GHENT UNIVERSITY

Lecture Notes

E016350: Artificial Intelligence

Part I: Basics of Machine Learning and Probabilistic Reasoning

Aleksandra Pizurica

Spring 2026

Contents

Disclaimer*: These lecture notes were written by Prof. Aleksandra Pizurica to accompany the slides of the course E016350: Artificial Intelligence, facilitating their understanding. The lecture notes are not meant to be self-contained, and do not cover all the study material in the course. They are by no means meant to replace the recommended textbook and do not necessarily cover all the relevant aspects that are presented in the slides and explained in the lectures. Some sections are adapted from the book of S. Russel and P. Norvig: Artificial Intelligence: A Modern Approach.*

1 Theory of learning

In supervised learning, we have a **training set** of N input-output pairs (training examples) $\mathcal{D}_{train} = \{(\mathbf{x}^{(i)}, y^{(i)}); i = 1, \dots N\}$. The input $\mathbf{x}^{(i)}$ is sometimes also called input **features**, and sometimes we rather use a separate notation $\phi(\mathbf{x})$ to stress that features ϕ are extracted from a (raw) input $\mathbf{x}$. The output $y^{(i)}$ is a **target** variable (also called **label**) that we are trying to predict.

When the output y is a number (such as the predicted arrival time), the learning problem is called **regression**. When the output is one of a finite set of values (e.g., land-cover class in a satellite photo: road, building, vegetation or water), the learning problem is called **classification**. In **Boolean** or **binary** classification there are only two values (e.g., classification of online comments to toxic or non-toxic or tissue classification in a medical image to normal or pathologic, etc.).

In the formal (mathematical) theory of learning, we say that given the set of training examples $\{(\mathbf{x}^{(i)}, y^{(i)}) \dots (\mathbf{x}^{(N)}, y^{(N)})\}$, where each $y^{(i)}$ was generated by an unknown function $y = f(\mathbf{x})$, the **goal of supervised learning** is to discover a function h that approximates the true function f.

The function h is a **hypothesis** about the world and is drawn from some **hypothesis space** $\mathcal{H}$. For example, the hypothesis space is the set of all polynomials up to some predefined degree. In different words, we say that h is a **model** of the data drawn from some **model class** $\mathcal{H}$, or, in general, it is a **function** drawn from some **function class**. For **parametric** models, different $h \in \mathcal{H}$ have different parameters $\mathbf{w}$. Here, the hypothesis is a parametrized function $h_{\mathbf{w}}(\mathbf{x})$, and the optimization consists in finding $\mathbf{w}$ that best fits the training data.

Fig. 1.1 illustrates the principle of supervised learning and Fig. 1.2 clarifies the notion of the hypthesis space and the functions within it. But how do we choose $\mathcal{H}$ to start with? We might exploit some prior knowledge about the underlying processes that generated the data. We can also perform **exploratory data analysis**: examining the data with statistical tests and visualizations –histograms, scatter plots, box plots – to get a feel for the data, and some insight into what hypothesis space might be appropriate [18]. Often we also try multiple hypothesis spaces (different types of models and/or different variants of the same kind of models) and evaluate which one works best for the given task.

The next question is how to choose a good hypothesis from within the hypothesis space. We say that a hypothesis is good if it correctly predicts the value of y for new examples. We then say that the hypothesis **generalizes well**. Intelligence can be seen as the ability to predict (e.g. the next sample) and generalize to unseen scenarios [17].

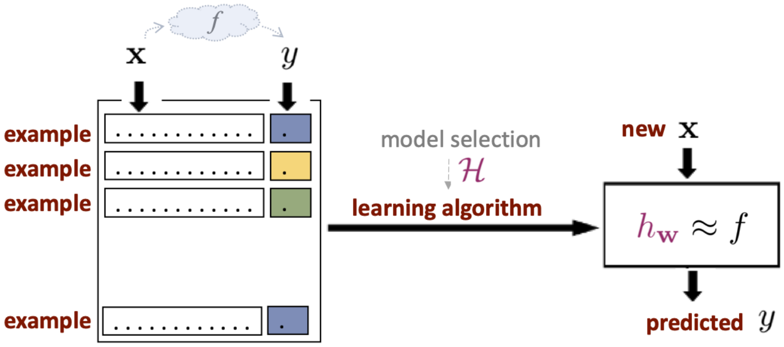


Figure 1.1: The principle of supervised learning, illustrated for a parameteric approach ($h = h_{\mathbf{w}}$).

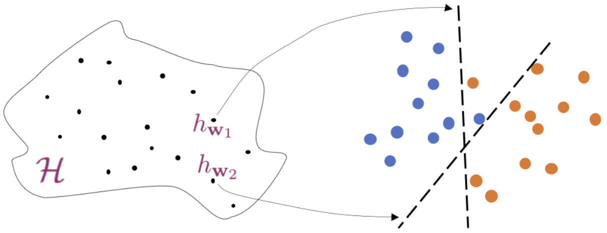


Figure 1.2: An illustration of the hypothesis space, exemplified with a parameteric model for binary linear classification where different hypotheses $h_{\mathbf{w}_i}$ are characterized with different parameters (weights) $\mathbf{w}_i$. Here, each $h_{\mathbf{w}_i}$ yields a different decision boundary. (e.g., $h_{\mathbf{w}_i}$ yields the predicted label for the input x as $y = 1$ if $w_{i,1}x + w_{i,0} > 0$ and $y = 0$ otherwise).

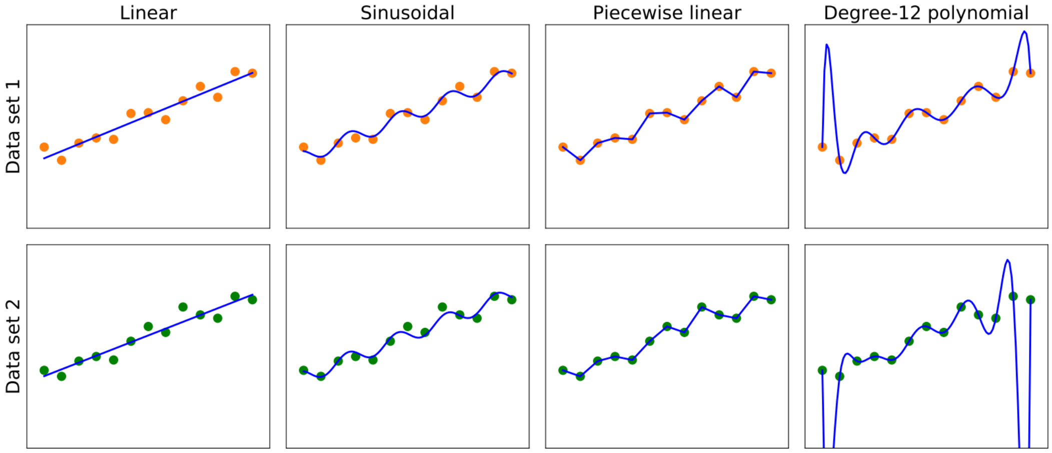


Figure 1.3: Finding hypotheses to fit data. Top row: four plots of best-fit functions from four different hypothesis spaces trained on data set 1. Bottom row: the same four functions, but trained on a slightly different data set (sampled from the same $f(x)$ function). Taken from [18].

The example in Fig. 1.3 shows how the best-fitted model differs depending on the chosen hypothesis space and it also illustrates the effect of the particular training set. Observe that the linear model has a **large bias** (i.e., a large deviation from the expected value when averaged over different training sets). It is because it allows only functions consisting of straight lines and thus fails to represent any patterns in the data other than the overall slope of a line. We say that such a hypothesis, which fails to find a pattern in the data, is **underfitting**. In the other extreme, a 12-order polynomial has a small bias but it has a **large variance**: a small fluctuation in the training data set translates into a large difference in the hypothesis. It means that at least one of the two found hypotheses must be a poor approximation for the true underlying f from which both datasets were drawn. Such a function that pays too much attention to the particular data set it is trained on is said to be **overfitting** and will perform poorly on unseen data.

Hence, we often face a **bias-variance trade-off**: a choice between more complex, low-bias hypotheses that fit the training data well and simpler, low-variance hypotheses that may generalize better [18]. A general principle, known as **Ockham's razor** tells us that the best models are simple models that fit the data well. In other words, simpler explanations are, other thing s being equal, generally better than more complex ones.

1.1 Model selection and optimization

The task of finding a good hypothesis consists of two main subtasks:

1. **Model selection**: selection of the model class, i.e, selection of the hypothesis space $\mathcal{H}$. This step determines the **hyperparameters** of the learning algorithm. For example, a learning algorithm typically involves a hyperparameter that determines the *size* of the model, like the number of layers in a neural network. If we learn a decision tree, the size could be the number of nodes in the tree; for polynomials, the maximal *degree* of the polynomial.

2. **Optimization**, also called **training**: finding the best hypothesis h within the selected $\mathcal{H}$. It involves a concrete *learning algorithm*, which needs to evaluate how good the predictors are based on some *loss function*.

1.2 Parametric models

For **parametric** models, different $h \in \mathcal{H}$ have different parameters $\mathbf{w}$. Here, the hypothesis is a parametrized function $h_{\mathbf{w}}(\mathbf{x})$, and the optimization consists in finding $\mathbf{w}$ that best fits the training data. Most of the machine learning approaches that we will cover in this course, ranging from the simplest linear regression and logistic regression to deep neural networks are parameteric models. In this chapter, we will deal with parameteric models but we will address nonparameteric machine learning models as well in some of the subsequent chapters.

1.3 Training, validation and test sets

In machine learning, we want to select a hypothesis that will optimally fit some future examples. Here we are implicitly making an assumption that the future examples will behave like the past ones. This is called **stationarity** assumption. Next, we need to define what is the *optimal* fit (i.e., the *best* hypothesis). We will say that it is a hypothesis that minimizes some **error rate**, being the proportion of times that $h(\mathbf{x}) \neq y$ for an $(\mathbf{x}, y)$ example. To estimate the error rate of a hypothesis, we need to test it by measuring its performance on a **test set** of examples. To ensure a fair evaluation, the simplest way is to split the set of all available examples into a **training set** (to create the hypothesis, i.e., to train the model) and a **test set** (to evaluate it). When we are creating only one model these two sets are sufficient. But if we want to compare multiple competing models, which can be entirely different machine learning models or variants of the same approach (with differently adjusted hyperparameters), then we need a third set of examples called the **validation set**. Hence, as a general rule, we need to split the set of all available examples $\mathcal{D}$ into three disjoint sets:

- **Training set** ($\mathcal{D}_{train}$) – to train candidate models
- **Validation set** ($\mathcal{D}_{val}$) – to evaluate the candidate models and choose the best one
- **Test set** ($\mathcal{D}_{test}$) – to do a final unbiased evaluation of the best model

When we don't have enough data to make properly all three of these data sets with sufficient sizes, we can use k-**fold cross-validation**. This technique enables us to "squeeze more" out of the data by allowing each sample to serve double duty – as training and validation data – but not at the same time [18]. We then perform k rounds of learning, each time $1/k$ of the data in $\mathcal{D} \setminus \mathcal{D}_{test}$ is held as a validation set and the remaining examples are used as the training examples. ($\mathcal{D}_{test}$ is

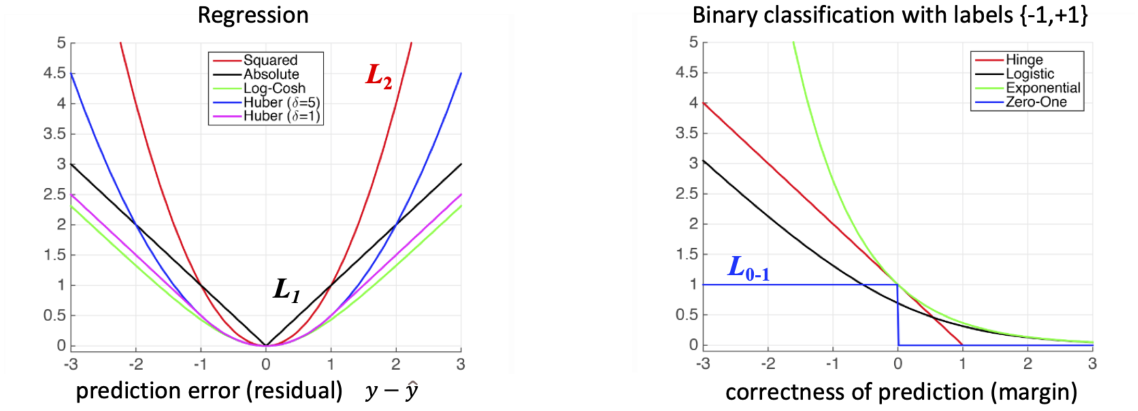


Figure 1.4: Common loss functions for the regression tasks (left) and binary classification (right).

always kept separately and not touched until the testing phase). Popular choices for k are 5 and 10. The extreme case with $k = n$ is known as **leave-one-out cross validation**.

1.4 Loss function

Since in AI the decision making (and optimal action) maximizes some form of **expected utility**, we can define the **loss function** in general terms as the amount of utility lost by replacing the correct answer $f(\mathbf{x}) = y$ by a hypothesis $h(\mathbf{x}) = \hat{y}$. This is the most general formulation. We will write the loss function as $L(y, h(\mathbf{x}))$ and often simply as $L(y, \hat{y})$. In the case of *parameteric* learning $\hat{y} = h_{\mathbf{w}}(x)$ we will use interchangeably $L(y, h_{\mathbf{w}}(\mathbf{x}))$ and $L(\mathbf{x}, y, \mathbf{w})$.

Fig. 1.4 illustrates some common loss functions for the regression tasks and for the binary classification tasks. For the regression task, the loss function needs to increase with the difference between the true and the predicted output. This increase is linear for the **absolute value loss** $L_1(y, \hat{y}) = |y - \hat{y}|$ and quadratic for the **squared-error loss**: $L_2(y, \hat{y}) = (y - \hat{y})^2$. The **Huber** loss function behaves as quadratic for small prediction errors and liner for large prediction errors:

$$L_\delta(y, \hat{y}) = \begin{cases} \frac{1}{2}(y - \hat{y})^2 & \text{if } |y - \hat{y}| \leq \delta \\ \delta(|y - \hat{y}| - \frac{1}{2}\delta) & \text{otherwise} \end{cases}$$

This makes it less sensitive to outliers in data than the squared error loss. Its parameter δ allows adjusting the transition between the two regions and the slope of the linear part.

For classification, we want to penalize the cases that will yield $\hat{y} \neq y$, but now it makes no sense to let the loss depend on the value of the difference $y - \hat{y}$ (think why). Rather, we now can reason how *confident* the model was when making a certain prediction and let the loss depend on the *correctness of prediction*, which is larger when the model is more confident about its correct prediction, and, conversely, is smaller when the model is more confident about its wrong prediction. Fig. 1.4 shows some common classification loss functions. For the **zero-one loss** $L_{0-1}(y, \hat{y})$ the penalty is 0 if $\hat{y} = y$ and 1 if $\hat{y} \neq y$ regardless of how confident the model was in predicting a particular value of $\hat{y}$. Other, more nuanced loss functions shown in the diagram on the right of Fig. 1.4 take this confidence into account.

1.5 Training loss

The **expected generalization loss** for a hypothesis h, with respect to loss function L is the mathematical expectation of the loss:

$$GenLoss_L(h) = \sum_{(\mathbf{x},y)\in\mathcal{E}} L(y, h(\mathbf{x}))P(\mathbf{x}, y) \tag{1.1}$$

where $\mathcal{E}$ denotes the set of all possible input-output pairs, and $P(\mathbf{x}, y)$ the joint probability of $\mathbf{x}$ and y. The **best hypothesis** is the one that yields the minimum expected generalization loss:

$$h^* = \arg\min_{h\in\mathcal{H}} GenLoss_L(h) \tag{1.2}$$

In reality, the true distribution $P(\mathbf{x}, y)$ is not known, so the learning agent can only estimate the generalization loss with an **empirical loss** on a set of available examples E:

$$EmpLoss_{L,E}(h) = \frac{1}{|E|} \sum_{(\mathbf{x},y)\in E} L(y, h(\mathbf{x})) \tag{1.3}$$

By minimizing the empirical loss, we obtain the **estimated best hypothesis**:

$$\hat{h}^* = \arg\min_{h\in\mathcal{H}} EmpLoss_{L,E}(h) \tag{1.4}$$

We define the **training loss** as the empirical loss over the set of training examples $\mathcal{D}_{train}$:

$$TrainLoss_{L,\mathcal{D}_{train}}(h) = \frac{1}{|\mathcal{D}_{train}|} \sum_{(\mathbf{x},y)\in\mathcal{D}_{train}} L(y, h(\mathbf{x})) \tag{1.5}$$

For compactness, we will suppress the subscripts L and $\mathcal{D}_{train}$, and for parameteric models $h_\mathbf{w}$, the loss function can be written both as $L(y, h_\mathbf{w}(\mathbf{x}))$ and $L(\mathbf{x}, y, \mathbf{w})$, so we also write:

$$TrainLoss(\mathbf{w}) = \frac{1}{|\mathcal{D}_{train}|} \sum_{(\mathbf{x},y)\in\mathcal{D}_{train}} L(\mathbf{x}, y, \mathbf{w}) \tag{1.6}$$

Sometimes we will find it more convenient to express the training set explicitly as N examples: $\mathcal{D}_{train} = \{(\mathbf{x}^{(i)}, y^{(i)}); i = 1, \ldots N\}$, and then write the training loss as

$$TrainLoss(\mathbf{w}) = \frac{1}{N}\sum_{i=1}^{N} L(y^{(i)}, h_\mathbf{w}(\mathbf{x}^{(i)})) = \frac{1}{N}\sum_{i=1}^{N} L(\mathbf{x}^{(i)}, y^{(i)}, \mathbf{w}) \tag{1.7}$$

which is equivalent to (1.6). Finally, we will sometimes use **explicit regularization** of a machine learning model, meaning that we will add a penalty term that penalizes the complexity of the solution. In other words, we will directly minimize a weighted sum of the empirical loss and the complexity of the hypothesis, which is also called the **total cost**:

$$Cost(h) = EmpLoss(h) + \lambda Complexity(h) \tag{1.8}$$

Here, $\lambda \geq 0$ is a parameter, often determined by cross-validation. This process explicitly penalizes complex hypotheses, promoting thus more 'regular' functions as solutions and is therefore called regularization. In practice, our training objective will become:

$$\min_{\mathbf{w}} \quad \sum_{i=1}^{N} L(y^{(i)}, h_\mathbf{w}(\mathbf{x}^{(i)})) + \lambda Reg(\mathbf{w}) \tag{1.9}$$

where $Reg(\mathbf{w})$ is some regularization function imposed on the weights, e.g., ℓ_1-regularization: $Reg(\mathbf{w}) = |\mathbf{w}|$ or ℓ_2-regularization: $Reg(\mathbf{w}) = \mathbf{w}^2$. We address regularization more concretely in Section 2.2.

