_k^\top \frac{\partial \boldsymbol{a}}{\partial \mathbf{W}} \\ &= z_k(1-z_k)\boldsymbol{e}_k\boldsymbol{x}^\top\end{aligned}$$

Since only k's row in $z_k(1-z_k)\boldsymbol{e}_k\boldsymbol{x}^\top$ is non-zero, so define $\frac{\partial \boldsymbol{z}}{\partial \mathbf{W}} = \sum_k \frac{\partial z_k}{\partial \mathbf{W}}$ which is a plain matrix instead of a full tensor form.

$$\frac{\partial \boldsymbol{z}}{\partial \mathbf{W}} = \sum_k z_k(1-z_k)\boldsymbol{e}_k\boldsymbol{x}^\top = \boldsymbol{z} \odot (1-\boldsymbol{z}) \cdot \boldsymbol{x}^\top$$

Hence, if z is near 0 or 1, the gradients wrt the parameters will go to 0.

Besides the saturating sigmoid, there are few types of non-saturating activation functions:

- ReLU. ReLU function turns off negative inputs, and passes positive inputs unchanged:

$$\text{ReLU}(a) = \max(a, 0) = a\mathbb{I}(a > 0)$$

$$\Rightarrow \text{ReLU}'(a) = \mathbb{I}(a > 0)$$

  Consider a layer to compute $\boldsymbol{z} = \text{ReLU}(\mathbf{W}\boldsymbol{x})$, the Jacobians look like:

$$\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}} = \mathbf{W}^\top \mathbb{I}(\boldsymbol{z} > 0), \quad \frac{\partial \boldsymbol{z}}{\partial \mathbf{W}} = \mathbb{I}(\boldsymbol{z} > 0)\boldsymbol{x}^\top$$

  Hence the gradient will not vanish, as long as $\boldsymbol{z}$ is positive. However, when weights are large and negative, there is a good chance that some element in $\mathbf{W}\boldsymbol{x}$ is large and negative, this will cause the gradient for the weights to go to 0 and cannot escape, i.e. dead ReLU problem.

- Leaky ReLU.

$$\text{LReLU}(a; \alpha) = \max(\alpha a, a) \quad \text{where } 0 < \alpha < 1$$

  The slope of this function is 1 for positive inputs, and α for negative inputs. If α is learnable rather than fixed, the leaky ReLU becomes parametric ReLU.

- ELU.

$$\text{ELU}(a; \alpha) = \begin{cases} \alpha(e^a - 1) & \text{if } a \leq 0 \\ a & \text{if } a > 0 \end{cases}$$

  ELU has the advantage of been a smooth function. A variant of ELU is self-normalising ELU,

$$\text{SELU}(a; \alpha, \lambda) = \lambda \text{ELU}(a; \alpha)$$

7.3.4 Residual connections

Another solution to the vanishing gradient problem for DNNs is to use a **residual network** or **ResNet**.

This is a feedforward model in which each layer has the form of a residual block, defined by

$$\mathcal{F}'_l(\boldsymbol{x}) = \mathcal{F}_l(\boldsymbol{x}) + \boldsymbol{x}$$

where $\mathcal{F}_l(\boldsymbol{x})$ is a standard shallow nonlinear mapping (eg. linear $\rightarrow$ activation $\rightarrow$ linear). Graphically,

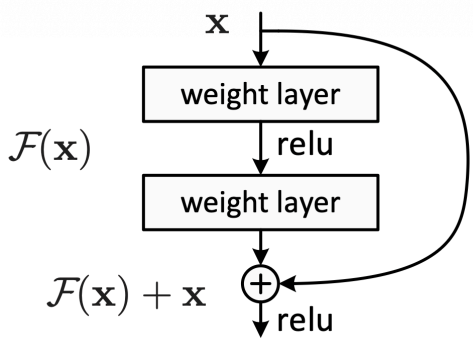


Figure 3: A residual block.

A model with residual connections has the same number of parameters as a model without residual connections, but it is easier to train. The reason is that gradients can flow directly from the output to earlier layers. Consider the activations of the output layer L, which can be derived in terms of any previous layer l as:

$$\boldsymbol{z}_L = \boldsymbol{z}_l + \sum_{i=l}^{L-1} \mathcal{F}_i(\boldsymbol{z}_i; \boldsymbol{\theta}_i)$$

Thus,

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}_l} &= \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}_l}\frac{\partial \boldsymbol{z}_l}{\partial \boldsymbol{\theta}_l} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}_L}\frac{\partial \boldsymbol{z}_L}{\partial \boldsymbol{z}_l}\frac{\partial \boldsymbol{z}_l}{\partial \boldsymbol{\theta}_l} \\
&= \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}_L}\left(1 + \sum_{i=l}^{L-1} \frac{\partial \mathcal{F}_i(\boldsymbol{z}_i; \boldsymbol{\theta}_i)}{\partial \boldsymbol{z}_l}\right)\frac{\partial \boldsymbol{z}_l}{\partial \boldsymbol{\theta}_l} \\
&= \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}_L}\frac{\partial \boldsymbol{z}_l}{\partial \boldsymbol{\theta}_l} + \text{other terms}
\end{aligned}$$

The gradient at layer l depends directly on the gradient at layer L, independent of the depth of the network.

7.3.5 Parameter initialization

The objective function for DNN training is non-convex, the way that we initialize the parameters affect the final optimization solution. It is shown that sampling parameters from a standard normal with fixed variance can result in exploding activations or gradients.

Consider a linear layer without an activation function given by $o_i = \sum_{j=1}^{n_{\text{in}}} w_{ij} x_j$; suppose $w_{ij} \sim \mathcal{N}(0, \sigma^2)$, and $\mathbb{E}[x_j] = 0$ and $\mathbb{V}[x_j] = \gamma^2$, where we assume x_j are independent of w_{ij}. The mean and variance of the output is given by

$$\begin{aligned}
\mathbb{E}[o_i] &= \sum_{j=1}^{n_{\text{in}}} \mathbb{E}[w_{ij} x_j] = \sum_{j=1}^{n_{\text{in}}} \mathbb{E}[w_{ij}]\mathbb{E}[x_j] = 0 \\
\mathbb{V}[o_i] &= \mathbb{E}[o_i^2] - (\mathbb{E}[o_i])^2 = \sum_{j=1}^{n_{\text{in}}} \mathbb{E}[w_{ij}^2 x_j^2] - 0 = \sum_{j=1}^{n_{\text{in}}} \mathbb{E}[w_{ij}^2]\mathbb{E}[x_j^2] = n_{\text{in}}\sigma^2\gamma^2
\end{aligned}$$

To keep the output variance from blowing up, we need to ensure $n_{\text{in}}\sigma^2 = 1$ (or some other constant), where n_{in} is the **fan-in** of a unit (number of incoming connections).

For the backwards pass, similarly, the variance of the gradients can blow up unless $n_{\text{out}}\sigma^2 = 1$, where n_{out} is the **fan-out** of a unit (number of outgoing connections). To satisfy both requirements at once, we set $\frac{1}{2}(n_{\text{in}} + n_{\text{out}})\sigma^2 = 1$, or equivalently

$$\sigma^2 = \frac{2}{n_{\text{in}} + n_{\text{out}}}$$

This is known as Xavier initialization or Glorot initialization.

Although the above derivation assumes Gaussian and linearity, the analysis applied to non-Gaussian initialization and nonlinear units. The best choice of initialization method depends on the activation function as well.

7.4 Regularization

One of the main statistical issues when training a neural network is overfitting, regularization is a way to avoid this. There are several regularization techniques.

Early stopping

The simplest way to prevent overfitting is called early stopping, which refers to the heuristic of stopping the training procedure when the error on the validation set starts to increase.

Weight decay

We can turn the maximum likelihood estimation into a maximum a posterior estimation by imposing a prior on the parameters. Commonly, using a Gaussian prior for the weights $\mathcal{N}(\boldsymbol{w}|\mathbf{0}, \alpha^2\mathbf{I})$ and the biases $\mathcal{N}(\boldsymbol{b}|\mathbf{0}, \beta^2\mathbf{I})$. This is called weight decay, since it encourages small weights by penalizing large wegihts.

Dropout

Dropout means randomly turn off all the outgoing connections from each neuron with probability p, on a per-example basis. Therefore, each neuron must learn to perform well even if some of the other units are missing at random, instead of learning complex but fragile dependencies on each other.

Denoting w_{lji} the real learned weight parameter through gradient descent, $\epsilon_{li} \sim \text{Ber}(1-p)$ is the random 0/1 mask resampled for every training example.

Viewing dropout as estimating a noisy version of the weights, $\theta_{lji} = w_{lji}\epsilon_{li}$, thus θ_{lji} is the effective weight that actually gets used in that one forward pass. In other words, 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.

At test time, we usually turn the noise off. 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 $w_{lji}^{\text{test}} = w_{lji}\mathbb{E}[\epsilon_{li}]$

We can keep the noise/dropouts during test as well. The result is an **ensemble** of networks, each with slightly different sparse graph structures. This is called **Monte Carlo dropout**,

$$p(\boldsymbol{y}|\boldsymbol{x}, \mathcal{D}) \approx \frac{1}{S}\sum_{s=1}^{S} p(\boldsymbol{y}|\boldsymbol{x}, \hat{\mathbf{W}}\epsilon^s + \hat{\boldsymbol{b}})$$

where S is the number of samples, and $\hat{\mathbf{W}}\epsilon^s$ indicates all the estimated weight matrices are multiplied by a sampled noise vector. This can sometimes provide a good approximation to the Bayesian posterior predictive distribution $p(\boldsymbol{y}|\boldsymbol{x}, \mathcal{D})$.

Regularization effects of (stochastic) gradient descent

Some optimization methods may finds sharp minima in the loss landscape (i.e. narrow but deep "hole"). However, such solutions generally correspond to a model that has overfit the data. It is better to flat minima, which are more robust and generalize better. 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. SGD often finds such flat minima by adding noises, which are called **implicit regularization**.

The continuous time gradient flow (ODE from $\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \epsilon \nabla \mathcal{L}(\boldsymbol{w}_t)$ as $\epsilon \to 0$) can be approximated by

$$\dot{\boldsymbol{w}} = -\nabla_{\boldsymbol{w}} \tilde{\mathcal{L}}_{GD}(\boldsymbol{w})$$

where

$$\tilde{\mathcal{L}}_{GD}(\boldsymbol{w}) = \mathcal{L}(\boldsymbol{w}) + \frac{\epsilon}{4} \|\nabla \mathcal{L}(\boldsymbol{w})\|^2$$

wherer $\mathcal{L}(\boldsymbol{w})$ is the original loss (finite $\epsilon > 0$), and the second term is an implicit regularization term that penalizes solutions with large gradients.

For the stochastic GD, the flow has the form $\dot{\boldsymbol{w}} = -\nabla_{\boldsymbol{w}} \tilde{\mathcal{L}}_{SGD}(\boldsymbol{w})$, where

$$\tilde{\mathcal{L}}_{SGD}(\boldsymbol{w}) = \mathcal{L}(\boldsymbol{w}) + \frac{\epsilon}{4m} \sum_{k=1}^{m} \|\nabla \mathcal{L}_k(\boldsymbol{w})\|^2$$

where m is the number of minibatches, and $\mathcal{L}_k(\boldsymbol{w})$ is the loss on the kth minibatch.

Comparing the loss of SGD to that of full-batch GD,

$$\tilde{\mathcal{L}}_{SGD}(\boldsymbol{w}) = \tilde{\mathcal{L}}_{GD}(\boldsymbol{w}) + \frac{\epsilon}{4m} \sum_{k=1}^{m} \|\nabla \mathcal{L}_k(\boldsymbol{w}) - \nabla \mathcal{L}(\boldsymbol{w})\|^2$$

The second term estimates the variance of the minibatch gradients, which is a measure of stability, and hence of generalization ability. Thus, SGD not only has computational advantages, but also statistical advantages.

8 Recurrent Neural Networks

A recurrent neural network (RNN) maps an input space of sequences to an output space of sequences in a stateful way, i.e. the prediction of output $\boldsymbol{y}_t$ depends on both the input $\boldsymbol{x}_t$ and the hidden state of the system $\boldsymbol{h}_t$, which gets updated over time as the sequence is processed.

8.1 Sequence generation: Vec2Seq

Vec2Seq mdoels map a vector to a sequence, which learn functions of the form: $f_{\boldsymbol{\theta}}$: $\mathbb{R}^D \to \mathbb{R}^{N_\infty C}$ with D is the size of the input vector, and the output is an arbitrary-length sequence of vectors each of size C.

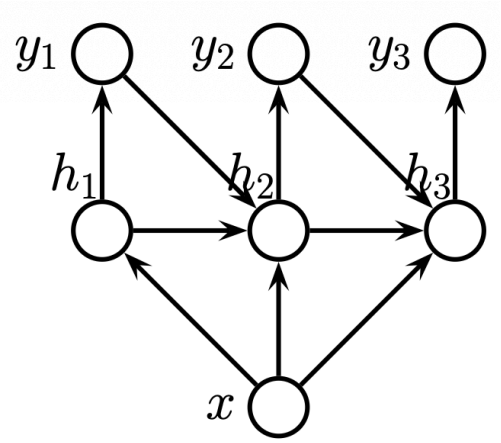


Figure 4: RNN for generating a variable length output sequence $\boldsymbol{y}_{1:T}$ given an optional fixed length input vector $\boldsymbol{x}$.

Let T be the length of the output, which is chosen dynamically. The RNN corresponds to the conditional generative model:

$$p(\boldsymbol{y}_{1:T}|\boldsymbol{x}) = \sum_{\boldsymbol{h}_{1:T}} p(\boldsymbol{y}_{1:T}, \boldsymbol{h}_{1:T}|\boldsymbol{x}) = \sum_{\boldsymbol{h}_{1:T}} \prod_{t=1}^{T} p(\boldsymbol{y}_t|\boldsymbol{h}_t) p(\boldsymbol{h}_t|\boldsymbol{h}_{t-1}, \boldsymbol{y}_{t-1}, \boldsymbol{x})$$

where $\boldsymbol{h}_t$ is the hidden state and $p(\boldsymbol{h}_1|\boldsymbol{h}_0, \boldsymbol{y}_0, \boldsymbol{x}) = p(\boldsymbol{h}_1|\boldsymbol{x})$ is the initial hidden state distribution (often deterministic). RNN can be used to generate sequences unconditionally ($\boldsymbol{x} = \emptyset$) or conditionally on $\boldsymbol{x}$.

- The output distribution $p(\boldsymbol{y}_t|\boldsymbol{h}_t)$ is defined as

$$p(\boldsymbol{y}_t|\boldsymbol{h}_t) = \text{Cat}(\boldsymbol{y}_t|\text{softmax}(\mathbf{W}_{hy}\boldsymbol{h}_t + \boldsymbol{b}_y))$$
$$p(\boldsymbol{y}_t|\boldsymbol{h}_t) = \mathcal{N}(\boldsymbol{y}_t|\mathbf{W}_{hy}\boldsymbol{h}_t + \boldsymbol{b}_y, \sigma^2\mathbf{I})$$

  for discrete or continuous output. $\mathbf{W}_{hy}$ is the hidden-to-output weights, and $\boldsymbol{b}_y$ is the bias.

- The hidden state distribution $p(\boldsymbol{h}_t|\boldsymbol{h}_{t-1}, \boldsymbol{y}_{t-1}, \boldsymbol{x})$ is assumed to be deterministic here.

$$p(\boldsymbol{h}_t|\boldsymbol{h}_{t-1}, \boldsymbol{y}_{t-1}, \boldsymbol{x}) = \mathbb{I}(\boldsymbol{h}_t = f(\boldsymbol{h}_{t-1}, \boldsymbol{y}_{t-1}, \boldsymbol{x}))$$

for some deterministic function f. The update function f is usually given by

$$\boldsymbol{h}_t = \varphi(\mathbf{W}_{xh}[\boldsymbol{x}; \boldsymbol{y}_{t-1}] + \mathbf{W}_{hh}\boldsymbol{h}_{t-1} + \boldsymbol{b}_h)$$

where $\mathbf{W}_{hh}$ are the hidden-to-hidden weights, $\mathbf{W}_{xh}$ are the input-to-hidden weights, and $\boldsymbol{b}_h$ are the bias terms. $[\boldsymbol{x}; \boldsymbol{y}_{t-1}]$ means stacking two vectors together to form a larger vector.

Note that $\boldsymbol{y}_t$ depends on $\boldsymbol{h}_t$, which depends on $\boldsymbol{y}_{t-1}$, which depends on $\boldsymbol{h}_{t-1}$, and so on. Thus, $\boldsymbol{y}_t$ implicitly depends on all past observations and $\boldsymbol{x}$.

Practically, when generating from RNN, first sample $\tilde{\boldsymbol{y}}_t = p(\boldsymbol{y}_t|\boldsymbol{h}_t)$, and then feed in the sampled value to the hidden state, to deterministically compute $\boldsymbol{h}_{t+1} = f(\boldsymbol{h}_t, \tilde{\boldsymbol{y}}_t, \boldsymbol{x})$, from which $\tilde{\boldsymbol{y}}_{t+1}$ is sampled, and so on. Current RNN model only has stochasticity in $p(\boldsymbol{y}_t|\boldsymbol{h}_t)$, the variational RNN adds stochasticity to $p(\boldsymbol{h}_t|\boldsymbol{h}_{t-1}, \boldsymbol{y}_{t-1}, \boldsymbol{x})$.

