UNIVERSITAT DE VALÈNCIA

NOTES

Machine Learning II

Author:
Pablo CARRERA

"People worry that computers will get too smart and take over the world, but the real problem is that they're too stupid and they've already taken over the world."

Pedro Domingos

Contents

Chapter 1

Component Analysis

1.1 What is Component Analysis?

Consider a dataset represented as a matrix (or a database table), such that each row represents a set of attributes (or features or dimensions) that describe a particular instance of something. If the number of attributes is large, then the space of unique possible rows is exponentially large. Thus, the larger the dimensionality, the more difficult it becomes to sample the space. This causes many problems. Algorithms that operate on high-dimensional data tend to have a very high time complexity. This has become known as the curse of dimensionality. Reducing data into fewer dimensions often makes analysis algorithms more efficient, and can help machine learning algorithms make more accurate predictions. Generally, there are three research streams for the purpose of dimensionality reduction:

- Component Analysis: try to find hidden components via data analysis techniques.
- Manifold Learning: construct a representation of data lying on a low dimensional manifold embedded in a high-dimensional space, modeling the local geometry of the data by constructing a graph.
- Functional Analysis: attempt to develop new forms of functional representation by recent advances in harmonic analysis.

According to the models and assumptions employed, Component Analysis methods could be further divided into three classes using:

- Generative models: methods in this class include Principal Component Analysis (PCA), Nonnegative Matrix Factorization (NMF), and Independent Component Analysis (ICA).
- Discriminative models: methods in this class include Linear Discriminant Analysis (LDA), Oriented Component Analysis (OCA), Canonical Correlation Analysis (CCA), and Relevant Component Analysis (RCA).
- Standard extensions of linear models: this class includes latent variable models, tensor factorization, and kernel methods.

1.2 Linear methods

First we define the notation, let

$$\mathbf{x} = \left[x^{(1)}, ..., x^{(d_x)}\right]^T$$
$$\mathbf{y} = \left[y^{(1)}, ..., y^{(d_y)}\right]^T$$

be column vectors, in such a way that matrices contains vectors in rows

$$\mathbf{X} = \begin{pmatrix} \mathbf{x}_1^T \\ \vdots \\ \mathbf{x}_n^T \end{pmatrix}$$
$$\mathbf{Y} = \begin{pmatrix} \mathbf{y}_1^T \\ \vdots \\ \mathbf{y}_n^T \end{pmatrix}$$

We can compute the mean value of the elements of the matrix as $E[\mathbf{X}] = \mu_X$, this allow us to define a centered matrix $\tilde{\mathbf{X}} = \mathbf{X} - \mu_X$, with zero mean in each column, and the empirical covariance matrix

$$C_{XX} = \frac{1}{n}\tilde{\mathbf{X}}^T\tilde{\mathbf{X}}$$

, where n represent the number of observations. Covariance is a measure of how much two variables change together, so a completely diagonal covariance matrix means that all features are uncorrelated, which is useful for learning, a diagonal covariance matrix means no redundant features.

1.2.1 Principal Component Analysis

Principal Component Analysis (PCA) is mathematically defined as an orthogonal linear transformation that transforms the data to a new coordinate system such that the greatest variance by some projection of the data comes to lie on the first coordinate (called the first principal component), the second greatest variance on the second coordinate, and so on. This allow us to keep few components without loosing too much information.

PCA is also used as a means of discovering unusual facets of a data set. This can be accomplished by plotting the top few pairs of principal component scores (those having largest variances) in a scatterplot. Such a scatterplot can identify whether X actually lives on a low-dimensional linear manifold of $\mathbb{R}^{d_X}$, as well as provide help identifying multivariate outliers, distributional peculiarities, and clusters of points. If the bottom set of principal components each have near-zero variance, then this implies that those principal components are essentially constant and, hence, can be used to identify the presence of collinearity and possibly outliers that might distort the intrinsic dimensionality of the vector X.

Suppose the input variables are the components of a vector $\mathbf{x} = (x_1^T, ..., x_{d_x}^T)^T$. PCA replace the input variables $\{x_1, ..., x_n\}$ by a new set of variables $\{\xi_1, ..., \xi_n\}$ that are

linear combinations of the original ones. that is $\mathbf{X}' = U\mathbf{X}$. The derived variables are constructed so as to be uncorrelated with each other and ordered by the decreasing values of their variances. To obtain the weights that define the principal components, we maximize the preserved variance of the data,

$$\begin{aligned}
&\max\left[\sum_{i=1}^{n}\text{var}(\tilde{\mathbf{X}})\right] = \max\left[\text{Tr}(\tilde{\mathbf{X}}^T{}'\tilde{\mathbf{X}}')\right] \\
&\implies \max\left[\sum_{i=1}^{n}\text{var}(\tilde{\mathbf{X}})\right] = \max\left[\text{Tr}\left((\tilde{\mathbf{X}}U)^T(\tilde{\mathbf{X}}U)\right)\right] \\
&\implies \max\left[\sum_{i=1}^{n}\text{var}(\tilde{\mathbf{X}})\right] = \max\left[\text{Tr}\left(U^T\tilde{\mathbf{X}}\tilde{\mathbf{X}}U\right)\right] \\
&\implies \max\left[\sum_{i=1}^{n}\text{var}(\tilde{\mathbf{X}})\right] = \max\left[\text{Tr}\left(U^TC_{XX}U\right)\right]
\end{aligned}$$

where $U^TU = I$, so, using Lagrange multipliers to solve this constrained maximization problem we obtain that the solution to U is given by the Eigenvalue Decomposition of C_{XX}, it will be given by the eigenvectors $\mathbf{u}_k$ of C_{XX} associated with the largest eigenvalues λ_k.

1.2.2 Partial Least Squares

In Partial Least Squares (PLS) instead of finding hyperplanes of maximum variance between the response and independent variables, we try to find a linear regression model by projecting the predicted variables and the observable variables to a new space. To obtain the weights of the decomposition, we maximize the covariance between the projected input and the output data in this new space,

$$\begin{aligned}
&\max\left[\sum_{i=1}^{n}\text{cov}(\tilde{\mathbf{X}},\mathbf{Y})\right] = \max\left[\text{Tr}(\tilde{\mathbf{X}}^T{}'\mathbf{Y}')\right] \\
&\implies \max\left[\sum_{i=1}^{n}\text{var}(\tilde{\mathbf{X}})\right] = \max\left[\text{Tr}\left((\tilde{\mathbf{X}}U)^T(\mathbf{Y}V)\right)\right] \\
&\implies \max\left[\sum_{i=1}^{n}\text{var}(\tilde{\mathbf{X}})\right] = \max\left[\text{Tr}\left(U^T\tilde{\mathbf{X}}\mathbf{Y}V\right)\right] \\
&\implies \max\left[\sum_{i=1}^{n}\text{var}(\tilde{\mathbf{X}})\right] = \max\left[\text{Tr}\left(U^TC_{XY}V\right)\right]
\end{aligned}$$

where $U^TU = I$ and $V^TV = I$, so, the solution to U and V is given by the Singular Value Decompostion of C_{XY}, it will be given by the $\mathbf{u}_k$ of C_{XX} associated with the largest singular values λ_k.

1.2.3 Orthonormalized Partial Least Squares

In Orthonormalized Partial Least Squares (OPLS) we choose the projection U that minimizes the MSE error using a linear regression,

$$U = \text{argmin}\left[||\mathbf{Y} - (\mathbf{XU})\mathbf{W}||_F^2\right]$$

, where $\mathbf{W} = (\mathbf{XU})^{\dagger}\mathbf{Y}$. With this in mind we can rewrite the problem as

$$\max\left[\text{Tr}\left(\mathbf{U}^T\mathbf{X}^T\mathbf{Y}\mathbf{Y}^T\mathbf{XU}\right)\right]$$

subject to $(\mathbf{XU})^T(\mathbf{XU}) = \boldsymbol{I}$.

1.2.4 Linear Discriminant Analysis

The objective of LDA is to perform dimensionality reduction while preserving as much of the class discriminatory information as possible. A possible objective function will be then the distance between the projected means, but is not a very good measure since it does not take into account the standard deviation within the classes. The solution is then is to maximize a function that represents the difference between the means, normalized by a measure of the within-class scatter,

$$\max\left[\text{Tr}\left((\mathbf{U}^T\mathbf{S}_B\mathbf{U})(\mathbf{U}^T\mathbf{S}_W\mathbf{U})^{-1}\right)\right]$$

where

$$\mathbf{S}_B = \sum_{i=1} n_i(\mu_i - \mu)(\mu_i - \mu)^T$$
$$\mathbf{S}_W = \sum_{i=1}\sum_{j\in\mathcal{C}_i}(x_j - \mu_i)(x_j - \mu_i)^T$$

are the between-class and within-class scatter matrices. Including Lagrange multipliers, this is equivalent to the generalized eigenvalue problem

$$\mathbf{S}_B\mathbf{u}_i = \lambda_i\mathbf{S}_W\mathbf{u}_i$$

where $\boldsymbol{S}_B$ is a sum of C matrices of rank one, so $\boldsymbol{S}_B$ will be of rank $C-1$, and only $C-1$ eigenvalues will be non-zero.

1.3 Nonlinear methods

1.3.1 Kernel Principal Component Analysis

The idea of Kernel Principal Component Analysis (KPCA) is to find the projection maximizing the variance of the mapped data

$$\max\left[\text{Tr}(\tilde{\boldsymbol{\Phi}}^T\tilde{\boldsymbol{\Phi}}')\right]$$

, where $\boldsymbol{U}^T\boldsymbol{U} = \boldsymbol{I}$, and using the representer's theorem $\boldsymbol{U} = \boldsymbol{\Phi}^T\boldsymbol{A}$ we obtain the equivalent problem

$$\max \left[\text{Tr}(\boldsymbol{A}^T\boldsymbol{K}\boldsymbol{K}\boldsymbol{A}) \right]$$

, where $\boldsymbol{A}^T\boldsymbol{K}\boldsymbol{A} = \boldsymbol{I}$. Including the Lagrange multipliers we obtain the same result that the original PCA, the eigenvector associated with the largest eigenvalue will produce the solution to U.

1.3.2 Kernel Orthonormalized Partial Least Squares

Finally the idea of Kernel Orthonormalized Partial Least Squares (KOPLS) is to choose the projection that minimizes the MSE in the mapped space

$$\max \left[\text{Tr} \left((\boldsymbol{\Phi}\boldsymbol{U})^T\boldsymbol{Y}\boldsymbol{Y}^T(\boldsymbol{\Phi}\boldsymbol{U}) \right) \right]$$

subject to $(\boldsymbol{\Phi}\boldsymbol{U})^T(\boldsymbol{\Phi}\boldsymbol{U}) = \boldsymbol{I}$. Then using the representer's theorem $\boldsymbol{U} = \boldsymbol{\Phi}^T\boldsymbol{A}$, this problem is equivalent to

$$\max \left[\text{Tr} \left(\boldsymbol{A}^T\boldsymbol{K}_X\boldsymbol{K}_Y\boldsymbol{K}_X\boldsymbol{A} \right) \right]$$

subject to $\boldsymbol{A}^T\boldsymbol{K}_X\boldsymbol{K}_X\boldsymbol{A} = \boldsymbol{I}$, which is solved again as a generalized eigenvalue problem.

Chapter 2

Manifold Learning

2.1 What is Manifold Learning?

High-dimensional data, meaning that requires more than two or three dimensions to represent, can be difficult to interpret. One approach to simplification is to assume that the data of interest lie on an embedded non-linear manifold within the higher-dimensional space. If the manifold is of low enough dimension, the data can be visualised in the low-dimensional space. The simplest description of manifold learning is then, that it is a class of algorithms for recovering a low-dimensional manifold embedded in a high-dimensional ambient space.

The primary tool of all embedding algorithms is the set of eigenvectors associated with the top few or bottom few eigenvalues of an appropriate random matrix. These algorithms are known as spectral embedding methods and are designed to recover linear or nonlinear manifolds, usually in high-dimensional spaces.

Linear methods, include Principal Component Analysis (PCA) and Multidimensional Scaling (MDS). But, when linear manifold learning does not result in a good low-dimensional representation of high-dimensional data, then we are led to consider the possibility that the data may lie on or near a nonlinear manifold. These nonlinear algorithms include Isomap, Local Linear Embedding, Laplacian Eigenmaps, Diffusion Maps, Hessian Eigenmaps, and different interpretations of what nonlinear PCA means.

Whereas linear manifold-learning methods seek to preserve global structure on the manifold (mapping close points on a manifold to close points in low-dimensional space, and distant points to distant points), most nonlinear manifold-learning methods are considered to be local methods, which seek to preserve local structure in small neighborhoods on the manifold.

2.2 Linear methods

Identifying a linear manifold embedded in a higher-dimensional space is closely related to the classical statistics problem of linear dimensionality reduction. The recommended way of accomplishing linear dimensionality reduction is to create a reduced set of linear transformations of the input variables. Linear transformations are projection methods, and so the problem is to derive a sequence of low-dimensional projections of the input data that possess some type of optimal properties.

Although data tend not to live on a linear manifold, we view the problem as having two kinds of motivations. The first such motivation is to assume that the data live close to a linear manifold, the distance off the manifold determined by a random error (or noise) component. A second way of thinking about linear manifold learning is that a linear manifold is really a simple linear approximation to a more complicated type of nonlinear manifold that would probably be a better fit to the data. In both scenarios, the intrinsic dimensionality of the linear manifold is taken to be much smaller than the dimensionality of the data.