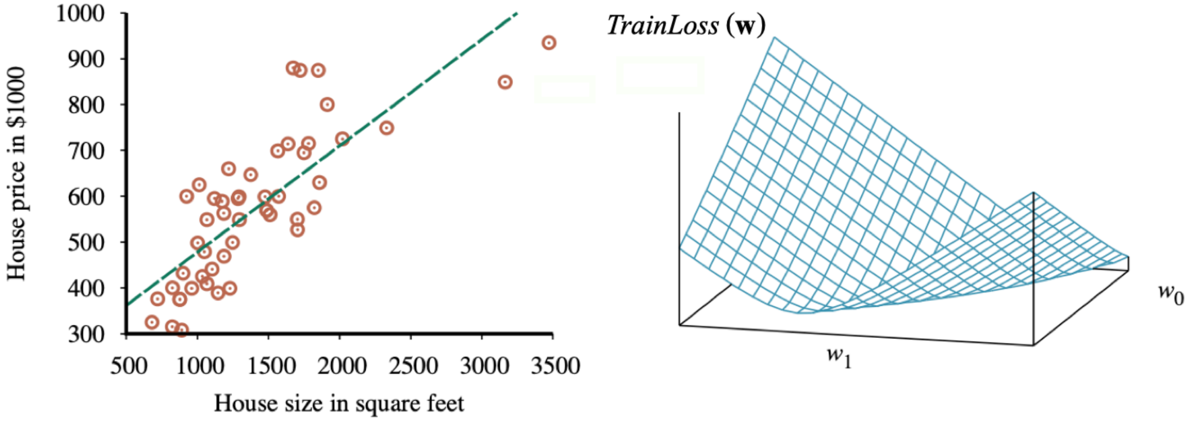


Figure 2.1: Left: data points and the fitted line under the squared error loss. Right: Plot of the training loss $\sum_i (y^{(i)} - (w_1 x^{(i)} + w_0))^2$ for various values of w_0, w_1. Observe that the training loss is a *convex* function. This is true for *every* linear regression problem with an L_2 loss function [18].

2 Linear regression

The task of linear regression is fitting a linear model through the training data. The hypothesis space is the space of all **linear functions** of continuous-valued inputs. The learning algorithm seeks to find the parameters $\mathbf{w}$, which characterize the best fitted line. The produced model $h_\mathbf{w}$ in the context of regression is called a **predictor**.

2.1 Univariate linear regression

In univariate linear regression, the inputs are one-dimensional (real numbers) x. The goal is to *fit a straight line*, i.e., to learn the coefficients w_0 and w_1 of a **univariate linear function**:

$$y = w_1 x + w_0 \tag{2.1}$$

The coefficients w_0 and w_1 can be seen as **weights**: the value of y is changed by changing the relative weight of one term or another.

Denoting the vector of weights by $\mathbf{w} = [w_0 \; w_1]^\top$, the predictor is

$$h_\mathbf{w}(x) = w_1 x + w_0 \tag{2.2}$$

and the task is to find $\mathbf{w}$ such that $h_\mathbf{w}(x)$ fits best the training data.

Typically, squared error loss function L_2 is used, which is then called **least squares linear regression**. An example is shown in Fig. 2.1.

A historical note: Carl Friedrich Gauss showed that when the noise in the outputs y is normally distributed, the most likely values of the weights are obtained using the L_2 loss, i.e., minimizing the sum of the squared errors [3]. It is believed that Gauss used this method in 1801 to model the orbit of Ceres (a dwarf planet) and to predict its location. The Ceres dataset consisted of 19 observations of the Ceres's locations, acquired over 42 days, where each data point consisted of a time stamp and the location on the sky. Ceres was located within 1/2 degree of Gauss's prediction, which was a far better prediction than made by other astronomers at the time. The least squares method was published by the French mathematician Adrien-Marie Legendre in 1805, and is attributed to Legendre but usually also co-credited to Gauss.

The training loss for the linear regression is thus commonly defined with the L_2 loss function. If the training set consists of N examples: $\mathcal{D}_{train} = \{(x^{(i)}, y^{(i)}); i = 1, \dots N\}$, the training loss is

$$TrainLoss(\mathbf{w}) = \sum_{i=1}^{N} (y^{(i)} - (w_1 x^{(i)} + w_0))^2 \tag{2.3}$$

We find the weights w_0 and w_1 by setting to zero the corresponding partial derivatives of the training loss, i.e.,

$$\frac{\partial}{\partial w_0} \sum_{i=1}^{N} (y^{(i)} - (w_1 x^{(i)} + w_0))^2 = 0; \qquad \frac{\partial}{\partial w_1} \sum_{i=1}^{N} (y^{(i)} - (w_1 x^{(i)} + w_0))^2 = 0 \tag{2.4}$$

This yields:

$$w_1 = \frac{N \sum_i x^{(i)} y^{(i)} - \sum_i x^{(i)} \sum_i y^{(i)}}{N \sum_i \left(x^{(i)}\right)^2 - \left(\sum_i x^{(i)}\right)^2}; \qquad w_0 = \left(\sum_i y^{(i)} - w_1 \sum_i x^{(i)}\right) / N \tag{2.5}$$

Although in this case we have a closed form solution, we will need a more general method for determining the weights that does not rely on solving to find the zeroes of the derivatives of the entire training loss. This will be needed for various reasons, e.g., when using other than L_2 loss functions and/or when the training examples arrive sequentially.

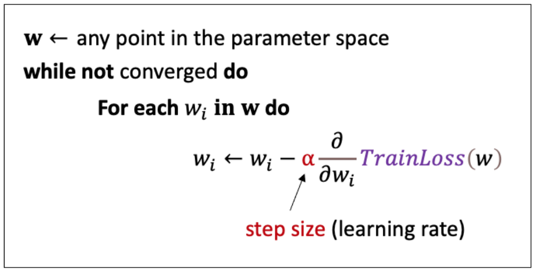
One such method that can be applied to any loss function – no matter how complex it is – is the **gradient descent**. It searches through a continuous weight space by incrementally modifying the parameters. Fig. 2.2 gives the pseudo-code and illustrates its operation. The parameter α, which is called the **step size** is usually called the **learning rate** when we are trying to minimize a loss in a learning problem. It can be a fixed constant, or chosen to decay as the learning process proceeds [18]. For linear regression with the L_2 loss function, the training loss is always **convex** and thus gradient descent will find the optimal solution.

Let us now derive the learning rule for the linear regression with the common L_2 loss. We start from a simplified case with one training example (x, y):

$$\frac{\partial}{\partial w_j} TrainLoss(\mathbf{w}) = \frac{\partial}{\partial w_j} (y - h_{\mathbf{w}}(x))^2 \tag{2.6}$$

Applying to both w_0 and w_1, we get

$$\frac{\partial}{\partial w_0} TrainLoss(\mathbf{w}) = -2(y - h_{\mathbf{w}}(x)); \qquad \frac{\partial}{\partial w_1} TrainLoss(\mathbf{w}) = -2(y - h_{\mathbf{w}}(x))x; \tag{2.7}$$



```
w ← any point in the parameter space
while not converged do
        For each $w_i$ in w do
                $w_i \leftarrow w_i - \alpha \frac{\partial}{\partial w_i} TrainLoss(w)$
                        step size (learning rate)
```

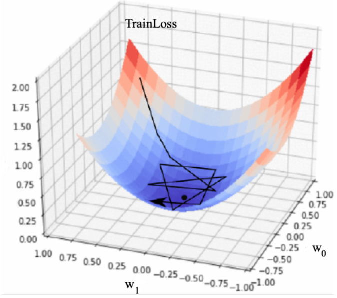


Figure 2.2: Left: Pseudo-code of the gradient descent algorithm. Right: An illustration of its operation (image from [3]).

Plugging this into the update rule of the gradient descent and folding the constant 2 into the unspecified learning rate[1] α, we obtain

$$w_0 \leftarrow w_0 + \alpha(y - h_{\mathbf{w}}(x)); \qquad w_1 \leftarrow w_1 + \alpha(y - h_{\mathbf{w}}(x))x; \tag{2.8}$$

These learning rules are intuitive: if $h_{\mathbf{w}}(x) = y$, the prediction is correct so don't change anything (keep the weights as they are). If $h_{\mathbf{w}}(x) > y$, i.e., the output of the hypothesis is too large, reduce w_0 a bit and reduce w_1 if x is positive but increase w_1 if x was negative. The opposite holds when $h_{\mathbf{w}}(x) < y$. While these updates were obtained for one training example, we can simply extend them to the case with N training examples. What changes is that in Eq (2.6) we will have the partial derivative of the sum $\sum_i (y^{(i)} - h_{\mathbf{w}}(x^{(i)}))^2$ instead of the partial derivative of a single element $(y - h_{\mathbf{w}}(x))^2$. Since the derivative of a sum is the sum of the derivatives, the expressions in Eq (2.8) generalize straightforwardly to

$$w_0 \leftarrow w_0 + \alpha \sum_{i=1}^{N} (y^{(i)} - h_{\mathbf{w}}(x^{(i)})); \qquad w_1 \leftarrow w_1 + \alpha \sum_{i=1}^{N} (y^{(i)} - h_{\mathbf{w}}(x^{(i)}))x^{(i)}; \tag{2.9}$$

Note that here we are summing over all the N training examples in each step. This is the so-called **batch gradient descent** learning rule for univariate linear regression. Since the loss function is convex, convergence to the optimum is guaranteed unless the learning rate is chosen too large so that it "overshoots" (see the following notes). A faster variant, called **stochastic gradient descent** makes in each step updates based on one randomly selected sample, according to Eq (2.8). Most commonly, the updates are made on a **minibatch** of m out of total N examples, and the resulting learning rule is known as the **minibatch gradient descent**. We return to these optimization aspects in the following notes.

2.2 Multivariate linear regression

In **multivariate** linear regression the input is a vector $\mathbf{x} = [x_1, \ldots, x_n]^\top$. Some authors use the terms multivariate, **multivariable** and **multiple** linear regression interchangeably. In some cases, a differentiation is made in the sense that the term *multivariate* denotes a more general case where the output is also a vector (this is the terminology, e.g., in [18]). We will here consider that the output is always a single number y.

[1] Note that we could have written this update rule with some learning rate $\alpha' = 2\alpha$ but since the learning rate is not specified, we simply denote the new constant with some generic α, and we could have chosen any other symbol.

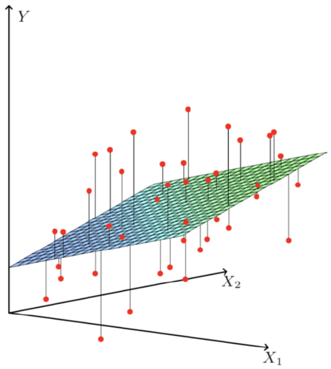


Figure 2.3: An illustration of the multivariate linear regression taken from [13].

We generalize the univariate linear regression to the case where each input $\mathbf{x}$ is a vector as:

$$h_{\mathbf{w}}(\mathbf{x}) = w_0 + w_1 x_1 + \cdots + w_n x_n = w_0 + \sum_j w_j x_j \tag{2.10}$$

In order to treat the intercept term w_0 in the same way as the others, we introduce a dummy input attribute x_0 , which is always 1. Then we can write compactly

$$h_{\mathbf{w}}(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x} = \mathbf{w}^\top \mathbf{x} = \sum_j w_j x_j \tag{2.11}$$

The inner product $\mathbf{w} \cdot \mathbf{x}$ between weights $\mathbf{w}$ and input $\mathbf{x}$ is often called the **score**.

The optimal weights can be obtained analytically, using the tools of linear algebra and vector calculus. Let $\mathbf{X}$ be the **data matrix** defined as the matrix of inputs, where each raw is one n-dimensional input example: $\mathbf{X}(:, i) = (\mathbf{x}^{(i)})^\top = [x_1^{(i)}, \ldots, x_n^{(i)}]$. One can show that

$$\mathbf{w}^* = \arg\min_{\mathbf{w}} \sum_i L_2(y^{(i)}, \mathbf{w} \cdot \mathbf{x}^{(i)}) = \arg\min_{\mathbf{w}} \|\mathbf{y} - \mathbf{X}\mathbf{w}\|_2^2 = \underbrace{(\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top}_{pseudoinverse} \mathbf{y} \tag{2.12}$$

So, we can obtain an analytical solution for the optimal (in the mean squared error sense) weights as the **pseudoinverse** $(\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top$ of the data matrix $\mathbf{X}$. In practice, it may be difficult to calculate the pseudoinverse of a large data matrix.

Instead, we can employ the gradient descent, which updates the weights as

$$w_j \leftarrow w_j + \alpha \sum_i (y^{(i)} - h_{\mathbf{w}}(\mathbf{x}^{(i)})) x_j^{(i)} \tag{2.13}$$

Gradient descent reaches the **unique minimum** because the training loss remains convex also for multivariate regression with the squared loss function.

2.3 Regularization

With multivariable linear functions it is common to use some form of **regularization** to avoid overfitting. This is because in high-dimensional spaces the data are more sparse and the chance is bigger that some dimensions that are in fact irrelevant are given more importance only because by chance they appeared to be useful.

In Eq (1.8), we defined a total cost by adding explicitly a regularization term to the optimization problem. For linear regression, we commonly specify the complexity of the hypthesis in terms of its weights, and particularly we consider the so-called ℓ_p family[2] of regularization functions:

$$Complexity(h_{\mathbf{w}}) = \ell_p(\mathbf{w}) = \sum_j |w_j|^p \tag{2.14}$$

Here, $\ell_p(\mathbf{w}) = \|w\|_p^p$, where $\|w\|_p$ is the L_p-norm:

$$\|\mathbf{w}\|_p = \Big(\sum_j w_j^p\Big)^{1/p} \tag{2.15}$$

With the squared error loss and ℓ_p-regularization, the total cost, being now the training loss, is

$$TrainLoss(\mathbf{w}) = Cost(h_{\mathbf{w}}) = \|\mathbf{y} - \mathbf{X}\mathbf{w}\|_2^2 + \lambda \ell_p(\mathbf{w}) \tag{2.16}$$

where $\lambda > 0$ is a regularization parameter. The optimal parameters follow as before from the minimization of the resulting loss:

$$\mathbf{w}^* = \arg\min_{\mathbf{w}} \|\mathbf{y} - \mathbf{X}\mathbf{w}\|_2^2 + \lambda \ell_p(\mathbf{w}) \tag{2.17}$$

Two special cases are of particular interest: $p = 2$ and $p = 1$. For $p = 2$, the resulting optimization problem is known as the **Ridge regression** or **Tikhonov regularization**:

$$\mathbf{w}^* = \arg\min_{\mathbf{w}} \|\mathbf{y} - \mathbf{X}\mathbf{w}\|_2^2 + \lambda\|\mathbf{w}\|_2^2 = \arg\min_{\mathbf{w}} \sum_{i=1}^{N} \Big(y^{(i)} - h_{\mathbf{w}}(\mathbf{x}^{(i)})\Big)^2 + \lambda \sum_j w_j^2 \tag{2.18}$$

For $p = 1$, the problem is known as the Least Absolute Shrinkage and Selection Operator (LASSO) regression:

$$\mathbf{w}^* = \arg\min_{\mathbf{w}} \|\mathbf{y} - \mathbf{X}\mathbf{w}\|_2^2 + \lambda\|\mathbf{w}\|_1 = \arg\min_{\mathbf{w}} \sum_{i=1}^{N} \Big(y^{(i)} - h_{\mathbf{w}}(\mathbf{x}^{(i)})\Big)^2 + \lambda \sum_j |w_j| \tag{2.19}$$

LASSO regression promotes sparse solutions. Fig. 2.4 explains this pictorially and illustrates the effect of both of these two regularization strategies. Note that we are minimizing the sum of two terms $Loss(\mathbf{w}) + \lambda Complexity(\mathbf{w})$, which is equivalent to minimizing $Loss(\mathbf{w})$ subject to the constraint that $Complexity(\mathbf{w}) \leq c$, for some constant c that is related to λ [18]. The

[2] In some cases, which are beyond the scope of this course, $\ell_{p,q}$ -regularization is used. For example, to impose 'structured sparsity', the coefficients $\mathbf{w}$ are structured into r groups $\mathbf{w} = [\mathbf{w}_{\mathcal{G}_1}^\top, \ldots, \mathbf{w}_{\mathcal{G}_r}^\top]^\top$ and different types of behaviour are promoted within and across the groups, which is expressed by adding an $\ell_{p,q}$-regularization term $\|\mathbf{w}\|_{p,q}^q$, where $\|\mathbf{w}\|_{p,q} = (\sum_i^r \|\mathbf{w}_{\mathcal{G}_i}\|_p^q)^{1/q}$ and $\mathcal{G}_i$ is the index set of the i-th group of coefficients. Particularly, $\ell_{2,1}$ regularization is popular in group sparse optimization. For more details, see [10].

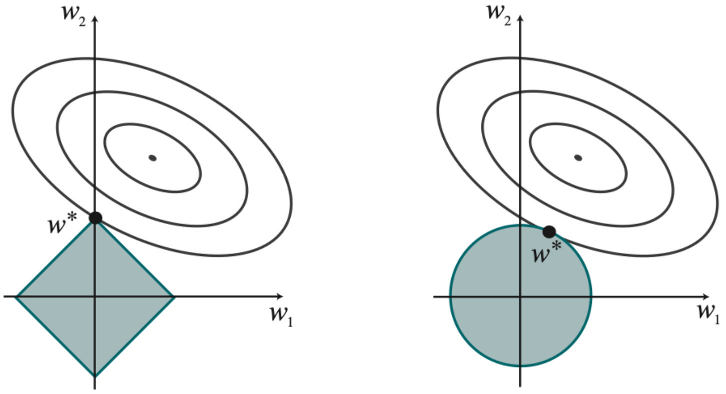


Figure 2.4: An illustration of the effects of ℓ_1 (left) and ℓ_2 (right) regularization. The concentric ovals are the contours of the loss function without regularization (minimum in the middle) and the shaded areas are the constraints for the corresponding regularization – the solution has to be within the shaded area. Observe that for ℓ_1 the solution will likely be on an axis (meaning other coefficients zero → sparse solution). Illustration from [18].

shaded regions in the figure (a diamond-shape for ℓ_1 and a circle for ℓ_2) represent the set of points that satisfy the constraint. For both LASSO and ridge regression $Loss(\mathbf{w})$ is squared-error loss, represented in the figure with concentric contours, with the minimum (smallest achievable loss) being the point in the middle. The optimal solution is where the shaded area touches the loss contour closest to the minimum. We can see that with ℓ_1 this will likely be along some of the axes, simply because the constraint area is pointy. It means that at least some of the components of this solution $\mathbf{w}^*$ will be zero, hence, the solution will be a **sparse** vector.

Why do we want to impose a sparsity constraint? Firstly, sparsity allows us to model naturally phenomena that are often appearing in real-world signals and images. Secondly, it also brings various practical (modelling and computational) advantages, as we will explain in the following.

In natural signals, sparsity refers to the phenomenon where only a small number of components or features in the signal are significant or carry essential information, while the majority are negligible or redundant. For example, in neuroscience, brain signals recorded from electrodes often exhibit sparsity because only certain neural events or activities are relevant to a particular cognitive process or behavior. In audio signals, such as speech or music, sparsity occurs because most sounds can be represented using a relatively small number of frequency components.

Incorporating a sparsity constraint in a computational model allows for more efficient representation and processing of the signals by focusing computational resources on the most relevant components while ignoring or compressing the redundant ones. When dealing with hight dimensional signal, sparsity provides also a way of mitigating the **curse of dimensionality**.[3]

In general, imposing sparse constraints in optimization and machine learning can be beneficial due to different reasons:

[3] The term *curse of dimensionality* refers to various challenges and phenomena that arise when working with high-dimensional data spaces. As the number of dimensions increases, the amount of data required to effectively cover the space increases exponentially. This leads to various issues, including an increased risk of overfitting, the need to gather and label large amounts of data, which can be costly and time-consuming, and processing of such data requires often huge (or even prohibitive) computation complexity.

- **Robustness to outliers**. Sparse regularization can enhance the robustness of models by filtering out noisy or irrelevant features. This can help in improving the model's performance in the presence of noise or outliers.