8.2 Sequence classification: Seq2Vec

Seq2Vec models map a sequence to a vector, which learn functions of the form: $f_{\boldsymbol{\theta}}$: $\mathbb{R}^{TD} \to \mathbb{R}^C$. In other words, Seq2Vec have a single fixed-length output vector $\boldsymbol{y}$ given a variable length sequence as input.

The simplest approach is to use the final state of the RNN as input to the classifier:

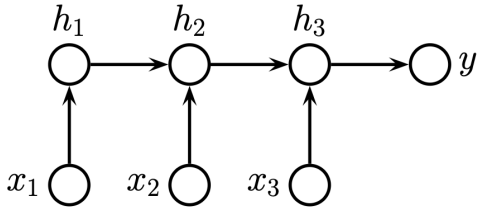


Figure 5: Simple Seq2Vec RNN for sequence classification.

with a generative model

$$p(y|\boldsymbol{x}_{1:T}) = \text{Cat}(y|\text{softmax}(\mathbf{W}\boldsymbol{h}_T))$$

where $y \in \{1, \cdots, C\}$ are class labels for notational simplicity.

Another variant called **bidirectional RNN** gives better results by letting the hidden states depend on the past and future context.

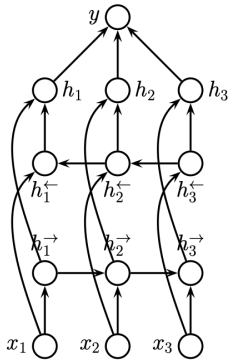


Figure 6: Bidirectional Seq2Vec RNN for sequence classification.

The hidden states are calculated deterministically via:

$$\boldsymbol{h}_t^{\rightarrow} = \varphi(\mathbf{W}_{xh}^{\rightarrow}\boldsymbol{x}_t + \mathbf{W}_{hh}^{\rightarrow}\boldsymbol{h}_{t-1}^{\rightarrow} + \boldsymbol{b}_h^{\rightarrow})$$

$$\boldsymbol{h}_t^{\leftarrow} = \varphi(\mathbf{W}_{xh}^{\leftarrow}\boldsymbol{x}_t + \mathbf{W}_{hh}^{\leftarrow}\boldsymbol{h}_{t+1}^{\leftarrow} + \boldsymbol{b}_h^{\leftarrow})$$

Define $\boldsymbol{h}_t = [\boldsymbol{h}_t^{\rightarrow}, \boldsymbol{h}_t^{\leftarrow}]$ to be the representation of the state at time t, taking into account past and future information. Then pool over these hidden states to get the final classifier:

$$p(y|\boldsymbol{x}_{1:T}) = \text{Cat}(y|\text{softmax}(\mathbf{W}\overline{\boldsymbol{h}}))$$

$$\overline{\boldsymbol{h}} = \frac{1}{T}\sum_{t=1}^{T}\boldsymbol{h}_t$$

8.3 Sequence translation: Seq2Seq

Seq2Seq models map a sequence to a sequence, which learn functions of the form: $f_{\boldsymbol{\theta}}$: $\mathbb{R}^{TD} \rightarrow \mathbb{R}^{T'C}$.

Aligned case ($T = T'$)

For the aligned case, the RNN predicts one label per position. The generative model looks like:

$$\begin{aligned} p(\boldsymbol{y}_{1:T}|\boldsymbol{x}_{1:T}) &= \sum_{\boldsymbol{h}_{1:T}}\prod_{t=1}^{T} p(\boldsymbol{y}_t|\boldsymbol{h}_t)p(\boldsymbol{h}_t|\boldsymbol{h}_{t-1}, \boldsymbol{x}_t) \\ &= \sum_{\boldsymbol{h}_{1:T}}\prod_{t=1}^{T} p(\boldsymbol{y}_t|\boldsymbol{h}_t)\mathbb{I}(\boldsymbol{h}_t = f(\boldsymbol{h}_{t-1}, \boldsymbol{x}_t)) \end{aligned}$$

with initial hidden state pre-defined as $\boldsymbol{h}_1 = f(\boldsymbol{h}_0, \boldsymbol{x}_1) = f_0(\boldsymbol{x}_1)$. Note that $\boldsymbol{y}_t$ depends on $\boldsymbol{h}_t$ which depends only on the past input $\boldsymbol{x}_{1:t}$, corresponding to (a) below. Bidirectinal Seq2Seq RNN can also be used to enable the model looks into the "future" of $\boldsymbol{x}$ as well as the past, corresponding to (b) below.

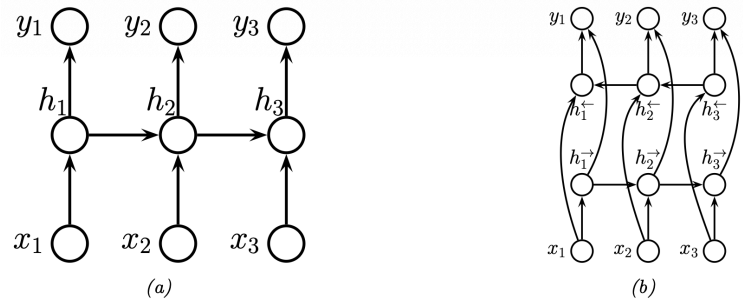


Figure 7: Seq2Seq RNN model.

Stacking multiple hidden chains on top of each other can produce more expressive RNN model.

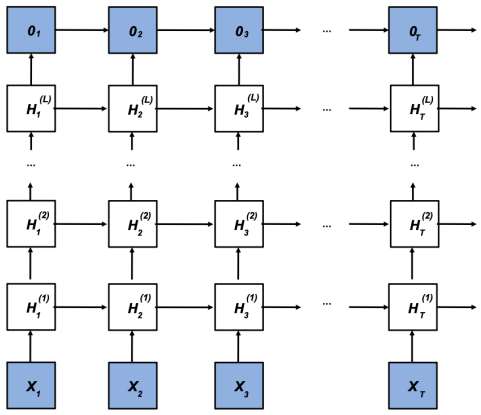


Figure 8

The hidden units for layer l at time t are computed using

$$\boldsymbol{h}_t^l = \varphi_l(\mathbf{W}_{xh}^l \boldsymbol{h}_t^{l-1} + \mathbf{W}_{hh}^l \boldsymbol{h}_{t-1}^l + \boldsymbol{b}_h^l)$$

The output is given by

$$\boldsymbol{o}_t = \mathbf{W}_{ho} \boldsymbol{h}_t^L + \boldsymbol{b}_o$$

Unaligned case ($T \neq T'$)

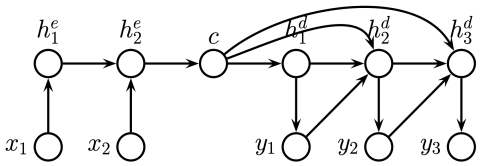


Figure 9: Encoder-decoder RNN architecture for mapping sequence $\boldsymbol{x}_{1:T}$ to sequence $\boldsymbol{y}_{1:T'}$

For unaligned cases, first encode the input sequence to get the context vector $\boldsymbol{c} = f_e(\boldsymbol{x}_{1:T})$, using the last state of an RNN (or average pooling over a biRNN). Then generate the output sequence using an RNN decoder $\boldsymbol{y}_{1:T'} = f_d(\boldsymbol{c})$

8.4 Teacher Forcing

When training a language model, the likelihood of a sequence of words $w_1, w_2, \ldots, w_T$, is given by

$$p(\boldsymbol{w}_{1:T}) = \prod_{t=1}^{T} p(w_t \mid \boldsymbol{w}_{1:t-1})$$

In an RNN, we therefore set the input to $x_t = w_{t-1}$ and the output to $y_t = w_t$. Note that we condition on the *ground truth* labels from the past, $\boldsymbol{w}_{1:t-1}$, not labels generated from the model. This is called **teacher forcing**, since the teacher's values are "force fed" into the model as input at each step (i.e., x_t is set to w_{t-1}).

Since all the inputs $\boldsymbol{x}_t$ are known in advance from the data, the prediction errors do not propagate through the inputs during training: a wrong prediction at step t does not change $\boldsymbol{x}_{t+1}$. This makes training stable, and all the per-step losses can be evaluated in a single forward pass.

However, at test time the true tokens are not available, so the model has to consume its own predictions. This mismatch between training and inference inputs is called exposure bias, and it means errors can compound during generation.

8.5 Backpropagation Through Time (BPTT)

To compute the maximum likelihood estimate of the parameters for an RNN,

$$\boldsymbol{\theta}^* = \underset{\boldsymbol{\theta}}{\operatorname{argmax}}\, p(\boldsymbol{y}_{1:T}|\boldsymbol{x}_{1:T}, \boldsymbol{\theta})$$

To compute the gradient of loss wrt the parameters for MLE, we need to unroll the computation graph of RNNs.

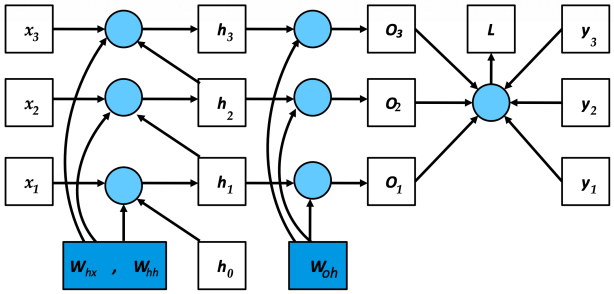


Figure 10: An RNN unrolled for 3 time steps, with the target output sequence and shown.

Specifically, the model is:

$$\boldsymbol{h}_t = \mathbf{W}_{hx}\boldsymbol{x}_t + \mathbf{W}_{hh}\boldsymbol{h}_{t-1}$$
$$\boldsymbol{o}_t = \mathbf{W}_{ho}\boldsymbol{h}_t$$

where $\boldsymbol{o}_t$ are the output logits, and the bias terms are dropped for notational simplicity. Assuming a single training sequence and $\boldsymbol{y}_t$ is the true target for each time step, so the loss is

$$L = \frac{1}{T}\sum_{t=1}^{T} \ell(\boldsymbol{y}_t, \boldsymbol{o}_t)$$

The derivative of loss wrt $\mathbf{W}_{ho}$ is

$$\frac{\partial L}{\partial \mathbf{W}_{ho}} = \frac{1}{T}\sum_{t=1}^{T} \frac{\partial \ell(\boldsymbol{y}_t, \boldsymbol{o}_t)}{\partial \boldsymbol{o}_t}\frac{\partial \boldsymbol{o}_t}{\partial \mathbf{W}_{ho}} = \frac{1}{T}\sum_{t=1}^{T} \frac{\partial \ell(\boldsymbol{y}_t, \boldsymbol{o}_t)}{\partial \boldsymbol{o}_t}\boldsymbol{h}_t^\top$$

The derivative of loss wrt $\mathbf{W}_{hx}$ and $\mathbf{W}_{hh}$ should be derived by flattening and stacking $\mathbf{W}_{hx}$ and $\mathbf{W}_{hh}$ to $\boldsymbol{w}_h$. Thus, $\boldsymbol{h}_t = f(\boldsymbol{x}_t, \boldsymbol{h}_{t-1}, \boldsymbol{w}_h)$ and $\boldsymbol{o}_t = g(\boldsymbol{h}_t, \boldsymbol{w}_o)$. Then,

$$\frac{\partial L}{\partial \boldsymbol{w}_h} = \frac{1}{T}\sum_{t=1}^{T} \frac{\partial \ell(y_t, \boldsymbol{o}_t)}{\partial \boldsymbol{w}_h} = \frac{1}{T}\sum_{t=1}^{T} \frac{\partial \ell(y_t, \boldsymbol{o}_t)}{\partial \boldsymbol{o}_t}\frac{\partial g(\boldsymbol{h}_t, \boldsymbol{w}_o)}{\partial \boldsymbol{h}_t}\frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{w}_h}$$

where

$$\begin{aligned}\frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{w}_h} &= \frac{\partial f(\boldsymbol{x}_t, \boldsymbol{h}_{t-1}, \boldsymbol{w}_h)}{\partial \boldsymbol{w}_h} + \frac{\partial f(\boldsymbol{x}_t, \boldsymbol{h}_{t-1}, \boldsymbol{w}_h)}{\partial \boldsymbol{h}_{t-1}}\frac{\partial \boldsymbol{h}_{t-1}}{\partial \boldsymbol{w}_h} \\ &= \frac{\partial f(\boldsymbol{x}_t, \boldsymbol{h}_{t-1}, \boldsymbol{w}_h)}{\partial \boldsymbol{w}_h} + \sum_{i=1}^{t-1}\left(\prod_{j=i+1}^{t}\frac{\partial f(\boldsymbol{x}_j, \boldsymbol{h}_{j-1}, \boldsymbol{w}_h)}{\partial \boldsymbol{h}_{j-1}}\right)\frac{\partial f(\boldsymbol{x}_i, \boldsymbol{h}_{i-1}, \boldsymbol{w}_h)}{\partial \boldsymbol{w}_h}\end{aligned}$$

It is standard to truncate the sum to the most recent K terms for saving computations.

8.6 Vanishing and Exploding Gradients

The activations in an RNN can decay or explode as going forwards in time, since $\mathbf{W}_{hh}$ is multiplied at each time step. Similarly, the gradients in an RNN can decay or explode as going backwards in time, since the Jacobians is multiplied at each time step (by chain rule).

A way to solve exploding gradient is to use gradient clipping

$$\boldsymbol{g}' = \max(1, \frac{c}{\|g\|})\boldsymbol{g}$$

Or there are some methods to control the spectral radius λ of the forward mapping $\mathbf{W}_{hh}$ and the backward mapping $\mathbf{J}_{hh}$. The simplest way to control the spectral radius is to randomly initialize $\mathbf{W}_{hh}$ in such a way as to ensure $\lambda \approx 1$, and then keep it fixed (i.e., we do not learn $\mathbf{W}_{hh}$). In this case, only the output matrix $\mathbf{W}_{ho}$ needs to be learned, resulting in a convex optimization problem. Or we can also learn $\mathbf{W}_{hh}$ by constraining it to be an orthogonal matrix (i.e. eigenvalues are 1).

An alternative to explicitly controlling the spectral radius is to modify the RNN architecture itself, to use additive rather than multiplicative updates to the hidden states, similar to the resnet idea.

Gated recurrent unit (GRU)

Assuming $\mathbf{X}_t \in \mathbb{R}^{D\times N}$, where N is the batch size and D is the input size. The vanilla RNN updates the hidden states via,

$$\mathbf{H}_t = \varphi\big(\mathbf{W}_{hx}\mathbf{X}_t + \mathbf{W}_{hh}\mathbf{H}_{t-1} + \boldsymbol{b}\big)$$

where $\mathbf{W}_{hx} \in \mathbb{R}^{H\times D}$ and $\mathbf{W}_{hh} \in \mathbb{R}^{H\times H}$, $\boldsymbol{b}$ is a bias vector which get broadcasted before adding to the matrices.

Instead of overwriting the hidden state, the GRU lets the network learn to selectively remember or forget some pieces of information in the past. Defining a new set of learnable parameters,

$$\begin{aligned}\mathbf{W}_{xr},\, \mathbf{W}_{xz},\, \mathbf{W}_{xh} &\in \mathbb{R}^{H\times D}\\ \mathbf{W}_{hr},\, \mathbf{W}_{hz},\, \mathbf{W}_{hh} &\in \mathbb{R}^{H\times H}\\ \boldsymbol{b}_r,\, \boldsymbol{b}_z,\, \boldsymbol{b}_h &\in \mathbb{R}^{H\times 1}\end{aligned}$$

Then, define two gates, with each element in $[0, 1]$

$$\begin{aligned}\mathbf{R}_t &= \sigma(\mathbf{W}_{xr}\mathbf{X}_t + \mathbf{W}_{hr}\mathbf{H}_{t-1} + \boldsymbol{b}_r) \in \mathbb{R}^{H\times N}\\ \mathbf{Z}_t &= \sigma(\mathbf{W}_{xz}\mathbf{X}_t + \mathbf{W}_{hz}\mathbf{H}_{t-1} + \boldsymbol{b}_z) \in \mathbb{R}^{H\times N}\end{aligned}$$

Given these gates, define a candidate state,

$$\tilde{\mathbf{H}}_t = \tanh(\mathbf{W}_{xh}\mathbf{X}_t + \mathbf{W}_{hh}\big(\mathbf{R}_t \odot \mathbf{H}_{t-1}\big) + \boldsymbol{b}_h)$$

If the entries of the reset gate $\mathbf{R}_t$ are close to 1, it is the standard RNN update rule; if the entries are close to 0, the model acts more like an MLP applied to $\mathbf{X}_t$. Thus, the candidate state captures the short-term dependencies. Then, the actual hidden state is,

$$\mathbf{H}_t = \mathbf{Z}_t \odot \mathbf{H}_{t-1} + \big(1 - \mathbf{Z}_t\big) \odot \tilde{\mathbf{H}}_t$$

This captures the long-term dependencies.