2.2.1 Principal Component Analysis

Principal component analysis (PCA) was introduced as a technique for deriving a reduced set of orthogonal linear projections of a single collection of correlated variables, $\boldsymbol{X} = (\boldsymbol{X_1}, ..., \boldsymbol{X}_{d_X})^T$, where the projections are ordered by decreasing variances. The amount of information in a random variable can be measured by its variance, which is a second-order property. PCA has also been referred to as a method for "decorrelating" $\boldsymbol{X}$. PCA is also used as a means of discovering unusual facets of a data set. This can be accomplished by plotting the top few pairs of principal component scores (those having largest variances) in a scatterplot. Such a scatterplot can identify whether $\boldsymbol{X}$ actually lives on a low-dimensional linear manifold of $\mathbb{R}^{d_X}$ as well as provide help identifying multivariate outliers, distributional peculiarities, and clusters of points. If the bottom set of principal components each have near-zero variance, then this implies that those principal components are essentially constant and, hence, can be used to identify the presence of collinearity and possibly outliers that might distort the intrinsic dimensionality of the vector $\boldsymbol{X}$.

Suppose that the input variables are the components of a random d_X-vector $\boldsymbol{X} = (\boldsymbol{X_1}, ..., \boldsymbol{X}_{d_X})^T$. Further, assuma that $\boldsymbol{X}$ has mean vector $\mathrm{E}[\boldsymbol{X}] = \boldsymbol{\mu}_X$, and a $d_X \times d_X$ covariance matrix $\mathrm{E}[(\boldsymbol{X} - \boldsymbol{\mu}_X)(\boldsymbol{X} - \boldsymbol{\mu}_X)] = \boldsymbol{\Sigma}_{XX}$. PCA replaces the input variables $X_1, ..., X_{d_X}$ by a new set of derived variables $\xi_1, ..., \xi_d$, with $d < d_X$, where

$$\xi_i = \boldsymbol{b}_j^T \boldsymbol{X} = b_{i1} X_1 + ... + b_{id_X} X_{d_X}$$

The derived variables are constructed so as to be uncorrelated with each other and ordered by decreasing vales of their variances. To obtain the vectors $\boldsymbol{b}_i$, which define the principal components,we minimize the loss of information due to replacement. In PCA information is interpreted as the total variation of the original input variables

$$\sum_{i=1}^{d_X} \mathrm{var}(X_i) = \mathrm{Tr}(\boldsymbol{\Sigma}_{XX})$$

, where from the spectral decomposition theorem we can write $\boldsymbol{\Sigma}_{XX} = \boldsymbol{U}\boldsymbol{\Lambda}\boldsymbol{U}^T$ with $\boldsymbol{U}^T\boldsymbol{U} = \boldsymbol{I}$, where the diagonal matrix $\boldsymbol{\Lambda}$ has as diagonal elements the eigenvalues $\lambda_1, ..., \lambda_{d_X}$ of $\boldsymbol{\Sigma}_{XX}$, and the columns of the matrix $\boldsymbol{U}$ are the eigenvectors $\boldsymbol{\Sigma}_{XX}$. Thus, the total variation is

$$\mathrm{Tr}(\boldsymbol{\Sigma}_{XX}) = \mathrm{Tr}(\boldsymbol{\Lambda}) = \sum_{i=1}^{d_X} \lambda_i$$

. The i-th coefficient vector $\boldsymbol{b}_i = (b_{1i}, ..., b_{d_X i})$ is chosen such that the top d linear projections ξ_i, with $i = 1, ..., d$ are ranked in importance through their variances,

which are listed in decreasing order of magnitude, $\text{var}(\xi_1) > ... > \text{var}(\xi_d)$. The first d linear projections are known as the top d principal components of $\boldsymbol{X}$.

Let $\boldsymbol{B} = (\boldsymbol{b}_1, ..., \boldsymbol{b}_{d_X})^T$ denote the $d \times d_X$ matrix of weitghts and let $\boldsymbol{\xi} = \boldsymbol{BX}$ be the vector of linear projections of $\boldsymbol{X}$. We wish to find a vector d_X-dimensional vector $\boldsymbol{\mu}$ and a $d_X \times d$ matrix $\boldsymbol{A}$ such that $\boldsymbol{X} \approx \boldsymbol{\mu} + \boldsymbol{A\xi}$ in some least square sense

$$\text{E}\left[(\boldsymbol{X} - \boldsymbol{\mu} - \boldsymbol{A\xi})^T(\boldsymbol{X} - \boldsymbol{\mu} - \boldsymbol{A\xi})\right]$$

, or substituting $\boldsymbol{\xi} = \boldsymbol{BX}$,

$$\text{E}\left[(\boldsymbol{X} - \boldsymbol{\mu} - \boldsymbol{ABX})^T(\boldsymbol{X} - \boldsymbol{\mu} - \boldsymbol{ABX})\right]$$

We can think about this minimization problem in another way. Because $\boldsymbol{A}$ is an $d_X \times d$ matrix and B is a $d \times d_x$ matrix, where $d < d_X$, then $\boldsymbol{C} = \boldsymbol{AB}$ is an $d_x \times d_X$ matrix of multivariate regression coefficients obtained by regressing $\boldsymbol{X}$ on itself while requiring $\boldsymbol{C}$ to have reduced rank d, that is, the rank of $\boldsymbol{C}$ is $r(\boldsymbol{C}) = d$. The rank condition implies that there may be a number of linear constraints on the set of regression coefficients. However, the value of the rank d and also the number and nature of those constraints may not be known prior to statistical analysis. We distinguish between the full-rank case when $d = d_X$ and the reduced-rank case when $d < d_X$.

The least squares criterion is minimized by

$$\begin{aligned}
\boldsymbol{A}^{(d)} &= (\boldsymbol{v}_1, ..., \boldsymbol{v}_d) \\
\boldsymbol{B}^{(d)} &= (\boldsymbol{v}_1, ..., \boldsymbol{v}_d) \\
\boldsymbol{\mu}^{(d)} &= (\boldsymbol{I} - \boldsymbol{A}^{(d)}\boldsymbol{B}^{(d)})\boldsymbol{\mu}_X
\end{aligned}$$

where $\boldsymbol{v}_i$ are the eigenvectors associated with the i-th largest eigenvalue λ_i of $\boldsymbol{\Sigma}_{XX}$. Thus, the best rank d reconstruction of the original data $\boldsymbol{X}$ is given by

$$\hat{\boldsymbol{X}}^{(t)} = \boldsymbol{\mu}^{(d)} + \boldsymbol{C}^{(d)}\boldsymbol{X} = \boldsymbol{\mu}_X + \boldsymbol{C}^{(d)}(\boldsymbol{X} - \boldsymbol{\mu}_X)$$

where

$$\boldsymbol{C}^{(d)} = \boldsymbol{A}^{(d)}\boldsymbol{B}^{(d)} = \sum_{i=1}^{d} \boldsymbol{v}_i\boldsymbol{v}_j^T$$

is the multivariate reduced-rank regression coefficient matrix. The first d principal components of $\boldsymbol{X}$ are given by the linear projections

$$\boldsymbol{\xi}_i = \boldsymbol{v}^T\boldsymbol{X}, \qquad i = 1, ..., d$$

As we just saw, the PCA applied to the population is constructed using the eigenvalues and eigenvectors of the population matrix $\boldsymbol{\Sigma}_{XX}$, however in practice it, and also $\boldsymbol{\mu}_X$ will be unknown and has to be estimated by sample data. So, suppose that we have n independent and identically distributed observations $\boldsymbol{X}_1, ..., \boldsymbol{X}_n$ on $\boldsymbol{X}$. First, we estimate the sample mean

$$\hat{\boldsymbol{\mu}}_X = \frac{1}{n}\sum_{i=1}^{n} \boldsymbol{X}_i$$

, now let $\tilde{\boldsymbol{X}}_i = \boldsymbol{X}_i - \hat{\boldsymbol{\mu}}_X$ and set $\tilde{\boldsymbol{X}} = (\tilde{\boldsymbol{X}}_1, ..., \tilde{\boldsymbol{X}}_n)$ to be a $n \times d_X$ matrix. We then estimate the sample covariance matrix

$$\hat{\boldsymbol{\Sigma}}_{XX} = \frac{1}{n} \tilde{\boldsymbol{X}} \tilde{\boldsymbol{X}}^T$$

. Then in d_X is fixed and n increases, the eigenvalues and eigenvectors of the sample $\hat{\boldsymbol{\Sigma}}_{XX}$ are consistent estimators of the corresponding eigenvalues and eigenvectors of the population $\boldsymbol{\Sigma}_{XX}$

2.2.2 Multidimensional Scaling

In Multidimensional Scaling (MDS), one is given a measure of proximity of the data, usually a Euclidean distance, and the problem is to reconstruct the distribution of the data as closely as possible while keeping these proximity. Here we have to determine the dimensionality of the underlying (linear) manifold that is consistent with the given measures of proximity.

Although there are several different versions of MDS, we describe here only the classical scaling method. Suppose we are given n points $\boldsymbol{X}_1, ..., \boldsymbol{X}_n \in \mathbb{R}^{d_X}$ from which we compute a $n \times n$ matrix of dissimilarities Δ, where

$$\delta_{ij} = ||\boldsymbol{X}_i - \boldsymbol{X}_j|| = \left[\sum_{k=1}^{n} (X_{ik} - X_{jk})^2 \right]^{1/2}$$

, is the dissimilarity between $\boldsymbol{X}_i = (X_{i1}, ..., X_{in})^T$ and $\boldsymbol{X}_j = (X_{j1}, ..., X_{jn})^T$. These dissimilarities are the Euclidean distances between all pairs of points in that space.

Squaring both sides and expanding

$$\delta_{ij}^2 = ||\boldsymbol{X}_i||^2 + ||\boldsymbol{X}_j||^2 - 2\boldsymbol{X}_i^T \boldsymbol{X}_j$$

we can then define $a_{ij} = -\frac{1}{2}\delta_{ij}^2$ and $b_{ij} = \boldsymbol{X}_i^T \boldsymbol{X}_j$ in such a way that

$$\begin{aligned} b_{ij} &= -\frac{1}{2}(\delta_{ij}^2 - \delta_{i0}^2 - \delta_{j0}^2) \\ \implies b_{ij} &= a_{ij} - \frac{1}{n}\sum_{j=1}^{n} a_{ij} - -\frac{1}{n}\sum_{i=1}^{n} a_{ij} + -\frac{1}{n^2}\sum_{i=1}^{n}\sum_{j=1}^{n} a_{ij} \end{aligned}$$

then defining $\boldsymbol{J}$ as a matrix of all ones and $\boldsymbol{H} = \boldsymbol{I} - \frac{1}{n}\boldsymbol{J}$ we can obtain a matrix representation of b_{ij}

$$\begin{aligned} \boldsymbol{B} &= \boldsymbol{A} - \frac{1}{n}\boldsymbol{A}\boldsymbol{J} - \frac{1}{n}\boldsymbol{J}\boldsymbol{A} + \frac{1}{n^2}\boldsymbol{J}\boldsymbol{A}\boldsymbol{J} \\ \implies \boldsymbol{B} &= \left(\boldsymbol{I} - \frac{1}{n}\boldsymbol{J}\right)\boldsymbol{A}\left(\boldsymbol{I} - \frac{1}{n}\boldsymbol{J}\right) \\ \implies \boldsymbol{B} &= \boldsymbol{H}\boldsymbol{A}\boldsymbol{H} \end{aligned}$$

where we can see that the mean of each row and each column is removed and replaced by the mean of the matrix, so it is usually said that $\boldsymbol{B}$ is double centered.

We would now like to find a set of d-dimensional points $\boldsymbol{Y}_1, ..., \boldsymbol{Y}_n \in \mathbb{R}^d$, called principal coordinates, that could represent the d_X-dimensional points $\boldsymbol{X}_1, ..., \boldsymbol{X}_n \in \mathbb{R}^{d_X}$ such that the interpoint distances in the new space match the ones in the original space. If we define dissimilarities as Euclidean distances between pairs of points, then the resulting representation will be equivalent to PCA because the principal coordinates are identical to the first d principal components scores of the $\boldsymbol{X}_1, ..., \boldsymbol{X}_n \in \mathbb{R}^{d_X}$.

Let $\boldsymbol{\Lambda} = \text{diag}(\lambda_1, ..., \lambda_n)$ be the diagonal matrix of eigenvalues of $\boldsymbol{B}$ and let $\boldsymbol{V} = (\boldsymbol{v}_1, ..., \boldsymbol{v}_n)$ be the matrix whose columns are the eigenvectors of $\boldsymbol{B}$. By the spectral theorem $\boldsymbol{B} = \boldsymbol{V}\boldsymbol{B}\boldsymbol{V}^T$. If $\boldsymbol{B}$ is nonnegative-definite with rank $r(\boldsymbol{B}) = d < n$, the largest d eigenvalues will be positive and the remaining $n - d$ eigenvalues will be zero. Let $\boldsymbol{\Lambda}_1 = \text{diag}(\lambda_1, ..., \lambda_d)$ be the diagonal matrix of positive eigenvalues and let $\boldsymbol{V}_1 = (\boldsymbol{v}_1, ..., \boldsymbol{v}_d)$ be the matrix of corresponding eigenvectors. Then

$$\boldsymbol{B} = \boldsymbol{V}_1\boldsymbol{\Lambda}_1\boldsymbol{V}_1^T = (\boldsymbol{V}_1\boldsymbol{\Lambda}_1^{1/2})(\boldsymbol{\Lambda}_1^{1/2}\boldsymbol{V}_1) = \boldsymbol{Y}\boldsymbol{Y}^T$$

where

$$\boldsymbol{Y} = \boldsymbol{V}_1\boldsymbol{\Lambda}_1^{1/2} = (\sqrt{\lambda_1}\boldsymbol{v}_1, ..., \sqrt{\lambda_d}\boldsymbol{v}_d) = (\boldsymbol{Y}_1, ..., \boldsymbol{Y}_d)^T$$

so, the principal components are the columns $\boldsymbol{Y}_1, ..., \boldsymbol{Y}_n \in \mathbb{R}^d$ of the $d \times n$ matrix $\boldsymbol{Y}^T$ whose interpoint distances $d_{ij}^2 = ||\boldsymbol{Y}_i - \boldsymbol{Y}_j||$ are equal to the distances δ_{ij} in the matrix Δ.