- **Dimensionality reduction**. Sparsity helps in reducing the dimensionality of the problem by selecting only a subset of the available features. This can mitigate the curse of dimensionality, making the models more tractable and less prone to overfitting, especially in high-dimensional spaces.

- **Feature selection**. Sparse regularization techniques naturally perform feature selection by encouraging many feature weights to be exactly zero. This automatic feature selection mechanism simplifies the model and can eliminate irrelevant or redundant features, leading to simpler and more efficient models.

- **Memory efficiency**. Sparse models require less memory storage compared to dense representations, which is especially important when dealing with large datasets. This makes sparse models more scalable and feasible for deployment in resource-constrained environments.

- **Improved generalization**. By focusing on the most informative features, sparse models often generalize better to unseen data. They are less likely to overfit to noise or irrelevant features present in the training data, leading to better performance on test or validation sets.

- **Improved interpretability**. Sparse regularization can also improve interpretability of the machine learning model by focusing on a subset of features that are deemed most relevant for the task at hand. For example, in LASSO regression, the non-zero coefficients indicate which features are most predictive of the outcome.

We have seen various advantages of sparse optimization. Tikhonov regularization shares some of these (it also mitigate overfitting and improves generalization) and has its distinctive advantages as well, especially in terms of stability, ease of implementation and computational efficiency. In particular, imposing Tikhonov (ℓ_2) regularization in optimization and machine learning can be beneficial mostly because of the following:

- **Stability in estimation**. The penalty term in Tikhonov regularization is proportional to the square of the parameter values, which provides more stability against Gaussian noise. Hence, Tikhonov regularization is less sensitive to small variations in the input data compared to LASSO.

- **Continuous solution**. Tikhonov regularization typically results in a solution with non-zero values for all parameters, albeit some may be very small. This continuous shrinkage of parameter values can be advantageous when the problem domain requires a smooth and continuous solution.

- **Simple implementation**. Tikhonov regularization is easy to implement and can be incorporated into various machine learning algorithms using standard techniques such as gradient descent or closed-form solutions. The additional computational cost is usually minimal compared to the benefits gained in terms of performance and stability.

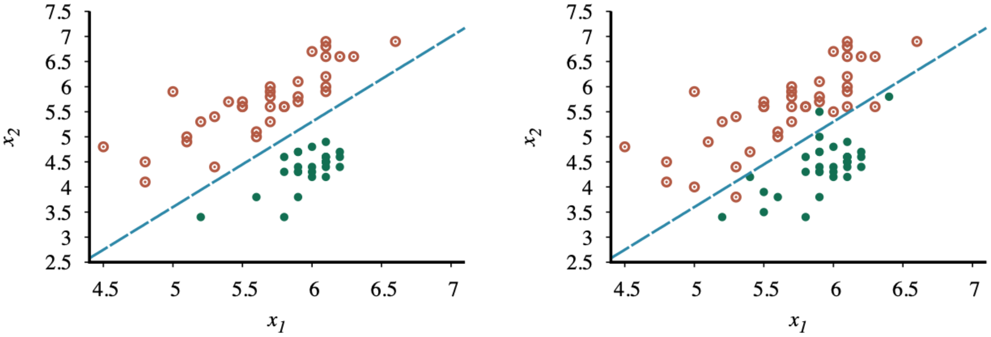


Figure 3.1: Left: A linearly separable dataset consisting of two classes and a linear decision boundary. x_1 and x_2 are two seismic parameters measured for earthquakes (orange circles) and explosions (green dots). Right: The same domain with more data, no longer linearly separable. This situation is common in real world. The image is taken from [18].

- **Efficient computation**. The optimization problem associated with Tikhonov regularization often has a closed-form solution or can be solved efficiently using standard optimization techniques. This makes Tikhonov regularization computationally less demanding compared to LASSO, especially for large-scale problems.

- **Theoretical interpretability**. The fact that the obtained solution often has a closed-form expression, allows for easier interpretation of model parameters and their significance. Tikhonov regularization has a well-understood theoretical foundation, especially in the context of linear regression.

- **Improved generalization**. Tikhonov regularization helps prevent overfitting by penalizing large parameter values. The regularization term encourages the model to capture the underlying structure of the data rather than fitting the noise present in the training set. By controlling overfitting, it enables improved generalization performance.

Moreover, Tikhonov regularization performs well in the cases where predictor variables are highly correlated. It tends to distribute the penalty evenly among correlated variables, whereas LASSO may arbitrarily select one of them and shrink the others to zero. The choice between Tikhonov regularization and LASSO depends on the specific characteristics of the dataset and the goals of the modeling task. LASSO may be preferred when feature selection or sparsity of the solution is desired, while Tikhonov regularization may be more suitable for problems where a continuous solution with less sensitivity to noise is required. Techniques such as **Elastic net regularization** combine the strengths of both Tikhonov regularization and LASSO, offering a more flexible regularization approach.

3 Linear Classification

Now we turn to the linear **classification framework**. As before, we are given training data, which consists of a set of examples $(\mathbf{x}, y)$, but y's are now some discrete class labels. We will focus first on the **binary classification** problem in which y can take only two values, 0 and 1. An example of this problem is shown in Fig. 3.1, where the input $\mathbf{x}$ consists of two components x_1

and x_2. The task of classification is to learn a model h that we call a **classifier** and that will for a new input $\mathbf{x}$ return the class label 0 or 1. A line (or surface, in higher dimensions) that separates the two classes is called a **decision boundary**. A linear decision boundary is called a **linear separator** and a dataset that can be ideally separated by at least one linear decision boundary is **linearly separable**.

3.1 Linear classification with a hard threshold

Consider the linearly separable case in Fig. 3.1 on the left. The depicted linear separator is:

$$x_2 = 1.7x_1 - 4.9 \quad \text{or} \quad -4.9 + 1.7x_1 - x_2 = 0 \tag{3.1}$$

We want to classify the explosions (green dots) with value 1. For these points $-4.9+1.7x_1-x_2 > 0$, while for earthquakes (orange circles) it holds $-4.9 + 1.7x_1 - x_2 < 0$. Introducing a dummy input $x_0 = 1$, we can write this compactly in vector form, with $\mathbf{w} = [-4.9, 1.7, -1]^\top$, as $\mathbf{w} \cdot \mathbf{x} > 0$ in one case and $\mathbf{w} \cdot \mathbf{x} < 0$ in the other. Thus, we can write the classification hypothesis as follows:

$$h_{\mathbf{w}}(\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{w} \cdot \mathbf{x} \geq 0 \\ 0 & \text{otherwise} \end{cases} \tag{3.2}$$

We can think of this classifier as the result of passing $\mathbf{w} \cdot \mathbf{x}$ through a threshold function:

$$h_{\mathbf{w}}(\mathbf{x}) = \mathit{Threshold}(\mathbf{w} \cdot \mathbf{x}), \quad \text{where} \quad \mathit{Threshold}(z) = \begin{cases} 1 & z \geq 0 \\ 0 & \text{otherwise} \end{cases} \tag{3.3}$$

The question is now how to choose the weights $\mathbf{w}$ to minimize the loss. For linear regression, we were able to determine the weights that minimize the squared error loss both analytically and using gradient descent algorithm. Here we cannot do that because with $h_{\mathbf{w}}(\mathbf{x})$ being a step function the gradient of the training loss is zero almost everywhere in the weight space, except at the transition $\mathbf{w} \cdot \mathbf{x} = 0$ where it is not defined.

However, it can be shown that the simple update rule called the **perceptron learning rule**

$$w_j \leftarrow w_j + \alpha(y - h_{\mathbf{w}}(\mathbf{x}))x_j \tag{3.4}$$

converges to the perfect linear separator (provided that data are linearly separable). Observe the following behaviour of this update:

- If the output is correct, i.e., $y = h_{\mathbf{w}}(x)$, the weights are not changed
- If $y = 1$ but $h_{\mathbf{w}}(\mathbf{x}) = 0$, then w_j is *increased* when x_j is positive and is *decreased* when x_j is negative. This is because we want to make $\mathbf{w} \cdot \mathbf{x}$ bigger so that $h_{\mathbf{w}}(x)$ outputs 1
- If $y = 0$ but $h_{\mathbf{w}}(\mathbf{x}) = 1$, then w_j is *decreased* when x_j is positive and is *increased* when x_j is negative. This way we make $\mathbf{w} \cdot \mathbf{x}$ smaller so that $h_{\mathbf{w}}(x)$ outputs 0

3.2 Soft classification with a logistic function

The hard nature of the classification threshold causes some problems. The fact that the hypothesis $h_{\mathbf{w}}(\mathbf{x})$ is not differentiable and is a discontinuous function of its inputs and its weights, makes learning with the perception rule very unpredictable [18]. Furthermore, the linear classifier always announces a "completely confident" prediction 0 or 1, while we often need more graduated predictions. These problems are alleviated by **softening** the threshold function: approximating a hard threshold with a continuous, differentiable function.

A widely used soft-threshold function is the **logistic** function, also called **sigmoid** function:

$$Logistic(z) = \frac{1}{1+e^{-z}} \tag{3.5}$$

Replacing the hard threshold with the logistic function, the classification hypothesis becomes

$$h_{\mathbf{w}}(\mathbf{x}) = Logistic(\mathbf{w} \cdot \mathbf{x}) = \frac{1}{1+e^{-\mathbf{w} \cdot \mathbf{x}}} \tag{3.6}$$

Logistic regression is the process of fitting the weights of this model to minimize loss on a data set [18]. Here we will show how the update rule is derived for the logistic regression under the L_2 loss. In the next note, we will address in more detail logistic regression, focusing on maximum likelihood estimation.

3.3 Least-square error logistic regression

We first derive the update rule for the logistic regression under L_2 loss. Let g denote the logistic function and g' its derivative. As we did for linear regression, we will use the chain rule for the derivatives: $\partial g(f(x))/\partial x = g'(f(x))(\partial f(x)/\partial x)$

We start again from a simplified case with *one* training example $(\mathbf{x}, y)$. The derivation is similar as for the linear regression but now $h_{\mathbf{w}}(\mathbf{x}) = g(\mathbf{w} \cdot \mathbf{x})$, so we have:

$$\begin{aligned}
\frac{\partial}{\partial w_j} TrainLoss(\mathbf{w}) &= \frac{\partial}{\partial w_j}(y - h_{\mathbf{w}}(\mathbf{x}))^2 \\
&= 2(y - h_{\mathbf{w}}(\mathbf{x}))\frac{\partial}{\partial w_j}(y - h_{\mathbf{w}}(\mathbf{x})) \\
&= -2(y - h_{\mathbf{w}}(\mathbf{x}))g'(\mathbf{w} \cdot \mathbf{x})\frac{\partial}{\partial w_j}(\mathbf{w} \cdot \mathbf{x}) \\
&= -2(y - h_{\mathbf{w}}(\mathbf{x}))g'(\mathbf{w} \cdot \mathbf{x})x_j
\end{aligned}$$

The derivative of the logistic function satisfies $g'(z) = g(z)(1 - g(z))$, so we have

$$g'(\mathbf{w} \cdot \mathbf{x}) = g(\mathbf{w} \cdot \mathbf{x})(1 - g(\mathbf{w} \cdot \mathbf{x})) = h_{\mathbf{w}}(\mathbf{x})(1 - h_{\mathbf{w}}(\mathbf{x})) \tag{3.7}$$

and the weight update for minimizing the loss is

$$w_j \leftarrow w_j + \alpha(y - h_{\mathbf{w}}(\mathbf{x}))h_{\mathbf{w}}(\mathbf{x})(1 - h_{\mathbf{w}}(\mathbf{x}))x_j \tag{3.8}$$

Note that this rule was derived for *one* training example (or for the stochastic gradient descent). You should know how to generalize it to the update rule based on N examples!

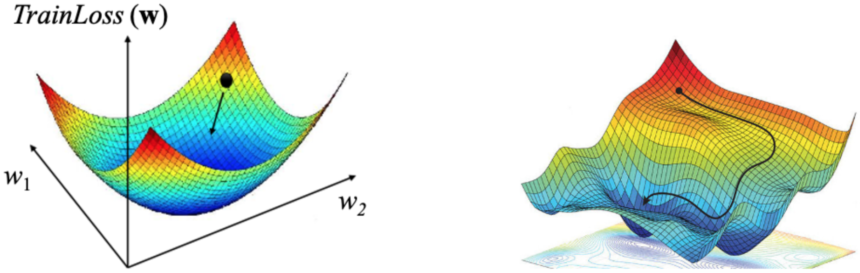


Figure 4.1: The optimization in ML aims at finding the weights that minimize the training loss. **Left**: A convex loss function (e.g., in the case of linear regression under the L_2 loss); **Right**: in general, the "loss landscape" is much more complex, non-convex with many local minima.

4 Optimization in machine learning

Our learning task is to determine the parameters (weights) $\mathbf{w}$ of a hypothesis $h_{\mathbf{w}}(\mathbf{x})$ that approximates the true, unknown function $y = f(\mathbf{x})$ that generated the data. We find the optimal $\mathbf{w}$ by minimizing the training loss

$$TrainLoss(\mathbf{w}) = \frac{1}{N}\sum_{i=1}^{N} L(\mathbf{x}^{(i)}, y^{(i)}, \mathbf{w}) \tag{4.1}$$

where $L(\mathbf{x}^{(i)}, y^{(i)}, \mathbf{w})$ is some loss function. Formally, we solve a minimization problem:

$$\mathbf{w}^* = \arg\min_{\mathbf{w}} TrainLoss(\mathbf{w}) \tag{4.2}$$

This is typically done by applying some variant of the **gradient descent** algorithm.

When the loss function is convex (like in the left of Fig. 4.1), the gradient descent, unless applied with a very wrong step size, is guaranteed to find the global optimum. In general, the training loss will have a much more complex landscape with many local minima, especially in deep learning. The illustration on the right of Fig. 4.1 gives some idea about such more complex training loss functions with only two weights, since we cannot visualize higher dimensional ones. The gradient descent algorithm will in these cases likely end up in a local optimum, but its variants, like the so-called **stochastic gradient descent** will in practice find good solutions even for very complex loss functions.

4.1 Gradient descent algorithm

We can minimize an arbitrary loss function by applying **iterative optimization**. The idea is to start with some $\mathbf{w}$ and keep on tweaking it to make the loss go down until we reach the minimum. To make the best "move" in the weight space at each step, we can use the gradient of the function. The gradient of a scalar-valued differentiable function of several variables is the vector field whose value at each point gives the **direction and the rate** of the **fastest increase** of the function at that point. Hence, moving along the direction of the **negative gradient** decreases the loss function. This iterative optimization procedure is called **gradient descent**. If the goal of the optimization procedure is to maximize an objective function, then we move in the direction

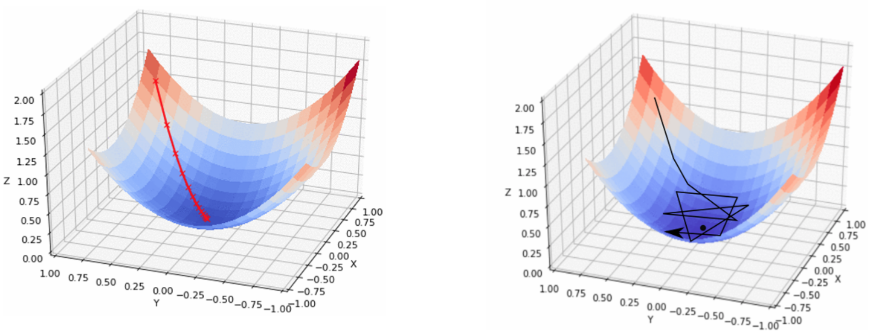


Figure 4.2: An illustartion of the gradient descent procedure with a good learning rate (left) and with a too large learning rate (right).

of the gradient to reach the maximum – this is known as the **gradient ascent** algorithm. We can use either of these two algorithms for the same problem if we can flip the objective function.

Thus, to minimize the training loss by the gradient descent, we will first initialize $\mathbf{w}$ to some value (say, all zeros) and then take a number of steps in the weight space, each time in the direction of the negative gradient. This means that we will each time subtract from $\mathbf{w}$ the gradient at that point $\nabla_{\mathbf{w}} TrainLoss(\mathbf{w})$ multiplied by some positive constant α that determines the step size. Concretely, the algorithm is as follows.

```
Algorithm: Gradient Descent (GD)
  initialize $\mathbf{w} = [0, \ldots, 0]$
  for iter $1, 2, \ldots$
  $\mathbf{w} \leftarrow \mathbf{w} - \alpha \nabla_{\mathbf{w}} TrainLoss(\mathbf{w})$
```

Observe that in each iteration *all* the training data are used. Therefore, each iteration here is an **epoch**, the term which refers to *using all the training data at once*. The step size $\alpha \geq 0$, also called the **learning rate**, specifies how aggressively we want to pursue the descent direction. The step size and the number of epochs are two hyperparameters of the optimization algorithm.

The loss minimization by the gradient descent procedure is illustrated in Fig. 4.2. In the case where the learning rate is well chosen, the algorithm steadily steps towards the minimum, while with a too large learning rate it will take too large sweeps, therefore "overshooting" and possibly even completely failing to reach the optimum (see also an illustration in Fig. 4.3). Generally, larger steps sizes are like driving fast: you can get faster convergence, but you might also get very unstable results and "crash". On the other hand, smaller step sizes give more stability , but the destination is reached more slowly. Note that when $\alpha = 0$, the weights don't change.

Some general strategies for choosing the learning rate include:

- set α such that update changes of $\mathbf{w}$ are about 0.1–1%
- decreasing: start with $\alpha = 1$ and then let $\alpha = 1/\sqrt{\#\text{updates made so far}}$
- more sophisticated – adapt α based on the data
    - e.g., *AdaGrad* and *Adam* optimizer

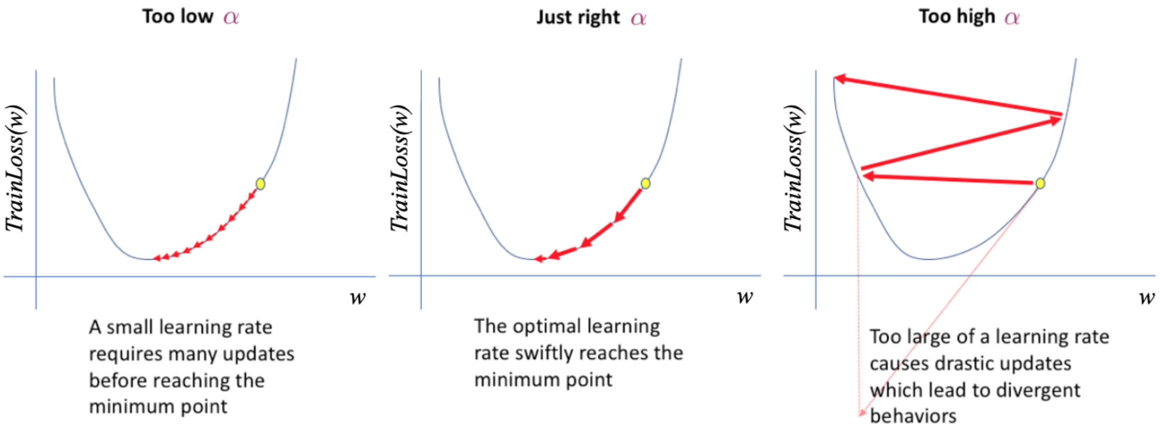

Too low α
Just right α
Too high α
TrainLoss(w)
TrainLoss(w)
TrainLoss(w)
w
w
w
A small learning rate requires many updates before reaching the minimum point
The optimal learning rate swiftly reaches the minimum point
Too large of a learning rate causes drastic updates which lead to divergent behaviors

Figure 4.3: The influence of the learning rate. Illustration Credit: E. Duchesnay.