Long short term memory (LSTM)

The long short term memory (LSTM) model augments the hidden state with a memory cell $\mathbf{C}_t \in \mathbb{R}^{H\times N}$, which acts as long-term memory. Assuming $\mathbf{X}_t \in \mathbb{R}^{D\times N}$, where N is the batch size and D is the input size, define a new set of learnable parameters,

$$\mathbf{W}_{xo},\, \mathbf{W}_{xi},\, \mathbf{W}_{xf},\, \mathbf{W}_{xc} \in \mathbb{R}^{H\times D}$$
$$\mathbf{W}_{ho},\, \mathbf{W}_{hi},\, \mathbf{W}_{hf},\, \mathbf{W}_{hc} \in \mathbb{R}^{H\times H}$$
$$\boldsymbol{b}_o,\, \boldsymbol{b}_i,\, \boldsymbol{b}_f,\, \boldsymbol{b}_c \in \mathbb{R}^{H\times 1}$$

where the biases are broadcasted before adding to the matrices.

Then, define three gates, with each element in $[0, 1]$: the output gate $\mathbf{O}_t$ determines what gets read out, the input gate $\mathbf{I}_t$ determines what gets written in, and the forget gate $\mathbf{F}_t$ determines how much of the old cell is kept,

$$\mathbf{O}_t = \sigma(\mathbf{W}_{xo}\mathbf{X}_t + \mathbf{W}_{ho}\mathbf{H}_{t-1} + \boldsymbol{b}_o) \in \mathbb{R}^{H\times N}$$
$$\mathbf{I}_t = \sigma(\mathbf{W}_{xi}\mathbf{X}_t + \mathbf{W}_{hi}\mathbf{H}_{t-1} + \boldsymbol{b}_i) \in \mathbb{R}^{H\times N}$$
$$\mathbf{F}_t = \sigma(\mathbf{W}_{xf}\mathbf{X}_t + \mathbf{W}_{hf}\mathbf{H}_{t-1} + \boldsymbol{b}_f) \in \mathbb{R}^{H\times N}$$

Given these gates, define a candidate cell state,

$$\tilde{\mathbf{C}}_t = \tanh(\mathbf{W}_{xc}\mathbf{X}_t + \mathbf{W}_{hc}\mathbf{H}_{t-1} + \boldsymbol{b}_c)$$

The actual cell update keeps part of the old cell (controlled by the forget gate) and adds part of the candidate (controlled by the input gate),

$$\mathbf{C}_t = \mathbf{F}_t \odot \mathbf{C}_{t-1} + \mathbf{I}_t \odot \tilde{\mathbf{C}}_t$$

If $\mathbf{F}_t$ is close to 1 and $\mathbf{I}_t$ is close to 0, the cell passes through unchanged, so the model can remember long-term information.

Finally, the hidden state is a transformed version of the cell, provided the output gate is on,

$$\mathbf{H}_t = \mathbf{O}_t \odot \tanh(\mathbf{C}_t)$$

Note that $\mathbf{H}_t$ is used both as the output of the unit and as the hidden state for the next time step. Thus $\mathbf{H}_t$ captures short-term memory (what the model has just output), whereas the cell $\mathbf{C}_t$ acts as long-term memory.

8.7 RNN Generation

The per-step predictive distribution of a RNN model is used for generation,

$$p(\boldsymbol{y}_t = \boldsymbol{y} \mid \hat{\boldsymbol{y}}_{1:t-1}, \boldsymbol{x}_{1:t}) = p(\boldsymbol{y}_t = \boldsymbol{y} \mid \boldsymbol{h}_t) = \text{softmax}(\mathbf{W}_{ho}\boldsymbol{h}_t)_{\boldsymbol{y}}$$

There are several ways to generate $\hat{\boldsymbol{y}}_t$,

- Greedy decoding.
$$\hat{\boldsymbol{y}}_t = \underset{\boldsymbol{y}}{\text{argmax}}\, p(\boldsymbol{y}_t = \boldsymbol{y} \mid \hat{\boldsymbol{y}}_{1:t-1}, \boldsymbol{x}_{1:t})$$
- Sampling $\hat{\boldsymbol{y}}_t$ from $p(\boldsymbol{y}_t = \boldsymbol{y} \mid \hat{\boldsymbol{y}}_{1:t-1}, \boldsymbol{x}_{1:t})$ directly. Optionally with temperature τ (using $\text{softmax}(\boldsymbol{o}_t/\tau)$) or truncated to the top-k or top-p tokens first.
- Beam search. Keep the top K partial sequences (beam width K), scored by their cumulative log-probability. At each step, expand each of them with all V possible next tokens (V is the vocabulary size) to get VK candidates, and keep the top K again. Each candidate carries its own hidden state. Greedy decoding is the special case $K = 1$.

Part III
Dimentionality Reduction

9 Principal Component Analysis

PCA algorithm is to find a **linear and orthogonal** projection of the high dimensional data $x \in \mathbb{R}^D$, to a low dimensional subspace $z \in \mathbb{R}^L$, such that the low dimensional representation is a "good approximation" to the original data. If we project or encode $\mathbf{x}$ to get $\mathbf{z} = \mathbf{W^T x}$, and then unproject or decode $\mathbf{z}$ to get $\widehat{\mathbf{x}} = \mathbf{Wz}$, then we want $\mathbf{x}$ to be close to $\mathbf{x}$ in ℓ_2 distance. In particular, we can define the following **reconstruction error**:

$$\mathcal{L}(\mathbf{W}) \triangleq \frac{1}{N}\sum_{n=1}^{N} \|\mathbf{x}_n - \text{decode}(\text{encode}(\mathbf{x}_n; \mathbf{W}); \mathbf{W})\|_2^2$$

PCA algorithm states that this objective can be minimized by setting $\mathbf{W} = \mathbf{U}_L$, where $\mathbf{U}_L$ contains the the L eigenvectors with lages eigenvalues of the empirical covariance matrix

$$\hat{\mathbf{\Sigma}} = \frac{1}{N}\sum_{n=1}^{N}(\boldsymbol{x}_n - \overline{\boldsymbol{x}})(\boldsymbol{x}_n - \overline{\boldsymbol{x}})^\mathsf{T} = \frac{1}{N}\mathbf{X}_c^\mathsf{T}\mathbf{X}_c$$

where $\mathbf{X}_c$ is a centered version of the $N \times D$ design matrix.

9.1 PCA setup

Suppose we have an (unlabeled) dataset $\mathcal{D} = \{\boldsymbol{x}_n : n = 1 : N\}$, where $\boldsymbol{x}_n \in \mathbb{R}^D$. We can represent this as an $N \times D$ data matrix $\mathbf{X}$. We will assume $\overline{\boldsymbol{x}} = \frac{1}{N}\sum_{n=1}^{N} \boldsymbol{x}_n = \mathbf{0}$, which we can ensure by centering the data.

We would like to approximate each $\boldsymbol{x}_n$ by a low dimensional representation, $\boldsymbol{z}_n \in \mathbb{R}^L$. We assume that each $\boldsymbol{x}_n$ can be explained in terms of a weighted combination of basis functions $\boldsymbol{w}_1, \ldots, \boldsymbol{w}_L$, where each $\boldsymbol{w}_k \in \mathbb{R}^D$, and where the weights are given by $\boldsymbol{z}_n \in \mathbb{R}^L$, i.e.

$$\boldsymbol{x}_n \approx \sum_{k=1}^{L} z_{nk}\boldsymbol{w}_k$$

The vector $\boldsymbol{z}_n$ is the low dimensional representation of $\boldsymbol{x}_n$, and is known as the **latent vector**, the collection of these latent variables are called the **latent factors**.

The error produced by this approximation:

$$\mathcal{L}(\mathbf{W}, \mathbf{Z}) = \frac{1}{N}\|\mathbf{X} - \mathbf{Z}\mathbf{W}^\mathsf{T}\|_F^2 = \frac{1}{N}\|\mathbf{X}^\mathsf{T} - \mathbf{W}\mathbf{Z}^\mathsf{T}\|_F^2 = \frac{1}{N}\sum_{n=1}^{N}\|\boldsymbol{x}_n - \mathbf{W}\boldsymbol{z}_n\|^2$$

where

- $\mathbf{X} \in \mathbb{R}^{N\times D}$: data matrix, rows are centered data points $\mathbf{x}_n^T$
- $\mathbf{Z} \in \mathbb{R}^{N\times L}$: latent code matrix, rows are centered data points $\mathbf{z}_n^T$
- $\mathbf{W} \in \mathbb{R}^{D\times L}$: basis matrix, columns $\boldsymbol{w}_1, \ldots, \boldsymbol{w}_L$ are orthonormal
- approximating each $\boldsymbol{x}_n$ by $\hat{\boldsymbol{x}}_n = \mathbf{W}\boldsymbol{z}_n$; or, $\hat{\mathbf{X}} = \mathbf{Z}\mathbf{W}^T$.
- the minimization of $\mathcal{L}$ is subject to constraints $\mathbf{w}_k^T\mathbf{w_j} = 0 \,\text{for}\, k \neq j; 1 \,\text{for}\, k = j$, i.e., $\mathbf{W}$ is orthogonal

9.2 Algorithm derivation

Base case

Starting by considering the best 1d solution , $\mathbf{w}_1 \in \mathbb{R}^D$, letting the coefficients for each datapoints associated with the first basis vector be denoted by $\tilde{\mathbf{z}}_1 = [z_{11}, \ldots, z_{N1}] \in \mathbb{R}^N$. The reconstruction error is

$$\begin{aligned}
\mathcal{L}(\boldsymbol{w}_1, \tilde{\boldsymbol{z}}_1) &= \frac{1}{N}\sum_{n=1}^{N} \|\boldsymbol{x}_n - z_{n1}\boldsymbol{w}_1\|^2 = \frac{1}{N}\sum_{n=1}^{N} (\boldsymbol{x}_n - z_{n1}\boldsymbol{w}_1)^\mathsf{T}(\boldsymbol{x}_n - z_{n1}\boldsymbol{w}_1) \\
&= \frac{1}{N}\sum_{n=1}^{N} [\boldsymbol{x}_n^\mathsf{T}\boldsymbol{x}_n - 2z_{n1}\boldsymbol{w}_1^\mathsf{T}\boldsymbol{x}_n + z_{n1}^2\boldsymbol{w}_1^\mathsf{T}\boldsymbol{w}_1] \\
&= \frac{1}{N}\sum_{n=1}^{N} [\boldsymbol{x}_n^\mathsf{T}\boldsymbol{x}_n - 2z_{n1}\boldsymbol{w}_1^\mathsf{T}\boldsymbol{x}_n + z_{n1}^2] \quad \text{since}\, \boldsymbol{w}_1^\mathsf{T}\boldsymbol{w}_1 = 1
\end{aligned}$$

Taking derivatives wrt z_{n1} and equating to zero gives

$$\frac{\partial}{\partial z_{n1}}\mathcal{L}(\boldsymbol{w}_1, \tilde{\boldsymbol{z}}_1) = \frac{1}{N}[-2\boldsymbol{w}_1^\mathsf{T}\boldsymbol{x}_n + 2z_{n1}] = 0 \Rightarrow z_{n1} = \boldsymbol{w}_1^\mathsf{T}\boldsymbol{x}_n$$

So the optimal embedding is obtained by orthogonally projecting the data onto $\boldsymbol{w}_1$. Plugging this back in gives the loss for the weights:

$$\mathcal{L}(\boldsymbol{w}_1) = \mathcal{L}(\boldsymbol{w}_1, \tilde{\boldsymbol{z}}_1^*(\boldsymbol{w}_1)) = \frac{1}{N}\sum_{n=1}^{N} [\boldsymbol{x}_n^\mathsf{T}\boldsymbol{x}_n - z_{n1}^2] = \text{const} - \frac{1}{N}\sum_{n=1}^{N} z_{n1}^2$$

To solve for $\boldsymbol{w}_1$, note that

$$\mathcal{L}(\boldsymbol{w}_1) = -\frac{1}{N}\sum_{n=1}^{N} z_{n1}^2 = -\frac{1}{N}\sum_{n=1}^{N} \boldsymbol{w}_1^\mathsf{T}\boldsymbol{x}_n\boldsymbol{x}_n^\mathsf{T}\boldsymbol{w}_1 = -\boldsymbol{w}_1^\mathsf{T}\hat{\boldsymbol{\Sigma}}\boldsymbol{w}_1$$

where $\boldsymbol{\Sigma}$ is the empirical covariance matrix (since we assumed the data is centered). We can trivially optimize this by letting $\|\boldsymbol{w}_1\| \to \infty$, so we impose the constraint $\|\boldsymbol{w}_1\| = 1$ and instead optimize

$$\tilde{\mathcal{L}}(\boldsymbol{w}_1) = \boldsymbol{w}_1^\mathsf{T}\hat{\boldsymbol{\Sigma}}\boldsymbol{w}_1 - \lambda_1(\boldsymbol{w}_1^\mathsf{T}\boldsymbol{w}_1 - 1)$$

where λ_1 is a Lagrange multiplier. Taking derivatives and equating to zero we have

$$\frac{\partial}{\partial \boldsymbol{w}_1}\tilde{\mathcal{L}}(\boldsymbol{w}_1) = 2\hat{\boldsymbol{\Sigma}}\boldsymbol{w}_1 - 2\lambda_1\boldsymbol{w}_1 = 0$$

$$\hat{\boldsymbol{\Sigma}}\boldsymbol{w}_1 = \lambda_1\boldsymbol{w}_1$$

Hence the optimal direction onto which we should project the data is an eigenvector of the covariance matrix. Left multiplying by $\boldsymbol{w}_1^\mathsf{T}$ (and using $\boldsymbol{w}_1^\mathsf{T}\boldsymbol{w}_1 = 1$) we find

$$\boldsymbol{w}_1^\mathsf{T}\hat{\boldsymbol{\Sigma}}\boldsymbol{w}_1 = \lambda_1$$

Since we want to maximize this quantity (minimize the loss), we pick the eigenvector which corresponds to the largest eigenvalue.

Induction Step

For another direction $\mathbf{w}_2$, the reconstructon error is

$$\mathcal{L}(\boldsymbol{w}_1, \tilde{\boldsymbol{z}}_1, \boldsymbol{w}_2, \tilde{\boldsymbol{z}}_2) = \frac{1}{N}\sum_{n=1}^{N} \|\boldsymbol{x}_n - z_{n1}\boldsymbol{w}_1 - z_{n2}\boldsymbol{w}_2\|^2$$

Optimizing wrt $\boldsymbol{w}_1$ and $\boldsymbol{z}_1$ gives the same solution as before. $\frac{\partial \mathcal{L}}{\partial \boldsymbol{z}_2} = 0$ yields $z_{n2} = \boldsymbol{w}_2^\mathsf{T}\boldsymbol{x}_n$. Substituting in yields

$$\mathcal{L}(\boldsymbol{w}_2) = \frac{1}{N}\sum_{n=1}^{N}[\boldsymbol{x}_n^\mathsf{T}\boldsymbol{x}_n - \boldsymbol{w}_1^\mathsf{T}\boldsymbol{x}_n\boldsymbol{x}_n^\mathsf{T}\boldsymbol{w}_1 - \boldsymbol{w}_2^\mathsf{T}\boldsymbol{x}_n\boldsymbol{x}_n^\mathsf{T}\boldsymbol{w}_2] = \text{const} - \boldsymbol{w}_2^\mathsf{T}\hat{\boldsymbol{\Sigma}}\boldsymbol{w}_2$$

Dropping the constant term, plugging in the optimal $\boldsymbol{w}_1$ and adding the constraints yields

$$\tilde{\mathcal{L}}(\boldsymbol{w}_2) = -\boldsymbol{w}_2^\mathsf{T}\hat{\boldsymbol{\Sigma}}\boldsymbol{w}_2 + \lambda_2(\boldsymbol{w}_2^\mathsf{T}\boldsymbol{w}_2 - 1) + \lambda_{12}(\boldsymbol{w}_2^\mathsf{T}\boldsymbol{w}_1 - 0)$$

with the extra constraint term, it can be shown that the solution is given by the eigenvector with the second largest eigenvalue:

$$\hat{\boldsymbol{\Sigma}}\boldsymbol{w}_2 = \lambda_2\boldsymbol{w}_2$$

The proof continues in this way to show that $\hat{\mathbf{W}} = \mathbf{U}_L$.