The most popular way of assessing dimensionality of the classical-scaling configuration is to plot the ordered eigenvalues (from largest to smallest) of $\boldsymbol{B}$ against order number (dimension), and then identify a dimension at which the eigenvalues stabilize. Eigenvalues become stable when they cease to change perceptively. At the dimension they become roughly constant, there will be an "elbow" in the plot where stability occurs. If $\boldsymbol{X}_i \in \mathbb{R}^d$, whit $i = 1, ..., n$, then we should see stability at dimension $d + 1$. Typically, one hopes that d is small, of the order of 2 or 3.

2.3 Nonlinear methods

The goal of these algorithms is to recover the full low-dimensional representation of an unknown nonlinear manifold $\mathcal{M}$, embedded in some high-dimensional space, where it is important to retain the neighborhood structure of $\mathcal{M}$. When $\mathcal{M}$ is highly nonlinear, these algorithms outperform the usual linear techniques. The nonlinear manifold-learning methods emphasize simplicity and avoid optimization problems that could produce local minima.

Assume that we have a finite random sample of data points, $\boldsymbol{y}_1, ..., \boldsymbol{y}_n$, from a smooth d-dimensional manifold $\mathcal{M}$ with metric given by the geodesic distance $d^{\mathcal{M}}$. These points are then nonlinearly embedded by a smooth map ψ into high-dimensional input space $\mathcal{X} = \mathbb{R}^{d_X}$ with Euclidean metric. This embedding provides us with the input data $\boldsymbol{x}_1, ..., \boldsymbol{x}_n$. Thus, $\phi : \mathcal{M} \longrightarrow \mathcal{X}$ is the embedding map, and a point on the manifold $\boldsymbol{y} \in \mathcal{M}$ can be expresed as $\boldsymbol{y} = \phi(\boldsymbol{x})$, where $\boldsymbol{x} \in \mathcal{X}$ and $\phi = \psi^{-1}$. The goal is then to recover $\mathcal{M}$ and find an implicit representation of the map ψ (and hence recover the points $\boldsymbol{y}_1, ..., \boldsymbol{y}_n$) given only the input data points $\boldsymbol{x}_1, ..., \boldsymbol{x}_n$ im $\mathcal{X}$.

Most of the nonlinear manifold learning algorithms that we discuss here are based upon different philosophies regarding how one should recover unknown nonlinear manifolds. However, they each consist of a three-step approach:

- The first is common to all algorithms, it incorporates neighborhood information at each data point to construct a weighted graph having the data points as vertices.
- The second step is specific to the algorithm, taking the weighted neighborhood graph and transforming it into suitable input for the spectral embedding step.
- The third is common to all algorithms, it is a spectral embedding step that involves an $n \times n$ eigenequation computation.

2.3.1 Isomap

The Isometric Feature Mapping (Isomap) assumes that the smooth manifold $\mathcal{M}$ is a convex region of $\mathbb{R}^d$ and that the embedding $\psi : \mathcal{M} \longrightarrow \mathcal{X}$ is an isometry. This means that the geodesic distance is invariant under the map ψ, for any pair of points on the manifold $\boldsymbol{y}, \boldsymbol{y}' \in \mathcal{M}$ the geodesic distance between those points equals the Euclidean distance between theis corresponding coordinates $\boldsymbol{x}, \boldsymbol{x}' \in \mathcal{X}$.

Isomap uses the isometry and convexity assumptions to form a nonlinear generalization of MDS by attempting to preserve the global geometric properties of the underlying nonlinear manifold, and it does this by approximating all pairwise geodesic distances on the manifold. In this sense, Isomap provides a global approach to manifold learning.

The Isomap algorithm consists of three steps:

- Nearest-neighbor search: select either an integer K or an $\epsilon > 0$, calculate the distances between all pairs of data points in $\mathcal{X}$

  $$d_{ij}^{\mathcal{X}} = d^{\mathcal{X}}(\boldsymbol{x}_i, \boldsymbol{x}_j) = ||\boldsymbol{x}_i - \boldsymbol{x}_j||_{\mathcal{X}}$$

  , and determine which data points are neighbors on the manifold $\mathcal{M}$ by connecting each point either to its K nearest neighbors or to all points lying withi a ball of radius ϵ of that point. Choice of K or ϵ controls neighborhood size and also the succes of Isomap.
- Compute graph distances: this gives us a weighted neighborhood graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where the set of vertices $\mathcal{V} = \{\boldsymbol{x}_1, ..., \boldsymbol{x}_n\}$ are the input data points, and the set of edges $\mathcal{E} = \{e_{ij}\}$ indicate neighborhood relationships between the points. The edge e_{ij} that joins the neighboring points $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ has a weight w_{ij} associated with it, and that weight is given by the distance $d_{ij}^{\mathcal{X}}$ between those points. If there is no edge present between a pair of points, the corresponding weight is zero. Estimate the unknown true geodesic distances, $\{d_{ij}^{\mathcal{M}}\}$, between pairs of points in $\mathcal{M}$ by graph distances $\{d_{ij}^{\mathcal{G}}\}$, with respect to the graph $\mathcal{G}$. The graph distances are the shortest path distances between all pairs of points in the graph $\mathcal{G}$. Points that are not neighbors of each other are connected by a sequence of neighbor-to-neighbor links, and the length of this path (sum of the link weights) is taken to approximate the distance between its endpoints on the manifold.

- Spectral embedding via multidimensional scaling: let $\boldsymbol{D}^{\mathcal{G}} = (d_{ij}^{\mathcal{G}})$ be the symmetric $n \times n$ matrix of graph distances. Apply the classical scaling algorithm of MDS to $\boldsymbol{D}^{\mathcal{G}}$ to give the reconstructed data points in a d-dimensional feature space $\mathcal{Y}$, so that the geodesic distances on $\mathcal{M}$ between data points are preserved as much as possible.

2.3.2 Local Linear Embedding

The Local Linear Embedding (LLE) algorithm for nonlinear dimensionality reduction is similar in spirit to the Isomap algorithm, but because it attempts to preserve local neighborhood information on the manifold (without estimating the true geodesic distances), we view LLE as a local approach rather than the global approach exemplified by Isomap. Like Isomap, the LLE algorithm also consists of three steps:

- Nearest-neighbor search: fix K and let N_i^K denote the neighborhood of $\boldsymbol{x}_i$ that contains only its K nearest points, as measured by Euclidean distance (K could be different for each point §$_i$). The success of LLE depends (as does Isomap) upon the choice of K, it must be sufficiently large so that the points can be well-reconstructed but also sufficient small for the manifold to have little curvature.
- Constrained least-squares fits: using the notion that every manifold is locally linear, we reconstruct $\boldsymbol{x}_i$ by a linear function of its K nearest neighbors

$$\hat{\boldsymbol{x}}_i = \sum_{j=1}^{n} w_{ij} \boldsymbol{x}_j$$

, where w_{ij} is a scalar weight for $\boldsymbol{x}_j$ with unit sum $\sum_{j=1}^{n} w_{ij} = 1$ for translation invariance. If $\boldsymbol{x}_l \notin N_i^K$, then set $w_{il} = 0$. Translation invariance implies that adding some contat $\boldsymbol{c}$ to each of the input vectors $\boldsymbol{x}_i$ does not change the minimizing quantity. Set $\boldsymbol{W} = (w_{ij})$ to be a sparse $n \times n$ of weights, the optimal weights $\hat{w}_{ij}$ are obtained by solving

$$\begin{aligned}
\hat{\boldsymbol{W}} &= \text{argmin}_{\boldsymbol{W}} \left[\sum_{i=1}^{n} ||\boldsymbol{x}_i - \sum_{j=1}^{n} w_{ij} \boldsymbol{x}_j||^2 \right] \\
\implies \hat{\boldsymbol{W}} &= \text{argmin}_{\boldsymbol{W}} \left[\sum_{i=1}^{n} || \sum_{j=1}^{n} w_{ij} (\boldsymbol{x}_i - \boldsymbol{x}_j)||^2 \right] \\
\implies \hat{\boldsymbol{W}} &= \text{argmin}_{\boldsymbol{W}} \left[\sum_{i=1}^{n} \boldsymbol{w}_i^T \boldsymbol{G}^{(i)} \boldsymbol{w}_i \right]
\end{aligned}$$

where $\boldsymbol{w}_i = (w_{i1}, ..., w_{in})$ only K of which are nonzero, and $\boldsymbol{G}^{(i)} = (G_{jk}^{(i)})$, where

$$G_{jk}^{(i)} = (\boldsymbol{x}_i - \boldsymbol{x}_j)^T (\boldsymbol{x}_i - \boldsymbol{x}_k), \qquad j, k \in N_i^K$$

is an $n \times n$ Gramm matrix. Using Lagrange multipliers to solve the constrained optimization problem we get that the optimal weights are given by

$$\hat{\boldsymbol{w}}_i = \frac{\boldsymbol{G}^{(i)-1} \mathbf{1}}{\mathbf{1}^T \boldsymbol{G}^{(i)-1} \mathbf{1}}$$

, where $\mathbf{1}$ is a vector of all ones. The resulting optimal weights for each data point, and all other zero-weights, are collected into a sparse $n \times n$ matrix $\hat{\boldsymbol{W}} = (\hat{w}_{ij})$ having only nK nonzero elements.

- Spectral embedding: consider the optimal weight matrix $\hat{\boldsymbol{W}}$ to be fixed. Now we find the $d \times n$ matrix $\boldsymbol{Y} = (\boldsymbol{y}_1, ..., \boldsymbol{y}_n)$ of embedding coordinates that solves

$$\hat{\boldsymbol{Y}} = \text{argmin}_{\boldsymbol{Y}} \left[\sum_{i=1}^{n} ||\boldsymbol{y}_i - \sum_{j=1}^{n} w_{ij} \boldsymbol{y}_j||^2 \right]$$

subject to the constraints that the mean vector is zero $\sum_{i=1}^{n} \boldsymbol{y}_i = \boldsymbol{Y}\mathbf{1} = 0$, and that the covariance matrix is the identity $\frac{1}{n} \sum_{i=1}^{n} \boldsymbol{y}_i \boldsymbol{y}_i^T = \frac{1}{n} \boldsymbol{Y}\boldsymbol{Y}^T = \boldsymbol{I}$. These constraints determine the translation, rotation and scale of the embedding coordinates, that helps to ensure that the objective function will be invariant. The matrix of embedding coordinates can be written as

$$\hat{\boldsymbol{Y}} = (\hat{y}_1, ..., \hat{y}_d)^T = (\boldsymbol{v}_{n-1}, ..., \boldsymbol{v}_{n-d})^T$$

, where $\boldsymbol{v}_{n-j}$ is the eigenvector corresponding to the $j+1$ smallest eigenvalue of $\boldsymbol{M}$. The sparseness of $\boldsymbol{M}$ enables eigencomputations to be carried out very efficiently.

Chapter 3

Active Learning

3.1 What is Active Learning?

The key hypothesis is that if the learning algorithm is allowed to choose the data from which it learns. Active learning systems attempt to overcome the labeling bottleneck by asking queries in the form of unlabeled instances to be labeled by an oracle (that is, a human annotator). In this way, the active learner aims to achieve high accuracy using as few labeled instances as possible, thereby minimizing the cost of obtaining labeled data.

3.2 Scenarios

There are several different problem scenarios in which the learner may be able to ask queries:

- Membership Query Synthesis: In this setting, the learner may request labels for any unlabeled instance in the input space, including (and typically assuming) queries that the learner generates itself, rather than those sampled from some underlying natural distribution. Query synthesis is reasonable for many problems, but labeling such arbitrary instances can be awkward if the oracle is a human annotator, since it can query unrecognizable patterns.
- Stream-Based Selective Sampling: The key assumption is that obtaining an unlabeled instance is free (or inexpensive), so it can first be sampled from the actual distribution, and then the learner can decide whether or not to request its label. The decision whether or not to query an instance can be framed several ways:
  - One approach is to evaluate samples using some "informativeness measure" or "query strategy" and make a biased random decision, such that more informative instances are more likely to be queried.
  - Another approach is to compute an explicit region of uncertainty, that is, the part of the instance space that is still ambiguous to the learner, and only query instances that fall within it.A naïve way of doing this is to set a minimum threshold on an informativeness measure which define the region, so instances whose evaluation is above this threshold are then queried. Another more principled approach is to define the region that is

still unknown to the overall model class, the set of hypotheses consistent with the current labeled training set called the version space.

- Pool-Based Sampling: it assumes that there is a small set of labeled data $\mathcal{L}$ and a large pool of unlabeled data $\mathcal{U}$ available. Queries are selectively drawn from the pool, which is usually assumed to be closed (i.e., static or non-changing), although this is not strictly necessary. Typically, instances are queried in a greedy fashion, according to an informativeness measure used to evaluate all instances in the pool (or, perhaps if is very large, some subsample thereof). The main difference between stream-based and pool-based active learning is that the former scans through the data sequentially and makes query decisions individually, whereas the latter evaluates and ranks the entire collection before selecting the best query.