4.2 Stochastic gradient descent

While gradient descent is a powerful general-purpose algorithm to optimize the training loss, one problem with it is that it's very slow. It is because it requires in each step the gradient of the full training loss, and the training loss is a sum over all the training data, see Eq (4.1). Thus, if we have millions of the training examples, each gradient computation requires going through those millions of examples, before we can make any small update of the weights.

The natural question is then – *Can we make progress before seeing all the data?* The answer to this question is – *yes*: rather than looping through all the training examples to compute a single gradient, we can make an update of the weights based on **each** example. This way the procedure will be much less stable and we will need many more steps, but each of these steps will be very cheap! This method is called the **stochastic gradient descent** (**SGD**).

Algorithm: Stochastic Gradient Descent (SGD)

```
• init $\mathbf{w} = [0, \dots, 0]$
• for iter $1, 2, \ldots$
    – For $(x, y) \in \mathcal{D}_{train}$:
        $\mathbf{w} \leftarrow \mathbf{w} - \alpha \nabla_{\mathbf{w}} L(\mathbf{x}, y, \mathbf{w})$
```

Each update now is not as good as with the (standard) gradient descent algorithm because we are only looking at one example at a time rather than taking all the examples. But the advantage is that each of these updates we compute very quickly so we can make many more steps this way.

There is a version between SGD and GD called **minibatch SGD**, where each update is made based on a **batch** of B examples. There are other variants of SGD. E.g., we can randomize the order in which we loop over the training data in each iteration. This is important, e.g., if in the training data we had all the positive examples first and the negative examples after that [4].

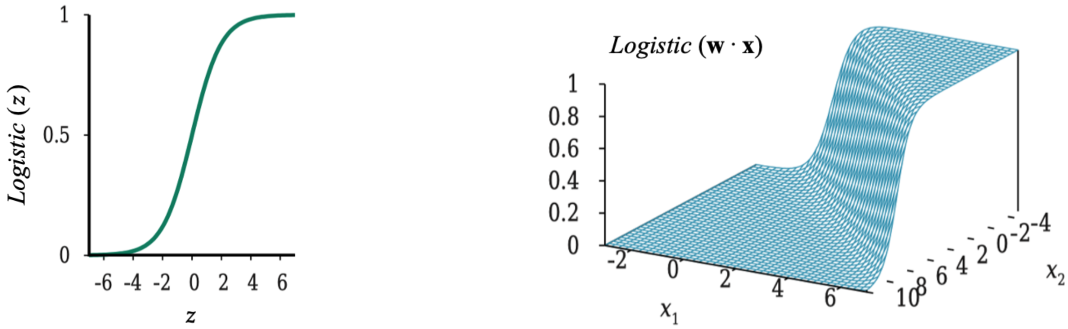


Figure 5.1: The logistic (sigmoid) function $Logistic(z) = 1/(1+e^{-z})$ and an example of a logistic regression hypothesis $h_{\mathbf{w}}(\mathbf{x}) = Logistic(\mathbf{w} \cdot \mathbf{x})$ for some weight vector $\mathbf{w} \in \mathbb{R}^2$. Figure from [18].

5 Logistic regression

Now we return to the task of binary classification. Previously we have seen that for some weight vector $\mathbf{w} \in \mathbb{R}^d$, the logistic regression hypothesis is

$$h_{\mathbf{w}}(\mathbf{x}) = Logistic(\mathbf{w} \cdot \mathbf{x}) = \frac{1}{1+e^{-\mathbf{w} \cdot \mathbf{x}}} = g(\mathbf{w} \cdot \mathbf{x}) \tag{5.1}$$

and we derived the update rule for the weights using (stochastic) gradient descent under the L_2 loss. Note that the loss function under the L_2 loss: $\sum_i (y^{(i)} - h_{\mathbf{w}}(\mathbf{x}^{(i)}))^2$ was convex for *linear* regression where $h_{\mathbf{w}}(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x}$. But with the nonlinear logistic regression hypothesis $h_{\mathbf{w}}(\mathbf{x})$ this loss is nonconvex with many local minima. So, although we could derive the update rule for logistic regression under the L_2 loss, the optimization with the gradient descent will be difficult (gradient descent may not find the global optimum – it may get stuck in a local minimum).

5.1 Logistic loss

For the reasons explained above, we will rarely use the logistic regression with square-error loss, but rather with the so-called **logistic loss**:

$$L(h_{\mathbf{w}}(\mathbf{x}), y) = \begin{cases} -\log(h_{\mathbf{w}}(\mathbf{x})) & \text{if } \ y = 1 \\ -\log(1 - h_{\mathbf{w}}(\mathbf{x})) & \text{if } \ y = 0 \end{cases} \tag{5.2}$$

which has nice properties for optimization and which can also be derived using the principle of **maximum likelihood estimation** as we will show next.

The logistic loss is illustrated schematically in Fig. 5.2. Note that $h_{\mathbf{w}}(\mathbf{x})$ from Eq (5.1) is between 0 and 1 and the logistic loss is a monotonic decreasing function with respect to the hypothesis when $y = 1$, and monotonically increasing when $y = 0$. Moreover, the loss is exactly zero when we are 100% confident while making the correct hypothesis and tends to infinity when we are 100% confident while making the wrong hypothesis.

For binary classification with $y \in \{0, 1\}$, the logistic loss function from Eq (5.2) can be written more compactly as:

$$L(h_{\mathbf{w}}(\mathbf{x}), y) = -y \log(h_{\mathbf{w}}(\mathbf{x})) - (1-y) \log(1 - h_{\mathbf{w}}(\mathbf{x})) \tag{5.3}$$

We will show now how we can derive this loss function using maximum-likelihood estimation.

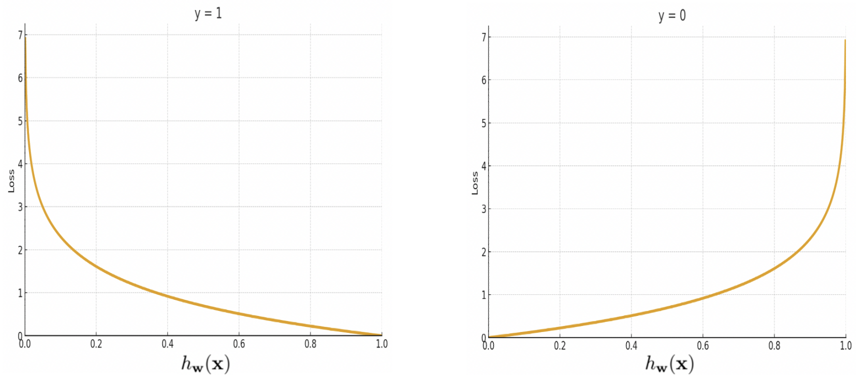


Figure 5.2: Logistic loss for $y = 1$ (left) and $y = 0$ (right).

5.2 Logistic regression under the maximum likelihood optimization

We already said earlier that the logistic regression $h_{\mathbf{w}}(\mathbf{x}) = Logistic(\mathbf{w} \cdot \mathbf{x})$ given in Eq (5.1) can be interpreted as the **probability** that $y = 1$. Let us now write this statement formally:

$$\begin{aligned} P(y = 1|\mathbf{x}, \mathbf{w}) &= h_{\mathbf{w}}(\mathbf{x}) \\ P(y = 0|\mathbf{x}, \mathbf{w}) &= 1 - h_{\mathbf{w}}(\mathbf{x}) \end{aligned} \tag{5.4}$$

Since y is always 1 or 0, we can write this more compactly as

$$P(y|\mathbf{x}, \mathbf{w}) = (h_{\mathbf{w}}(\mathbf{x}))^y (1 - h_{\mathbf{w}}(\mathbf{x}))^{(1-y)} \tag{5.5}$$

If the training examples were generated independently, the **likelihood** of the weights is:

$$\mathcal{L}(\mathbf{w}) = \prod_{i=1}^{N} P(y^{(i)}|\mathbf{x}^{(i)}, \mathbf{w}) = \prod_{i=1}^{N} \left(h_{\mathbf{w}}(\mathbf{x}^{(i)})\right)^{y^{(i)}} \left(1 - h_{\mathbf{w}}(\mathbf{x}^{(i)}\right)^{1-y^{(i)}} \tag{5.6}$$

In the maximum-likelihood philosophy, the optimal weights are those that are most likely given the data, i.e., those that yield the maximum likelihood. It is easier to maximize the logarithm of this likelihood and it will yield exactly the same solution as maximizing the likelihood itself, since the logarithm is a monotonic function. Therefore, we express first the **log likelihood**:

$$\ell(\mathbf{w}) = \log \mathcal{L}(\mathbf{w}) = \sum_{i=1}^{N} y^{(i)} \log h_{\mathbf{w}}(\mathbf{x}^{(i)}) \;+\; (1 - y^{(i)}) \log(1 - h_{\mathbf{w}}(\mathbf{x}^{(i)})$$

Observe that this is in fact the *logistic loss* from Eq (5.3) which was there written for one example only.

Now we can determine the update rule for the logistic regression by maximizing the log-likelihood of the weights. This is **the most common form of the logistic regression**.

Note that now $TrainLoss(\mathbf{w}) = -\ell(\mathbf{w})$, so we are applying the gradient descent algorithm to $-\ell(\mathbf{w})$, or equivalently, we are applying the **gradient ascent** to $\ell(\mathbf{w})$:

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha \nabla_{\mathbf{w}} \ell(\mathbf{w}) \tag{5.7}$$

We start with **one** training example $(\mathbf{x}, y)$:

$$\begin{aligned}
\frac{\partial}{\partial w_j}\ell(\mathbf{w}) &= \Big(y\frac{1}{g(\mathbf{w}\cdot\mathbf{x})} - (1-y)\frac{1}{1-g(\mathbf{w}\cdot\mathbf{x})}\Big)\frac{\partial}{\partial w_j}g(\mathbf{w}\cdot\mathbf{x}) \\
&= \Big(y\frac{1}{g(\mathbf{w}\cdot\mathbf{x})} - (1-y)\frac{1}{1-g(\mathbf{w}\cdot\mathbf{x})}\Big)g(\mathbf{w}\cdot\mathbf{x})(1-g(\mathbf{w}\cdot\mathbf{x}))\frac{\partial}{\partial w_j}(\mathbf{w}\cdot\mathbf{x}) \\
&= (y(1-g(\mathbf{w}\cdot\mathbf{x}))\ -\ (1-y)g(\mathbf{w}\cdot\mathbf{x}))x_j \\
&= (y - h_{\mathbf{w}}(\mathbf{x}))x_j
\end{aligned}$$

In the derivation above, we used the fact that $g'(z) = g(z)(1-g(z))$. Hence, the maximum-likelihood update rule for the logistic regression, with one example, is

$$w_j \leftarrow w_j + \alpha(y - h_{\mathbf{w}}(\mathbf{x}))x_j$$

and with all training examples

$$w_j \leftarrow w_j + \alpha\sum_{i=1}^{N}(y^{(i)} - h_{\mathbf{w}}(\mathbf{x}^{(i)}))x_j^{(i)}$$

Note that this update looks exactly the same as for the least-squares linear regression but, of course, $h_{\mathbf{w}}$ is different. We followed here the derivation from [14], where you can find more details about the logistic regression, including an alternative algorithm for the maximization of $\ell(\mathbf{w})$.

6 Multiclass linear classification

So far we considered only binary linear classification. Now we turn to a more general case where we can have more than two classes. For example, we want to predict the value of a handwritten digit or to classify newspaper articles into categories culture, science, sports, politics etc. We still want to define the decision boundaries based on **linear functions** of the input, i.e., based on linear combinations of input features. This task is called **multiclass linear classification**.

Let our input be a d-dimensional vector as before $\mathbf{x} \in \mathbb{R}^d$. We now have a weight vector $\mathbf{w}_y \in \mathbb{R}^d$ for each output class $y \in \{1, \dots, K\}$, and a new input $\mathbf{x}$ is classified based on the **scores** $\mathbf{w}_y \cdot \mathbf{x}$ that are computed for every class. We will put all the weight vectors together in one long vector $\mathbf{w} = [(\mathbf{w}_1)^\top, \dots, (\mathbf{w}_K)^\top]^\top \in \mathbb{R}^{Kd}$ and we'll denote the prediction same as before by $h_{\mathbf{w}}(\mathbf{x})$.

6.1 Multiclass perceptron

Given the setup above, the prediction rule "the highest score wins":

$$h_{\mathbf{w}}(\mathbf{x}) = \arg\max_i\ \mathbf{w}_i\cdot\mathbf{x} \tag{6.1}$$

extends directly the linear binary classification with a **hard threshold** to multiple classes. This classification approach is illustrated in Fig. 6.1 and is often referred to as the **multiclass perceptron**. The scores $z_i = \mathbf{w}_i \cdot \mathbf{x}$ are also called **activations** (this is the terminology that we will use commonly with neural networks).

Often it is convenient to represent multiclass classification with **one-hot encoding**. This means that the target output (the correct classification result) is represented as a vector $\mathbf{t}$ with

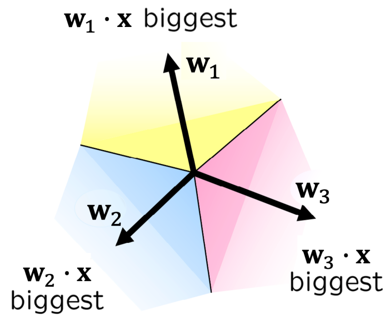

$\mathbf{w}_1 \cdot \mathbf{x}$ biggest
$\mathbf{w}_1$
$\mathbf{w}_2$
$\mathbf{w}_3$
$\mathbf{w}_2 \cdot \mathbf{x}$ biggest
$\mathbf{w}_3 \cdot \mathbf{x}$ biggest

Figure 6.1: The concept of multiclass linear classification illustrated on a case with three classes. The input data point $\mathbf{x}$ is assigned to the class that gives the biggest score. Credit: D. Klein & P. Abbeel [11].

all zeroes except one entry "1", which indicates the correct class. For example, if the correct class out of K possibles classes is the k-th class, the one-hot encoded target output is

$$\mathbf{t} = \underbrace{[0,\ldots,0,1,0,\ldots,0]^\top}_{\text{entry } k \text{ is one}} \in \mathbb{R}^K$$

We can represent the multiclass perceptron prediction with one-hot encoding as follows. Let $\mathbf{o} \in \mathbb{R}^K$ be the one-hot encoded output vector. Then for the multiclass perceptron

$$o_k = \begin{cases} 1 & \text{if } k = \arg\max_i \ \mathbf{w}_i \cdot \mathbf{x} \\ 0 & \text{otherwise} \end{cases} \tag{6.2}$$

6.2 Multiclass logistic regression and the softmax rule

The question now is how to turn the *hard* multiclass classification that we defined above into a soft one. In the binary case we replaced the hard threshold with the sigmoid function and we called the resulting model logistic regression. The nice property of the sigmoid (logistic) function was that it provided a probabilistic interpretation of the output as the probability of belonging to class "1".

We can equivalently turn the scores for multiclass classification into probabilities for belonging to the corresponding classes by using the **softmax** rule:

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{k=1}^K e^{z_k}}, \quad i = 1,\ldots,K \tag{6.3}$$

The original activations z_i are transformed this way to **softmax activations**. The resulting approach is **multiclass logistic regression** (also called **multinomial logistic regression** or **softmax regression**) where the hypothesis is defined as:

$$h_\mathbf{w}(\mathbf{x}) = \begin{bmatrix} P(y=1|\mathbf{x},\mathbf{w}) \\ P(y=2|\mathbf{x},\mathbf{w}) \\ \vdots \\ P(y=K|\mathbf{x},\mathbf{w}) \end{bmatrix} = \frac{1}{\sum_{k=1}^K e^{\mathbf{w}_k \cdot \mathbf{x}}} \begin{bmatrix} e^{\mathbf{w}_1 \cdot \mathbf{x}} \\ e^{\mathbf{w}_2 \cdot \mathbf{x}} \\ \vdots \\ e^{\mathbf{w}_K \cdot \mathbf{x}} \end{bmatrix} \tag{6.4}$$

where $\mathbf{w} = [(\mathbf{w}_1)^\top, \dots, (\mathbf{w}_K)^\top]^\top$. If we denote the output vector by $\mathbf{o} = h_\mathbf{w}(\mathbf{x})$ we can write

$$o_k = softmax(\mathbf{w}_k \cdot \mathbf{x}), \quad k = 1, \dots, K \tag{6.5}$$

Note how this "softens" the hard classification rule in Eq (6.2).

6.3 Multiclass perceptron learning rule

Let us now see how we learn the weights from the training data for the two above presented multiclass classification methods. Remember the perceptron learning rule for binary classification with $y \in \{0, 1\}$, which can be written in a vector form as $\mathbf{w} \leftarrow \mathbf{w} + \alpha(y - h_\mathbf{w}(\mathbf{x}))\mathbf{x}$. It did nothing if the output was correct, and otherwise the weights were either increased or decreased by $\alpha\mathbf{x}$ to nudge them in the right direction (increasing if $y = 1$ and $h_\mathbf{w}(\mathbf{x}) = 0$ and decreasing in $y = 0$ and $h_\mathbf{w}(\mathbf{x}) = 1$). This is simply extended to the case with multiple classes as follows:

- If $h_\mathbf{w}(\mathbf{x}) = y$ do nothing
- If $h_\mathbf{w}(\mathbf{x}) \neq y$ update the weights for the true class y and for the predicted class $y^* = h_\mathbf{w}(\mathbf{x})$
  - Update the **correct** class vector as $\mathbf{w}_y \leftarrow \mathbf{w}_y + \alpha\mathbf{x}$
  - Update the **wrong** class vector as $\mathbf{w}_{y^*} \leftarrow \mathbf{w}_{y^*} - \alpha\mathbf{x}$
  - Do **not** change the weights of any other class

6.4 Optimization for multiclass logistic regression

For multiclass logistic regression we optimize the weights similarly as we did with the logistic regression in the binary case: by maximizing the likelihood of the weights given the training data:

$$\mathbf{w}^* = \arg\max_{\mathbf{w}} \mathcal{L}(\mathbf{w})$$

Assuming as before that the training examples were generated independently, the likelihood is:

$$\mathcal{L}(\mathbf{w}) = \prod_{i=1}^{N} P(y^{(i)}|\mathbf{x}^{(i)}, \underbrace{\mathbf{w}_1, \dots, \mathbf{w}_K}_{\mathbf{w}}) = \prod_{i=1}^{N} P(y^{(i)}|\mathbf{x}^{(i)}, \mathbf{w}) \tag{6.6}$$

where

$$P(y^{(i)}|\mathbf{x}^{(i)}, \mathbf{w}) = \frac{e^{\mathbf{w}_{y^{(i)}} \cdot \mathbf{x}^{(i)}}}{\sum\limits_{y} e^{\mathbf{w}_{y^{(i)}} \cdot \mathbf{x}^{(i)}}}$$

Again, as was the case with the binary logistic regression, we will perform the desired optimization easier on =the logarithm of the likelihood:

$$\ell(\mathbf{w}) = \log \mathcal{L}(\mathbf{w}) = \sum_{i=1}^{N} \log P(y^{(i)}|\mathbf{x}^{(i)}, \mathbf{w}) \tag{6.7}$$

The optimization objective is now equivalently expressed as maximizing the likelihood or minimizing the negative log-likelihood, i.e., the training loss is now the negative log-likelihood and we have that:

$$\mathbf{w}^* = \arg\min_{\mathbf{w}} \; -\ell(\mathbf{w}) = \arg\min_{\mathbf{w}} \; -\sum_{i=1}^{N} \log P(y^{(i)}|\mathbf{x}^{(i)}, \mathbf{w}) \tag{6.8}$$

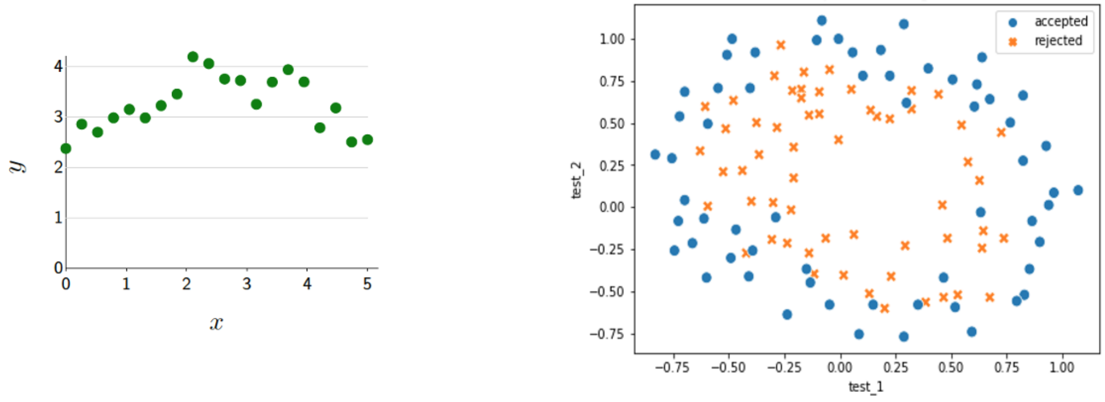


Figure 6.2: Examples of more complex data where a non-linear predictor is needed for regression (left) or classification (right). Figures from [4].