9.3 Computational tricks

1. **High-dimensional data.** PCA finds the eigenvectors of the $D \times D$ covariance matrix $\mathbf{X}^\mathsf{T}\mathbf{X}$. If $D > N$, it is faster to work with the $N \times N$ matrix $\mathbf{X}\mathbf{X}^\mathsf{T}$. Let $\mathbf{U}$ be an orthonormal matrix containing the eigenvectors of $\mathbf{X}\mathbf{X}^\mathsf{T}$ with corresponding eigenvalues in $\boldsymbol{\Lambda}$. By definition we have $(\mathbf{X}\mathbf{X}^\mathsf{T})\mathbf{U} = \mathbf{U}\boldsymbol{\Lambda}$. Pre-multiplying by $\mathbf{X}^\mathsf{T}$ gives

   $$(\mathbf{X}^\mathsf{T}\mathbf{X})(\mathbf{X}^\mathsf{T}\mathbf{U}) = (\mathbf{X}^\mathsf{T}\mathbf{U})\boldsymbol{\Lambda}$$

   from which we see that the eigenvectors of $\mathbf{X}^\mathsf{T}\mathbf{X}$ are $\mathbf{V} = \mathbf{X}^\mathsf{T}\mathbf{U}$, with eigenvalues $\boldsymbol{\Lambda}$ as before. However, these eigenvectors are not normalized, since $\|\boldsymbol{v}_j\|^2 = \boldsymbol{u}_j^\mathsf{T}\mathbf{X}\mathbf{X}^\mathsf{T}\boldsymbol{u}_j = \lambda_j\boldsymbol{u}_j^\mathsf{T}\boldsymbol{u}_j = \lambda_j$. The normalized eigenvectors are given by

   $$\mathbf{V} = \mathbf{X}^\mathsf{T}\mathbf{U}\boldsymbol{\Lambda}^{-\frac{1}{2}}$$

2. **Computing PCA using SVD.** Let $\mathbf{U}_\Sigma\boldsymbol{\Lambda}_\Sigma\mathbf{U}_\Sigma^\mathsf{T}$ be the top L eigendecomposition of the covariance matrix $\boldsymbol{\Sigma} \propto \mathbf{X}^\mathsf{T}\mathbf{X}$ (assume $\mathbf{X}$ is centered). Recall the optimal estimate of the projection weights $\mathbf{W}$ is given by the top L eigenvalues, so $\mathbf{W} = \mathbf{U}_\Sigma$. Now let $\mathbf{U}_X\mathbf{S}_X\mathbf{V}_X^\mathsf{T} \approx \mathbf{X}$ be the L-truncated SVD approximation to the data matrix $\mathbf{X}$. By SVD definition, the columns of $\mathbf{V}_X$ are the eigenvectors of $\mathbf{X}^\mathsf{T}\mathbf{X}$, so

   $$\mathbf{V}_X = \mathbf{U}_\Sigma = \mathbf{W}.$$

   In addition, the eigenvalues of the covariance matrix are related to the singular values of the data matrix via $\lambda_k = s_k^2/N$. Now suppose we are interested in the principal components (or PC scores), rather than the projection matrix. We have

   $$\mathbf{Z} = \mathbf{X}\mathbf{W} = \mathbf{U}_X\mathbf{S}_X\mathbf{V}_X^\mathsf{T}\mathbf{V}_X = \mathbf{U}_X\mathbf{S}_X$$

   Finally, if we want to approximately reconstruct the data, we have

   $$\hat{\mathbf{X}} = \mathbf{Z}\mathbf{W}^\mathsf{T} = \mathbf{U}_X\mathbf{S}_X\mathbf{V}_X^\mathsf{T}$$

   This is precisely the same as a **truncated SVD approximation**.

3. **Latent dimentions selection**. Recall that the reconstruction error can be written in terms of eigenvalues

$$\mathcal{L}_L = \frac{1}{|\mathcal{D}|} \sum_{n \in \mathcal{D}} \|\boldsymbol{x}_n - \hat{\boldsymbol{x}}_n\|^2 = \sum_{j=L+1}^{D} \lambda_j$$

where $\hat{\mathbf{x}}_n = \mathbf{W}\mathbf{z}_n + \mu$ and μ is the empirical mean. Thus, as the number of dimentions incrceases, the eigenvalues get smaller, and so does the reconstruction error. Therefore, if we give it more latent dimenions, it will be able to approximate the test date more accurately.

Thus, PCA can be performed either using an eigendecomposition of $\mathbf{\Sigma}$ or an SVD decomposition of $\mathbf{X}$. The latter is often preferable, for computational reasons.

10 Factor Analysis

Factor analysis is a generalization of PCA corresponds to the following linear-Gaussian latent variable probabilistic **generative** model:

$$\mathbf{x} = \mathbf{W}\mathbf{z} + \boldsymbol{\mu} + \boldsymbol{\epsilon}, \quad \boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{\Psi})$$

$$p(\boldsymbol{z}) = \mathcal{N}(\boldsymbol{z} \mid \boldsymbol{\mu}_0, \boldsymbol{\Sigma}_0)$$

$$p(\boldsymbol{x} \mid \boldsymbol{z}, \boldsymbol{\theta}) = \mathcal{N}(\boldsymbol{x} \mid \mathbf{W}\boldsymbol{z} + \boldsymbol{\mu}, \boldsymbol{\Psi})$$

where $\mathbf{W}$ is a $D \times L$ matrix, known as the **factor loading matrix**, $\boldsymbol{\Psi}$ is a diagonal $D \times D$ noise covariance matrix, and $\boldsymbol{\mu}$ is a $D \times 1$ mean of observed data

FA can be thought of as a low-rank version of a Gaussian distribution. To see this, note that the induced marginal distribution $p(\boldsymbol{x} \mid \boldsymbol{\theta})$ is a Gaussian:

$$p(\boldsymbol{x} \mid \boldsymbol{\theta}) = \int \mathcal{N}(\boldsymbol{x} \mid \mathbf{W}\boldsymbol{z} + \boldsymbol{\mu}, \boldsymbol{\Psi}) \mathcal{N}(\boldsymbol{z} \mid \boldsymbol{\mu}_0, \boldsymbol{\Sigma}_0)\, d\boldsymbol{z}$$

$$= \mathcal{N}(\boldsymbol{x} \mid \mathbf{W}\boldsymbol{\mu}_0 + \boldsymbol{\mu}, \boldsymbol{\Psi} + \mathbf{W}\boldsymbol{\Sigma}_0\mathbf{W}^\mathsf{T})$$

Hence $\mathbb{E}[\boldsymbol{x}] = \mathbf{W}\boldsymbol{\mu}_0 + \boldsymbol{\mu}$ and $\text{Cov}[\boldsymbol{x}] = \mathbf{W}\text{Cov}[\boldsymbol{z}]\mathbf{W}^\mathsf{T} + \boldsymbol{\Psi} = \mathbf{W}\boldsymbol{\Sigma}_0\mathbf{W}^\mathsf{T} + \boldsymbol{\Psi}$. Without loss of generality, we can set $\boldsymbol{\mu}_0 = \mathbf{0}$ and $\boldsymbol{\Sigma}_0 = \boldsymbol{I}$. Thus,

$$p(\boldsymbol{z}) = \mathcal{N}(\boldsymbol{z} \mid \mathbf{0}, \mathbf{I})$$

$$p(\boldsymbol{x} \mid \boldsymbol{z}) = \mathcal{N}(\boldsymbol{x} \mid \mathbf{W}\boldsymbol{z} + \boldsymbol{\mu}, \boldsymbol{\Psi})$$

$$p(\boldsymbol{x}) = \mathcal{N}(\boldsymbol{x} \mid \boldsymbol{\mu}, \mathbf{W}\mathbf{W}^\mathsf{T} + \boldsymbol{\Psi})$$

where FA approximates the covariance matrix of the visible vector using a low-rank decomposition $\mathbf{C} = \text{Cov}[\boldsymbol{x}] = \mathbf{W}\mathbf{W}^\mathsf{T} + \boldsymbol{\Psi}$. The marginal variance of each visible variable is given by $\mathbb{V}[x_d] = \sum_{k=1}^{L} w_{dk}^2 + \psi_d$ where the first term is the variance due to the common factors, and the second ψ_d term is dimension-specific uniqueness.

Using EM algorithm to estimate the parameters $\mathbf{W}, \boldsymbol{\mu}, \boldsymbol{\Psi}$, then we can compute probabilistic latent embeddings using $p(\boldsymbol{z} \mid \boldsymbol{x})$ using Bayes rule for Gaussians,

$$p(\boldsymbol{z} \mid \boldsymbol{x}) = \mathcal{N}(\boldsymbol{z} \mid \mathbf{W}^\mathsf{T}\mathbf{C}^{-1}(\boldsymbol{x} - \boldsymbol{\mu}), \mathbf{I} - \mathbf{W}^\mathsf{T}\mathbf{C}^{-1}\mathbf{W})$$

10.1 Prerequisite: evidence lower bound (ELBO)

Let data x been generated by some latent variable z through a process $p(z) \to p(x \mid z)$.

- To compute the marginal likelihood:

  $$p(x) = \int p(x \mid z)\, p(z)\, dz$$

  which tells how well the model explains the data, and should be maximized during training. However, the integral is usually intractable (eg. $p(x \mid z)$ is a neural network).

- To compute posterior $p(z \mid x)$ (i.e. given this data point, what latent value likely produced it). By Bayes' rule,

  $$p(z \mid x) = \frac{p(x \mid z)p(z)}{p(x)}$$

  and this also requires the intractable $p(x)$ in the denominator.

Since $p(z \mid x)$ cannot be computed exactly, an approximate posterior $q(z \mid x)$ is chosen and try to make it as close as possible to the true posterior, measured by KL divergence:

$$D_{KL}(q(z \mid x) \,\|\, p(z \mid x)) \geq 0$$

expand from the definition of KL divergence and re-arrange,

$$\begin{aligned}\log p(x) &= \underbrace{\mathbb{E}_{q(z|x)}[\log p(x \mid z)] - D_{KL}(q(z \mid x) \,\|\, p(z))}_{\text{ELBO}} + D_{KL}(q(z \mid x) \,\|\, p(z \mid x)) \\ &= \mathcal{L}(x) + D_{\mathrm{KL}}\big(q(z|x) \,\|\, p(z|x)\big)\end{aligned}$$

for which,

$$\log p(x) \geq \mathcal{L}(x) \quad \text{since } D_{\mathrm{KL}}(\cdot\|\cdot) \geq 0$$

ELBO is an useful variational trick,

- $\log p(x)$ itself is intractable, but $\mathcal{L}(x)$ only involves $q(z \mid x)$, $p(z)$, and $p(x \mid z)$ — all things you choose/model directly.
- Maximizing the ELBO does two things simultaneously: it pushes $\log p(x)$ up (better data likelihood), and it pushes $q(z \mid x)$ closer to the true posterior $p(z \mid x)$ (since the gap between ELBO and $\log p(x)$ *is* that KL term).

In the EM algorithm for FA below, expressing ELBO equivalently as $\mathcal{L} = \mathbb{E}_q[\log p(x, z)] - \mathbb{E}_q[\log q(z)]$, according to the lower bound theorem above, maximizing $\log p(x)$ can be done by maximizing $\mathbb{E}_q[\log p(x, z)]$.

10.2 EM algorithm for FA

The FA generative model:

$$p(\mathbf{z}_i) = \mathcal{N}(\mathbf{z}_i \mid \mathbf{0}, \mathbf{I}), \qquad p(\mathbf{x}_i \mid \mathbf{z}_i) = \mathcal{N}(\mathbf{x}_i \mid \mathbf{W}\mathbf{z}_i + \boldsymbol{\mu}, \boldsymbol{\Psi})$$

with parameters $\boldsymbol{\theta} = (\mathbf{W}, \boldsymbol{\mu}, \boldsymbol{\Psi})$ shared across all datapoints $i = 1, \ldots, N$.

E-step: compute the posterior $p(\mathbf{z}_i \mid \mathbf{x}_i, \boldsymbol{\theta})$. Because both distributions above are Gaussian and linearly related, this posterior is available in closed form by completing the square:

$$p(\mathbf{z}_i \mid \mathbf{x}_i, \boldsymbol{\theta}) = \mathcal{N}(\mathbf{z}_i \mid \mathbf{m}_i, \boldsymbol{\Sigma}_i)$$

$$\boldsymbol{\Sigma}_i = (\mathbf{I}_L + \mathbf{W}^T\boldsymbol{\Psi}^{-1}\mathbf{W})^{-1} \in \mathbb{R}^{L\times L}, \qquad \mathbf{m}_i = \boldsymbol{\Sigma}_i\mathbf{W}^T\boldsymbol{\Psi}^{-1}(\mathbf{x}_i - \boldsymbol{\mu}) \in \mathbb{R}^{L\times 1}$$

Note $\boldsymbol{\Sigma}_i$ does not depend on i in practice (no $\mathbf{x}_i$ term appears), so it can be computed once per iteration.

M-step: maximize the expected complete-data log-likelihood $\mathbb{E}[\log p(\mathbf{x}_i, \mathbf{z}_i \mid \boldsymbol{\theta})]$. It is convenient to estimate $\mathbf{W}$ and $\boldsymbol{\mu}$ jointly by defining

$$\tilde{\mathbf{W}} = (\mathbf{W}, \boldsymbol{\mu}), \qquad \tilde{\mathbf{z}} = (\mathbf{z}, 1), \quad \text{so that } \mathbf{x} = \mathbf{W}\mathbf{z} + \boldsymbol{\mu} = \tilde{\mathbf{W}}\tilde{\mathbf{z}}$$

Define:

$$\mathbf{b}_i \triangleq \mathbb{E}[\tilde{\mathbf{z}}_i \mid \mathbf{x}_i] = [\mathbf{m}_i; 1]$$

$$\mathbf{C}_i \triangleq \mathbb{E}[\tilde{\mathbf{z}}_i\tilde{\mathbf{z}}_i^T \mid \mathbf{x}_i] = \begin{pmatrix} \mathbb{E}[\mathbf{z}_i\mathbf{z}_i^T \mid \mathbf{x}_i] & \mathbb{E}[\mathbf{z}_i \mid \mathbf{x}_i] \\ \mathbb{E}[\mathbf{z}_i \mid \mathbf{x}_i]^T & 1 \end{pmatrix} = \begin{pmatrix} \boldsymbol{\Sigma}_i + \mathbf{m}_i\mathbf{m}_i^T & \mathbf{m}_i \\ \mathbf{m}_i^T & 1 \end{pmatrix}$$

Maximizing the expected complete-data log-likelihood with respect to $\tilde{\mathbf{W}}$ and $\boldsymbol{\Psi}$ (a weighted-least-squares-type problem, since it is quadratic in $\tilde{\mathbf{W}}$ given $\mathbf{b}_i, \mathbf{C}_i$) yields:

$$\begin{aligned}\arg\max_{\boldsymbol{\theta}} \sum_i \mathbb{E}[\log p(\mathbf{x}_i, \mathbf{z}_i \mid \boldsymbol{\theta})] &= \arg\max_{\boldsymbol{\theta}} \sum_i \mathbb{E}[\log p(\mathbf{z}_i) + \log p(\mathbf{x}_i \mid \mathbf{z}_i, \boldsymbol{\theta})] \\ &= \arg\max_{\boldsymbol{\theta}} \sum_i \mathbb{E}[\log p(\mathbf{x}_i \mid \mathbf{z}_i, \boldsymbol{\theta})] \\ &= \arg\max_{\boldsymbol{\theta}} \left\{ -\frac{N}{2}\log|\Psi| - \frac{1}{2}\sum_{i=1}^{N}\left[\mathbf{x}_i^\top \Psi^{-1}\mathbf{x}_i - 2\mathbf{x}_i^\top \Psi^{-1}\tilde{\mathbf{W}}\boldsymbol{b}_i + \operatorname{tr}\left(\tilde{\mathbf{W}}^\top \Psi^{-1}\tilde{\mathbf{W}}\mathbf{C}_i\right)\right]\right\}\end{aligned}$$

Solve for $\boldsymbol{\theta} = (\tilde{\mathbf{W}}, \boldsymbol{\Psi})$ by setting $\partial Q/\partial \tilde{\mathbf{W}} = 0$ to get $\hat{\tilde{\mathbf{W}}}$, then substituting back and setting $\partial Q/\partial \Psi^{-1} = 0$ to get $\hat{\Psi}$,

$$\hat{\tilde{\mathbf{W}}} = \left[\sum_i \mathbf{x}_i \mathbf{b}_i^T\right]\left[\sum_i \mathbf{C}_i\right]^{-1}$$

$$\hat{\boldsymbol{\Psi}} = \frac{1}{N}\operatorname{diag}\left\{\sum_i \left(\mathbf{x}_i - \hat{\tilde{\mathbf{W}}}\mathbf{b}_i\right)\mathbf{x}_i^T\right\}$$

the $\texttt{diag}\{\cdot\}$ enforces the modeling assumption that $\boldsymbol{\Psi}$ remains diagonal.

The loop. Alternate E-step (recompute $\mathbf{m}_i, \boldsymbol{\Sigma}_i, \mathbf{b}_i, \mathbf{C}_i$ using current $\boldsymbol{\theta}$) and M-step (recompute $\mathbf{W}, \boldsymbol{\mu}, \boldsymbol{\Psi}$ using current $\mathbf{b}_i, \mathbf{C}_i$) until the marginal log-likelihood converges.

10.3 EM algorithm for PPCA

PPCA is the special case of FA with $\boldsymbol{\Psi} = \sigma^2\mathbf{I}$. Substituting into the general FA-EM equations:

E-step.

$$\boldsymbol{\Sigma} = \left(\mathbf{I}_L + \sigma^{-2}\mathbf{W}^T\mathbf{W}\right)^{-1} = \sigma^2\mathbf{M}^{-1}, \qquad \mathbf{M} := \mathbf{W}^T\mathbf{W} + \sigma^2\mathbf{I}$$

$$\mathbf{m}_i = \mathbf{M}^{-1}\mathbf{W}^T(\mathbf{x}_i - \boldsymbol{\mu})$$

($\boldsymbol{\Sigma}$ has no subscript i: it does not depend on $\mathbf{x}_i$, only on $\mathbf{W}, \sigma^2$.)