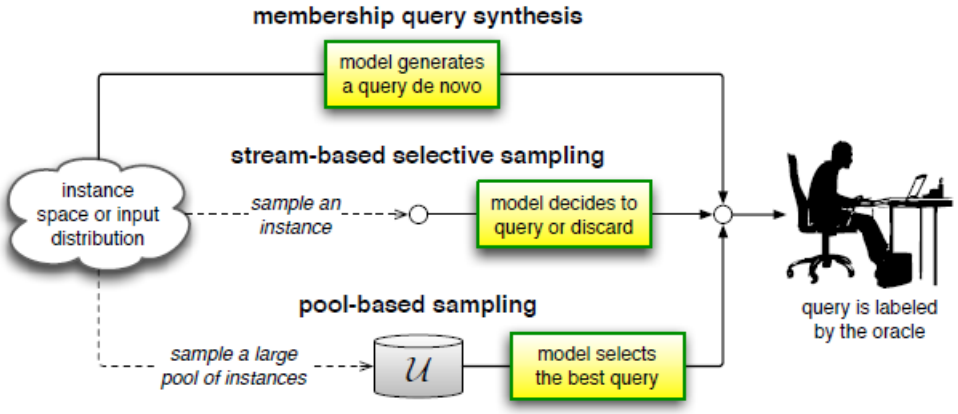

FIGURE 3.1: Diagram illustrating the three main active learning scenarios.

3.3 Query Strategy Frameworks

All active learning scenarios involve evaluating the informativeness of unlabeled instances, which can either be generated de novo or sampled from a given distribution. There have been many proposed ways of formulating such query strategies in the literature.

3.3.1 Uncertainty Sampling

In this framework, an active learner queries the instances about which it is least certain how to label. The general uncertainty sampling variant might query the instance whose prediction is the least confident,

$$x^*_{LC} = \text{argmax}_x \left[1 - P_\theta(\hat{y}|x)\right]$$

, where $\hat{y} = \text{argmax}_y P_\theta(y|x)$ is the class label with the highest posterior probability under the model θ.

However, the criterion for the least confident strategy only considers information about the most probable label. Thus, it effectively "throws away" information about the remaining label distribution. To correct for this, there is a multi-class uncertainty sampling variant called margin sampling,

$$x^*_{MS} = \text{argmin}_x \left[P_\theta(\hat{y}_1|x) - P_\theta(\hat{y}_2|x) \right]$$

, where $\hat{y}_1$ and $\hat{y}_2$ are the first and second more probable class labels under the model respectively. Margin sampling aims to correct for a shortcoming in least confident strategy, by incorporating the posterior of the second most likely label. Instances with large margins are easy, since the classifier has little doubt in differentiating between the two most likely class labels. Instances with small margins are more ambiguous, thus knowing the true label would help the model discriminate more effectively between them.

A more general uncertainty sampling strategy (and possibly the most popular) uses entropy as an uncertainty measure,

$$x^*_H = \text{argmax}_x \left[-\sum_l P_\theta(y_l|x) \log P_\theta(y_l|x) \right]$$

, where y_l ranges over all possible labels. Entropy is an information-theoretic that represents the amount of information needed to "encode" a distribution. As such, it is often thought of as a measure of uncertainty or impurity in machine learning.

Intuitively, entropy seems appropriate if the objective function is to minimize log-loss, while the other two (particularly margin) are more appropriate if we aim to reduce classification error, since they prefer instances that would help the model better discriminate among specific classes.

3.3.2 Query-By-Committee

The QBC approach involves maintaining a committee $\mathcal{C} = \{\theta^{(1)}, ..., \theta^{(C)}\}$ of models which are all trained on the current labeled set $\mathcal{L}$, but represent competing hypotheses. Each committee member is then allowed to vote on the labelings of query candidates. The most informative query is considered to be the instance about which they most disagree. The fundamental premise behind the QBC framework is minimizing the version space, which is the set of hypotheses that are consistent with the current labeled training data $\mathcal{L}$. If we view machine learning as a search for the best model within the version space, then our goal in active learning is to constrain the size of this space as much as possible with as few labeled instances as possible.

In order to implement a QBC selection algorithm, one must be able to construct a committee of models that represent different regions of the version space and have some measure of disagreement among committee members. It is highly encouraged to have diversity among committee members.

For measuring the level of disagreement, two main approaches have been pro posed:

- Vote entropy

$$x^*_{VE} = \text{argmax} \left[-\sum_l \frac{V(y_l)}{C} \log \frac{V(y_l)}{C} \right]$$

, where again y_l ranges over all possible labelings, and $V(y_l)$ is the number of "votes" that a label receives from among the committee members' predictions, and C is the committee size.

- Kullback-Leibler (KL) divergence

$$x^*_{VE} = \text{argmax} \left[-\frac{1}{C} \sum_{c=1}^{C} D(P_{\theta^{(c)}} || P_C) \right]$$

where

$$D(P_{\theta^{(c)}} || P_C) = \sum_l P_{\theta^{(c)}}(y_l|x) \log \frac{P_{\theta^{(c)}}(y_l|x)}{P_C(y_l|x)}$$

. Here $\theta^{(c)}$ represents a particular model in the committee, and C represents the committee as a whole, thus $P_C(y_l|x) = \frac{1}{C} P_{\theta^{(c)}}(y_l|x)$ is the "consensus" probability that y_l is the correct label. KL divergence is an information-theoretic measure of the difference between two probability distributions. So this disagreement measure considers the most informative query to be the one with the largest average difference between the label distributions of any one committee member and the consensus.

3.3.3 Expected Model Change

Another general active learning framework uses a decision-theoretic approach, selecting the instance that would impart the greatest change to the current model if we knew its label. An example query strategy in this framework is the "expected gradient length" (EGL), which can be applied to any learning problem where gradient-based training is used. Since discriminative probabilistic models are usually trained using gradient-based optimization, the "change" imparted to the model can be measured by the length of the training gradient.

The learner should query the instance x which, if labeled and added to $\mathcal{L}$, would result in the new training gradient of the largest magnitude. Let $\nabla l_\theta(\mathcal{L})$ be the gradient of the objective function l with respect to the model parameters θ. Now let $\nabla l_\theta(\mathcal{L} \cup (x, y))$ be the new gradient that would be obtained by adding the training tuple (x, y) to $\mathcal{L}$. Since the query algorithm does not know the true label in advance, we must instead calculate the length as an expectation over the possible labelings

$$x^*_{EGL} = \text{argmax}_x \left[\sum_l P_C(y_l|x) || \nabla l_\theta(\mathcal{L} \cup (x, y_l)) || \right]$$

where $||.||$ is, in this case, the Euclidean norm of each resulting gradient vector. Note that, at query time, $||\nabla l_\theta(\mathcal{L})||$ should be nearly zero since l converged at the previous round of training. Thus, we can approximate $||\nabla l_\theta(\mathcal{L} \cup (x, y_l))|| \approx ||\nabla l_\theta((x, y_l))||$ for computational efficiency, because training instances are usually assumed to be independent.

The intuition behind this framework is that it prefers instances that are likely to most influence the model, that is, to have greatest impact on its parameters, regardless of the resulting query label.

3.3.4 Expected Error Reduction

Another decision-theoretic approach aims to measure not how much the model is likely to change, but how much its generalization error is likely to be reduced. The idea it to estimate the expected future error of a model trained using $\mathcal{L} \cup < x, y >$ on the remaining unlabeled instances in $\mathcal{U}$ (which is assumed to be representative of the test distribution, and used as a sort of validation set), and query the instance with minimal expected future error (sometimes called risk). One approach is to minimize the expected 0/1-loss

$$x^*_{0/1} = \text{argmin}_x \left[\sum_i P_\theta(y_i|x) \left(1 - P_{\theta^{+<x,y_i>}}(\hat{y}|x^u)\right) \right]$$

, where $\theta^{+<x,y_i>}$ refers to the the new model after it has been re-trained with the training tuple $< x, y_i >$ added to $\mathcal{L}$.

As with EGL in the previous section, we do not know the true label for each query instance, so we approximate using expectation over all possible labels under the current model θ. The objective here is to reduce the expected total number of incorrect predictions.

A less stringent objective is to minimize the expected log-loss

$$x^*_{0/1} = \text{argmin}_x \left[\sum_i P_\theta(y_i|x) \left(- \sum_{u=1}^{U} \sum_j P_{\theta^{+<x,y_i>}}(y_j|x^u) \log \left(P_{\theta^{+<x,y_i>}}(y_j|x^u) \right) \right) \right]$$

, which is equivalent to reducing the expected entropy over $\mathcal{U}$. Another interpretation of this strategy is maximizing the expected information gain of the query x or,equivalently, the mutual information of the output variables over x and $\mathcal{U}$.

3.3.5 Variance Reduction

3.3.6 Density-Weighted Methods

Chapter 4

Deep Learning

4.1 What is Deep Learning?

The name Deep Learning can mean different things for different people. For many researchers, it means that the neural network has more than 2 layers. For many other researchers, it is also associated with the fact that the model makes use of unlabeled data. For many people in the industry, it means that there is no need for human-invented features. But in general, it means a set of algorithms that use neural networks as an architecture, and learn the features automatically.

When the problem does exhibit nonlinear properties, deep networks seem computationally more attractive than shallow networks, it has been observed empirically that in order to get to the same level of performances of a deep network, one has to use a shallow network with many more connections. An intuition of why this is the case is that a deep network can be thought of as a program in which the functions computed by the lower-layered neurons can be thought of as subroutines that are re-used many times in the computation of the final program. Therefore, using a shallow network is similar to writing a program without the ability of calling subroutines. Without this ability, at any place we could otherwise call the subroutine, we need to explicitly write the code for the subroutine. In terms of the number of lines of code, the program for a shallow network is therefore longer than a deep network.

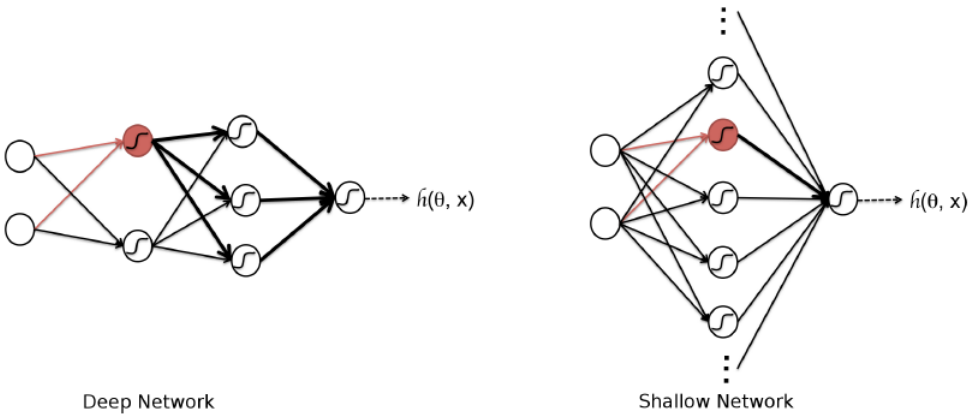

FIGURE 4.1: Diagram of a deep network and a shallow network. In the deep network the function computed by the red neuron in the first layer is re-used three times. In contrast, in the shallow network, the function computed by the red neuron is only used once.

4.2 Autoencoders

Suppose we have a set of data points $\{x^{(1)}, x^{(2)}, \ldots, x^{(m)}\}$ where each data point has many dimensions. The question becomes whether there is a general way to map them to another set of data points $\{z^{(1)}, z^{(2)}, \ldots, z^{(m)}\}$, where z's have lower dimensionality than x's and z's can faithfully reconstruct x's. To map data back and forth more systematically, we can propose that z and $\tilde{x}$ are functions of their inputs, in the following manner:

$$z^{(i)} = W_1 x^{(i)} + b_1$$
$$\tilde{x}^{(i)} = W_2 z^{(i)} + b_2$$

Then, as our goal is to have $\tilde{x}$ to approximate x , we can set up the following objective function,

$$J(W_1, b_1, W_2, b_2) = \sum_{i=1}^{m} \left(\tilde{x}^{(i)} - x^{(i)}\right)^2 = \sum_{i=1}^{m} \left(W_2(W_1 x^{(i)} + b_1) + b_2 - x^{(i)}\right)^2$$

which can be minimized using stochastic gradient descent.

This particular architecture is also known as a linear autoencoder, which works for the case that the data lie on a linear surface. If the data lie on a nonlinear surface, it makes more sense to use a nonlinear autoencoder. If the data is highly nonlinear, one could add more hidden layers to the network to have a deep autoencoder.

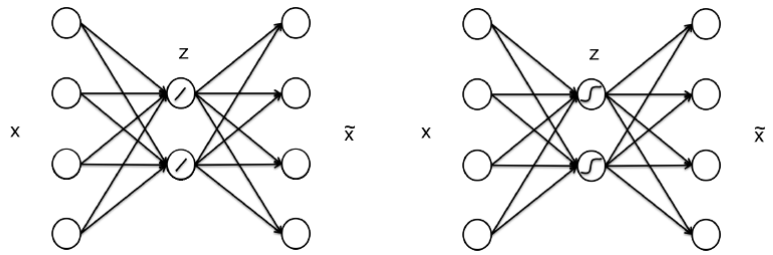

FIGURE 4.2: Diagram of a linear autoencoder and a nonlinear autoencoder. We are trying to map data from 4 dimensions to 2 dimensions using a neural network with one hidden layer.