Thus the update rule with the stochastic gradient descent is

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha \sum_{i=1}^{N} \nabla \log P(y^{(i)}|\mathbf{x}^{(i)}, \mathbf{w}) \tag{6.9}$$

It is possible to express this update rule analytically and to show that it is a direct extension of the update rule for the weights in the case of binary logistic regression. Let $\mathbf{t}^{(i)}$ and $\mathbf{o}^{(i)}$ denote **one-hot** encoded target and predicted output for the ith example. The update rule for multiclass logistic regression is:

$$\mathbf{w}_k \leftarrow \mathbf{w}_k + \alpha \sum_{i=1}^{N} (t_k^{(i)} - o_k^{(i)}))\mathbf{x}^{(i)}, \qquad k = 1, \ldots, K$$

The log-likelihood loss in the logistic regression, which is often called the **logistic loss** or just **log loss**) is in the literature often called also **cross-entropy loss** (although strictly speaking the logistic loss is an approximation of the true cross-entropy loss, which would require the actual (unknown) distribution of the examples, and we are approximating this unknown distribution by its samples contained in the training set). Nevertheless, these terms are now often used interchangeably and the term cross-entropy loss is common in the machine learning community.

7 Linear predictors with nonlinear features

So far we were dealing with linear regression and linear classification. However, in real life data are often more complex and a linear predictor may not be a satisfactory fit (see examples in Fig. 6.2). In this case, we can turn to more advanced models like decision trees and neural networks (that we will study next). Before doing so, let's see how we can tackle these tasks still with the machinery of linear predictors but then feeding them with nonlinear features. You will see that in some cases this can work pretty well!

The main idea is to extract a vector of **nonlinear features** $\phi(\mathbf{x}) \in \mathbb{R}^n$ from the input $\mathbf{x} \in \mathbb{R}^d$ and to feed these nonlinear features to a linear predictor. The prediction will be non-linear in $\mathbf{x}$! With appropriately selected nonlinear features we can fit the data as illustrated in Fig. 7.1.

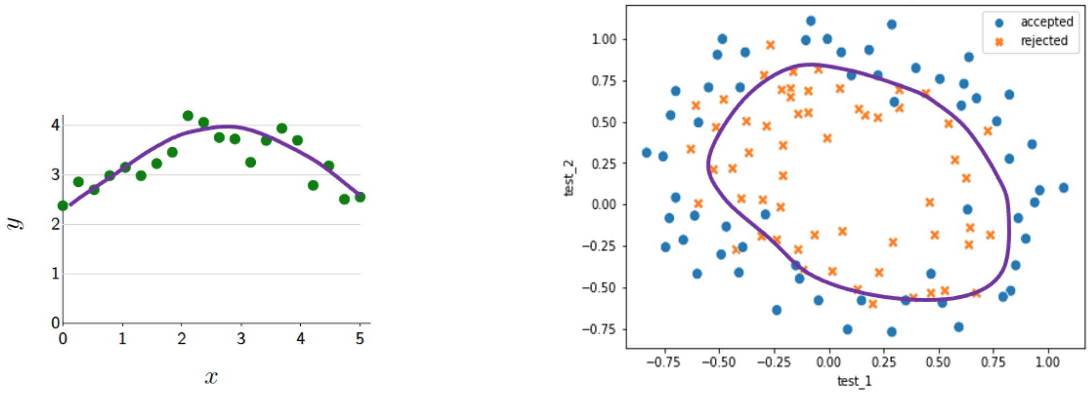
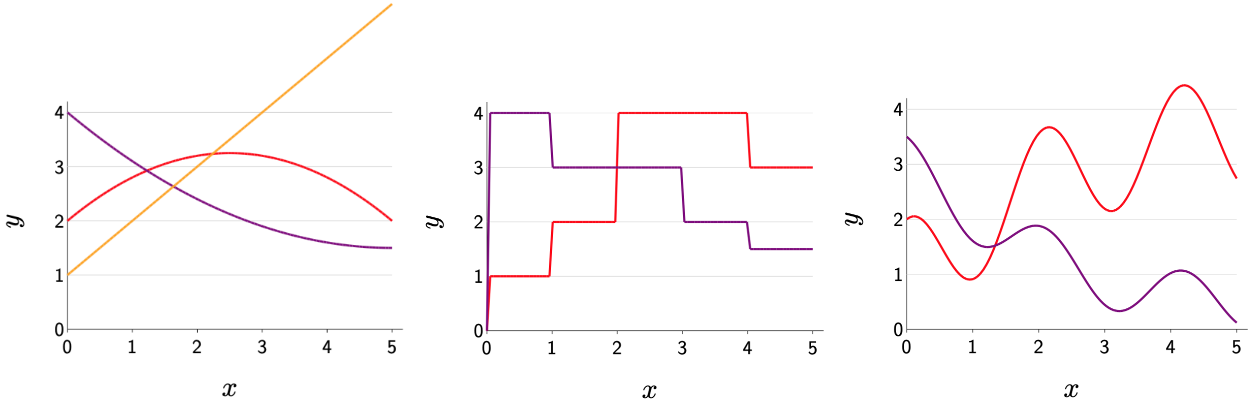


Figure 7.1: By extracting nonlinear features $\phi(\mathbf{x})$ from the input $\mathbf{x}$ and feeding those to linear regression as $h_{\mathbf{w}}(\mathbf{x}) = \phi(\mathbf{x}) \cdot \mathbf{w}$ or to logistic regression as $h_{\mathbf{w}}(\mathbf{x}) = \mathit{Logistic}(\phi(\mathbf{x}) \cdot \mathbf{w})$, we obtain predictions that are nonlinear in $\mathbf{x}$. Illustrations from [4].



(a) (b) (c)

Figure 7.2: Examples of predictors with (a) quadratic features; (b) piece-wise constant features and (c) features with periodicity structure. Illustrations from [4].

7.1 Regression with nonlinear features

We generalize linear regression $\mathbf{x} \cdot \mathbf{w}$ by replacing the "raw" input $\mathbf{x}$ by some feature vector $\phi(\mathbf{x})$. The resulting predictor is

$$h_{\mathbf{w}}(\mathbf{x}) = \phi(\mathbf{x}) \cdot \mathbf{w} \tag{7.1}$$

The feature vector $\phi(\mathbf{x})$ can be arbitrary. We will illustrate the use of nonlinear features for univariate regression only, i.e., for the case where the input is scalar x from which we will construct a n-dimensional feature vector $\phi(x)$. Fig. 7.2 illustrates three classes of nonlinear predictors that are obtained with different feature vectors.

Note that with $\phi(x) = [1, x]^\top$ the predictor in Eq (7.1) would simply be univariate linear regression (the dummy variable $x_0 = 1$ allows us to include the intercept term w_0 in the vector $\mathbf{w}$). Now, if we construct a nonlinear feature vector by adding a quadratic term x^2:

$$\phi(x) = [1, x, x^2]^\top$$

we obtain **quadratic predictors** illustrated in Fig. 7.1(a). The different curves there correspond

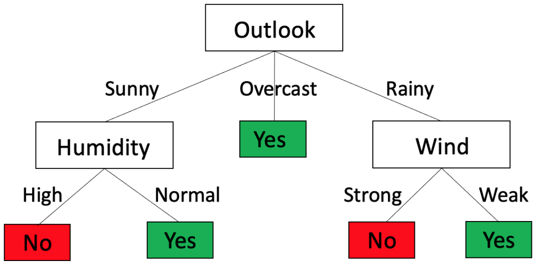


Figure 8.1: An example of a simple decision tree.

to different weight vectors $\mathbf{w}$. The line corresponds to $\mathbf{w} = [1, 1, 0]^\top$ which sets the quadratic term to zero.

The **piecewise constant** predictors in Fig. 7.1(b) are obtained with feature extractors that divide the input space into regions and allow the predicted value of each region to vary independently. Specifically, each component of the feature vector corresponds to one region, e.g., (0, 1], and is 1 if x lies in that region and 0 otherwise:

$$\phi(x) = [\mathbf{1}[0 < x \leq 1], \mathbf{1}[1 < x \leq 2], \mathbf{1}[2 < x \leq 3], \mathbf{1}[3 < x \leq 4], \mathbf{1}[4 < x \leq 5]]^\top$$

Assuming the regions are disjoint, the weight associated with a component/region is exactly the predicted value. E.g., the predictor shown in red corresponds to $\mathbf{w} = [1, 2, 4, 4, 3]^\top$. As we make the regions smaller, we get more features, and the expressiveness of our hypothesis class increases. In the limit, we can essentially capture any predictor we want.

This sounds very nice but think what happens if x were not a scalar, but a d-dimensional vector $\mathbf{x}$? Then if each component gets broken up into B bins, then there will be B d features! For each feature, we need to fit its weight, and there will ingenerally be too few examples to fit all the features.

Fig. 7.1(c) shows yet another family of the predictors, and these have some **periodicity structure**. In particular, these were obtained with

$$\phi(x) = [1, x, x^2, \cos(3x)]^\top$$

We showed three examples but there is an unboundedly large design space of possible feature extractors. In practice, the choice of features is informed by the prediction task that we wish to solve (either prior knowledge or preliminary data exploration) [4].

8 Decision trees

Decision trees underlay many of today's most successful learning approaches. They are able to learn complex, **nonlinear** relationships between variables, using a series of simple, **intuitive** decision rules: start with one test, and depending on its outcome decide what the next test will be. This process continues until a decision is reached.

8.1 Interpretation and basic types of decision trees

Fig. 8.1 shows a simple example of a decision tree, which is often given in the introductory materials on this topic. Here, the decision on "*Should I play tennis today?*" is based on the values of three attributes: outlook from the window (with three possible outcomes sunny, overcast or rainy), humidity (which can be high or normal) and wind (which can be strong or weak). The training will be done based on some training set that contains examples with different combinations of the attribute values and "play" ("yes") or "not play" ("no") decisions for each of them.

Formally, like in any supervised ML approach, a decision tree is learned from examples $(\mathbf{x}, y) \in \mathcal{D}_{train}$, where $\mathbf{x}$ are the values of some **features** (or **attributes**) $\mathbf{X}$ and y is the output label. The structure and the elements of a decision tree have the following interpretation:

- **Root and internal nodes** test a feature X_i. In our tree: $X_1 = Outlook$, $X_2 = Humidity$, $X_3 = Wind$
- **Branching** is determined by the feature value E.g. $x_3 = wind \in \{strong, weak\}$
- **Leaf nodes** are outputs (decisions, predictions)

We will interpret the decision as a prediction – we use the decision tree as a *predictive model.* When a nominal (categorical) variable is predicted, the tree is called a classification tree (like the tree in Fig. 8.1). We can also use decision trees to predict a numerical variable, this is then a regression tree. In principle, the tree can also predict multiple variables at once (or, equivalently, a tuple-valued variable); such trees are sometimes called multi-target trees. Classification trees that do not merely predict a class, but define a conditional probability for each class given the input, are called probability estimation trees. In summary, the output of a decision tree can be of different types, including:

- **numerical** (our model is then a **regression tree**)
- **categorical** (we call it then a **classification tree**)
- **tuple-valued** (in the so-called **multi-target trees**)
- $P(y|\mathbf{x})$ (we call these models **probability estimation trees**)

The decision trees can be used in many settings and we can define other sub-categories of these models next to those that are listed above. Notably, as it repeatedly divides a data set into subsets, a tree implicitly defines a hierarchical clustering. Trees learned for this purpose are called **clustering trees**. The difference between a hierarchical clustering defined by a clustering tree, and one defined by other clustering algorithms, is that each cluster in a clustering tree is defined precisely by a set of test outcomes. **Density estimation trees** partition the dataset into regions of high and low density, and as such can be used to describe the joint probability distribution of the data.

The decision trees are widely used because they are **easy to understand and interpret** and because they **require little or no data preparation**. Moreover, they provide basis to some of the best performing ML models today: **Random forests** or random decision forests is an ensemble learning method for classification, regression and other tasks that operates by constructing a multitude of decision trees at training time. In classification tasks, the output of the random forest is the class selected by most trees. For regression tasks, the mean or average prediction of the individual trees is returned [8].

8.2 Case study: Restaurant domain

We will study decision trees on a use case *Restaurant domain* from [18]: the problem of deciding whether to wait for a table at a restaurant based on the following attributes:

1. *Alternate (Alt)*: Is there a suitable alternative restaurant nearby?
2. *Bar (Bar)*: Is there a comfortable bar area in the restaurant, where I can wait?
3. *Fri/Sat (Fri)*: True on Fridays/Saturdays
4. *Hungry (Hun)*: Are we hungry?
5. *Patrons (Pat)*: How many people are in the restaurant (*None*, *Some* or *Full*)
6. *Price (Price)*: the restaurant's price range ($, $$, $$$)
7. *Raining (Rain)*: Is it raining outside?
8. *Reservation (Res)*: Did we make a reservation?
9. *Type (Type)*: the kind of restaurant (French, Italian, Thai or burger)
10. *WaitEstimate (Est)*: the wait time estimated by the host (0-10, 10-30, 30-60, or>60 min)

The training set consists of 12 examples that are given in the following table [18]:

	Input Attributes										*Output*
Example	*Alt*	*Bar*	*Fri*	*Hun*	*Pat*	*Price*	*Rain*	*Res*	*Type*	*Est*	*WillWait*
1	*T*	*F*	*F*	*T*	*Some*	*$$$*	*F*	*T*	*French*	*0–10*	*T*
2	*T*	*F*	*F*	*T*	*Full*	*$*	*F*	*F*	*Thai*	*30–60*	*F*
3	*F*	*T*	*F*	*F*	*Some*	*$*	*F*	*F*	*Burger*	*0–10*	*T*
4	*T*	*F*	*T*	*T*	*Full*	*$*	*F*	*F*	*Thai*	*10–30*	*T*
5	*T*	*F*	*T*	*F*	*Full*	*$$$*	*F*	*T*	*French*	*>60*	*F*
6	*F*	*T*	*F*	*T*	*Some*	*$$*	*T*	*T*	*Italian*	*0–10*	*T*
7	*F*	*T*	*F*	*F*	*None*	*$*	*T*	*F*	*Burger*	*0–10*	*F*
8	*F*	*F*	*F*	*T*	*Some*	*$$*	*T*	*T*	*Thai*	*0–10*	*T*
9	*F*	*T*	*T*	*F*	*Full*	*$*	*T*	*F*	*Burger*	*>60*	*F*
10	*T*	*T*	*T*	*T*	*Full*	*$$$*	*F*	*T*	*Italian*	*10–30*	*F*
11	*F*	*F*	*F*	*F*	*None*	*$*	*F*	*F*	*Thai*	*0–10*	*F*
12	*T*	*T*	*T*	*T*	*Full*	*$*	*F*	*F*	*Burger*	*30–60*	*T*

Each raw in this table is an example $(\mathbf{x}^{(i)}, y^{(i)})$, where $\mathbf{x}^{(i)}$ contains values of the 10 attributes and the output $y^{(i)}$ is true (T) or false (F). One possible tree that correctly represents these examples is shown in Fig. 8.2. This is also the 'ground truth' tree, which represents exactly the actual decision function that was used by the person (Stuart Russel) who gave these examples [18].

We suppose our AI system doesn't have access to this 'true' tree but needs to *learn* a good decision tree from the supplied examples. We will then compare the learned tree to this true one to see how well our system learned a good decision model.

Note that there are $2^6 \times 3^2 \times 4^2 = 9216$ combinations for the attributes in this problem while we are given only 12. This is the essence of *induction*: make the best guess for many missing output values given only the evidence of few examples.

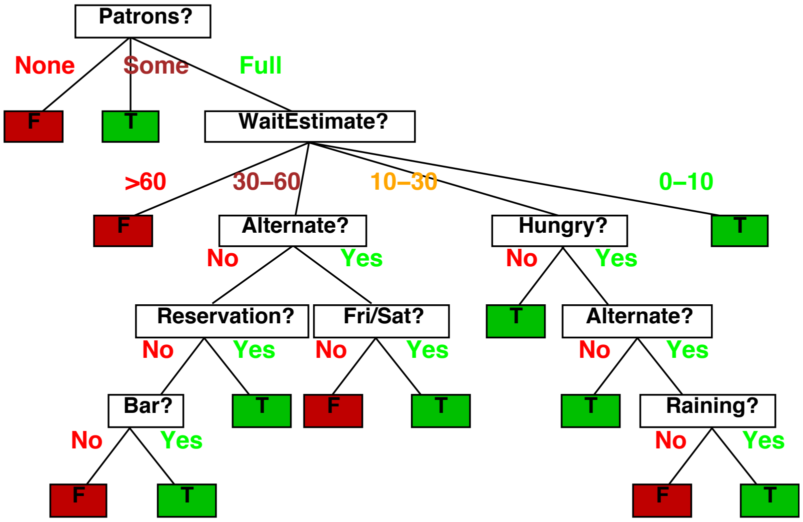
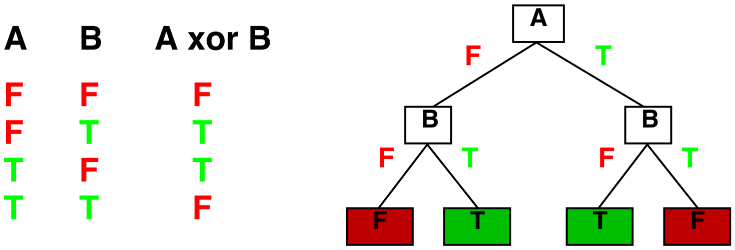


Figure 8.2: Ground truth tree for the restaurant problem.