M-step. The $\tilde{\mathbf{W}}$ update is unchanged in form:

$$\hat{\tilde{\mathbf{W}}} = \left[\sum_i \mathbf{x}_i \mathbf{b}_i^T\right]\left[\sum_i \mathbf{C}_i\right]^{-1}$$

Since noise is isotropic, σ^2 is a single scalar (average over all D dimensions, rather than a diagonal $\boldsymbol{\Psi}$):

$$\hat{\sigma}^2 = \frac{1}{ND}\sum_{i=1}^{N}\operatorname{tr}\left[(\mathbf{x}_i - \hat{\tilde{\mathbf{W}}}\mathbf{b}_i)\mathbf{x}_i^T\right]$$

10.4 Limiting case: EM algorithm for PCA

Noise-free limit ($\sigma^2 \to 0$): EM for PCA. The posterior collapses to a point estimate ($\boldsymbol{\Sigma} \to \mathbf{0}$), reducing to alternating least squares. With data centered and column-stacked as $\tilde{\mathbf{X}} \in \mathbb{R}^{D\times N}$:

1. E-step: given current $\mathbf{W}$, project data onto it via least squares $\to \tilde{\mathbf{Z}}$

$$\tilde{\mathbf{Z}} = (\mathbf{W}^T\mathbf{W})^{-1}\mathbf{W}^T\tilde{\mathbf{X}}$$

2. M-step: given current $\tilde{\mathbf{Z}}$, re-fit $\mathbf{W}$ via least-squares regression of $\mathbf{X}$ onto $\mathbf{Z}$

$$\mathbf{W} = \tilde{\mathbf{X}}\tilde{\mathbf{Z}}^T(\tilde{\mathbf{Z}}\tilde{\mathbf{Z}}^T)^{-1}$$

Why use EM instead of the closed-form eigendecomposition? Each EM iteration only requires $L \times L$ inversions ($L \ll D$ typically), costing $O(NDL + L^3)$ per iteration, versus $O(D^3)$ for eigendecomposing the full covariance matrix $\mathbf{S}$. This matters when D is large (e.g. many neurons) but L is small. EM also extends naturally to missing data, unlike the closed-form solution.

Sanity check. As $\sigma^2 \to 0$, both EM-derived quantities converge to the standard PCA solution: $\hat{\mathbf{W}} \to \mathbf{U}_L\mathbf{L}_L^{1/2}$ and $\mathbb{E}[\mathbf{z} \mid \mathbf{x}] \to (\mathbf{W}^T\mathbf{W})^{-1}\mathbf{W}^T(\mathbf{x} - \overline{\mathbf{x}})$, matching the closed-form Tipping & Bishop result.

10.5 Special case: Probabilistic PCA

Probabilistic PCA (PPCA) is a special case of the factor analysis model in which $\mathbf{W}$ has orthonormal columns and $\mathbf{\Psi} = \sigma^2\mathbf{I}$. Then, the full-form marginal distribution on the visible variables reduces to:

$$\begin{aligned} p(\boldsymbol{x} \mid \boldsymbol{\mu}, \mathbf{W}, \sigma^2) &= \int \mathcal{N}(\boldsymbol{x} \mid \mathbf{W}\boldsymbol{z} + \boldsymbol{\mu}, \mathbf{\Psi})\mathcal{N}(\boldsymbol{z} \mid \boldsymbol{\mu}_0, \mathbf{\Sigma}_0)\, d\boldsymbol{z} \\ &= \int \mathcal{N}(\boldsymbol{x} \mid \mathbf{W}\boldsymbol{z}, \sigma^2\mathbf{I})\mathcal{N}(\boldsymbol{z} \mid \mathbf{0}, \boldsymbol{I})\, d\boldsymbol{z} \\ &= \mathcal{N}(\boldsymbol{x} \mid \boldsymbol{\mu}, \mathbf{W}\mathbf{W}^{\mathsf{T}} + \sigma^2\mathbf{I}) = \mathcal{N}(\boldsymbol{x} \mid \boldsymbol{\mu}, \mathbf{C}) \end{aligned}$$

Then, the log-likelihood for PPCA is:

$$\begin{aligned} \log p(\mathbf{X} \mid \boldsymbol{\mu}, \mathbf{W}, \sigma^2) &= \sum_{n=1}^{N} \log p(\mathbf{x}_n \mid \boldsymbol{\mu}, \mathbf{W}, \sigma^2) \\ &= -\frac{ND}{2}\log(2\pi) - \frac{N}{2}\log|\mathbf{C}| - \frac{1}{2}\sum_{n=1}^{N}(\mathbf{x}_n - \boldsymbol{\mu})^T\mathbf{C}^{-1}(\mathbf{x}_n - \boldsymbol{\mu}) \end{aligned}$$

Solve for $\boldsymbol{\mu}$

Differentiate the log-likelihood w.r.t. $\boldsymbol{\mu}$ gives

$$\hat{\boldsymbol{\mu}} = \overline{\mathbf{x}} = \frac{1}{N}\sum_{n=1}^{N}\mathbf{x}_n$$

Substituting this back gives the simplified objective in terms of the sample covariance matrix $\mathbf{S} = \frac{1}{N}\sum_{n=1}^{N}(\mathbf{x}_n - \overline{\mathbf{x}})(\mathbf{x}_n - \overline{\mathbf{x}})^T$:

$$\log p(\mathbf{X} \mid \boldsymbol{\mu}, \mathbf{W}, \sigma^2) = -\frac{N}{2}\left[D\log(2\pi) + \log|\mathbf{C}| + \mathrm{tr}(\mathbf{C}^{-1}\mathbf{S})\right]$$

Solve for $\mathbf{W}$ and σ^2 jointly

Rather than iterative fitting, the maximizer is derived directly (Tipping & Bishop 1999):

$$\mathbf{W} = \mathbf{U}_L(\mathbf{L}_L - \sigma^2\mathbf{I})^{1/2}\mathbf{R}$$

where:

- $\mathbf{U}_L$: $D \times L$ matrix of the top L eigenvectors of $\mathbf{S}$
- $\mathbf{L}_L$: $L \times L$ diagonal matrix of the corresponding top L eigenvalues
- $\mathbf{R}$: arbitrary $L \times L$ rotation matrix (WLOG set $\mathbf{R} = \mathbf{I}$)

and the noise variance:

$$\hat{\sigma}^2 = \frac{1}{D-L} \sum_{i=L+1}^{D} \lambda_i$$

i.e., the *average of the discarded eigenvalues* (the variance in directions not kept as signal).

Both $\hat{\mathbf{W}}$ and $\hat{\sigma}^2$ come from *one* eigendecomposition of $\mathbf{S}$: compute all D eigenvalues/eigenvectors once, keep the top L for $\hat{\mathbf{W}}$, and average the rest for $\hat{\sigma}^2$.

Sanity check

As $\sigma^2 \to 0$:

$$\hat{\mathbf{W}} \to \mathbf{U}_L \mathbf{L}_L^{1/2}$$

which is proportional to the PCA solution.

Getting the latent representation

Unlike PCA's deterministic projection, PPCA computes the *posterior* $p(\mathbf{z} \mid \mathbf{x})$ via Bayes' rule:

$$p(\mathbf{z} \mid \mathbf{x}) = \frac{p(\mathbf{x} \mid \mathbf{z}) p(\mathbf{z})}{\int p(\mathbf{x} \mid \mathbf{z}') p(\mathbf{z}') \, d\mathbf{z}'} = \mathcal{N}(\mathbf{z} \mid \mathbf{M}^{-1} \mathbf{W}^T (\mathbf{x} - \boldsymbol{\mu}),\ \sigma^2 \mathbf{M}^{-1})$$

where $p(\mathbf{z}) = \mathcal{N}(\mathbf{z} \mid \mathbf{0}, \mathbf{I})$, $p(\mathbf{x} \mid \mathbf{z}) = \mathcal{N}(\mathbf{x} \mid \mathbf{W}\mathbf{z} + \boldsymbol{\mu}, \sigma^2 \mathbf{I})$ and $\mathbf{M} := \mathbf{W}^T \mathbf{W} + \sigma^2 \mathbf{I}$.

The posterior mean serves as the latent embedding (PPCA's analogue of PCA's $\mathbf{z}_n = \mathbf{W}^T \mathbf{x}_n$). In the $\sigma^2 \to 0$ limit:

$$\mathbb{E}[\mathbf{z} \mid \mathbf{x}] = (\mathbf{W}^T \mathbf{W})^{-1} \mathbf{W}^T (\mathbf{x} - \overline{\mathbf{x}})$$

which reduces exactly to the orthogonal projection used in standard PCA.

10.6 Summary

	PCA	**PPCA**	**Factor Analysis**
Decoder	$\mathbf{x}_n = \mathbf{W}\mathbf{z}_n$ (deterministic)	$\mathbf{x} = \mathbf{W}\mathbf{z} + \boldsymbol{\mu} + \boldsymbol{\epsilon},\ \boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \sigma^2\mathbf{I})$	$\mathbf{x} = \mathbf{W}\mathbf{z} + \boldsymbol{\mu} + \boldsymbol{\epsilon},\ \boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{\Psi})$
Noise structure	none	isotropic: same variance σ^2 in every dimension	diagonal: different variance ψ_i per dimension
Prior on z	none	$p(\mathbf{z}) = \mathcal{N}(\mathbf{0}, \mathbf{I})$	$p(\mathbf{z}) = \mathcal{N}(\mathbf{0}, \mathbf{I})$
Constraint on W	orthonormal columns	orthonormal columns	none
Encoder (inference)	deterministic projection: $\mathbf{z}_n = \mathbf{W}^T\mathbf{x}_n$	posterior mean: $\mathbb{E}[\mathbf{z} \mid \mathbf{x}] = \mathbf{M}^{-1}\mathbf{W}^T(\mathbf{x} - \boldsymbol{\mu})$, $\mathbf{M} = \mathbf{W}^T\mathbf{W} + \sigma^2\mathbf{I}$	posterior mean: $\mathbb{E}[\mathbf{z} \mid \mathbf{x}] = \mathbf{W}^T\boldsymbol{\Sigma}_x^{-1}(\mathbf{x} - \boldsymbol{\mu})$, using full $\boldsymbol{\Sigma}_x = \mathbf{W}\mathbf{W}^T + \boldsymbol{\Psi}$
Encoder uncertainty	none	yes: $\mathrm{Cov}(\mathbf{z} \mid \mathbf{x}) = \sigma^2\mathbf{M}^{-1}$	yes: full posterior covariance, generally non-diagonal
Fitting method	eigendecomposition of $\mathbf{S}$ (direct, no likelihood)	closed-form MLE via eigendecomposition of $\mathbf{S}$ (Tipping & Bishop)	no closed form — requires EM algorithm
Marginal $p(\mathbf{x})$	none	$\mathcal{N}(\boldsymbol{\mu}, \mathbf{W}\mathbf{W}^T + \sigma^2\mathbf{I})$	$\mathcal{N}(\boldsymbol{\mu}, \mathbf{W}\mathbf{W}^T + \boldsymbol{\Psi})$
Generative?	No	Yes	Yes

Table 1: Comparison of PCA, PPCA, and Factor Analysis in encoder/decoder terms.

11 Autoencoders

11.1 Deterministic autoencoders

Recall that PCA and FA as learning a linear mapping from $\boldsymbol{x} \to \boldsymbol{z}$, called the **encoder**, f_e, and learning another (linear) mapping $\boldsymbol{z} \to \boldsymbol{x}$, called the **decoder**, f_d. The overall reconstruction function has the form $r(\boldsymbol{x}) = f_d(f_e(\boldsymbol{x}))$. The model is trained to minimize $\mathcal{L}(\boldsymbol{\theta}) = \|r(\boldsymbol{x}) - \boldsymbol{x}\|_2^2$, if isotropic generative model's noise variance is assumed. More generally, $\mathcal{L}(\boldsymbol{\theta}) = -\log p(\boldsymbol{x} \mid r(\boldsymbol{x}))$.

Autoencoders uses neural networks to implement non-linear mappings in the encoder and decoder.

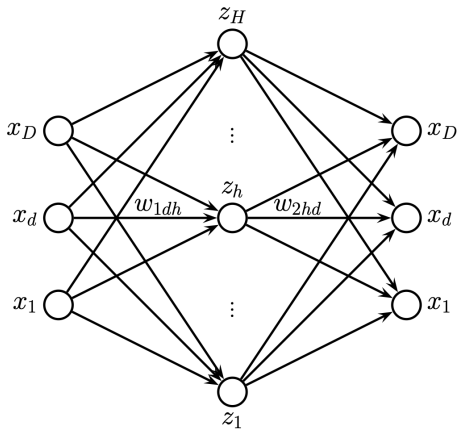


Figure 11: An autoencoder with one hidden layer MLP

The hidden units in the middle is a low-dimensional bottleneck between the input and its reconstruction.

If the hidden layer is wide enough, the model may learn the identity function. Thus, a narrow bottleneck layer with $L \ll D$ can be used, called **undercomplete representation**; a wider bottleneck layer with $L \gg D$ can also be used but to impose other kind of regularization, called **overcomplete representation**.

Regularization techniques to avoid trivial memorization,

1. **Bottleneck autoencoders**: This is a special case of a **linear autoencoder**, in which there is one hidden layer, the hidden units are computed using $\boldsymbol{z} = \mathbf{W}_1\boldsymbol{x}$, and the output is reconstructed using $\hat{\boldsymbol{x}} = \mathbf{W}_2\boldsymbol{z}$, where $\mathbf{W}_1$ is a $L \times D$ matrix, $\mathbf{W}_2$ is a $D \times L$ matrix, and $L < D$ (restricting model capacity). Hence $\hat{\boldsymbol{x}} = \mathbf{W}_2\mathbf{W}_1\boldsymbol{x} = \mathbf{W}\boldsymbol{x}$ is the output of the model. If we train this model to minimize the squared reconstruction error, $\mathcal{L}(\mathbf{W}) = \sum_{n=1}^{N} \|\boldsymbol{x}_n - \mathbf{W}\boldsymbol{x}_n\|_2^2$, literature shows that $\hat{\mathbf{W}}$ is an orthogonal projection onto the first L eigenvectors of the empirical covariance matrix of the data, equivalent to PCA. If non-linearities are introduced, such as replacing MLP with CNN, the model is strictly more powerful than PCA without significantly changes the model size and training time.

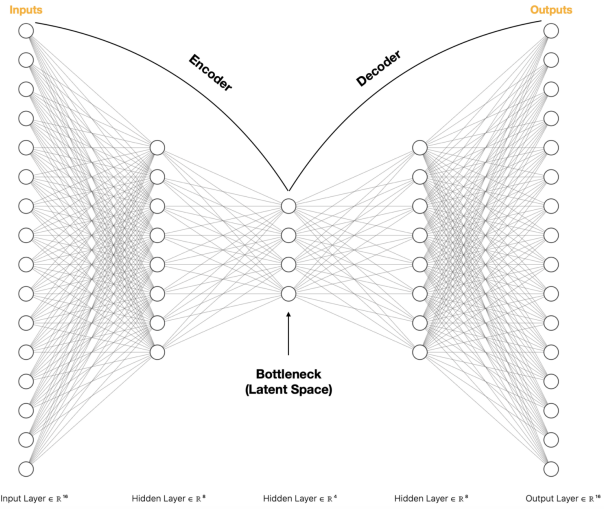


Figure 12: Schematic illustration of an undercomplete autoencoder.

2. **Denoising autoencoders**: Adding noises such as Gaussian noise to the input, the model is able to recover details that are missing in the input, since it has seen similar images before, and can store this information in the parameters of the model. Suppose we train a DAE using Gaussian corruption and squared error reconstruction, i.e., we use $p_c(\tilde{\boldsymbol{x}} \mid \boldsymbol{x}) = \mathcal{N}(\tilde{\boldsymbol{x}} \mid \boldsymbol{x}, \sigma^2\mathbf{I})$ and $\ell(\boldsymbol{x}, r(\tilde{\boldsymbol{x}})) = \|\boldsymbol{r}(\tilde{\boldsymbol{x}}) - \boldsymbol{x}\|_2^2$ for sample $\boldsymbol{x}$. DAE can learn a vector field, corresponding to the gradient of the log data density $\boldsymbol{e}(\boldsymbol{x}) = r(\tilde{\boldsymbol{x}}) - \boldsymbol{x} \approx \nabla_{\boldsymbol{x}} \log p(\boldsymbol{x})$. Thus points that are close to the data manifold will be projected onto it via the sampling process.

3. **Contractive autoencoders**: Adding a penalty term to the reconstruction loss, $\lambda \left\| \frac{\partial f_e(\boldsymbol{x})}{\partial \boldsymbol{x}} \right\|_F^2 = \lambda \sum_k \|\nabla_{\boldsymbol{x}} h_k(\boldsymbol{x})\|_2^2$ (i.e. penalize the Frobenius norm of the encoder's Jacobian), forcing the encoding to be insensitive to small input perturbations.