Autoencoders have many interesting applications, such as data compression or visualization, but they are mainly used as a way to pretrain neural networks. The reason is that training very deep neural networks is difficult, the magnitudes of gradients in the lower layers and in higher layers are different, the landscape of curvature of the objective function is difficult for stochastic gradient descent to find a good local optimum, and there are too many parameters, which can make the network remember training data and do not generalize well.

The goal of pretraining is to address these problems. With pretraining, the process of training a deep network is divided in a sequence of steps:

- Pretraining step: train a sequence of shallow autoencoders, greedily one layer at a time, using unsupervised data,

- Fine-tuning step 1: train the last layer using supervised data,
- Fine-tuning step 2: use backpropagation to fine-tune the entire network using supervised data.

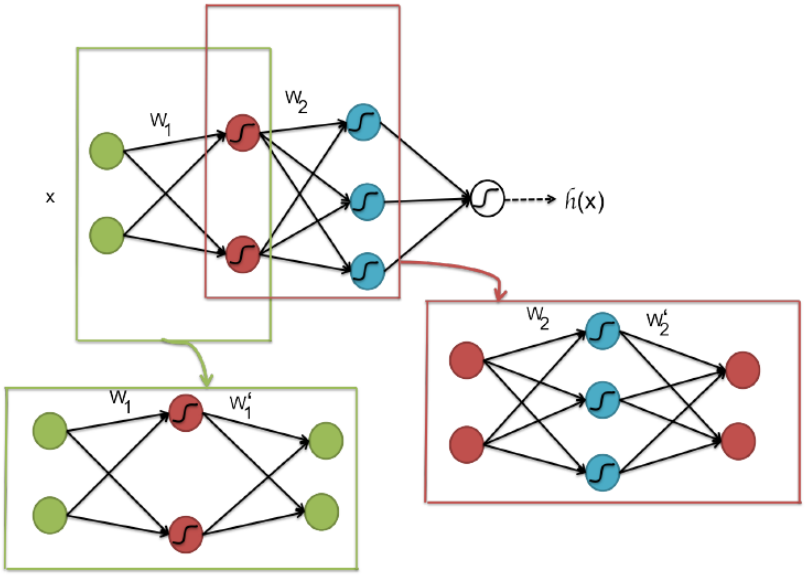

FIGURE 4.3: Example of pretraining of a deep neuronal network. To train the red neurons, we will train an autoencoder that has parameters W_1 and W_1'. After this, we will use W_1 to compute the values for the red neurons for all of our data, which will then be used as input data to the subsequent autoencoder. The parameters of the decoding process W_1' will be discarded. The next autoencoder uses the values for the red neurons as inputs, and trains an autoencoder to predict those values by adding a decoding layer with parameters W_2'.

4.3 Convolutional Neural Networks

4.3.1 Locally connected networks

In all networks that we have seen so far, every neuron in the first hidden layer connects to all the neurons in the inputs. This does not work when input data is high-dimensional, because every neuron ends up with many connections. For example, when input is a small image of 100x100 pixels, every neuron has 10,000 parameters. To make this more efficient, we can force each neuron to have a small number of connections to the input. The connection patterns can be designed to fit some structure in the inputs. For example, in the case of images, the connection patterns are that neurons can only look at adjacent pixels in the input image.

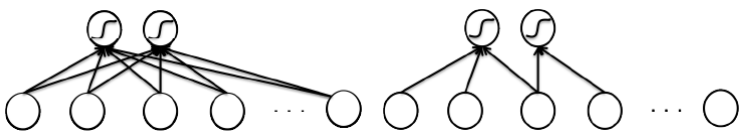

FIGURE 4.4: Diagram of totally connected neurons and locally connected neurons.

We can extend this idea to force local connectivity in many layers, to obtain a deep locally connected network. Training with gradient descent is possible because we can modify the backpropagation algorithm to deal with local connectivity: in the forward pass, we can compute the values of neurons by assuming that the empty connections have weights of zeros, wheareas in the backward pass, we do not need to compute the gradients for the empty connections.

4.3.2 Translational invariance

We have seen that using locality structures significantly reduces the number of connections. But even further reduction can be achieved via another technique called weight sharing. In weight sharing, some of the parameters in the model are constrained to be equal to each other. This idea of sharing the weights resembles an important operation in signal processing known as convolution. In convolution, we can apply a filter, a set of weights, to many positions in the input signals.

In practice, this type of networks also comes with another layer known as the max-pooling layer. The max-pooling layer computes the max value of a selected set of output neurons from the convolutional layer and uses these as inputs to higher layers. An interesting property of this approach is that the output of the max-pooling neurons are invariant to shifts in the inputs, that is, the outputs of the system are invariant to translation. This is also known as translational invariance.

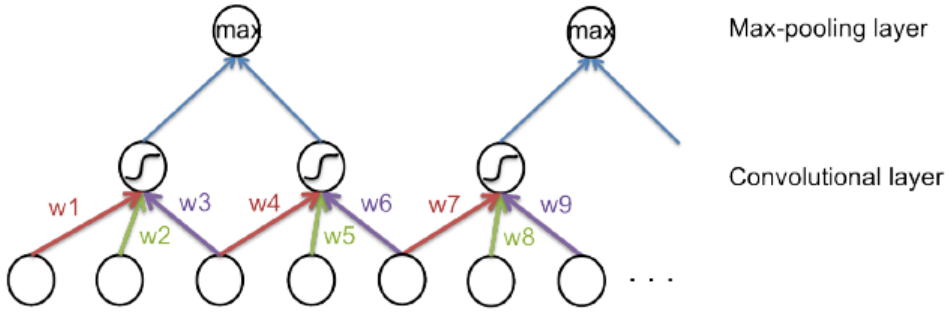

FIGURE 4.5: Example of locally connected neurons with weight sharing and a max-pooling layer. We have the following constraints $w_1 = w_4 = w_7$, $w_2 = w_5 = w_8$ and $w_3 = w_6 = w_9$, that is, the edges that have the same color have the same weight. With these constraints, the model can be quite compact in terms of number of actual parameters. Instead of storing all the weights from w_1 to w_9, we only need to store w_1, w_2 and w_3. Values for other connections can then be derived from these three values.

Translational invariant systems are typically effective for natural data (such as images, sounds ...) because a major source of distortions in natural data is typically translation. This type of networks are also known as Convolutional Neural Networks (CNN). The max pooling layer in the convolutional neural networks is also known as the subsampling layer because it dramatically reduces the size of the input data. The max operation can sometimes be replaced by the average operation. We can also construct a deep convolutional neural networks by treating the outputs of a max-pooling layer as a new input vector, and adding a new convolutional layer and a new max-pooling layer on top of this vector.

Many recent convolutional neural networks also have another type of layers, called Local Contrast Normalization (LCN). This layer operates on the outputs of the max-pooling layer. Its goal is to subtract the mean and divide the standard deviation of the incoming neurons (in the same manner with max-pooling). This operation allows brightness invariance, which is useful for image recognition.

Finally, it is possible to modify the backpropagation algorithm to work with these layers:

- In the forward pass, perform explicit computation with all the weights, w_1, ..., w_9.
- In the backward pass, compute the gradient for all the weights, $\frac{\partial J}{\partial w_1}$, ..., $\frac{\partial J}{\partial w_9}$.
- When updating the weights, use the average of the gradients from the shared weights, $w_1 = w_1 - \alpha \left(\frac{\partial J}{\partial w_1} + \frac{\partial J}{\partial w_4} + \frac{\partial J}{\partial w_7} \right)$, ..., $w_9 = w_9 - \alpha \left(\frac{\partial J}{\partial w_3} + \frac{\partial J}{\partial w_6} + \frac{\partial J}{\partial w_9} \right)$.
- For the max-pooling layer, in the forward pass, we need to remember what branch gives the max value, so that in the backward pass, we only compute the gradient for that branch.

4.3.3 Further modifications

Images typically have multiple channels, so it is possible to modify the convolutional architecture above to work with multiple channel inputs. The modification is essentially to have a filter that looks at multiple channels. The weights are often not shared across channel.

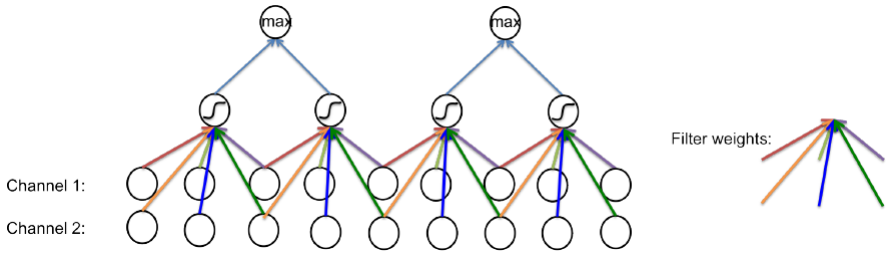

FIGURE 4.6: Example of a CNN with two input channels.

Finally, the current convolutional architecture has only one filter per position of the input. We can extend that to have many filters per location. For instance, at one location we can have two filters looking at exactly the same input. Each set of output produced by each filter is called a map.

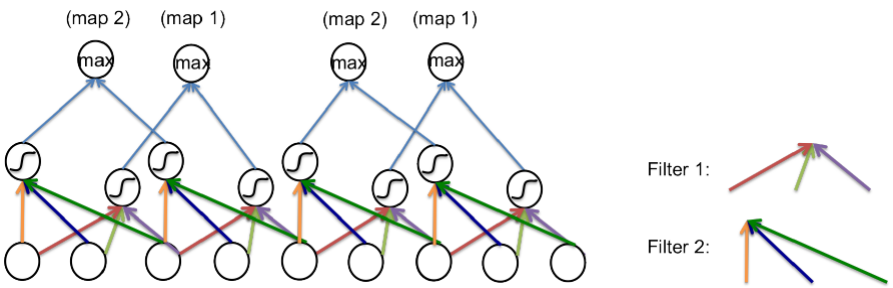

FIGURE 4.7: Example of a CNN with two maps.

4.4 Recurrent Neuronal Networks

Translation invariance is acceptable for images because the output of an object recognition system should be invariant to translation. In contrast, in a time series prediction model, this is an undesirable property, because we want to make use of the precise temporal information. A proper way to deal with these inputs is to use a Recurrent Neural Network (RNN).

Let $x_0, x_1, ..., x_T$ be the temporal values of a time series, and let $h_0, h_1, ..., h_T$ be the hidden states of the recurrent network. For a recurrent neural network, there are typically three sets of parameters: the input to hidden weights W, the hidden to hidden weights U, and the hidden to label weight V. Notice that all the W's are shared, all the U's are shared and all the V 's are shared. The weight sharing property makes our network suitable for variable-sized inputs. Even if T grows, the size of our parameters stay the same: it's only W, U and V.

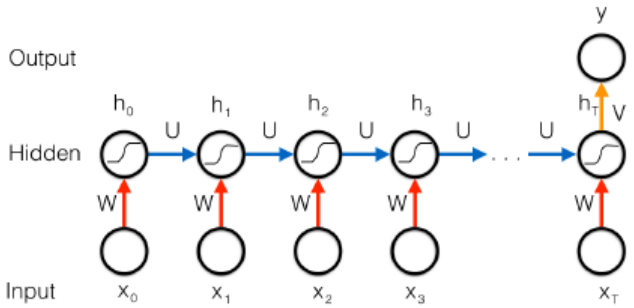

FIGURE 4.8: Example of a RNN, note that each bubble indicate a layer.

With these notations, the hidden states are recursively computed as:

$$
\begin{aligned}
&h_0 = \sigma(Wx_0) \\
&... \\
&h_t = \sigma(Uh_{t-1} + Wx_t) \\
&... \\
&f(x) = Vh_T
\end{aligned}
$$

then we can define a cost function $J = (y - f(x))^2$ and minimize it to obtain the appropriate weights. To compute the gradient of the recurrent neural network, we can use backpropagation again, but with a modification, backpropagation through time (BPTT).

Since there are tied-weights in the network, we can apply the same idea in the convolutional neural network. In the forward pass, pretend that the weights are not shared (so that we have weights $W_0, W_1, ..., W_T$ and $U_0, U_1, ..., U_T$). In the backward pass, we compute the gradient with respect to all W's and all U's. The final gradient of W is the sum of all gradients for $W_0, W_1, ..., W_T$, and the final gradient of U is the sum of all gradients for $U_0, U_1, ..., U_T$.

In practice, when computing the gradient for the recurrent neural network, one can find that the gradient is either very large or very small. This can cause the optimizer to converge slowly. To speed up training, it is important to clip the gradient at certain values. For example, any dimension of the gradient should be smaller than 1, if the value of a dimension is larger than 1 we should set it to be 1.

4.4.1 Long Short Term Memory Networks

Backpropagation through time (BPTT) for RNNs is usually difficult due to a problem known as vanishing/exploding gradient, the magnitude of the gradient becomes extremely small or extremely large towards the first or the last time steps in the network. This problem makes the training of RNNs challenging, especially when there are a lot of long term dependencies (the output prediction is influenced by long-distance pieces of information in the inputs). In terms of optimization, this means that the gradient magnitude in some certain dimension is extremely small, and in some other dimension is extremely large.

The reason for this vanishing/exploding gradient problem is due to the use of sigmoidal activation functions in recurrent networks. As the error derivatives are back-propagated backwards, it has to be multiplied by the derivative of the sigmoid (or tanh) function which can saturate quickly. One can imagine that the ReLU activation function can help here as its derivative allows for better gradient flow. However, another problem arises as we use ReLU as the activation function, in the forward prop computation, if we are not careful with the initialization of the U matrix, the hidden values can explode or vanish depending on whether the eigenvalues of U are bigger or smaller than 1.