A	B	A xor B
F	F	F
F	T	T
T	F	T
T	T	F

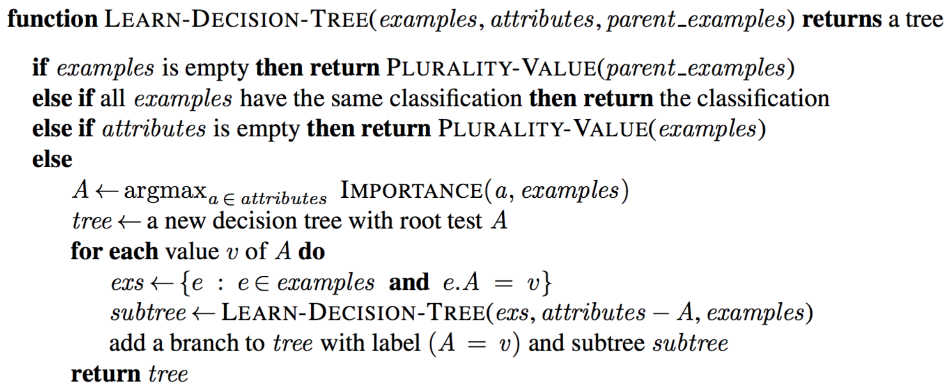
Figure 8.3: Truth table for the logical operation ‘XOR’ and the corresponding decision tree.

8.3 Expressiveness of decision trees

Decision trees can express any function of the input attributes. For Boolean functions, each row in the truth table is one path to the leaf (see an illustration in Fig. 8.3). For many problems, the decision tree format yields a nice, concise, understandable result. But some functions cannot be represented concisely. For example, the majority function, which returns true if and only if more than half of the inputs are true, requires an exponentially large decision tree [18].

In general, we will increase the expressiveness of the tree by using more attributes. With more attributes, the decision tree has more potential splitting points to choose from. This allows to model more complex relationships in the data. However, the number of possible trees grows combinatorially. For example, for a Boolean function with n attributes the truth table has 2^n rows, which means there are 2^{2n} distinct truth tables. With only 10 Boolean attributes there are 10^{308} possible trees. Finding the best hypothesis in a such a huge hypothesis space becomes very difficult. Moreover, the risk of overfitting increases.

In summary, more attributes generally increase the tree’s expressiveness but at the cost of higher risks of overfitting and computational complexity. Conversely, fewer attributes lead to simpler, potentially less expressive models but with reduced risks of overfitting and improved interpretability. Effective feature selection and regularization techniques are essential to balance the expressiveness and generalization of decision trees.



```
function LEARN-DECISION-TREE($examples$, $attributes$, $parent\_examples$) returns a tree
  if $examples$ is empty then return PLURALITY-VALUE($parent\_examples$)
  else if all $examples$ have the same classification then return the classification
  else if $attributes$ is empty then return PLURALITY-VALUE($examples$)
  else
    $A \leftarrow \mathrm{argmax}_{a \in attributes}$ IMPORTANCE($a$, $examples$)
    $tree \leftarrow$ a new decision tree with root test $A$
    for each value $v$ of $A$ do
      $exs \leftarrow \{e : e \in examples$ and $e.A = v\}$
      $subtree \leftarrow$ LEARN-DECISION-TREE($exs$, $attributes - A$, $examples$)
      add a branch to $tree$ with label $(A = v)$ and subtree $subtree$
    return $tree$
```

Figure 8.4: Pseudo-code of the decision tree learning algorithm [18].

8.4 Decision tree learning

In general, the goal of decision tree learning is to find a tree that is consistent with the provided examples and is as small as possible. Unfortunately, it is intractable to find a guaranteed smallest consistent tree [18]. Therefore we resort to a greedy approach and it turns out that with some simple heuristics, we can efficiently find a tree that is close to the smallest consistent tree.

8.4.1 Decision tree learning algorithm

Here we describe a greedy algorithm known as the decision tree learning algorithm. Its idea is to choose the "most significant" attribute as the root and repeat this recursively for each subtree. We start with the whole training set and an empty decision tree. Then pick the feature that gives the best split. We split on that feature and repeat the process on the sub-partitions.

Fig. 8.4 gives a pseudo-code for the decision tree learning algorithm. The function IMPORTANCE measures the importance of attributes (as explained next). The PLURALITY-VALUE function selects the most common output value among a set of examples, breaking ties randomly.

8.4.2 Choosing important attributes based on the information gain

The key question is how to choose the most *important* attribute in each phase of the decision tree learning. The general idea is that a good (i.e., important) attribute is one that makes the most difference to the classification of an example. With Boolean attributes, this means an attribute that splits well the examples into subsets that are (ideally)"all positive" or "all negative".

Common techniques and criteria used to identify important attributes include **information gain** (which is equivalent to **entropy reduction**) and the **Gini index** (a measure for the "impurity" of a dataset). We will focus our attention to the first criterion.

Information answers questions – the more clueless we are about the answer initially, the more information is contained in the answer. In information theory, *entropy* is a measure of the uncertainty of a random variable, the "expected surprisal". The more information, the less entropy.

Let us denote the **entropy of an information source** with n outcomes occurring with probabilities $P_1, \ldots P_n$ as

$$H(\langle P_1, \ldots, P_n \rangle) = \sum_{i=1}^{n} -P_i \log_2 P_i$$

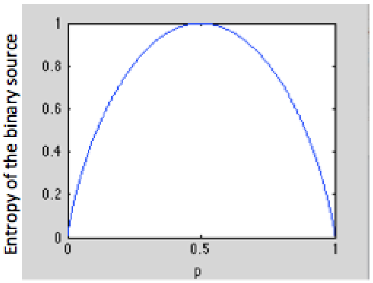


Figure 8.5: Entropy of a binary information source $H(\langle p, 1-p\rangle)$.

This is information in an answer when the prior is $\langle P_1, \dots, P_n\rangle$. For the binary source we have:

$$H(\langle p, 1-p\rangle) = -p\log_2(p) - (1-p)\log_2(1-p)$$

The corresponding plot is shown in Fig. 8.5. Note that $H(\langle 1, 1\rangle) = 1$, i.e., 1 **bit** is the information entropy of a random binary variable that takes values 0 and 1 with equal probability. Or, put in other words, 1 bit is an answer to Boolean question with prior $\langle 0.5, 0.5\rangle$.

Suppose we have p positive and n negative examples at the root. Then

$$B(\frac{p}{p+n}) = H(\langle\frac{p}{p+n}, \frac{n}{p+n}\rangle)$$

bits are needed to classify a new example. For the restaurant use case, out of the 12 training examples we had six positive and six negative ones, so $p = n = 6$, and thus we need exactly 1 bit of information to classify a new example. The result of a test on an attribute A will give us some information, thus reducing the overall entropy by some amount. We can measure this reduction by looking at the entropy remaining after the attribute test.

An attribute A with d distinct values divides the training set E into subsets $E_1, \dots, E_d$ each of which (we hope) needs less information to complete the classification. Let E_i have p_i positive and n_i negative examples. This means that if we go along that branch, we will need

$$H(\langle\frac{p_i}{p_i+n_i}, \frac{n_i}{p_i+n_i}\rangle)$$

bits of information to answer the question. A randomly chosen example from the training set has the kth value for the attribute (i.e., is in E_k with probability $(p_k+n_k)/(p+n)$). Thus, the expected number of bits (EBS) needed if A is at the root is

$$EBS(A) = \sum_i \ \frac{p_i+n_i}{p+n} \ H\Big(\Big\langle\frac{p_i}{p_i+n_i}, \frac{n_i}{p_i+n_i}\Big\rangle\Big)$$

This is also the expected entropy remaining after testing attribute A. The **information gain** from the attribute test on A is the expected **reduction in entropy**:

$$Gain(A) = B(\frac{p}{p+n}) - EBS(A)$$

Take for example the attribute *Patrons* from the *Restaurant* domain. It has three possible outcomes and the splitting of the training examples based on this attribute (as can be read from

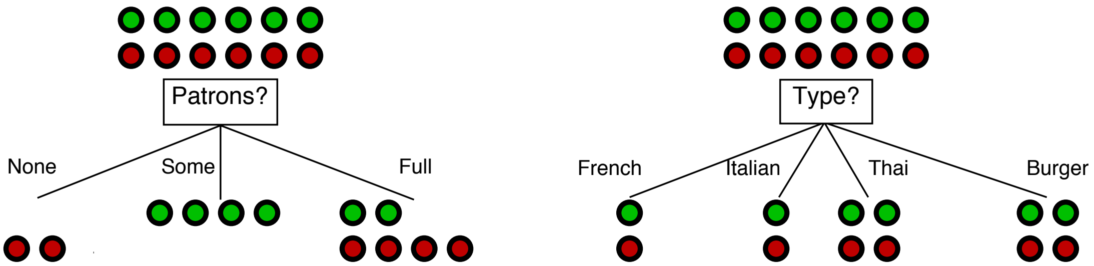


Figure 8.6: Splitting the examples from the *Restaurant* domain by testing on two different attributes. Note that splitting on *Type* brings us no nearer to distinguishing between positive and negative examples while splitting on *Patrons* does a good job of separating positive and negative examples. Example from [18].

the Table in Section 8.2) is shown in Fig. 8.6. For the value *None* (set E_1) we have $p_1 = 0$; $n_1 = 2$; for *Some* (set E_2): $p_2 = 4$; $n_2 = 0$ and for *Full* (set E_3): $p_3 = 4$; $n_3 = 0$. Thus,

$$
\begin{aligned}
EBS(Patrons) &= \sum_i \frac{p_i + n_i}{p+n}\, H\Big(\Big\langle \frac{p_i}{p_i+n_i}, \frac{n_i}{p_i+n_i}\Big\rangle\Big) \\
&= \frac{2}{12}\underbrace{H\Big(\Big\langle \frac{0}{2}, \frac{2}{2}\Big\rangle\Big)}_{0} + \frac{4}{12}\underbrace{H\Big(\Big\langle \frac{4}{4}, \frac{0}{4}\Big\rangle\Big)}_{0} + \frac{6}{12}H\Big(\Big\langle \frac{4}{6}, \frac{2}{6}\Big\rangle\Big) \\
&= \frac{1}{2}H\Big(\Big\langle \frac{2}{3}, \frac{1}{3}\Big\rangle\Big) = 0.4591 \text{ bits}
\end{aligned}
$$

the information gain of this attribute is

$$
Gain(Patrons) = B(\frac{p}{p+n}) - EBS(Patrons) = 1 - 0.4591 \approx 0.541 \text{ bits}
$$

It is easy to verify that for the attribute *Type* (for which the splitting is also shown in Fig. 8.6) the expected number of bits is still 1 and thus *Gain*(*Type*) = 0 bits. This confirms our intuition that *Patrons* is a better attribute to split on first. In fact, *Patrons* has the maximum information gain of any of the attributes and thus would be chosen by the decision tree learning algorithm as the root [18].

8.4.3 Some considerations

Another important question when dealing with decision trees is *when is it no longer useful to split a subset into smaller subsets?* For classification trees, it is clear that when a subset has zero class-entropy (that is, all cases in the subset have the same class), further splitting is no longer useful. For regression trees, the equivalent would be zero variance, but that is almost never achievable. Some learners stop splitting when the best test does not lead to a significant reduction of entropy or variance. When the subset to be split is very small, further reductions are almost certainly not significant; for that reason, many learners only split subsets whose size is above some threshold value.

It is known that too large trees tend to overfit the data: they fit the training data well, but tend to perform worse on other data. Fig. 8.7 illustrates this problem. Ideally, a tree learner stops splitting just before such overfitting occurs. However, it turns out it is very hard to determine the

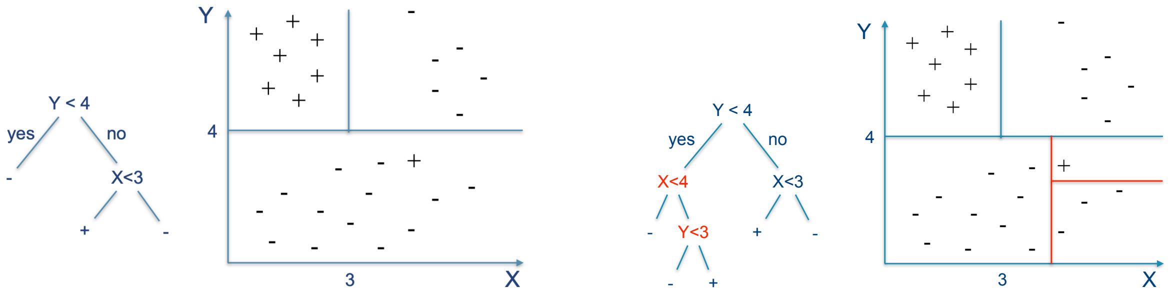


Figure 8.7: Left: a small tree fits the training data almost perfectly. It can be grown to fit perfectly (right), but a relatively large area to the right will then be predicted positive, while the data contains very little evidence for this. Example from [1].

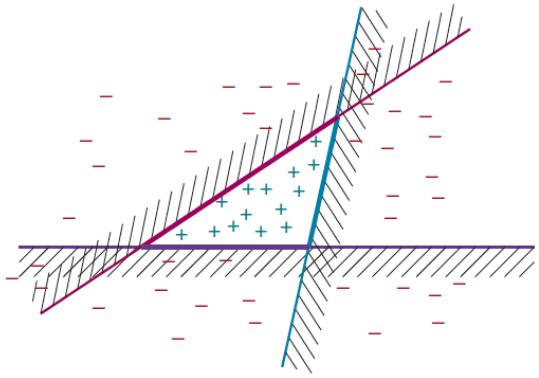


Figure 9.1: An example from [18] showing how ensemble learning can increase expressive power of simple (in this case, linear) models. By combining three linear models a triangular decision region can be achieved, which is beyond the possibility of any of the three base models alone.

right moment. Statistical significance tests do not work well in this context, and it is perfectly possible that even if no single test leads to a substantial improvement, a combination of tests will. For this reason, many learners grow the tree beyond its optimal size, and prune away useless branches afterwards. This pruning process typically makes use of a so-called validation set: a set of data not used for learning the tree, but used to estimate the quality of the full tree and its pruned variants during the pruning process. Since the validation set was not used while growing the tree, it provides an unbiased view of the actual predictive accuracy of the tree. The pruning process then consists of pruning away branches that did not lead to a higher accuracy on the validation set (i.e., the improvement they gave on the training set was most likely due to overfitting) [1].

9 Ensemble learning

Traditional learning methods use a single hypothesis to make predictions. **Ensemble learning** aims to improve performance by combining multiple hypotheses (called *base models*), denoted $h_1, h_2, \ldots, h_n$, into a single **ensemble model** [18]. The models are typically aggregated using techniques such as averaging, majority voting, or by meta-learning [**?**].

The motivation behind this approach is twofold:

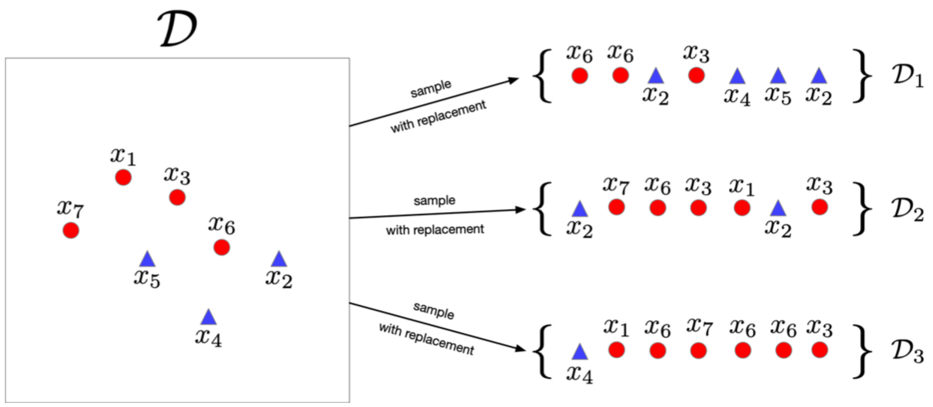


Figure 9.2: An illustration of sampling with replacement in the bagging procedure.

1. **Reducing bias** – A single model may be too simplistic, resulting in high bias (e.g., a linear classifier's limitations). An ensemble can represent more complex decision boundaries. Take an example from Fig. 9.1 where three linear classifiers together define a triangular region, which a single linear model cannot capture. Using n base models adds only n times more computation, which is often more efficient than training a more general, highly flexible model that may require exponential resources.

2. **Reducing variance** – Ensembles can also reduce variance of the prediction. Consider an example from [18]:
   Let an ensemble consist of $K = 5$ binary classifiers that we combine by majority voting. For a misclassification to occur, at least 3 out of 5 must be wrong, which is less likely than a single classifier making an error.

   Suppose each individual classifier is 80% accurate. If we train 5 classifiers on different data subsets to promote independence (though it may slightly reduce individual performance to, say, 75%), the ensemble can still achieve 89% accuracy. With 17 classifiers, accuracy can reach 99%, assuming independence. Verify this yourself!

   Note: In practice, full independence is rare—models often share data or assumptions and thus may make correlated errors. However, if base models are sufficiently diverse, ensembles can still reduce misclassifications.

Widely used ensemble models include **bagging**, **random forests**, **stacking** and **boosting**.

9.1 Bagging

The term **bagging** stems from "**b**ootstrap **agg**regat**ing**". Bootstrapping is a statistical method that involves resampling data *with replacement* (it is allowed to draw the same data point multiple times) to estimate the sampling distribution of a statistic or the uncertainty of a model.

The bagging procedure can be conceptually explained as consisting of the two steps:

1. **Generate K training sets by sampling with replacement and train K models**. We randomly pick N examples from the training set $\mathcal{D}$, allowing to pick the same examples multiple times. We then run a machine learning algorithm an these N examples and repeat

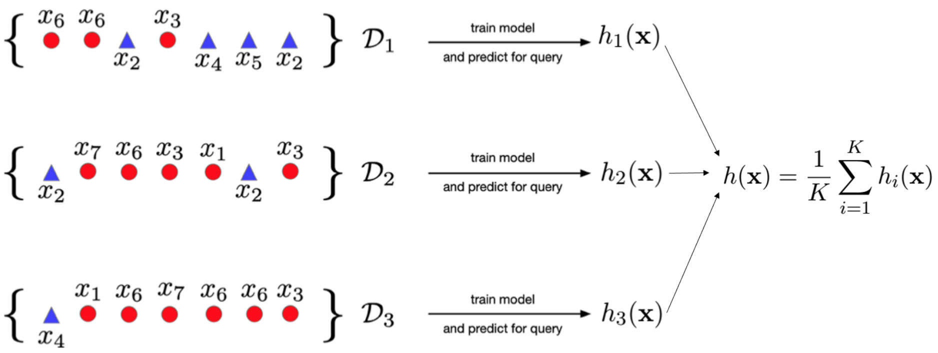


Figure 9.3: An illustration of sampling with replacement in the bagging procedure.

the process K times. This way we obtain K distinct training sets and K corresponding models. In the example from Fig. 9.2, the training set consists of the data points $\mathbf{x}_i$ from two classes (circles and triangles), $K = 3$ and $N = 7$.

2. **Aggregate the predictions of the K models**: For a new input $\mathbf{x}$, each of the K trained models gives its own prediction $h_i(\mathbf{x})$, $i = 1, \ldots, K$. In classification problems, the different h_i's are aggregated by a voting procedure and in regression problems the average is taken as the final hypothesis:

$$h(\mathbf{x}) = \frac{1}{K}\sum_{i=1}^{K} h_i(\mathbf{x})$$

Fig. 9.3 illustrates this aggregation for the regression task with $K = 3$ models, continuing on the example from Fig. 9.2.