4. **Sparse autoencoders**: Adding a penalty term to the reconstruction loss, $\lambda \|\boldsymbol{z}\|_1$, forcing some embeddings to be small/zero. Thus, even under the overcomplete representation ($L > D$), it restricts how much of the model capacity gets used at any one time, which can actually give a richer, more interpretable representation

In summary, the workflow of a deterministic autoencoder is as follows
Setup

- Input $\boldsymbol{x} \in \mathbb{R}^D$ (high-dim), latent $\boldsymbol{z} \in \mathbb{R}^K$ with $K \ll D$
- Encoder $f_e(\boldsymbol{x}; \boldsymbol{\phi}) : \mathbb{R}^D \to \mathbb{R}^K$
- Decoder $f_d(\boldsymbol{z}; \boldsymbol{\theta}) : \mathbb{R}^K \to \mathbb{R}^D$

Training steps

1. Sample a minibatch $\{\boldsymbol{x}_i\}$ from the dataset
2. Forward pass: $\boldsymbol{z}_i = f_e(\boldsymbol{x}_i; \boldsymbol{\phi})$ (deterministic, single vector — no sampling)
3. Reconstruct: $\hat{\boldsymbol{x}}_i = f_d(\boldsymbol{z}_i; \boldsymbol{\theta})$

4. Compute loss:

$$\mathcal{L} = \frac{1}{N} \sum_i \|\boldsymbol{x}_i - \hat{\boldsymbol{x}}_i\|^2$$

   (or cross-entropy, depending on data type)

5. Backprop $\nabla_{\boldsymbol{\theta}} \mathcal{L}, \nabla_{\boldsymbol{\phi}} \mathcal{L}$, and update $\boldsymbol{\theta}, \boldsymbol{\phi}$ with an optimizer (Adam, SGD, etc.)

6. Repeat over epochs until convergence

Inference Taking a new/held-out point $\boldsymbol{x}$, passing it through the encoder $\boldsymbol{z} = f_e(\boldsymbol{x}; \boldsymbol{\phi})$, where $\boldsymbol{z} \in \mathbb{R}^K$ is the reduced-dimension representation. Passing through the decoder if reconstruction is needed.

11.2 Variational autoencoders (VAE)

	Deterministic	**Probabilistic**
Linear	PCA	PPCA, FA
Nonlinear	Bottleneck/Denoising/Contractive/Sparse AE	VAE

Table 2: Comparison of dimensionality reduction algorithms.

VAE can be thought of as a probabilistic version of the deterministic autoencoder. Thus, as a generative model, it can generate new samples instead of just computing embeddings of input vectors.

Also, VAE is the non-linear extension of the factor analysis generative model. Replacing $p(\boldsymbol{x} \mid \boldsymbol{z}) = \mathcal{N}(\boldsymbol{x} \mid \mathbf{W}\boldsymbol{z}, \sigma^2 \mathbf{I})$ with

$$p_{\boldsymbol{\theta}}(\boldsymbol{x} \mid \boldsymbol{z}) = \mathcal{N}(\boldsymbol{x} \mid f_d(\boldsymbol{z}; \boldsymbol{\theta}), \sigma^2 \mathbf{I})$$

where f_d is a non-linear decoder with parameter $\boldsymbol{\theta}$.

Since the posterior $p(\boldsymbol{z} \mid \boldsymbol{x})$ is intractable, so another model $q(\boldsymbol{z} \mid \boldsymbol{x})$ is created to approximate the posterior, called recognition network or inference network. If the posterior is assumed to be Gaussian, with diagonal covariance, then,

$$q_{\boldsymbol{\phi}}(\boldsymbol{z} \mid \boldsymbol{x}) = \mathcal{N}(\boldsymbol{z} \mid f_{e,\mu}(\boldsymbol{x}; \boldsymbol{\phi}), \mathrm{diag}(f_{e,\sigma}(\boldsymbol{x}; \boldsymbol{\phi})))$$

where $f_{e,\mu/\sigma}$ is the encoder for mean and variance respectively with parameter $\boldsymbol{\phi}$. This 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.

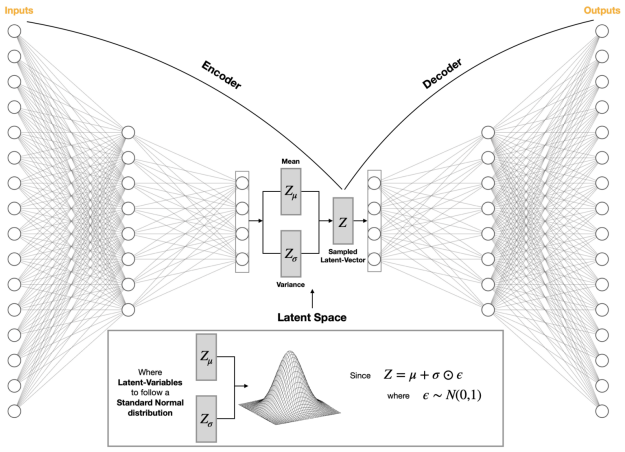


Figure 13: Schematic illustration of a VAE.

Training VAEs

The computation of exact $p_{\boldsymbol{\theta}}(\boldsymbol{x}) = \int p_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{z}) d\boldsymbol{z} = \int p_{\boldsymbol{\theta}}(\boldsymbol{x} \mid \boldsymbol{z}) p(\boldsymbol{z}) d\boldsymbol{z}$ for MLE training is in possible because the porterior in a nonlinear FA model is intractable. However, by making use of $q(\boldsymbol{z} \mid \boldsymbol{x})$, ELBO can be computed. For a single example $\boldsymbol{x}$, the ELBO as the objective function is,

$$\begin{aligned} \mathrm{L}(\boldsymbol{\theta}, \boldsymbol{\phi} | \boldsymbol{x}) &= \mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})} \left[\log p_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{z}) - \log q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}) \right] \\ &= \mathbb{E}_{q(\boldsymbol{z}|\boldsymbol{x}, \boldsymbol{\phi})} \left[\log p(\boldsymbol{x}|\boldsymbol{z}, \boldsymbol{\theta}) \right] - D_{KL} \left(q(\boldsymbol{z}|\boldsymbol{x}, \boldsymbol{\phi}) \,\|\, p(\boldsymbol{z}) \right) \end{aligned}$$

where $q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}) = q(\boldsymbol{z}|\boldsymbol{x}, \boldsymbol{\phi})$ are different notation conventions. This can be interpreted as the expected log-likelihood plus a regularizer which penalizes the posterior from deviating too much from the prior.

ELBO is a lower bound of the evidence as shown by the Jensen's inquality,

$$\begin{aligned} \mathrm{L}(\boldsymbol{\theta}, \boldsymbol{\phi} | \boldsymbol{x}) &= \int q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}) \log \frac{p_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{z})}{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})} d\boldsymbol{z} \\ &\leq \log \int q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}) \frac{p_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{z})}{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})} d\boldsymbol{z} = \log p_{\boldsymbol{\theta}}(\boldsymbol{x}) \end{aligned}$$

Thus, the optimization task for training VAE is

$$\boldsymbol{\theta}^*, \boldsymbol{\phi}^* = \arg\max_{\boldsymbol{\theta}, \boldsymbol{\phi}} \sum_i \mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi} \mid \boldsymbol{x})$$

A trick in computing ELBO is **reparameterization**. Since $q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})$ is Gaussian, then

$$\boldsymbol{z} = f_{e,\mu}(\boldsymbol{x}; \boldsymbol{\phi}) + f_{e,\sigma}(\boldsymbol{x}; \boldsymbol{\phi}) \odot \boldsymbol{\epsilon}$$

where $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$ and $\odot$ denotes element-wise multiplication. Hence,

$$\mathrm{L}(\boldsymbol{\theta}, \boldsymbol{\phi} | \boldsymbol{x}) = \mathbb{E}_{\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})} \left[\log p\boldsymbol{\theta}(\boldsymbol{x}|\boldsymbol{z} = \mu_{\boldsymbol{\phi}}(\boldsymbol{x}) + \sigma_{\boldsymbol{\phi}}(\boldsymbol{x}) \odot \boldsymbol{\epsilon}) \right] - D_{KL} \left(q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}) \mid p(\boldsymbol{z}) \right)$$

The expectation is now over a fixed distribution $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$. Before reparameterization, every gradient step would change $q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})$ itself (the ground samples are taking from shifts). After reparameterization, always sampling from the same fixed source of randomness $\boldsymbol{\epsilon}$, and all the

parameter($\boldsymbol{\phi}$)-dependence has been moved into a deterministic, differentiable path inside the expectation (i.e. once $\boldsymbol{\epsilon}$ is sampled, only a deterministic and differentiable neural network $f_{e,\mu/\sigma}$ is left). Therefore, the backpropagation for training can be used as usual.

The first term in the ELBO can be approximated by sampling $\boldsymbol{\epsilon}$, scaling it by the output of the inference network to get $\boldsymbol{z}$, and then evaluating $\log p(\boldsymbol{x}|\boldsymbol{z})$ using the decoder network f_d.

The second term in the ELBO is the KL of two Gaussians, which has a closed form expression. With $p(\boldsymbol{z}) = \mathcal{N}(\boldsymbol{z}|\mathbf{0}, \mathbf{I})$ and $q_{\boldsymbol{\phi}}(\boldsymbol{z} \mid \boldsymbol{x}) = \mathcal{N}(\boldsymbol{z}|\boldsymbol{\mu}, \text{diag}(\boldsymbol{\sigma}))$,

$$D_{KL}\left(q \| p\right) = \sum_{k=1}^{K}\left[\log\left(\frac{1}{\sigma_k}\right) + \frac{\sigma_k^2 + (\mu_k - 0)^2}{2 \cdot 1} - \frac{1}{2}\right] = -\frac{1}{2}\sum_{k=1}^{K}\left[\log\sigma_k^2 - \sigma_k^2 - \mu_k^2 + 1\right]$$

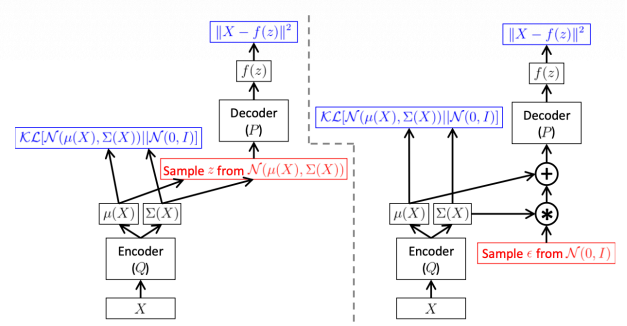


Figure 14: Computation graphs for VAEs. Computation graph for VAEs, where $p(\boldsymbol{z}) = \mathcal{N}(\boldsymbol{z}|\mathbf{0}, \mathbf{I})$, $p(\boldsymbol{x}|\boldsymbol{z}, \boldsymbol{\theta}) = \mathcal{N}(\boldsymbol{x}|f(\boldsymbol{z}), \sigma^2\mathbf{I})$, and $q(\boldsymbol{z}|\boldsymbol{x}, \boldsymbol{\phi}) = \mathcal{N}(\boldsymbol{z}|\mu(\boldsymbol{x}), \Sigma(\boldsymbol{x}))$. Red boxes show sampling operations which are not differentiable. Blue boxes show loss layers (we assume Gaussian likelihoods and priors). (Left) Without the reparameterization trick. (Right) With the reparameterization trick. Gradients can flow from the output loss, back through the decoder and into the encoder.

In summary,

Setup

- $\boldsymbol{x} \in \mathbb{R}^D$, $\boldsymbol{z} \in \mathbb{R}^K$
- Encoder outputs distribution params: $\mu_{\boldsymbol{\phi}}(\boldsymbol{x}), \sigma_{\boldsymbol{\phi}}(\boldsymbol{x}) \in \mathbb{R}^K$
- Decoder $f_d(\boldsymbol{z}; \boldsymbol{\theta})$ outputs likelihood params for $p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})$
- Fixed prior $p(\boldsymbol{z}) = \mathcal{N}(0, I)$

Training steps

1. Sample a minibatch $\{\boldsymbol{x}_i\}$, then forward pass through encoder: $(\mu_{\boldsymbol{\phi}}(\boldsymbol{x}_i), \sigma_{\boldsymbol{\phi}}(\boldsymbol{x}_i)) = f_e(\boldsymbol{x}_i; \boldsymbol{\phi})$
2. Reparameterize: $\boldsymbol{z}_i = \mu_{\boldsymbol{\phi}}(\boldsymbol{x}_i) + \sigma_{\boldsymbol{\phi}}(\boldsymbol{x}_i) \odot \boldsymbol{\epsilon}$ (deterministic given $\boldsymbol{\epsilon}$, differentiable w.r.t. $\boldsymbol{\phi}$) with a sample from $\boldsymbol{\epsilon} \sim \mathcal{N}(0, I)$
3. Decode: pass $\boldsymbol{z}_i$ through decoder to get $p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z}_i)$'s parameters, which can be used for ELBO first term calculation.
4. Compute ELBO:

$$\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi} \mid \boldsymbol{x}_i) = \log p_{\boldsymbol{\theta}}(\boldsymbol{x}_i \mid \boldsymbol{z}_i) - D_{KL}\big(q_{\boldsymbol{\phi}}(\boldsymbol{z} \mid \boldsymbol{x}_i) \,\|\, p(\boldsymbol{z})\big)$$

5. Minimize $-\frac{1}{N}\sum_i \mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi} \mid \boldsymbol{x}_i)$ via backprop — gradients flow through the reparameterized $\boldsymbol{z}_i$ into both $\boldsymbol{\phi}$ and $\boldsymbol{\theta}$, and update $\boldsymbol{\theta}, \boldsymbol{\phi}$ with an optimizer
6. Repeat over epochs

Inference Since the encoder gives a *distribution*, not a point:

- Option A: use the mean only, $\boldsymbol{z} = \mu_{\boldsymbol{\phi}}(\boldsymbol{x})$, as the reduced representation — deterministic, ignores $\sigma_{\boldsymbol{\phi}}(\boldsymbol{x})$ entirely.
- Option B: sample $\boldsymbol{z} \sim q_{\boldsymbol{\phi}}(\boldsymbol{z} \mid \boldsymbol{x}) = \mathcal{N}(\mu_{\boldsymbol{\phi}}(\boldsymbol{x}), \sigma_{\boldsymbol{\phi}}(\boldsymbol{x})^2)$ each time — introduces stochasticity into the reduced representation.

Key property: because of the KL regularization during training, the resulting $\boldsymbol{z}$-space (using $\mu_{\boldsymbol{\phi}}(\boldsymbol{x})$) tends to be smoother and better organized around the origin than a plain AE's latent space, at some cost to reconstruction fidelity (since the KL term competes with the reconstruction term).

Part IV
Kernel Methods

Kernel methods are a class of nonparametric methods for regression and classification. Such methods do not assume a fixed parametric form for the prediction function, but instead try to estimate the function itself directly from data.

For a given dataset $\mathcal{D} = \{(\boldsymbol{x}_n, y_n)\}_{n=1}^N$, where $y_n = f(\boldsymbol{x}_n)$ and f is unknown. To predict the function value at a new point $\boldsymbol{x}_*$, kernel methods predicts that $f(\boldsymbol{x}_*)$ with some weighted combination of the $\{f(\boldsymbol{x}_n)\}$ values, i.e. remembering the entire training set $\mathcal{D} = \{(\boldsymbol{x}_n, y_n)\}_{n=1}^N$ to make predictions.

12 Mercer Kernels and Mercer's Theorem

The key to nonparametric methods is that we need a way to encode prior knowlegde about the similarity of two input vectors. To define similarity, we use a **kernel function**.

Here, 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

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

for all $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N \in \mathcal{X}$ which are vectors in $\mathbb{R}^D$ and any choice of numbers $c_i \in \mathbb{R}$. Assuming $\mathcal{K}(\boldsymbol{x}_i, \boldsymbol{x}_j) > 0$, so that only equality if $c_i = 0$ for all i.

Another way to understand this condition: given a set of N datapoints, define the Gram matrix as the following $N \times N$ similarity matrix:

$$\mathbf{K} = \begin{bmatrix} \mathcal{K}(\boldsymbol{x}_1, \boldsymbol{x}_1) & \cdots & \mathcal{K}(\boldsymbol{x}_1, \boldsymbol{x}_N) \\ \vdots & \vdots & \vdots \\ \mathcal{K}(\boldsymbol{x}_N, \boldsymbol{x}_1) & \cdots & \mathcal{K}(\boldsymbol{x}_N, \boldsymbol{x}_N) \end{bmatrix} \in \mathbb{R}^{N \times N}$$

then, the condition becomes that $\mathbf{K}$ is positive definite, i.e. all its eigenvalues are positive, for any set of distinct inputs $\{\boldsymbol{x}_i\}_{i=1}^N$.

12.1 Mercer's theorem

Any symmetric matrix $\mathbf{K}$ can be represented using an eigendecomposition $\mathbf{K} = \mathbf{U}\boldsymbol{\Lambda}\mathbf{U}^T$ where $\mathbf{U}$ is an orthonormal matrix of eigenvectors and $\boldsymbol{\Lambda}$ is a diagonal matrix of eigenvalues $\lambda_i > 0$.

Consider an element (i, j) of $\mathbf{K}$,

$$k_{ij} = \mathcal{K}(\boldsymbol{x}_i, \boldsymbol{x}_j) = (\boldsymbol{\Lambda}^{\frac{1}{2}} \mathbf{U}_{:i})^\top (\boldsymbol{\Lambda}^{\frac{1}{2}} \mathbf{U}_{:j})$$

where $\mathbf{U}_{:i}$ is the i-th column of $\mathbf{U}$. Define a mapping $\boldsymbol{\phi} : \mathbb{R}^D \to \mathbb{R}^N$ as $\boldsymbol{\phi}(\boldsymbol{x}_i) = \boldsymbol{\Lambda}^{\frac{1}{2}} \mathbf{U}_{:i}$ Then,

$$k_{ij} = \mathcal{K}(\boldsymbol{x}_i, \boldsymbol{x}_j) = \boldsymbol{\phi}(\boldsymbol{x}_i)^\top \boldsymbol{\phi}(\boldsymbol{x}_j) = \sum_{m=1}^N \phi_m(\boldsymbol{x}_i) \phi_m(\boldsymbol{x}_j)$$

In theory, the kernel function can be generalized to $N \to \infty$ with $\lambda_i > 0$ as a prerequisite.