Perhaps the most successful attempt to improve the learning of recurrent networks to date is Long Short Term Memory (LSTM) Recurrent Networks. The idea behind LSTM is to modify the architecture of recurrent networks to allow the error derivatives to flow better. At the heart of an LSTM is the concept of memory cells which act as an integrator over time. Let x_t be the input data at time t and h_t be the hidden state at the previous timestep, then the memory cells at time t have values

$$m_t = \alpha \odot m_{t-1} + \beta \odot f(x_t, h_{t-1})$$

, where $\odot$ is an element-wise multiplication between two vectors. Just think of m_t as a linearly weighted combination between m_{t-1} and f. A nice property of m_t is that it is computed additively and not associated with any nonlinearity. So if the error derivatives cannot pass through the function f at time step t (to subsequently flow through h_{t-1}), it has an opportunity to propagated backward further through m_{t-1}. In other words, it allows another path for the error derivatives to flow. To prevent m from exploding, f is often associated with a sigmoid/tanh function.

Built on this construct, the hidden state of the network at time t can be computed as

$$h_t = \tanh m_t$$

, where note again that both, h_t and m_t, will be used as inputs to the next time steps, and that means the gradient has more opportunities to flow through different paths.

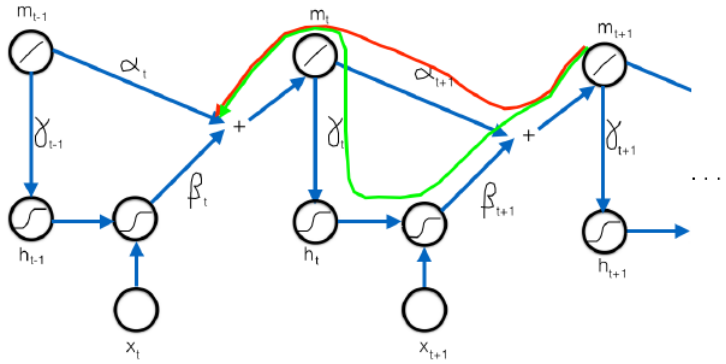

FIGURE 4.9: Example of the construction of a LSTM CNN. The green and the red paths are the two paths that gradient can flow back from m_{t+1} to m_t. The green path, which generates nonlinear ouputs, is a difficult path for gradient to flow, whereas the red path, which only generates linear functions, is an easy path for gradient to flow.

The variables α, β and γ are often called gates, as they modulate the contribution of input, memory, and output to the next time step. At a time step t, they are computed and implemented as:

$$\begin{aligned}
\alpha(t) &= g_1(x_t, h_{t-1}, m_{t-1}) = \sigma(W_{x\alpha}x_t + W_{h\alpha}h_{t-1} + W_{m\alpha}m_{t-1} + b_\alpha) \\
\beta(t) &= g_2(x_t, h_{t-1}, m_{t-1}) = \sigma(W_{x\beta}x_t + W_{h\beta}h_{t-1} + W_{m\beta}m_{t-1} + b_\beta) \\
\gamma(t) &= g_3(x_t, h_{t-1}, m_{t-1}) = \sigma(W_{x\gamma}x_t + W_{h\gamma}h_{t-1} + W_{m\gamma}m_t + b_\gamma) \\
\implies f(x_t, h_{t-1}) &= \tanh(W_{xm}x_t + W_{hm}h_{t-1} + b_n)
\end{aligned}$$

4.4.2 Sequence output prediction

In the above sections, the prediction outputs have always been a scalar (classification, and regression) or a fixed-length vector (autoencoders). But this is restrictive since in many applications the desired outputs can be a variable-length sequence. For example, in the task of machine translation, we would like to predict the target sentence from the source sentence, or in the task of automatic image captioning, we would like to predict the caption given the input image.

Suppose during training, we are presented with an input vector x (a hidden layer in convolutional or recurrent networks), and a target output sequence y_1, y_2, y_3. We would like to train a RNN to go from x to y_1, y_2, y_3. We know that we can use a recurrent network to predict y_1 from x and the use it as the input of y_2, and so on. This means that during training we feed the ground-truth targets as inputs to the next step prediction. This is good because during training, we see the ground-truth targets anyway.

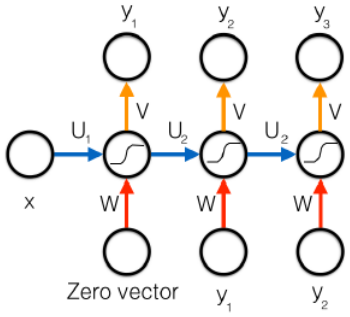

FIGURE 4.10: Example of a CNN to predict variable length sequences. Zero vector is used as input to the first step of the network to keep it simple and consistent with later steps. The first set of weights U_1 and other set of weights U_2 can be different.

The problem is that during inference, we will not see the ground-truth targets. So we can do something rather greedy, we simply take the best prediction of the previous timestep as the input to the current timestep, this is sometimes called greedy search. The problem with greedy search, as its name suggested, is that it may not be optimal. More concretely, the joint probability of the output sequence may not be highest possible. To do better than greedy search, one can do a full search, we use the recurrent networks to compute the joint probability of every possible output sequence. This can guarantee to find the best possible sequence with a large computation cost.

Greedy search and full search lie at two extremes. One is very fast but not optimal, the other is extremely slow but optimal. To be practical, maybe we can do something in between. This brings us to another trick, left-to-right beam search. In beam search, we can keep a list of k possible sequences sorted by the joint probability. We will generate output predictions, or decode, from left to right starting from the first output. During the decode procedure, any sequence that does not belong to the top-k highest joint probability will be removed from our list of candidates. Even though it's not guaranteed for the beam search to achieve the optimal sequence, in practice, the generated sequences in the top-k list are generally very good.

4.4.3 The Attention Model

As such it feels alright if the input network, that gives rise to x, takes some other fixed-length vectors as inputs. But a problem arises if the input network takes some variable-length sequences as well. The problem is that we have to push all variable-length information into a fixed-length vector x. Perhaps a better model is to allow the output recurrent network to pay attention to the certain part of the input network so that it doesn't have to rely on this bottleneck in x. The modification to our model goes as follows. At every time step in the output, we will predict an auxiliary hidden state called u. This u vector will be used to compute the dot product with the hidden states at all timesteps in the input network. These dot products are used as weights to combine all hidden states into a vector v. This vector can be then used as additional inputs to our prediction at the current time step.

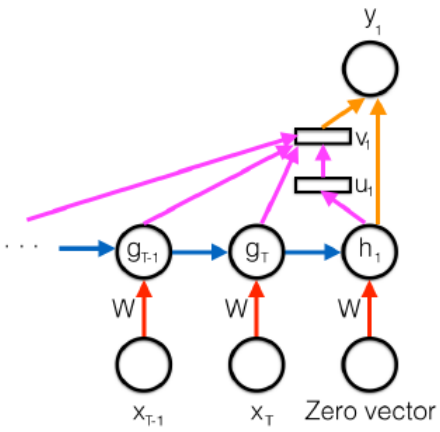

FIGURE 4.11: Example of a representation of the attention model.

More concretely, let's assume that we are at step τ in our output recurrent network. Suppose also that the input sequence is represented by another recurrent network, whose the input sequence has T steps $(x_1, ..., x_T)$. The hidden states of the output network is denoted by h and the hidden states of the input network is denoted by g. Then

$$
\begin{aligned}
u_\tau &= \Theta h_\tau \\
\beta_\tau &= u_\tau^T, t = 1, ..., T \\
\alpha_t &= \frac{\exp(\beta_t)}{\sum_{t=1}^{T} \exp(\beta_t)}, t = 1, ..., T \\
v_\tau &= \sum_{t=1}^{T} \alpha_t g_t \\
y_\tau &=
\end{aligned}
$$

where note that all variables have to be recomputed in each timestep.

4.5 Parallelism with neural networks

Training neural networks takes a long time, especially when the training set is large. It therefore makes sense to use many machines to train our neural networks. DistBelief is one of the most well-known frameworks for scaling up the training neural networks using many machines. DistBelief has two levels of parallelism:

- Model parallelism: every model in DistBelief is partitioned into multiple machines. There will be communications across machines, but if the network is locally connected, the communication cost is small compared to the computation cost.

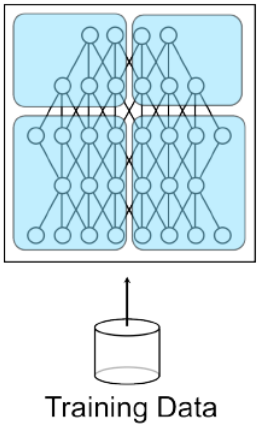

FIGURE 4.12: Representaiton of parallelism by model parallelism.

- Data parallelism: the data is partitioned into different shards, and training different copies of the model on each shard. To keep them in sync, we can have a set of machines known as the parameter server that stores the parameters. The communication to the parameter server is asynchronous, at every iteration, each copy of the model will compute the gradient on its data, and then send the gradient Δp to the parameter server, the parameter server will grab the gradient and update its p own parameter copy to arrive at p'. After several updates, the parameter server will broadcast the new parameters to the network. All replicas of the model will replace its parameters with the arriving parameters. This way of parallelizing the training is known as data parallelism via asynchronous communication. The asynchronous communication can be thought of as a soft way to average the gradient without waiting. An advantage of this approach is that it is insensitive to machine slowness: if a replica of the model is slow, it won't delay the entire training.

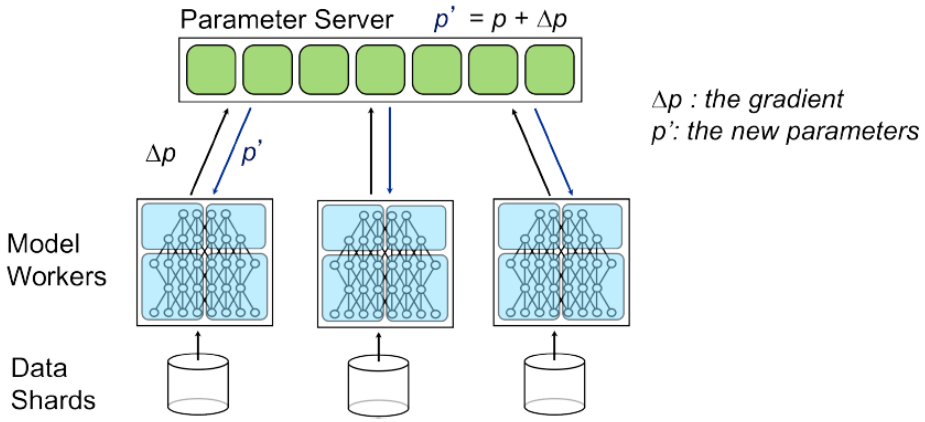

FIGURE 4.13: Representaiton of parallelism by data parallelism.

4.6 Generative Adversarial Networks

Aproximaciones generativas frente a discriminativas

- Todos los metodos que hemos estado viendo son discriminativos; la idea es, de diferentes maneras, mapear unas ciertas caracterısticas con unas etiquetas para resolver un problema
- Las aproximaciones generativas funcionan justamente al reves; la idea serıa predecir las caracterısticas a partir de las etiquetas
- En un ejemplo de detectar mail spam, una aproximacion discriminativa intentar ıa predecir si es spam usando informacion de un conjunto de variables mientras que una generativa, intentarıa averiguar partiendo de un correo spam cuales son las caracterısticas que lo describen
- Otra manera de verlo serıa pensar en terminos de clasificacion: mientras que un clasificador discriminativo buscarıa la frontera entre clases uno generativo estarıa modelando la distribucion de las clases individuales

Funcionamiento de las GAN

- La primera red (generador) produce nuevas muestras de los datos, mientras que la otra (discriminador) evalua la autenticidad de estas (decide si pertenecen al conjunto de datos de entrenamiento o no)
- El objetivo del generador no es tanto que las muestras sean de hecho del conjunto de entrenamiento sino que el discriminador se lo crea, y el objetivo del discriminador es justamente identificar generaciones que no son correctas

Ejemplo en clasificacion de imagenes

- El generador crea una imagen aleatoria
- La imagen se mete en el discriminador dentro de una secuencia de imagenes del conjunto real
- El discriminador devuelve un valor de probabilidad que representa la autenticidad de cada una de las imagenes

Doble bucle

- El discriminador esta en bucle con las imagenes reales del conjunto de datos
- El generador esta en bucle con el discriminador

Implementacion

- El discriminador es una CNN que categoriza las imagenes en reales o falsas (se deshace de los datos mediante maxpooling)
- El generador es una CNN inversa en el sentido de que a partir de un vector de ruido aleatorio genera una imagen (crea datos en lugar de deshacerlos)

Chapter 5

Association Rule Learning

5.1 What is Association Rule Learning?

5.2 Categorical Data Analysis

A categorical variable has a measurement scale consisting of a set of categories. Categorical scales are pervasive in social sciences and health sciences, but also in another disciplines such as behavioral sciences, public health, zoology, education or marketing.

5.2.1 Nominal and Ordinal scales

Categorical variables have two main types of measurement scales. Many categorical scales have a natural ordering, and are called ordinal variables, then categorical variables having unordered scales are called nominal variables.