Bagging lowers variance and is useful when there is limited data or when a model is prone to overfitting. While it can be applied to any model, it is most often used with decision trees due to their sensitivity to small changes in data. Bagging mitigates this instability, and is particularly efficient when computations are done in parallel across multiple machines [18].

9.2 Random forests

Random forests are an enhanced version of decision tree bagging designed to reduce variance and prevent overfitting. Unlike regular bagging, which often leads to correlated trees due to the dominance of certain attributes, random forests introduce additional randomness to create a more diverse ensemble. At each split, a random subset of attributes is considered, and for some variations, like **extremely randomized trees (ExtraTrees)**, the values at each split are also randomly sampled. This added randomness helps ensure that the trees are diverse and reduces the chance of overfitting. Random forests are also computationally efficient, as they can be built in parallel across multiple processors. Furthermore, they do not require pruning, as the ensemble approach naturally mitigates overfitting. Key hyperparameters, such as the number of trees and the number of attributes per split, can be tuned using cross-validation or by measuring out-of-bag error [18].

Despite their complexity, random forests are surprisingly resistant to overfitting, with error rates typically improving as more trees are added to the model. This is because the random selection of attributes leads to trees that cover different areas of the input space, making the model more robust and less likely to be overly sensitive to individual data points. However, while random forests are not immune to overfitting, the model's performance generally stabilizes as more trees are added, and the error does not grow indefinitely. Random forests have found widespread use across a variety of domains, from Kaggle data science competitions to practical applications in finance (credit card default prediction, income prediction) and bioinformatics (diabetic retinopathy, gene expression analysis). Although deep learning is becoming a dominant approach in AI, random forests remain a powerful and versatile machine learning tool offering strong performance across a wide range of tasks. For more in-depth coverage, see [2, 7, 8, 18].

9.3 Boosting

Boosting is a powerful ensemble method that improves the performance of "weak learners" by focusing on the training examples that are hardest to classify. It works by assigning weights to each training example, initially treating all examples equally. After training the first model, the algorithm increases the weights of misclassified examples and decreases the weights of correctly classified ones, encouraging subsequent models to focus on the hard cases. This process continues for a predefined number of iterations, building a sequence of models (hypotheses), each trained on a different distribution of weights. Unlike bagging, boosting is a sequential and greedy algorithm, meaning it cannot parallelize model training. In the final ensemble, each model contributes to the prediction with a weight based on its accuracy:

$$h(\mathbf{x}) = \frac{1}{K}\sum_{i=1}^{K} z_i h_i(\mathbf{x})$$

A widely used variant of boosting is *AdaBoost*, which is especially effective when using simple models as the base models. It is usually applied with decision trees as the component hypotheses. A key theoretical result of *AdaBoost* is that if the base model performs just slightly better than random guessing (a weak learner), then the boosting process can combine them into a **strong learner** that achieves *perfect accuracy on the training set*, for large enough K, regardless of the base model's simplicity or the complexity of the function to be learned [18] Thus the algorithm guarantees to overcome bias in the base model, as long as the base model is better than random guessing.

Boosting demonstrates a counter-intuitive but powerful behavior: even after the ensemble perfectly fits the training data, adding more weak learners can continue to improve test performance. This challenges traditional intuitions like Ockham's razor, which cautions against increasing model complexity unnecessarily. The insight here is that boosting doesn't just fit the data—it refines confidence in predictions, particularly around difficult examples. Theoretical interpretations suggest this may stem from boosting's resemblance to Bayesian learning [18], gradually improving approximation to an optimal classifier. As a result, boosting can generalize better even as the ensemble grows more complex, although the improvements may become smaller or stabilize after a certain number of weak learners.

10 Neural Networks

Artificial neural networks are computational models composed of simple processing units, or *neurons*, arranged in layers and connected by weighted edges. Despite the simplicity of each individual neuron, the collective behavior of a large network can represent highly complex functions. This section introduces the building blocks of neural networks, explains the role of nonlinear activation functions, and illustrates how layered connections lead to increasingly expressive models.

10.1 Artificial Neurons and Activation Functions

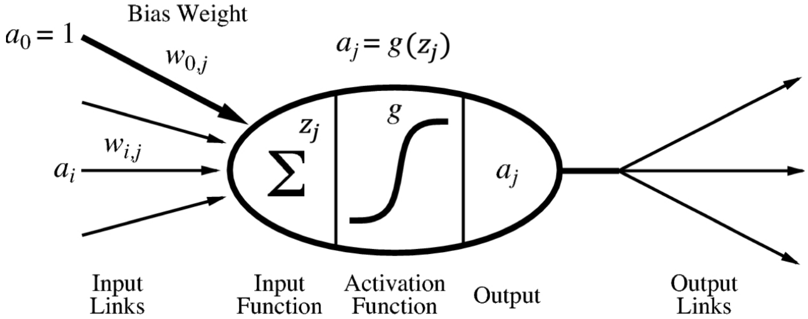


Figure 10.1: The McCulloch–Pitts neuron, the classical abstraction of a biological neuron. It computes a weighted sum of its inputs and applies a nonlinear activation. Adapted from [18].

The computational core of a neural network is the artificial neuron, illustrated in Fig. 10.1. Each neuron receives scalar inputs a_i, multiplies them by learned weights $w_{i,j}$, and aggregates them into a scalar being the weighed sum of the inputs:

$$z_j = \sum_i w_{i,j} a_i.$$

This value is then passed through a nonlinear activation function g, producing the neuron's output

$$a_j = g(z_j).$$

This nonlinear transformation is essential. If all activation functions were linear, the network would collapse into a single linear function, regardless of depth. The expressive power of neural networks arises precisely because the activation distorts the linear combinations in a way that cannot be undone by further linear transformations. In this sense, nonlinearity is the mechanism by which networks escape the limitations of classical linear models.

Fig. 2 illustrates several activation functions widely used in modern neural networks. The logistic sigmoid was historically motivated by biological activation curves; today it plays an important role in probabilistic models. The hyperbolic tangent function rescales the sigmoid to the interval $(-1, 1)$ and is often more convenient when activations are centered around zero.

The rectified linear unit (ReLU), defined by $\max(0, x)$, has become the default activation in deep learning because it avoids the saturation problems inherent in sigmoids and thus facilitates gradient-based optimization. Softplus $\log(1 + e^x)$ provides a smooth approximation to ReLU and is useful when differentiability is important. Each activation introduces a different kind of nonlinearity, affecting how the network partitions or bends the input space.

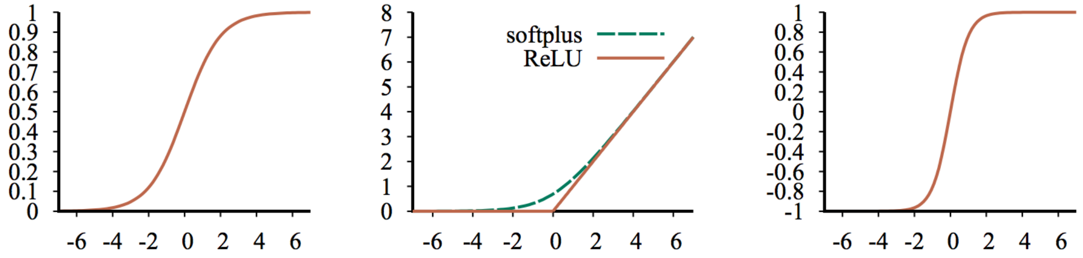
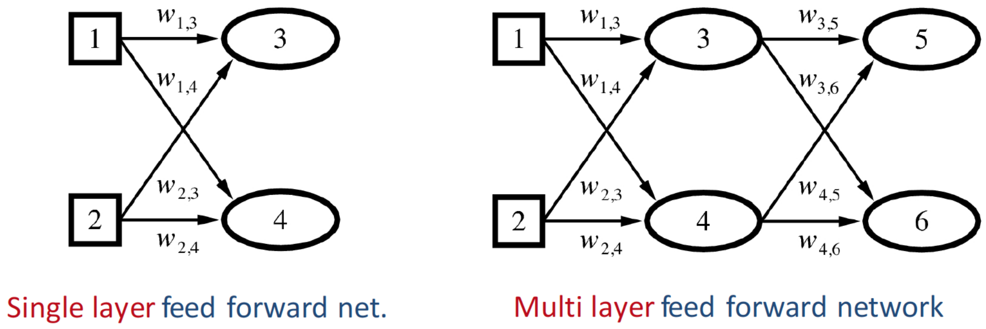


Figure 10.2: Common activation functions. Left: logistic (sigmoid); Middle: ReLU and softplus; Right: tanh. Illustration from [18].



Single layer feed forward net.

Multi layer feed forward network

Figure 10.3: Examples of feed-forward neural networks. Multilayer networks have one or more "hidden" layers between the input and the output layers (the network on the right has one hidden layer). Each additional layer increases the representational capacity of the network.

The bias weight associated with each neuron shifts the activation function horizontally. Adjusting the bias moves the threshold at which the neuron becomes active, giving the network control over where nonlinearities occur. This simple mechanism enables neurons to respond to different regions of the input space and is crucial for forming complex decision boundaries.

10.2 Feed-Forward Architectures

Neural networks differ not only in their choice of activation functions but also in their structural connectivity. We shall focus on *feed-forward* architectures, where information flows strictly from the input layer to one or more hidden layers and finally to the output layer.

- In a feed-forward network, each neuron computes a function of its inputs and passes the result to neurons in the next layer.
- There are no cycles: the output of a neuron is never fed back into itself or any preceding layer.
- Networks of this type include single-layer and multi-layer perceptrons (see Fig. 10.3). Multilayer networks have one or more "hidden" layers.

Feed forward network architectures may vary in width and depth. Increasing the number of neurons in a layer allows the model to capture finer distinctions among input patterns, while increasing the number of layers allows the network to construct hierarchical representations: lower

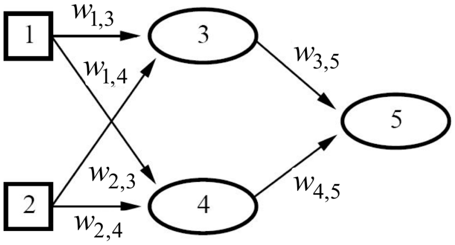


Figure 10.4: A two-layer feed-forward network illustrating how nonlinearities compose.

layers extract elementary features, while higher layers combine them into abstract concepts. This hierarchical organization is central to the success of deep learning.

In these notes, the terms "neuron" and "network node"(or simply "node") will be used interchangeably, as they refer to the same computational unit in any layer of the network. We shall use the following notation.

- We denote by $a_j^{(l)}$ the *activation* of the j-th node in the l-th layer of the network, which is also the signal at the output of that node.
- The input to that same node, that we also call *preactivation* (or sometimes the *score*) is a weighted sum of the activations of the nodes from the previous layer:

$$z_j^{(l)} = \sum_i w_{i,j}\, a_i^{(l-1)}$$

- The nonlinear activation function $g^{(l)}$ can be different in each layer l, but unless necessary to express this explicitly, we will write only g.

Inputs are denoted x_i, so the first layer computes

$$a_j^{(1)} = g\left(\sum_i w_{i,j} x_i\right) = g(\mathbf{w}_j \cdot \mathbf{x}). \tag{10.1}$$

When context makes the layer clear, we omit the superscript and write simply a_j and z_j.

Example: Consider as an example the network in Fig. 10.4. The final activation a_5 depends on the outputs of neurons a_3 and a_4, which themselves depend on the original inputs:

$$\begin{aligned} a_5 &= g(w_{3,5}a_3 + w_{4,5}a_4), \\ &= g(w_{3,5}g(w_{1,3}x_1 + w_{2,3}x_2) + w_{4,5}g(w_{1,4}x_1 + w_{2,4}x_2))\,. \end{aligned}$$

Observe the recursive nature of neural networks: each layer transforms the output of the previous one, creating a nested composition of nonlinear maps. Through this nested composition of nonlinear functions, even shallow networks can produce highly nontrivial transformations of the input space.

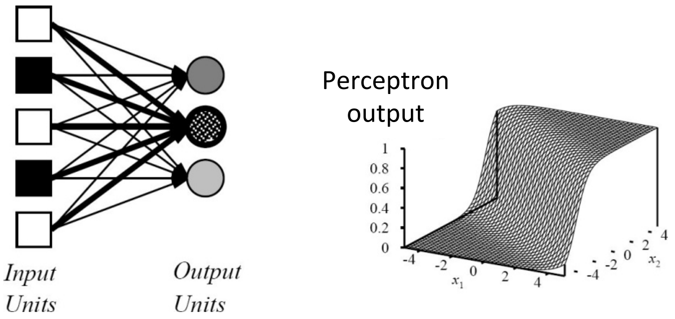


Figure 10.5: A single-layer neural network. Such networks can only produce a single linear decision boundary after the nonlinearity.

10.3 Limitations of single-layer networks

Single-layer perceptrons compute the output activations as $a_j = g(\mathbf{w}_j \cdot \mathbf{x})$, which means their decision surfaces are linear (or affine) in the input space. The nonlinearity is applied *after* the weighted sum, so it cannot generate complex partitions of the plane. This restricts single-layer networks to learning linearly separable (or nearly separable) datasets.

A geometrical interpretation, illustrated in Fig. 10.5, is that the model cannot bend or warp the space: it can shift and rescale a linear boundary but cannot change its topology. This is why problems such as XOR cannot be solved by a single-layer network. Adding hidden layers overcomes this limitation by introducing intermediate nonlinear transformations.

10.4 Multilayer neural networks

Deeper feed-forward networks manipulate the input through sequences of nonlinear transformations. Early layers extract simple directional or contrast-based features, while deeper layers develop class-specific abstractions. Fig. 10.6 depicts this progressive separation of classes in feature space, culminating in a linearly separable configuration at the output layer. In essence, such a deep network effectively learns a set of nonlinear feature mappings $\phi_i(\mathbf{x})$ from the raw input $\mathbf{x}$. In doing so, it eliminates the need for manual feature engineering, which would otherwise be required to supply nonlinear features to an equivalent multiclass logistic regression classifier.

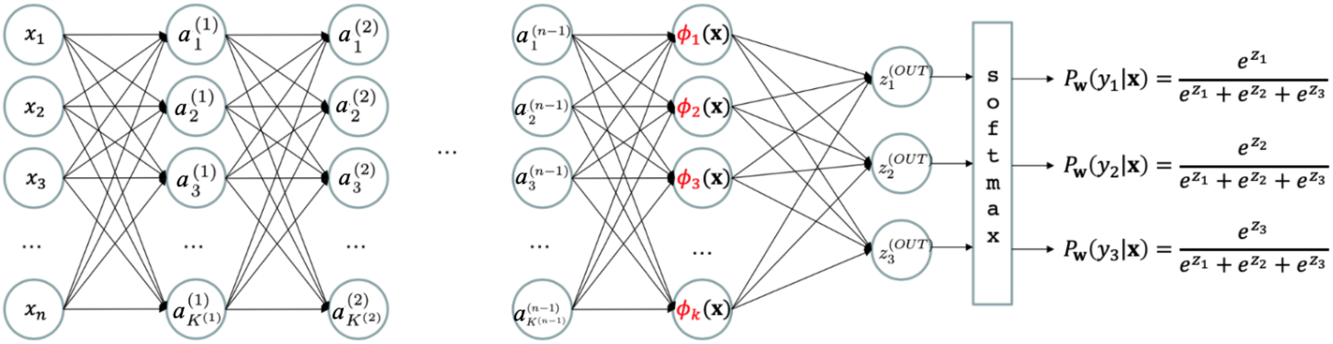


Figure 10.6: Feed-forward network applied to multi-class classification. Hidden layers gradually reshape the representation so that classes become separable.

10.4.1 Classical backpropagation algorithm

Deep neural networks contain many parameters, often millions or more, and training consists of adjusting the weights so as to minimize a loss function. Gradient descent is the method most commonly used, but computing the gradient of the loss with respect to all weights requires a systematic and efficient procedure. This is provided by the *backpropagation* algorithm, which applies the chain rule of calculus layer by layer.

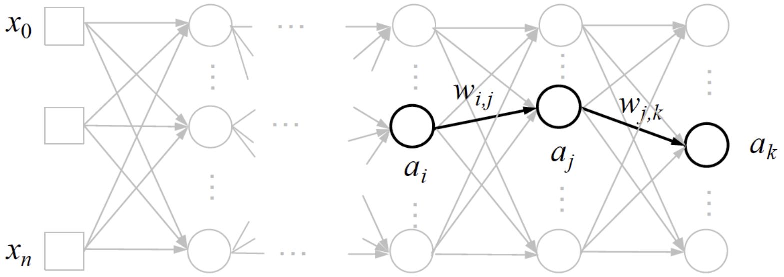


Figure 10.7: Simplified notation illustrating the structure of the computational graph for backpropagation. Inputs a_i correspond to x_i, hidden activations to a_j, and outputs to a_k.

Fig. 10.7 presents a simplified computational graph that captures the dependency structure of a feed-forward network. Backpropagation begins at the output layer, where the loss is directly defined, and moves backward through the graph, passing error signals to earlier layers.

Remark: *With some abuse of notation, and to simplify the presentation of the derivation, we here use a_i for inputs, a_j for hidden activations and a_k for outputs.*

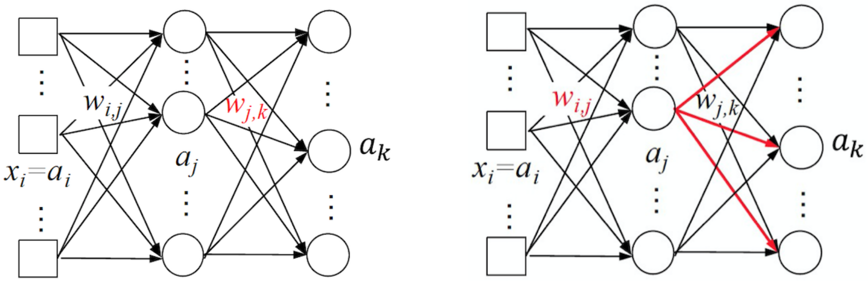


Figure 10.8: Left: Updates for the output-layer weights. Right: Backpropagation of error signals to inner-layer weights.

For an output neuron with preactivation z_k and output $a_k = g(z_k)$, the gradient of the squared loss function $Loss = \sum_k (y_k - a_k)^2$ with respect to a weight $w_{j,k}$ is

$$\frac{\partial Loss}{\partial w_{j,k}} = \frac{\partial Loss}{\partial a_k}\frac{\partial a_k}{\partial z_k}\frac{\partial z_k}{\partial w_{j,k}} = a_j\, g'(z_k)\,[-2(y_k - a_k)] = a_j \Delta_k.$$

Thus the output weights are updated as $w_{j,k} \leftarrow w_{j,k} - \alpha a_j \Delta_k$. The quantity $\Delta_k = g'(z_k)[-2(y_k - a_k)]$ is the *error signal* for output neuron k.

For hidden units, the loss does not depend on the weight $w_{i,j}$ directly, but only through the activations of the downstream units that receive a_j as an input. Thus the gradient must be assembled by propagating error information backward from all units to which neuron j contributes.