$\boldsymbol{\phi}$ is a mapping from input space to the feature space, which is a Hilbert space. **The kernel function $\mathcal{K}$ can be interpreted as an inner product in the feature space**, i.e. $\mathcal{K}(\boldsymbol{x}_i, \boldsymbol{x}_j) = \langle \boldsymbol{\phi}(\boldsymbol{x}_i), \boldsymbol{\phi}(\boldsymbol{x}_j) \rangle$. The kernel trick is that $\boldsymbol{\phi}$ is never explicitly computed, but from eigendecomposition of $\mathbf{K}$.

12.2 Mercer kernels examples

For real-valued inputs $\boldsymbol{x}_i, \boldsymbol{x}_j \in \mathbb{R}^D$, it is common to use stationary kernels, which are functions of the form $\mathcal{K}(\boldsymbol{x}_i, \boldsymbol{x}_j) = \mathcal{K}(\|\boldsymbol{x}_i - \boldsymbol{x}_j\|)$, i.e. the value depends only on the elementwise distance between the two inputs.

Gaussian Kernel

Also called the squared exponential kernel, exponentiated quadratic kernel, RBF kernel, defined as

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

ℓ is the length scale of the kernel, i.e., the distance over which we expect differences to matter. This is known as the bandwidth parameter.

ARD Kernel

Automatic relevancy determination kernel is a generalization of the Gaussian kernel by replacing Euclidean distance with Mahalanobis distance, defined as

$$\mathcal{K}(\boldsymbol{x}, \boldsymbol{x}') = \sigma^2 \exp\left(-\frac{1}{2}\boldsymbol{r}^\top \boldsymbol{\Sigma}^{-1} \boldsymbol{r}\right)$$

Typically, $\boldsymbol{\Sigma}$ is a diagonal matrix with ℓ_i^2 on the diagonal, thus

$$\mathcal{K}(\boldsymbol{x}, \boldsymbol{x}') = \sigma^2 \exp\left(-\sum_{i=1}^{D} \frac{(x_i - x_i')^2}{2\ell_i^2}\right)$$

which allows each input dimension i to have its own characteristic length scale. For high-dimensional inputs where some dimensions may be more relevant than others (eg. $\ell_i = \infty$ makes that dimension been ignored).

12.3 Combining kernels

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.

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

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.

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

13 Gaussian Processes

Gaussian processes is a way to define distributions over functions of the form $f : \mathcal{X} \to \mathbb{R}$, where $\mathcal{X}$ is any domain. The key assumption is that the function values at a set of $M > 0$ inputs, $\boldsymbol{f} = [f(\boldsymbol{x}_1), \ldots, f(\boldsymbol{x}_M)]$, is jointly Gaussian, with mean $(\boldsymbol{\mu} = m(\boldsymbol{x}_1), \ldots, m(\boldsymbol{x}_M))$ and covariance $\boldsymbol{\Sigma}_{ij} = \mathcal{K}(\boldsymbol{x}_i, \boldsymbol{x}_j)$, where m is a mean function and $\mathcal{K}$ is a positive definite (Mercer) kernel.

Since we assume this holds for any $M > 0$, this includes the case where $M = N + 1$, containing N training points $\boldsymbol{x}_n$ and 1 test point $\boldsymbol{x}_*$. Then, $f(\boldsymbol{x}_*)$ can be inferred from knowledge of $f(\boldsymbol{x}_1), \ldots, f(\boldsymbol{x}_n)$ by manipulating the joint Gaussian distribution $p(f(\boldsymbol{x}_1), \ldots, f(\boldsymbol{x}_N), f(\boldsymbol{x}_*))$. Also, extending this to noisy functions of $f(\boldsymbol{x}_n)$ can deal with problems such as regression or classification.

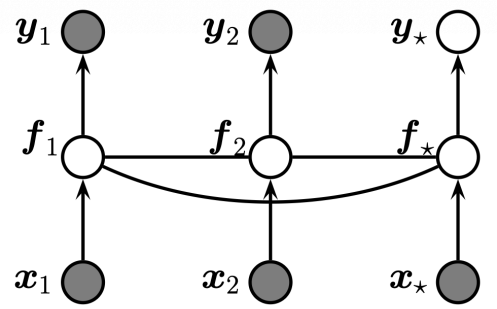


Figure 15: A Gaussian process for 2 training points and 1 test point. The hidden nodes $f_i = f(\boldsymbol{x}_i)$ represent the value of the function at each of the data points. These hidden nodes are fully interconnected with the edge represent the covaiance terms $\Sigma_{ij} = \mathcal{K}(\boldsymbol{x}_i, \boldsymbol{x}_j)$. If the testing points $\boldsymbol{x}_*$ is close to the training points, then the predicted function value $f(\boldsymbol{x}_*)$ will be similar to the training points $f(\boldsymbol{x}_1)$ and $f(\boldsymbol{x}_2)$.

Compactly, the Gaussian Process defined over a function f is:

$$f \sim \mathcal{GP}(m, \mathcal{K})$$

where f, m are $\mathcal{X} \to \mathbb{R}$ functions and $\mathcal{K}$ is a $\mathcal{X} \times \mathcal{X} \to \mathbb{R}$ function. Then, for the N training points $\boldsymbol{x}_n$ plus one test point $\boldsymbol{x}_*$ case,

$$\begin{pmatrix} f(\mathbf{x}_1) \\ \vdots \\ f(\mathbf{x}_N) \\ f(\mathbf{x}_*) \end{pmatrix} \sim \mathcal{N}\left(\begin{pmatrix} m(\mathbf{x}_1) \\ \vdots \\ m(\mathbf{x}_N) \\ m(\mathbf{x}_*) \end{pmatrix}, \begin{pmatrix} \mathbf{K} & \mathbf{k}_* \\ \mathbf{k}_*^\mathsf{T} & K(\mathbf{x}_*, \mathbf{x}_*) \end{pmatrix} \right)$$

13.1 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.

Now for the case of predicting the outputs for new inputs. Specifically, given a test set $\mathbf{X}_*$ with N_* new data points, 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

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

where $\boldsymbol{\mu}_X = [m(\boldsymbol{x}_1), \ldots, m(\boldsymbol{x}_N)]$, $\boldsymbol{\mu}_* = [m(\boldsymbol{x}_{*1}), \ldots, m(\boldsymbol{x}_{*N_*})]$, $\mathbf{K}_{X,X} = \mathcal{K}(\mathbf{X}, \mathbf{X})$ is $N \times N$, $\mathbf{K}_{X,*} = \mathcal{K}(\mathbf{X}, \mathbf{X}_*)$ is $N \times N_*$, and $\mathbf{K}_{*,*} = \mathcal{K}(\mathbf{X}_*, \mathbf{X}_*)$ is $N_* \times N_*$. By the standard rules for conditioning Gaussians, the posterior has the following form

$$\begin{aligned} p(\boldsymbol{f}_*|\mathbf{X}_*, \mathcal{D}) &= \mathcal{N}(\boldsymbol{f}_*|\boldsymbol{\mu}_{*|X}, \boldsymbol{\Sigma}_{*|X}) \\ \boldsymbol{\mu}_{*|X} &= \boldsymbol{\mu}_* + \mathbf{K}_{X,*}^{\mathsf{T}} \mathbf{K}_{X,X}^{-1} (\boldsymbol{f}_X - \boldsymbol{\mu}_X) \\ \boldsymbol{\Sigma}_{*|X} &= \mathbf{K}_{*,*} - \mathbf{K}_{X,*}^{\mathsf{T}} \mathbf{K}_{X,X}^{-1} \mathbf{K}_{X,*} \end{aligned}$$

13.2 Noisy observations

Suppose we observe a training set $\mathcal{D} = \{(\boldsymbol{x}_n, y_n) : n = 1 : N\}$, where $y_n = f(\boldsymbol{x}_n) + \epsilon_n$ is the noisy observation of the function evaluated at $\boldsymbol{x}_n$, and $\epsilon_n \sim \mathcal{N}(0, \sigma_y^2)$ is i.i.d. Gaussian noise. Consider, the covariance of the observed noisy responses is

$$\begin{aligned} \mathrm{Cov}[y_i, y_j] &= \mathrm{Cov}[f(\boldsymbol{x}_i) + \epsilon_i, f(\boldsymbol{x}_j) + \epsilon_j] \\ &= \mathrm{Cov}[f(\boldsymbol{x}_i), f(\boldsymbol{x}_j)] + \mathrm{Cov}[\epsilon_i, \epsilon_j] \\ &= \mathcal{K}(\boldsymbol{x}_i, \boldsymbol{x}_j) + \sigma_y^2 \delta_{ij} \end{aligned}$$

Writing them compactly as,

$$\mathrm{Cov}[\boldsymbol{y} \mid \mathbf{X}] = \mathbf{K}_{X,X} + \sigma_y^2 \mathbf{I}_N \triangleq \mathbf{K}_\sigma$$

The joint density of the observed data and the latent function on the test points is given by

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

Therefore, on a set of test points $\mathbf{X}_*$, the posterior predictive distribution is

$$\begin{aligned} p(\boldsymbol{f}_*|\mathbf{X}_*, \mathcal{D}) &= \mathcal{N}(\boldsymbol{f}_*|\boldsymbol{\mu}_{*|X}, \boldsymbol{\Sigma}_{*|X}) \\ \boldsymbol{\mu}_{*|X} &= \boldsymbol{\mu}_* + \mathbf{K}_{X,*}^{\mathsf{T}} \mathbf{K}_\sigma^{-1} (\boldsymbol{f}_X - \boldsymbol{\mu}_X) \\ \boldsymbol{\Sigma}_{*|X} &= \mathbf{K}_{*,*} - \mathbf{K}_{X,*}^{\mathsf{T}} \mathbf{K}_\sigma^{-1} \mathbf{K}_{X,*} \end{aligned}$$

13.3 Bayesian linear regression

Bayesian linear regression is a special case of Gaussian processes, and can be viewed in weight space or function space. In the following example, lets consider the case where we have N training points $\mathcal{D} = \{(\boldsymbol{x}_n, \boldsymbol{y}_n) : n = 1 : N\}$ and 1 testing points $\boldsymbol{x}_*$.

Weight space view

Consider the linear regression model $y = f(\boldsymbol{x}) + \epsilon$, where $f(\boldsymbol{x}) = \boldsymbol{w}^{\mathsf{T}} \boldsymbol{\phi}(\boldsymbol{x})$ and $\epsilon \sim \mathcal{N}(0, \sigma_y^2)$. With vectorized form:

$$\boldsymbol{y} = \boldsymbol{\Phi}\boldsymbol{w} + \boldsymbol{\epsilon} \implies p(\boldsymbol{y}|\boldsymbol{w}) = \mathcal{N}(\boldsymbol{y} \mid \boldsymbol{\Phi}\boldsymbol{w}, \sigma_y^2 \mathbf{I})$$

where $\boldsymbol{\Phi}$ is the $N \times D$ design matrix including features of all N training points. Plain linear regression stops by maximizing this w.r.t. $\boldsymbol{w}$, but the Bayesian pushes this forward.

Assuming a Gaussian prior $p(\boldsymbol{w}) = \mathcal{N}(\boldsymbol{w}|\mathbf{0}, \boldsymbol{\Sigma}_w)$ on the model parameter $\boldsymbol{w} \in \mathbb{R}^{D \times 1}$, then the posterior is as follows:

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

which is derived by completing the square with $\mathbf{A} = \sigma_y^{-2}\mathbf{\Phi}^\mathsf{T}\mathbf{\Phi} + \mathbf{\Sigma}_w^{-1}$.

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

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

using the standard rules for linear transforms of Gaussians ($\boldsymbol{z} = \boldsymbol{c}^\mathsf{T}\boldsymbol{w} \implies \mathbb{E}[z] = \boldsymbol{c}^\mathsf{T}\boldsymbol{\mu}_w$, $\text{Cov}(z) = \boldsymbol{c}^\mathsf{T}\mathbf{\Sigma}_w\boldsymbol{c}$).

Function space view

Bayesian linear regression can be viewed as a GP with a kernel $\mathcal{K}(\boldsymbol{x}, \boldsymbol{x}') = \boldsymbol{\phi}(\boldsymbol{x})^\mathsf{T}\mathbf{\Sigma}_w\boldsymbol{\phi}(\boldsymbol{x}')$, which measures the similarity between two inputs $\boldsymbol{x}$ and $\boldsymbol{x}'$ in the feature space defined by $\boldsymbol{\phi}$ and is also a valid kernel since covariance $\mathbf{\Sigma}_w$ is always positive semi-definite.

Then, defining the training-training, training-test, and test-test covariance:

$$\mathbf{K} = \mathbf{\Phi}^\top\mathbf{\Sigma}_w\mathbf{\Phi} \in \mathbb{R}^{N\times N}, \quad \mathbf{k}_* = \mathbf{\Phi}^\top\mathbf{\Sigma}_w\boldsymbol{\phi}_* \in \mathbb{R}^{N\times 1}, \quad k_{**} = \boldsymbol{\phi}_*^\top\mathbf{\Sigma}_w\boldsymbol{\phi}_* \in \mathbb{R}$$

where $\boldsymbol{\phi}_*$ is a $D \times 1$ is the feature vector of the single test point. Using these notation, and the matrix inversion lemma, we can rewrite the predictive posterior in the weight space view section as:

$$\begin{aligned}
p(f_*|\mathcal{D}, \boldsymbol{x}_*) &= \mathcal{N}(f_*|\boldsymbol{\mu}_{*|X}, \mathbf{\Sigma}_{*|X}) \\
\boldsymbol{\mu}_{*|X} &= \boldsymbol{\phi}_*^\mathsf{T}\mathbf{\Sigma}_w\mathbf{\Phi}^\mathsf{T}(\mathbf{K} + \sigma_y^2\mathbf{I})^{-1}\boldsymbol{y} = \boldsymbol{k}_*^\mathsf{T}\mathbf{K}_\sigma^{-1}\boldsymbol{y} \\
\mathbf{\Sigma}_{*|X} &= \boldsymbol{\phi}_*^\mathsf{T}\mathbf{\Sigma}_w\boldsymbol{\phi}_* - \boldsymbol{\phi}_*^\mathsf{T}\mathbf{\Sigma}_w\mathbf{\Phi}^\mathsf{T}(\mathbf{K} + \sigma_y^2\mathbf{I})^{-1}\mathbf{\Phi}\mathbf{\Sigma}_w\boldsymbol{\phi}_* = k_{**} - \boldsymbol{k}_*^\mathsf{T}\mathbf{K}_\sigma^{-1}\boldsymbol{k}_*
\end{aligned}$$

which matches the predictive posterior in the GP with noisy observations section with only one testing point $\boldsymbol{x}_*$ instead of N_* testing points $\mathbf{X}_*$.

With kernel trick and a valid kernel $\mathcal{K}$, we can compute the predictive posterior without explicitly computing the feature mapping $\boldsymbol{\phi}$ and allowing for a infinite length feature space.

13.4 Estimating the kernel

Most kernels have some free parameters, which can have a large effect on the predictions from the model. To estimate the kernel parameters $\boldsymbol{\theta}$ (i.e. hyperparameters), we could use exhaustive search over a discrete grid of values, with validation loss as an objective, but this can be quite slow.

Empirical Bayes

For a fully Bayes approach, the kernel hyperparameter $\boldsymbol{\theta}$ is a random variable as well, then

$$p(\boldsymbol{y}|\boldsymbol{X}) = \int\int p(\boldsymbol{y}|\boldsymbol{f}, \boldsymbol{X})p(\boldsymbol{f}|\boldsymbol{X}, \boldsymbol{\theta})p(\boldsymbol{\theta})d\boldsymbol{f}d\boldsymbol{\theta}$$

But this double integral is typically untractable, MCMC or other approximate inference should be used.

Consider an empirical Bayes approach by treating $\boldsymbol{\theta}$ as a fixed but unknown constant, which allows the use of gradient-based optimization methods, which are much faster. In particular, we will maximize the marginal likelihood

$$p(\boldsymbol{y}|\boldsymbol{X}, \boldsymbol{\theta}) = \int p(\boldsymbol{y}|\boldsymbol{f}, \boldsymbol{X})p(\boldsymbol{f}|\boldsymbol{X}, \boldsymbol{\theta})d\boldsymbol{f}$$

The latent Gaussian vector $\boldsymbol{f}$ have been marginalized out, so this is called the marginal likelihood rather than likelihood.

Recall that the prior $p(\boldsymbol{f}|\boldsymbol{X}) = \mathcal{N}(\boldsymbol{f}|\boldsymbol{0}, \boldsymbol{K}_{X,X})$ and the likelihood $p(\boldsymbol{y}|\boldsymbol{f}) = \mathcal{N}(\boldsymbol{y}|\boldsymbol{f}, \sigma_y^2\mathbf{I}_N)$, in which we assume the mean function is 0, then the marginal likelihood is given by

$$\log p(\boldsymbol{y}|\mathbf{X}, \boldsymbol{\theta}) = \log \mathcal{N}(\boldsymbol{y}|\boldsymbol{0}, \mathbf{K}_\sigma) = -\frac{1}{2}\boldsymbol{y}^\mathsf{T}\mathbf{K}_\sigma^{-1}\boldsymbol{y} - \frac{1}{2}\log|\mathbf{K}_\sigma| - \frac{N}{2}\log(2\pi)$$

where the dependence of $\mathbf{K}_\sigma = \mathbf{K}_{X,X} + \sigma_y^2\mathbf{I}_N$ 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 a constant. There is a tradeoff between the first and the second term.