For nominal variables, the order of listing the categories is irrelevant. The statistical analysis should not depend on that ordering. Methods designed for nominal variables give the same results no matter how the categories are listed. Methods designed for ordinal variables utilize the category ordering. Methods designed for ordinal variables cannot be used with nominal variables, since nominal variables do not have ordered categories. Methods designed for nominal variables can be used with nominal or ordinal variables, since they only require a categorical scale. When used with ordinal variables, however, they do not use the information about that ordering. This can result in serious loss of power. It is usually best to apply methods appropriate for the actual scale.

Categorical variables are often referred to as qualitative, to distinguish them from numerical-valued or quantitative, however, it is often advantageous to treat ordinal data in a quantitative manner, for instance by assigning ordered scores to the categories.

5.2.2 Discretization

5.2.3 Codification

5.3 Bivariate analysis

5.3.1 Contingency tables

For a single categorical variable, we can summarize the data by counting the number of observations in each category. The sample proportions in the categories estimate the category probabilities.

Suppose there are two categorical variables, denoted by X and Y. Let I denote the number of categories of X and J the number of categories of Y. A rectangular table having I rows for the categories of X and J columns for the categories of Y has $I \times J$ cells that display the possible combinations of outcomes. A table of this form that displays counts of outcomes in the cells is called a contingency table. A table that cross classifies two variables is called a two-way contingency table; one that cross classifies three variables is called a three-way contingency table, and so forth.

Denote the cell counts by n_{ij}, the marginal counts of the row totals by $n_{i+} = \sum_j n_{ij}$, the marginal counts of the column totals by $n_{+j} = \sum_i n_{ij}$, and the total sample size by $n = \sum_{ij} n_{ij}$. The sample cell proportion relate to the cell counts by $p_{ij} = n_{ij}/n$.

5.3.2 Mosaic plots

5.4 Association Rules

The problem of association rule mining is defined as follows: let $I = \{i_1, i_2, \ldots, i_n\}$ be a set of n binary attributes called items and let $T = \{t_1, t_2, \ldots, t_m\}$ be a set of transactions called the database. Each transaction in D has a unique transaction ID and contains a subset of the items in I.

A rule is defined as an implication of the form $X \Rightarrow Y$ where $X, Y \subseteq I$. Every rule is composed by two different sets of items, also known as itemsets, X and Y, where X is called antecedent or left-hand-side (LHS) and Y consequent or right-hand-side (RHS).

Rules typically take the form of an `{IF 'condition' THEN 'result'}` expression. An individual rule is not in itself a model, since the rule is only applicable when its condition is satisfied. Therefore rule-based machine learning methods typically identify a set of rules that collectively comprise the prediction model, or the knowledge base.

In order to select interesting rules from the set of all possible rules, constraints on various measures of significance and interest are used. Let X, Y be itemsets, $X \Rightarrow Y$ an association rule and T a set of transactions of a given database. Then we define:

- Support: is an indication of how frequently the itemset appears in the dataset. The support of X with respect to T is defined as the proportion of transactions

in the dataset which contains the itemset X.

$$supp(X) = \frac{|\{t \in T | X \in t\}|}{|T|}$$

- Confidence: Confidence is an indication of how often the rule has been found to be true. The confidence value of a rule, $X \Rightarrow Y$, with respect to a set of transactions T, is the proportion of the transactions that contains X which also contains Y.

$$conf(X \Rightarrow Y) = \frac{supp(X \cup Y)}{supp(X)}$$

- Lift: is defined as the ratio of the observed support to that expected if X and Y were independent. If the rule had a lift of 1, it would imply that the probability of occurrence of the antecedent and that of the consequent are independent of each other. When two events are independent of each other, no rule can be drawn involving those two events. If the lift is > 1, that lets us know the degree to which those two occurrences are dependent on one another, and makes those rules potentially useful for predicting the consequent in future data sets. If the lift is < 1, that lets us know the items are substitute to each other. This means that presence of one item has negative effect on presence of other item and vice versa.

$$lift(X \Rightarrow Y) = \frac{supp(X \cup Y)}{supp(X)supp(y)}$$

The best-known constraints are minimum thresholds on support and confidence. Association rules are usually required to satisfy a user-specified minimum support and a user-specified minimum confidence at the same time. Association rule generation is usually split up into two separate steps:

- A minimum support threshold is applied to find all frequent item sets in a database.
- A minimum confidence constraint is applied to these frequent itemsets in order to form rules.

wile the second step is straightforward, the first step needs more attention.

5.5 Association Rule Mining

Association rule algorithms count the frequency of complimentary occurrences, or associations, across a large collection of items or actions. The goal is to find associations that take place together far more often than you would find in a random sampling of possibilities.

5.5.1 ZeroR

5.5.2 OneR

5.5.3 A priori algorithm

Chapter 6

Probabilistic Graphical Models

6.1 What are Probabilistic Graphical Models?

6.2 Probability

6.2.1 Probability distributions

Before we discuss the representation of probability, we need to define what the events are to which we want to assign a probability. Formally, we define events by assuming that there is an agreed upon space of possible outcomes, which we denote by Ω. In addition, we assume that there is a set of measurable events $\mathcal{S}$ to which we are willing to assign probabilities. Formally, each event $\alpha \in \mathcal{S}$ is a subset of Ω.

Probability theory requires that the event space satisfy three basic properties:

- It contains the empty event $\emptyset$, and the trivial event Ω.
- It is closed under union. That is, if $\alpha, \beta \in \mathcal{S}$, then so is $\alpha \cup \beta$
- It is closed under complementation. That is, if $\alpha \in \mathcal{S}$, then so is $\Omega - \alpha$.

where note that the requirement that the event space is closed under union and complementation implies that it is also closed under other Boolean operations, such as intersection and set difference.

A probability distribution P over $(\Omega, \mathcal{S})$ is a mapping from events in $\mathcal{S}$ to real values that satisfies the following conditions:

- $P(\alpha) \geq 0$ for all $\alpha \in \mathcal{S}$.
- $P(\Omega) = 1$.
- If $\alpha, \beta \in \mathcal{S}$, and $\alpha \cap \beta \neq 0$, then $P(\alpha \cup \beta) = P(\alpha) + P(\beta)$.

where the first condition states that probabilities are not negative, the second states that the trivial event, which allows all possible outcomes, has the maximal possible probability of 1, and the third condition states that the probability that one of two mutually disjoint events will occur is the sum of the probabilities of each event. These two latest conditions imply many other conditions, of particular interest are $P(\emptyset) = 0$ and $P(\alpha \cup \beta) = P(\alpha) + P(\beta) - P(\alpha \cap \beta)$.

6.2.2 Concepts in probability

After learning that an event α is true, we can change our probability about occurring β via the notion of conditional probability. Formally, the conditional probability of β given α is defined as

$$P(\alpha|\beta) = \frac{P(\alpha \cup \beta)}{P(\alpha)}$$

, that is, the probability that β is true given that we know α is the relative proportion of outcomes satisfying β among these that satisfy α. Note then that the conditional probability is not defined when $P(\alpha) = 0$.

From the definition of the conditional distribution, we immediately see that

$$P(\alpha \cap \beta) = P(\alpha)P(\beta|\alpha)$$

, this equality is known as the chain rule of conditional probabilities. More generally, if $\alpha_1, ..., \alpha_n$ are events, then we can write

$$P(\alpha \cap ... \cap \alpha_n) = P(\alpha_1)P(\alpha_2|\alpha_1)...P(\alpha_n|\alpha_1 \cap ... \cap \alpha_{n-1})$$

, we can express the probability of a combination of several events in terms of the probability of the first, the probability of the second given the first, and so on. It is important to notice that we can expand this expression using any order of events and the result will remain the same.

Another immediate consequence of the definition of conditional probability is Bayes' rule

$$P(\alpha|\beta) = \frac{P(\alpha)P(\beta|\alpha)}{P(\beta)}$$

, which is important in that it allows us to compute the conditional probability $P(\alpha|\beta)$ from the "inverse" conditional probability $P(\beta|\alpha)$.

6.2.3 Random variables

6.2.4 Independence

6.3 Graphs

6.3.1 Nodes and edges

A graph is a data structure $\mathcal{K}$ consisting of a set of nodes $\mathcal{X} = \{X_1, ..., X_m\}$ and a set of edges $\mathcal{E}$. A pair of nodes X_i, X_j can be connected by a directed edge $X_i \longrightarrow X_j$ or by an undirected edge X_i—X_j. Thus the set of edges $\mathcal{E}$ is a set of pairs, where each pair is one of $X_i \longrightarrow X_j$, $X_j \longrightarrow X_i$ or X_i—X_j for $X_i, X_j \in \mathcal{X}$ for $i < j$. We assume that for each pair of nodes at most one type of edge exists. The notation $X_i \longrightarrow X_j$ is equivalent to $X_j \longleftarrow X_i$, and the notation X_i—X_j is equivalent to X_j—X_i. Moreover, we will use the notation $X_i \rightleftharpoons X_j$ to denote any relation, directer or undirected.

In many cases, we want to restrict attention to graphs that contain only edges of one kind or another. We say that a graph is directed if all edges are directed and undirected if all edges are undirected. We usually denote directed graphs as $\mathcal{G}$ and

undirected graphs as $\mathcal{H}$. We sometimes convert a general graph to an undirected graph by ignoring the directions on the edges. Given a graph $\mathcal{K} = (\mathcal{X}, \mathcal{E})$, its undirected version is a graph $\mathcal{H} = (\mathcal{X}, \mathcal{E}')$, where $\mathcal{E}' = \{X—Y : X \rightleftharpoons Y \in \mathcal{E}\}$.

Whenever we have that $X_i \Longrightarrow X_j \in \mathcal{E}$, we say that X_j is the child of X_i in $\mathcal{K}$ and that X_i is the parent of $X_j in \mathcal{K}$. When we have $X_i—X_j \in \mathcal{E}$ we say that X_i is a neighbor of X_j in $\mathcal{K}$ and viceversa. Finally we say that X_i and X_j are adjacent whenever $X_i \rightleftharpoons X_j \in \mathcal{E}$. We use Pa_X to denote the parents of X, Ch_X to denote its children and Nb_X to denote its neighbors. We define the boundary of X as $\text{Boundary}_X = \text{Pa}_X \cup \text{Nb}_X$.

6.3.2 Subgraphs

In many cases, we want to consider only the part of the graph that is associated with a particular subset of the nodes. Let $\mathcal{K} = (\boldsymbol{X}, \mathcal{E})$, and let $\boldsymbol{X} \subset \mathcal{X}$, we define the induced subgraph $\mathcal{K}[\boldsymbol{X}]$ to be the graph $(\boldsymbol{X}, \mathcal{E}')$ where $\mathcal{E}'$ are all the edges $\boldsymbol{X} \rightleftharpoons \boldsymbol{Y}$ such that $X, Y \in \boldsymbol{X}$.

A type of subgraph that is often of particular interest is one that contains all possible edges. A subgraph over $\boldsymbol{X}$ is complete if every two nodes in $\boldsymbol{X}$ are connected by some edge. The set $\boldsymbol{X}$ is often called a clique. We say that a clique $\boldsymbol{X}$ is maximal if for any superset of nodes $\boldsymbol{Y} \supset \boldsymbol{X}$, $\boldsymbol{Y}$ is not a clique.

Although the subset of nodes $\boldsymbol{X}$ can be arbitrary, we are often interested in sets of nodes that preserve certain aspects of the graph structure. We say that a subset of nodes $\boldsymbol{X} \in \mathcal{X}$ is upwardly closed in $\mathcal{K}$ if, for any $X \in \boldsymbol{X}$, we have that $\text{Boundary}_X \subset X$. We define the upward closure of $\boldsymbol{X}$ to be the minimal upwardly closed subset $\boldsymbol{Y}$ that contains $\boldsymbol{X}$. We define the upwardly closed subgraph of $\boldsymbol{X}$, denoted as $\mathcal{K}^+[\boldsymbol{X}]$, to be the induced subgraph over $\boldsymbol{Y}$, $\mathcal{K}[\boldsymbol{Y}]$

6.3.3 Paths

Using the basic notion of edges, we can define different types of longer-range connections in the graph. We say that $X_1, ..., X_k$ form a path in the graph $\mathcal{K} = (\mathcal{X}, \mathcal{E})$ if, for every $i = 1, ..., k-1$ we have that either $X_i \longrightarrow X_{i+1}$ or $X_i—X_{i+1}$. A path is directed if, for at least one i, we have $X_i \longrightarrow X_{i+1}$. We say that $X_1, ..., X_k$ form a trail in the graph $\mathcal{K} = (\mathcal{X}, \mathcal{E})$ if, for every $i = 1, ..., k-1$, we have that $X_i \rightleftharpoons X_{i-1}$. Then we say that a graph is connected if for every X_i, X_j there is a trail between X_i and X_j.

We can now define longer-range relationships in the graph. We say that X is an ancestor of Y in $\mathcal{K} = (\mathcal{X}, \mathcal{E})$, and that Y is a descendant of X, if there exists a directed path $X_1, ..., X_k$ with $X_1 = X$ and $X_k = Y$. We use Descendants_X to denote X's descendants, Ancestors_X to denote X's ancestors, and NonDescendants_X to denote the set of nodes in $\mathcal{X} - \text{NonDescendants}_X$.

A final useful notion is that of an ordering of the nodes in a directed graph that is consistent with the directionality its edges. Let $\mathcal{K} = (\mathcal{X}, \mathcal{E})$ be a graph, an ordering of the nodes $X_1, ..., X_k$ is a topological ordering relative to $\mathcal{K}$ if, whenever we have $X_i \longrightarrow X_j \in \mathcal{E}$, then $i < j$.

6.3.4 Cycles and loops

Note that, in general, we can have a cyclic path that leads from a node to itself, making that node its own descendant. A cycle in $\mathcal{K}$ is a directed path $X_1, ..., X_k$ where $X_1 = X_k$. A graph is acyclic if it contains no cycles. We will restrict attention to graphs that do not allow such cycles, since it is quite diffcult to define a coherent probabilistic model over graphs with directed cycles.

An acyclic graph containing only directed edges is called a Directed Acyclic Graph (DAG). An acyclic graph containing both directed and undirected edges is called a Partially Directed Acyclic Graph or (PDAG). The acyclicity requirement on a PDAG implies that the graph can be decomposed into a directed graph of chain components, where the nodes within each chain component are connected to each other only with undirected edges. The acyclicity of a PDAG guarantees us that we can order the components so that all edges point from lower-numbered components to higher-numbered ones.

Let $\mathcal{K}$ be a PDAG over $\mathcal{X}$. Let $\boldsymbol{K}_1, ..., \mathcal{K}_k$ be a disjoint partition of $\mathcal{X}$. such that the induced subgraph over $\boldsymbol{K}_i$ contains no directed edges and for any pair of nodes $\boldsymbol{X} \in \boldsymbol{K}_i$ and $\boldsymbol{Y} \in \boldsymbol{K}_j$ fro $i < j$, an edge between X and Y can only be a directed edge $X \longrightarrow Y$. Each component $\boldsymbol{K}_i$ is called a chain component.

A loop in $\mathcal{K}$ is a trail$X_1, ..., X_k$ where $X_1 = X_k$. A graph is singly connected if it contains no loops. A node in a singly connected graph is called a leaf if it has exactly one adjacent node. A singly connected directed graph is also called a polytree. A singly connected undirected graph is called a forest, and if it is also connected, it is called a tree. We can also define a notion of a forest, or of a tree, for directed graph, a directed graph is a forest if each node has at most one parent, and a directed forest is a tree if it is also connected.

6.4 Bayesian Networks

Our goal is to represent a joint distribution P over some set of random variables $\mathcal{X} = \{X_1, ..., X_n\}$. For all but the smallest n, the explicit representation of the joint distribution is unmanageable from every perspective. Computationally, it is very expensive to manipulate and generally too large to store in memory. Cognitively, it is impossible to acquire so many numbers from a human expert, moreover, the numbers are very small and do not correspond to events that people can reasonably contemplate. Statistically, if we want to learn the distribution from data, we would need ridiculously large amounts of data to estimate this many parameters robustly. Bayesian Networks use the independence properties in the distribution to represent such high-dimensional distributions much more compactly, the combinatorial data structure of a DAG can provide us with a general purpose modeling language for exploiting this type of structure in our representation.

6.4.1 The Naïve Bayes model

The Naïve Bayes model (NB) is the simplest example where a conditional parameterization is combined with conditional independence assumptions to produce a very compact representation of a high-dimensional probability distribution.

The naive Bayes model assumes that instances of a feature fall into one of a number of mutually exclusive and exhaustive classes. Thus, we have a class variable C that takes on values in some set $\{c^1, ..., c^k\}$. The model also includes some number of features $X_1, ..., X_n$ whose values are typically observed. The naive Bayes assumption is that the features are conditionally independent given the instance's class, within each class of instances, the different properties can be determined independently. Then, based on these independence assumptions, we can show that the model factorizes as

$$P(C, X_1, ..., X_n) = P(C) \prod_{i=1}^{n} P(X_i|C)$$

6.4.2 Bayesian Networks

Bayesian networks build on the same intuitions as the naive Bayes model by exploiting conditional independence properties of the distribution in order to allow a compact and natural representation. However, they are not restricted to representing distributions satisfying the strong independence assumptions implicit in the naive Bayes model. They allow us the flexibility to tailor our representation of the distribution to the independence properties that appear reasonable in the current setting.

The core of the Bayesian Network representation is a DAG $\mathcal{G}$, whose nodes are the random variables in our domain and whose edges correspond, intuitively, to direct influence of on a variable on another. This graph $\mathcal{G}$ can be viewed as a data structure that provides the skeleton for representing a joint distribution compactly in a factorized way and as a compact representation for a set of conditional independence assumptions about a distribution

The second component of the Bayesian network representation is a set of local probability models that represent the nature of the dependence of each variable on its parents In general, each variable X in the model is associated with a Conditional Probability Distribution (CPD) that specifies a distribution over the values of X given each possible joint assignment of values to its parents in the model, in such a way that the non-descendants of F are then conditionally independent of X. For a node with no parents, the CPD is conditioned on the empty set of variables. Hence, the CPD turns into a marginal distribution.

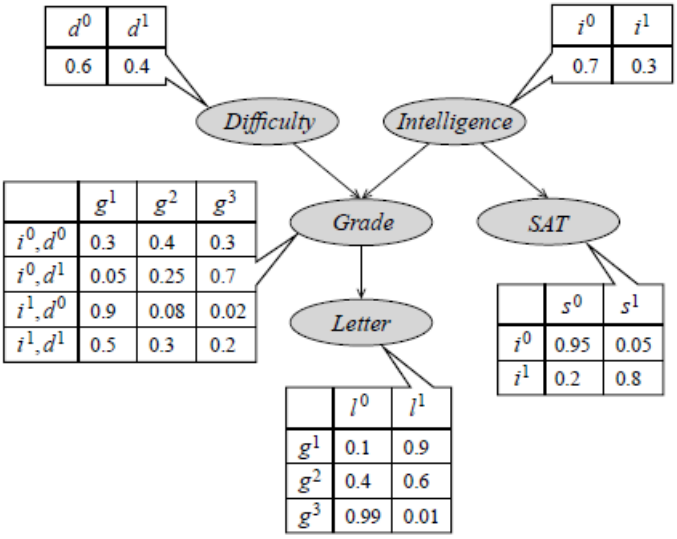

FIGURE 6.1: Example of a Bayesian Network for the decision of hiring a student by a company $\mathcal{B}$ with the correspondent CPDs. We have five random variables in this domain: the student's intelligence (I), the course difficulty (D), the grade (G), the student's SAT score (S), and the quality of the recommendation letter (L).

6.4.3 Inference

A joint distribution specifies the probability $P_{\mathcal{B}}(\boldsymbol{Y} = \boldsymbol{y}|\boldsymbol{X} = \boldsymbol{x})$ of any event $\boldsymbol{y}$ given any observations $\boldsymbol{x}$. We condition the joint distribution on the event $\boldsymbol{X} = \boldsymbol{x}$ by eliminating the entries in the joint inconsistent with our observation $\boldsymbol{x}$, and renormalizing the resulting entries to sum to 1. We then compute the probability of the event $\boldsymbol{y}$ by summing the probabilities of all of the entries in the resulting posterior distribution that are consistent with $\boldsymbol{y}$. To illustrate this process, let us consider a network $\mathcal{B}$ and see how the probabilities of various events change as evidence is obtained:

- Queries in which we predict the downstream effects of various factors are instances of causal reasoning or prediction.
- Queries in which we use evidences to reason upstream from effects to causes are instances of evidential reasoning or explanation.
- Queries in which different causes of the same effect can interact are instances of a general reasoning pattern called intercausal reasoning, where different causes of the same effect can interact.

6.5 Learning

6.6 Classification

6.7 Decision Trees

A decision tree is a representation of the different scenarios that might be encountered by the decision maker in the context of a particular decision problem. A decision tree has two types of internal nodes (denoted t-nodes to distinguish them from nodes in a graphical model), one set encoding decision points of the agent, and the other set encoding decisions of nature. The outgoing edges at an agent's t-node correspond to different decisions that the agent might make. The outgoing edges at one of nature's t-nodes correspond to random choices that are made by nature. The leaves of the tree are associated with outcomes, and they are annotated with the agent's utility for that outcome.

More formally a decision tree is a rooted tree with a set of internal t-nodes $\mathcal{V}$ and leaves $\mathcal{V}_L$. The set $\mathcal{V}$ is partitioned into two disjoint sets, agent t-nodes $\mathcal{V}_A$ and nature t-nodes $\mathcal{V}_N$. Each t-node has some set of choices $\mathcal{C}[v]$, associated with its outgoing edges. We let $\text{succ}(v, c)$ denote the child of v reached via the edge labeled with c. Each of nature's t-nodes is associated with a probability distribution P_v over $\mathcal{C}[v]$. Each leaf $v \in \mathcal{V}_L$ in the tree is annotated with a numerica value $U(v)$ corresponding to the agents's utility for reaching that leaf.

The decision-tree representation allows us to encode decision scenarios in a way that reveals much more of their internal structure than the abstract setting of outcomes and utilities. In particular, it allows us to encode explicitly sequential decision settings, where the agent makes several decisions. It also allows us to encode information that is available to the agent at the time a decision must be made.

We can encode the agent's overall behavior in a decision problem encoded as a decision tree as a strategy. There are several possible definitions of a strategy. One that is simple and suitable for our purposes is a mapping from agent t-nodes to possible choices at that t-node. A decision-tree strategy specifies, for each $v \in \mathcal{V}_A$ one of the choices labeling its outgoing edges.

6.8 Influence diagrams

The decision-tree representation is a significant improvement over representing the problem as a set of abstract outcomes. However, much of the structure of the problem is still not made explicit. An alternative representation is the influence diagram (sometimes also called a decision network), a natural extension of the Bayesian network framework. It encodes the decision scenario via a set of variables, each of which takes on values in some space. Some of the variables are random variables, as we have seen so far, and their values are selected by nature using some probabilistic model. Others are under the control of the agent, and their value reflects a choice made by him. Finally, we also have numerically valued variables encoding the agent's utility. This type of model can be encoded graphically, using a DAG containing three types of nodes corresponding to chance variables, decision variables,

and utility variables. These different node types are represented as ovals, rectangles, and diamonds, respectively. An influence diagram $\mathcal{I}$ is a DAG over these nodes, such that the utility nodes have no children.

More formally, in an influence diagram, the world in which the agent acts is represented by the set of chance variables $\mathcal{X}$, and by a set of decision variables $\mathcal{D}$. Chance variables are those whose values are chosen by nature. The decision variables are variables whose values the agent gets to choose. Each variable $Vin\mathcal{X} \cup \mathcal{D}$ has a finite domain $\text{Val}(V)$ of possible values. The possible actions $\mathcal{A}$ are all of the possible assignments $\text{Val}(\mathcal{D})$, and the possible outcomes are all the joint assignments in $\text{Val}(\mathcal{X} \cup \mathcal{D})$. Thus, this framework provides a factored representation of both the action and the outcome space.

So, summarizing, an influence diagram $\mathcal{I}$ over $\mathcal{Z} = \mathcal{X} \cup \mathcal{D} \cup \mathcal{U}$ is a DAG whose nodes correspond to $\mathcal{Z}$ and where nodes corresponding to utility $\mathcal{U}$ have no children. Each chance variable $X \in \mathcal{X}$ is associated with a CPD $P(X|\text{Pa}_X)$ and each utility variable $U \in \mathcal{U}$ is associated with a deterministic funtion $U(\text{Pa}_U)$.

6.8.1 Decision rules

For a decision variable $D \in \mathcal{D}$, Pa_D is the set of variables whose values the agent knows when he chooses a value for D. The edges incoming into a decision variable are often called information edges. The influence diagram representation captures the causal structure of the problem and its parameterization in a much more natural way than the decision tree. The choice that the agent makes for a decision variable D can be contingent only on the values of its parents. More precisely, in any trajectory through the decision scenario, the agent will encounter D in some particular information information state states, where each information state is an assignment of values to Pa_D. An agent's strategy for D must tell the agent how to act at D, at each of these information states.

A decision rule δ_D for a decision variable D is a conditional probability $P(D|\text{Pa}_D)$, a function that maps each instantiation pa_D of Pa_D to a probability distribution over $\text{Val}(D)$. A decision rule is deterministic if each probability distribution $\delta_D(D|\text{Pa}_D)$ assings nonzero probability to exactly one value of D. A complete assignment σ of decision rules to every decsion $D \in \mathcal{D}$ is called a complete strategy. We use σ_D to denote the decision rule at D.

Bibliography

[1] Alan Agresti. *An Introduction to Categorical Data Analysis*. Wiley Series in Probability and Statistics. Wiley, 2006. ISBN: 9780471226185. DOI: 10.1002/0470114754.

[2] Alan Agresti. *Statistical Learning and Pattern Analysis for Image and Video Processing*. Wiley Series in Probability and Statistics. Wiley, 2006. ISBN: 9780471226185. DOI: 10.1002/0470114754.

[3] Fabian Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[4] Daphne Koller and Nir Friedman. *Probabilistic Graphical Models: Principles and Techniques*. Adaptive Computation and Machine Learning. The MIT Press, 2009. ISBN: 9780262013192.

[5] Quoc V. Le, Google Brain, and Google Inc. *A Tutorial on Deep Learning Part 1: Nonlinear Classifiers and The Backpropagation Algorithm*. 2015. URL: https://cs.stanford.edu/~quocle/tutorial1.pdf.

[6] Quoc V. Le, Google Brain, and Google Inc. *A Tutorial on Deep Learning Part 2: Autoencoders, Convolutional Neural Networks and Recurrent Neural Networks*. 2015. URL: https://cs.stanford.edu/~quocle/tutorial2.pdf.

[7] Yunqian Ma and Yun Fu. *Manifold Learning Theory and Applications*. CRC Press, 2011. ISBN: 9781439871096. DOI: 10.1201/b11431.

[8] Burr Settles. *Active Learning Literature Survey*. Computer Sciences Technical Report 1648. University of Wisconsin–Madison, 2009. URL: http://burrsettles.com/pub/settles.activelearning.pdf.