To maximize the marginal likelihood,

$$\begin{aligned}\frac{\partial}{\partial\theta_j}\log p(\boldsymbol{y}|\mathbf{X}, \boldsymbol{\theta}) &= \frac{1}{2}\boldsymbol{y}^\mathsf{T}\mathbf{K}_\sigma^{-1}\frac{\partial\mathbf{K}_\sigma}{\partial\theta_j}\mathbf{K}_\sigma^{-1}\boldsymbol{y} - \frac{1}{2}\mathrm{tr}\left(\mathbf{K}_\sigma^{-1}\frac{\partial\mathbf{K}_\sigma}{\partial\theta_j}\right) \\ &= \frac{1}{2}\mathrm{tr}\left((\boldsymbol{\alpha}\boldsymbol{\alpha}^\mathsf{T} - \mathbf{K}_\sigma^{-1})\frac{\partial\mathbf{K}_\sigma}{\partial\theta_j}\right)\end{aligned}$$

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. Therefore, 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, so multiple restarts might be used.

13.5 GP for classifications

So far, the focus is on GP for regressions with Gaussian likelihoods (eg. $p(\boldsymbol{y}|\boldsymbol{f}) = \mathcal{N}(\boldsymbol{y} \mid \boldsymbol{f}, \sigma_y^2\mathbf{I})$). In this case, the posterior is also a GP, and all computation can be performed analytically. However, if the likelihood is non-Gaussian, such as the Bernoulli likelihood for binary classification, the posterior doesn't have a closed form.

To make classfication inference, the predictive distribution on test data $\boldsymbol{x}_*$ is required,

$$p(f_* \mid \mathcal{D}, \boldsymbol{x}_*) = \int p(f_* \mid \boldsymbol{f}, \boldsymbol{\theta}, \boldsymbol{x}_*)p(\boldsymbol{f}, \boldsymbol{\theta} \mid \mathbf{X}, \boldsymbol{y})\, d\boldsymbol{f}\, d\boldsymbol{\theta}$$

Expanding the posterior distribution term,

$$\begin{aligned}\log(p(\boldsymbol{f}, \boldsymbol{\theta} \mid \mathbf{X}, \boldsymbol{y})) &= \log p(\boldsymbol{y} \mid \boldsymbol{f}) + \log p(\boldsymbol{f} \mid \mathbf{X}) + \log p(\boldsymbol{\theta}) - \log p(\boldsymbol{y} \mid \mathbf{X}) \\ &= \underbrace{\log\mathcal{N}(\boldsymbol{f} \mid \boldsymbol{0}, \mathbf{K}(\mathbf{X}, \mathbf{X})) + \sum_{n=1}^{N}\log\mathrm{Ber}(y_n \mid f_n(\boldsymbol{x}_n)) + \log p(\boldsymbol{\theta})}_{-\mathcal{E}(\boldsymbol{f},\boldsymbol{\theta}),\text{ which has closed form}} - \underbrace{\log p(\boldsymbol{y} \mid \mathbf{X})}_{\text{no closed form}}\end{aligned}$$

Thus, via autograd, $\nabla_{\boldsymbol{f}}\mathcal{E}(\boldsymbol{f}, \boldsymbol{\theta})$ and $\nabla_{\boldsymbol{\theta}}\mathcal{E}(\boldsymbol{f}, \boldsymbol{\theta})$ can be calculated, and use these gradients to perform Hamiltonian Monte Carlo simulations to approximate the posterior $p(\boldsymbol{f}, \boldsymbol{\theta} \mid \mathbf{X}, \boldsymbol{y}) = p(\boldsymbol{f}, \boldsymbol{\theta} \mid \mathcal{D})$.

Intuitively, HMC use $\nabla_{\boldsymbol{f},\boldsymbol{\theta}}\mathcal{E}(\boldsymbol{f}, \boldsymbol{\theta})$ to simulate a physical particle moving through the $(\boldsymbol{f}, \boldsymbol{\theta})$ landscape defined by $-\mathcal{E}$, using gradients to guide it toward high-probability regions. Recording the particle's positions over time yields samples $\{(\boldsymbol{f}^{(s)}, \boldsymbol{\theta}^{(s)})\}_{s=1}^S$. Then, the intractable predictive distribution can be approximated by averaging over the S samples:

$$p(f_* \mid \mathcal{D}, \boldsymbol{x}_*) \approx \frac{1}{S}\sum_{s=1}^{S} p(f_* \mid \boldsymbol{f}^{(s)}, \boldsymbol{\theta}^{(s)}, \boldsymbol{x}_*)$$

where each term in the sum is just a noisy-free observations, which can be calculated analytically using Gaussian conditioning formulas.

13.6 GP and deep learning

There are many connections and similarities between GPs and deep neural networks.

Consider a neural network with a single layer of J RBF units, i.e. the output of this layer in terms of the input vector $\boldsymbol{x}$ is

$$f(\boldsymbol{x}) = \sum_{j=1}^{J} v_j \boldsymbol{\phi}_j(\boldsymbol{x}) \in \mathbb{R}, \quad \boldsymbol{\phi}_j(\boldsymbol{x}) = \boldsymbol{\phi}(\boldsymbol{x}; \boldsymbol{u}_j) = \exp\left(-\frac{\|\boldsymbol{x} - \boldsymbol{u}_j\|^2}{2\ell^2}\right)$$

where $v_j \overset{iid}{\sim} \mathcal{N}(0, \sigma_v^2)$ and $\boldsymbol{u}_j \overset{iid}{\sim} p(\boldsymbol{u})$ are the output weights and RBF centers for each neuron. Therefore,

$$\mathbb{E}[f(\boldsymbol{x})] = \sum_j \mathbb{E}[v_j]\, \mathbb{E}[\boldsymbol{\phi}(\boldsymbol{x}; \boldsymbol{u}_j)] = 0$$

$$\mathrm{Cov}[f(\boldsymbol{x}), f(\boldsymbol{x}')] = \sum_{j,k} \mathbb{E}[v_j v_k]\, \mathbb{E}[\boldsymbol{\phi}_j(\boldsymbol{x}) \boldsymbol{\phi}_k(\boldsymbol{x}')] = \sigma_v^2 \sum_{j=1}^{J} \mathbb{E}_{\boldsymbol{u}}[\boldsymbol{\phi}(\boldsymbol{x}; \boldsymbol{u}) \boldsymbol{\phi}(\boldsymbol{x}'; \boldsymbol{u})]$$

(cross terms vanish since $\mathbb{E}[v_j v_k] = 0$ for $j \neq k$). Rescale $\sigma_v^2 = \omega^2 / J$,

$$\mathrm{Cov}[f(\boldsymbol{x}), f(\boldsymbol{x}')] = \omega^2\, \mathbb{E}_{\boldsymbol{u}}\left[\boldsymbol{\phi}(\boldsymbol{x}; \boldsymbol{u}) \boldsymbol{\phi}(\boldsymbol{x}'; \boldsymbol{u})\right] = \omega^2 \int \boldsymbol{\phi}(\boldsymbol{x}; \boldsymbol{u})\, \boldsymbol{\phi}(\boldsymbol{x}'; \boldsymbol{u})\, p(\boldsymbol{u})\, d\boldsymbol{u}$$

which is an integral (weighted by $p(\boldsymbol{u})$) over a continuous index $\boldsymbol{u}$ of the feature space $\boldsymbol{\phi}$. This makes it into a valid kernel.

For any finite set of test points $\{\boldsymbol{x}_1, \cdots, \boldsymbol{x}_M\}$, define a random vector $\boldsymbol{f}$ such that it is a sum of J random vectors,

$$\boldsymbol{f} = [f(\boldsymbol{x}_1), \cdots, f(\boldsymbol{x}_M)] = \sum_{j=1}^{J} v_j [\boldsymbol{\phi}_j(\boldsymbol{x}_1), \cdots, \boldsymbol{\phi}_j(\boldsymbol{x}_M)]$$

with mean is $\mathbf{0}$ and covariance $\omega^2\, \mathbb{E}_{\boldsymbol{u}}\left[\boldsymbol{\phi}(\boldsymbol{x}_i; \boldsymbol{u}) \boldsymbol{\phi}(\boldsymbol{x}_j; \boldsymbol{u})\right]$ for any entry (i, j). By the Central Limit Theorem, as $J \to \infty$ this sum converges to an exact multivariate Gaussian with the same mean and covariance. By assuming $p(\boldsymbol{u}) \sim \mathcal{N}(\mathbf{0}, s^2 \mathbf{I})$ with $s \to \infty$ (i.e. uninformative prior on the RBF centers), the integral can be expressed as,

$$\mathrm{Cov}[f(\boldsymbol{x}), f(\boldsymbol{x}')] = \mathcal{K}(\boldsymbol{x}, \boldsymbol{x}') = \omega^2 C \exp\left(-\frac{\|\boldsymbol{x} - \boldsymbol{x}'\|^2}{4\ell^2}\right)$$

where C is a normalizing constant. Therefore, we see that a neural network with a single, infinitely wide layer of RBF units is equivalent to a GP with an RBF kernel.

13.7 GP for dimensionality reduction

Gaussian Process Factor Analysis (GPFA) is an useful tool for reducing dimensionality of a time series data. Recall that:

Factor Analysis (FA) models a collection of high-dimensional observations as noisy linear combinations of a small number of shared latent factors. For a single observation $\mathbf{y}_t \in \mathbb{R}^q$,

$$\mathbf{y}_t = \mathbf{W}\mathbf{x}_t + \boldsymbol{\mu} + \boldsymbol{\epsilon}_t, \qquad \boldsymbol{\epsilon}_t \sim \mathcal{N}(\mathbf{0}, \boldsymbol{\Psi}),$$

where

- $\mathbf{x}_t \in \mathbb{R}^p$ is the latent factor vector, $p \ll q$, with $x_i(t)$ denotes the ith component in $\mathbf{x}_t$
- $\mathbf{W} \in \mathbb{R}^{q \times p}$ is the factor loading matrix, $\boldsymbol{\mu} \in \mathbb{R}^q$ is a mean offset,
- $\boldsymbol{\Psi} \in \mathbb{R}^{q \times q}$ is a diagonal covariance matrix, so that any correlation among the observed dimensions of $\mathbf{y}_t$ is explained entirely through the shared latents $\mathbf{x}_t$, not through correlated noise.

The defining assumption of FA is that the latents are drawn independently and identically at each observation,

$$\mathbf{x}_t \sim \mathcal{N}(\mathbf{0}, \mathbf{I}_p), \qquad \text{i.i.d. across } t.$$

Ordinary FA has no notion of correlation between observations: permuting the index t does not change the model or its likelihood.

A Gaussian Process (GP) is a distribution over functions, fully specified by a mean function $m(t)$ and a covariance (kernel) function $k(t, t')$:

$$f(t) \sim \mathcal{GP}\big(m(t), \mathcal{K}(t, t')\big).$$

For any finite collection of input points $\{t_i\}_{i=1}^T$, the corresponding function values are jointly Gaussian,

$$\big(f(t_1), \ldots, f(t_T)\big) \sim \mathcal{N}(\mathbf{m}, \mathbf{K}), \qquad \mathbf{m}_i = m(t_i), \quad \mathbf{K}_{ij} = \mathcal{K}(t_i, t_j).$$

Commonly, the squared-exponential (RBF) kernel is used throughout GPFA,

$$k(t, t') = \sigma^2 \exp\left(-\frac{(t - t')^2}{2\tau^2}\right),$$

where σ^2 sets the marginal variance and τ is a characteristic timescale: pairs of points with $|t - t'| \ll \tau$ are nearly perfectly correlated (enforcing smoothness), while pairs with $|t - t'| \gg \tau$ are nearly independent.

The GPFA Model

GPFA combines the observation model of FA with a GP prior over each latent dimension in place of the i.i.d. Gaussian prior. Concretely, for a sequence of observations $\mathbf{y}(t_1), \ldots, \mathbf{y}(t_T)$ at times $t_1, \ldots, t_T$,

$$\begin{aligned} \mathbf{y}(t) &= \mathbf{W}\mathbf{x}(t) + \boldsymbol{\mu} + \boldsymbol{\epsilon}(t), \qquad \boldsymbol{\epsilon}(t) \sim \mathcal{N}(\mathbf{0}, \boldsymbol{\Psi}) \\ x_i(t) &\sim \mathcal{GP}\big(0, \mathcal{K}_i(t, t')\big), \qquad i = 1, \ldots, p \end{aligned}$$

with the following features:

- Each latent dimension i has its own kernel $\mathcal{K}_i(t, t')$ (typically RBF), with its own timescale τ_i and variance σ_i^2, allowing different latents to evolve at different characteristic speeds.
- The p latent GPs are assumed independent.
- $\mathbf{W}$, $\boldsymbol{\mu}$, and $\boldsymbol{\Psi}$ (diagonal) play exactly the same role as in standard FA.

The only change relative to FA is thus replacing the independent-across-time prior on $\mathbf{x}_t$ with a temporally correlated GP prior on each $x_i(t)$.

Fixing the T observation times $t_1, \ldots, t_T$ of a trial, stack the latent values for dimension i into $\mathbf{x}_i = \big(x_i(t_1), \ldots, x_i(t_T)\big)^\top \in \mathbb{R}^T$. By the GP marginalization property,

$$\mathbf{x}_i \sim \mathcal{N}(\mathbf{0}, \mathbf{K}_i), \qquad [\mathbf{K}_i]_{ab} = \mathcal{K}_i(t_a, t_b).$$

Concatenating all p latent trajectories into a single vector $\mathbf{x} = (\mathbf{x}_1^\top, \ldots, \mathbf{x}_p^\top)^\top \in \mathbb{R}^{pT}$, and using the independence across latent dimensions,

$$\mathbf{x} \sim \mathcal{N}(\mathbf{0}, \tilde{\mathbf{K}}), \qquad \tilde{K} = \mathrm{blkdiag}(\mathbf{K}_1, \ldots, \mathbf{K}_p) \in \mathbb{R}^{pT \times pT}$$

Similarly stacking the observations across time, $\mathbf{y} = \big(\mathbf{y}(t_1)^\top, \ldots, \mathbf{y}(t_T)^\top\big)^\top \in \mathbb{R}^{qT}$, the linear-Gaussian observation model can be written compactly as

$$\mathbf{y} = \tilde{\mathbf{W}}\,\mathbf{x} + \tilde{\boldsymbol{\mu}} + \boldsymbol{\epsilon}, \qquad \boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \tilde{\boldsymbol{\Psi}}),$$

where $\tilde{\mathbf{W}}$ places a copy of $\mathbf{W}$ (suitably permuted to match the stacking of $\mathbf{x}$) at each time block, $\tilde{\boldsymbol{\mu}}$ repeats $\boldsymbol{\mu}$ across the T blocks, and $\tilde{\boldsymbol{\Psi}} = I_T \otimes \boldsymbol{\Psi}$.

Because both the prior $p(\mathbf{x})$ and the observation model are linear-Gaussian, $\mathbf{y}$ and $\mathbf{x}$ are jointly Gaussian, and the posterior $p(\mathbf{x} \mid \mathbf{y})$ is again Gaussian with closed-form mean and covariance obtained via the standard Gaussian conditioning formulas.

$$p(\mathbf{x} \mid \mathbf{y}) = \mathcal{N}\Big(\mathbf{x} \;\Big|\; \tilde{\mathbf{K}}\tilde{\mathbf{W}}^\top \tilde{\mathbf{Q}}^{-1}(\mathbf{y} - \tilde{\boldsymbol{\mu}}),\;\; \tilde{\mathbf{K}} - \tilde{\mathbf{K}}\tilde{\mathbf{W}}^\top \tilde{\mathbf{Q}}^{-1}\tilde{\mathbf{W}}\tilde{\mathbf{K}}\Big), \qquad \tilde{\mathbf{Q}} = \tilde{\mathbf{W}}\tilde{\mathbf{K}}\tilde{\mathbf{W}}^\top + \tilde{\boldsymbol{\Psi}}$$

GPFA parameters $\theta = \{\mathbf{W}, \boldsymbol{\mu}, \boldsymbol{\Psi}, \{\sigma_i^2, \tau_i\}_{i=1}^p\}$ are typically fit by Expectation-Maximization, mirroring FA but with two changes:

E-step. Given current θ, compute the posterior mean and covariance of $p(\mathbf{x} \mid \mathbf{y})$. This step performs GP regression across all p latents and T time points jointly, rather than the simple pointwise Gaussian conditioning used in ordinary FA's E-step.

M-step. Update $\mathbf{W}$, $\boldsymbol{\mu}$, $\boldsymbol{\Psi}$ in closed form same as in FA. The GP kernel hyperparameters $\{\sigma_i^2, \tau_i\}$ do not have closed-form updates and are instead optimized numerically (e.g., by gradient ascent on the expected complete-data log-likelihood).