Recall that

$$z_j = \sum_i w_{i,j} a_i, \qquad a_j = g(z_j).$$

We seek the derivative of the loss with respect to $w_{i,j}$. The chain rule gives

$$\frac{\partial Loss}{\partial w_{i,j}} = \frac{\partial z_j}{\partial w_{i,j}} \frac{\partial a_j}{\partial z_j} \frac{\partial Loss}{\partial a_j}.$$

The first two factors are straightforward:

$$\frac{\partial z_j}{\partial w_{i,j}} = a_i, \qquad \frac{\partial a_j}{\partial z_j} = g'(z_j).$$

The remaining term $\frac{\partial Loss}{\partial a_j}$ requires more care. The activation a_j influences the loss only through the units k in the next layer for which a_j is an input. Thus we must sum over all such downstream units:

$$\frac{\partial Loss}{\partial a_j} = \sum_k \frac{\partial Loss}{\partial z_k} \frac{\partial z_k}{\partial a_j}.$$

Since $z_k = \sum_j w_{j,k} a_j$, we have $\frac{\partial z_k}{\partial a_j} = w_{j,k}$, and, for the output of unit k,

$$\frac{\partial Loss}{\partial z_k} = \frac{\partial Loss}{\partial a_k} \frac{\partial a_k}{\partial z_k} = \Delta_k,$$

where $\Delta_k = g'(z_k)\,[-2(y_k - a_k)]$ is the error signal at neuron k. Hence,

$$\frac{\partial Loss}{\partial a_j} = \sum_k w_{j,k} \Delta_k.$$

Substituting into the original chain rule expression yields

$$\frac{\partial Loss}{\partial w_{i,j}} = a_i g'(z_j) \sum_k w_{j,k} \Delta_k.$$

This motivates defining the hidden-layer error signal

$$\Delta_j = g'(z_j) \sum_k w_{j,k} \Delta_k,$$

so that the gradient simplifies to

$$\frac{\partial Loss}{\partial w_{i,j}} = a_i \Delta_j.$$

This expression mirrors the outer-layer gradient but with the error signal Δ_j obtained by back-propagating contributions from deeper layers.

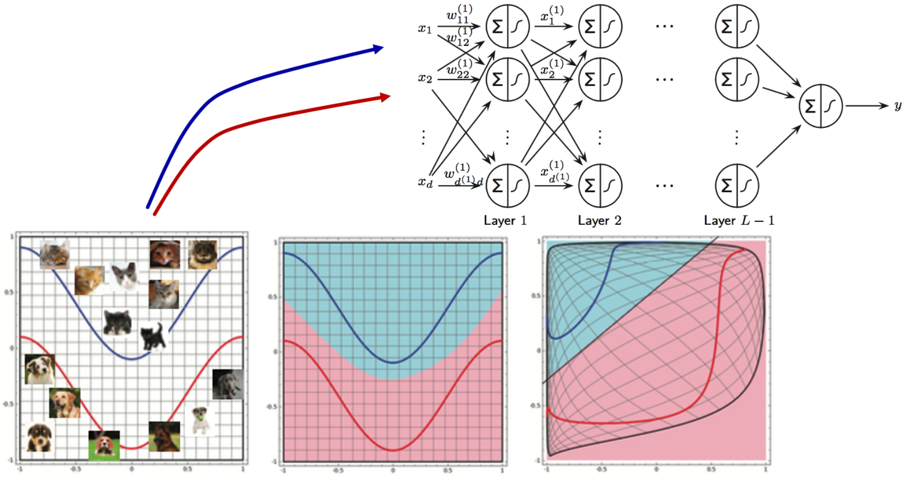


Figure 10.9: Deep networks progressively warp the input space, making complex class boundaries linearly separable in the transformed feature space.

10.4.2 Geometric View and Universal Approximation

One of the most important insights about neural networks is that their power does not come from the raw number of parameters, but from the *composition* of nonlinear transformations. Each layer applies a deformation of the input space, stretching some regions while contracting others, folding and twisting the geometry so that classes that were intertwined in the original space become separable in the transformed one.

Fig. 9 illustrates this perspective: successive layers reshape the representation to disentangle complex manifolds. This geometric viewpoint provides intuition for why deep networks, despite their apparent complexity, can generalize well: the learned transformations align data with simple, often linear, decision surface. See the Nature review by LeCun, Bengio and Hinton [12].

The **Universal Approximation Theorem** states that neural networks with a certain structure can, in principle, approximate any continuous function to any desired degree of accuracy. Furthermore, it can be shown that a neural network with a single hidden layer and sufficiently many neurons can approximate any continuous function on a compact domain [9]. Depth, however, offers a more efficient representation by enabling hierarchical structure. Rather than approximating the target function in one step, the network builds it gradually through intermediate features.

11 Deep neural networks

Modern neural networks used in practice often consist of many layers, ranging from a few tens to, in carefully designed architectures such as residual networks, hundreds of successive transformations. For an input vector $\mathbf{x}$, such models apply a long sequence of linear mappings and elementwise nonlinearities. As the depth increases, both the description of the model and the derivation of learning algorithms become more involved. For this reason, a compact and systematic mathematical notation is essential. In this section, we introduce matrix notation for multilayer (deep) neural networks, which allows their forward computation and gradient-based training to be described in a concise and scalable way.

11.1 Matrix notation for multilayer neural networks

To describe deep neural networks compactly, it is convenient to collect the parameters of each layer into matrices and write the forward computation in matrix form.

Weight matrix. Let layer l have m neurons and d inputs. Its parameters are collected in a *weight matrix* $\mathbf{W}^{(l)} \in \mathbb{R}^{m\times(d+1)}$, whose rows are the weight vectors of the individual neurons:

$$\mathbf{W}^{(l)} = \begin{bmatrix} \mathbf{w}_1^{(l)\top} \\ \vdots \\ \mathbf{w}_m^{(l)\top} \end{bmatrix}.$$

Each weight vector $\mathbf{w}_j^{(l)}$ contains the weights connecting all inputs to neuron j. For the most compact mathematical formulation, we can include the bias weights also in the weight matrix, following the notation from Fig. 10.1, where the inputs to each neuron are augmented with a constant component (dummy input) equal to 1 that is fed into all the neurons in layer l. Then the j-th row of the weight matrix is

$$\mathbf{W}^{(l)}(j,:) = \mathbf{w}_j^{(l)\top} = [\, w_{0,j}^{(l)}, \ldots, w_{d,j}^{(l)}\,].$$

where $w_{0,j}^{(l)}$ is the bias term for the j-th neuron in the l-th layer, multiplying the dummy input $a_0^{(l-1)} = 1$ that is fed into it and into all the other neurons at layer l. In this convention, for $l = 1$, we have $a_0^{(0)} = x_0 = 1$, i.e., the input is also augmented with the constant input 1, the same notation that we used throughout these lecture notes.

With this convention, for an input $\mathbf{x} = [\,0, x_1, \ldots, x_d\,]$, the forward propagation through a network with L layers is given by

$$\begin{aligned}
\mathbf{a}^{(1)} &= g\big(\mathbf{W}^{(1)}\mathbf{x}\big)\,, \quad \text{(append } a_0^{(1)} = 1\text{)},\\
\mathbf{a}^{(2)} &= g\big(\mathbf{W}^{(2)}\mathbf{a}^{(1)}\big)\,, \text{ (append } a_0^{(2)} = 1\text{)},\\
&\vdots\\
\mathbf{a}^{(L-1)} &= g\big(\mathbf{W}^{(L-1)}\mathbf{a}^{(L-2)}\big)\,, \text{ (append } a_0^{(L-1)} = 1\text{)},\\
\mathbf{z}^{OUT} &= \mathbf{W}^{(L)}\mathbf{a}^{(L-1)}. \qquad (11.1)
\end{aligned}$$

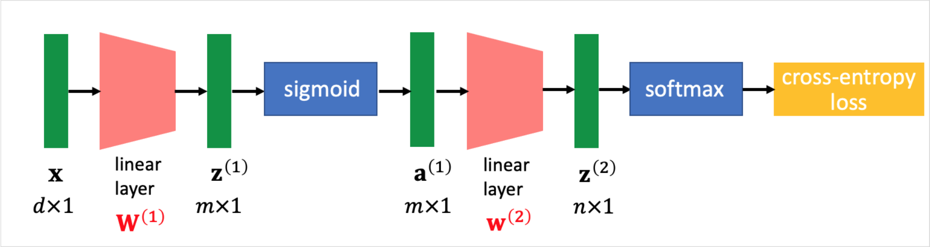


Figure 11.1: A compactly represented neural network. Adapted from Sergey Levine: Backpropagation - Designing, Visualizing and Understanding Deep Neural Networks.

In regression, the prediction is given by the output score,

$$h_{\mathbf{w}}(\mathbf{x}) = \mathbf{z}^{OUT}.$$

In classification, the output score is further transformed, for example by a threshold or sigmoid function in the binary case,

$$h_{\mathbf{w}}(\mathbf{x}) = \sigma(\mathbf{z}^{OUT}),$$

or by a softmax function in the multiclass setting.

11.2 Backpropagation in a matrix form

We derived before the classical backpropagation algorithm. Let's revisit backpropagation using the vector and matrix notation. Consider a simplified two-layer network, schematically represented in Fig. 11.1. The gradient with respect to the first-layer weights can be expressed using the chain rule as

$$\frac{\partial Loss}{\partial \mathbf{W}^{(1)}} = \frac{\partial \mathbf{z}^{(1)}}{\partial \mathbf{W}^{(1)}} \frac{\partial \mathbf{a}^{(1)}}{\partial \mathbf{z}^{(1)}} \frac{\partial \mathbf{z}^{(2)}}{\partial \mathbf{a}^{(1)}} \frac{\partial Loss}{\partial \mathbf{z}^{(2)}}. \tag{11.2}$$

Each factor in this expression is a Jacobian matrix. If each layer has size n, then

$$\frac{\partial \mathbf{a}^{(1)}}{\partial \mathbf{z}^{(1)}} \in \mathbb{R}^{n\times n}, \qquad \frac{\partial \mathbf{z}^{(2)}}{\partial \mathbf{a}^{(1)}} \in \mathbb{R}^{n\times n}.$$

Multiplying these Jacobians explicitly requires $O(n^3)$ time, which is infeasible for typical layer sizes (e.g. $n = 4096$). Backpropagation avoids this cost by evaluating the expression in Eq. (11.2) in reverse order. Instead of forming products of Jacobian matrices, the gradient of the loss is propagated backward through the network via successive Jacobian–vector products. Initialize the process by computing:

$$\boldsymbol{\delta}_{\text{init}} = \frac{\partial Loss}{\partial \mathbf{z}^{(2)}}$$

and then proceed sequentially (here we need only two update steps, but for a deeper network we would compute in the same way more):

$$\boldsymbol{\delta} = \frac{\partial \mathbf{z}^{(2)}}{\partial \mathbf{a}^{(1)}} \boldsymbol{\delta}_{\text{init}},$$

$$\boldsymbol{\delta}_{\text{new}} = \frac{\partial \mathbf{a}^{(1)}}{\partial \mathbf{z}^{(1)}} \boldsymbol{\delta}.$$

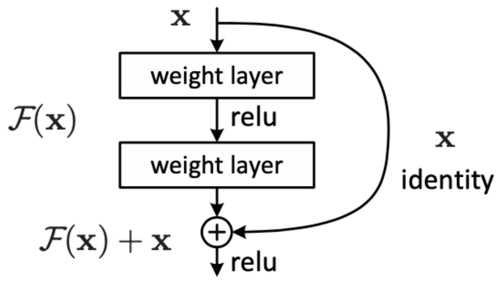


Figure 11.2: Residual block. The stacked layers learn a residual $\mathcal{F}(\mathbf{x})$, which is added to the identity mapping $\mathbf{x}$ via a skip connection.

Each step requires only a matrix–vector product, hence of the order $O(n^2)$, not a matrix–matrix product that would require $O(n^3)$ computations. Finally,

$$\frac{\partial Loss}{\partial \mathbf{W}^{(1)}} = \frac{\partial \mathbf{z}^{(1)}}{\partial \mathbf{W}^{(1)}} \boldsymbol{\delta}_{\text{new}}.$$

Thus, the recursion used in backpropagation is essential for computational efficiency. It yields the exact gradient while avoiding costly multiplication of large Jacobians. This mechanism is the basis of automatic differentiation systems used in modern deep learning frameworks.

11.3 Residual neural networks

Residual neural networks are a popular approach to building very deep neural networks. Instead of learning the desired mapping $h(\mathbf{x})$ directly, the stacked nonlinear layers learn a residual function

$$\mathcal{F}(\mathbf{x}) = h(\mathbf{x}) - \mathbf{x},$$

so that the original mapping can be written as

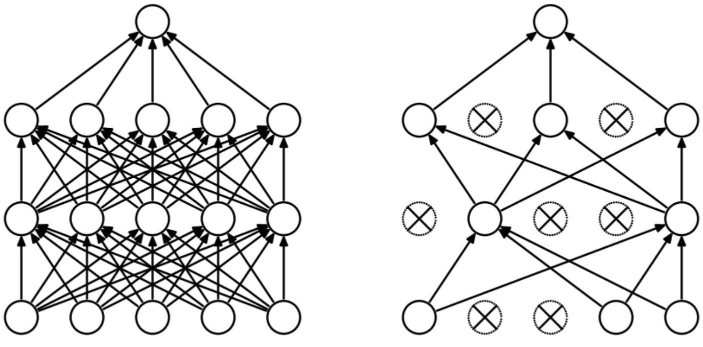
$$h(\mathbf{x}) = \mathcal{F}(\mathbf{x}) + \mathbf{x}.$$

This reformulation often makes optimization easier than learning $h(\mathbf{x})$ in an unreferenced form. In particular, if the identity mapping were optimal, it is easier to push the residual $\mathcal{F}(\mathbf{x})$ toward zero than to fit the identity mapping using a stack of nonlinear layers.

11.4 Dropout regularization

Deep neural networks typically contain a very large number of parameters and are therefore prone to overfitting. Dropout is a simple and widely used regularization technique that helps mitigate this problem.

During training, dropout randomly deactivates a subset of hidden units by setting their activations to zero. As a result, each training step effectively updates a different subnetwork. At test time, all units are active and their outputs are appropriately rescaled, which can be interpreted as implicitly averaging the predictions of many subnetworks.



(a) Standard Neural Net

(b) After applying dropout.

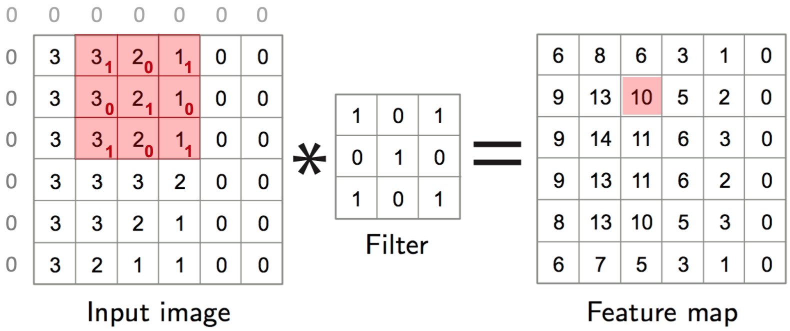
Figure 11.3: Dropout regularization. During training, a randomly selected subset of units is deactivated, preventing strong co-adaptation between neurons.

11.5 Convolutional neural networks

Convolutional neural networks (CNNs) are a class of neural network architectures specifically designed to process data with a grid-like structure, most notably images. A key motivation for CNNs arises from the limitations of fully connected neural networks when applied to realistic image sizes. For example, even a modest color image of size 224×224 contains more than 150,000 input values; connecting such an input fully to just a single hidden layer would already require tens of millions of parameters. This leads to excessive memory requirements, slow training, and a high risk of overfitting.

CNNs address this problem by exploiting two fundamental properties of images. First, informative patterns such as edges, corners, and textures are typically local: they depend on small neighborhoods of pixels rather than on the entire image. Second, these patterns tend to reappear at many different spatial locations. Convolutional layers explicitly encode these assumptions by restricting each neuron to a small receptive field and by reusing the same set of weights across the image. This architectural choice dramatically reduces the number of parameters, introduces a powerful inductive bias, and makes it possible to train very deep networks that scale to high-resolution images.

Fig. 11.4 illustrates the basic convolution mechanism. A small filter (or kernel), typically of size 3×3, is positioned at a particular location of the input image. The filter values are multiplied elementwise with the corresponding input pixels, and the resulting products are summed. This sum



0	0	0	0	0	0	0
0	3	3_1	2_0	1_1	0	0
0	3	3_0	2_1	1_0	0	0
0	3	3_1	2_0	1_1	0	0
0	3	3	3	2	0	0
0	3	3	2	1	0	0
0	3	2	1	1	0	0

Input image

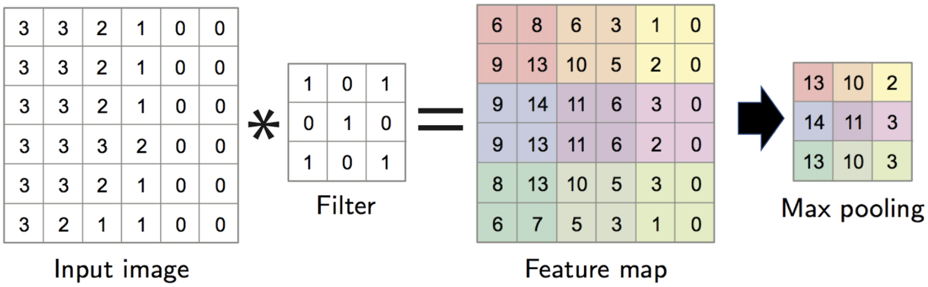
1	0	1
0	1	0
1	0	1

Filter

6	8	6	3	1	0
9	13	10	5	2	0
9	14	11	6	3	0
9	13	11	6	2	0
8	13	10	5	3	0
6	7	5	3	1	0

Feature map

Figure 11.4: An illustration of the convolutional operation at the heart of CNNs.



3	3	2	1	0	0
3	3	2	1	0	0
3	3	2	1	0	0
3	3	3	2	0	0
3	3	2	1	0	0
3	2	1	1	0	0

1	0	1
0	1	0
1	0	1

6	8	6	3	1	0
9	13	10	5	2	0
9	14	11	6	3	0
9	13	11	6	2	0
8	13	10	5	3	0
6	7	5	3	1	0

13	10	2
14	11	3
13	10	3

Figure 11.5: Convolution followed by max pooling, producing a downsampled feature map.

measures how strongly the pattern encoded by the filter matches the local content of the image. By sliding the filter across all spatial positions, the convolution operation produces a *feature map*, whose entries indicate the presence of that pattern at different locations.

In typical convolutional neural networks, the resulting feature maps are then subjected to a *pooling* operation. Pooling aggregates responses over small spatial neighborhoods of the feature map, reducing its spatial resolution while retaining the most salient activations. This further decreases computational cost and introduces robustness to small translations or local distortions in the input. Common pooling strategies include *max pooling* and *average pooling*. Fig. 11.5 illustrates the max-pooling operation.

In the example shown in Fig. 11.6, the kernel is designed to respond strongly to vertical intensity changes and therefore acts as a vertical edge detector. As a result, the corresponding feature map highlights vertical structures present in the input image. Different kernels can be learned to detect other local patterns, such as edges with different orientations, corners, or more complex texture-like features, depending on the task for which the network is trained.

Because the same filter is applied at every spatial location, CNNs benefit from *weight sharing*. Instead of learning separate weights for each pixel position, the network learns a single filter that characterizes a feature of interest, such as a vertical edge, a horizontal transition, or a repeating texture. This not only leads to a drastic reduction in the number of parameters, but also induces translation equivariance: a shift in the input image produces a corresponding shift in the feature map. For visual data, this is a desirable property, since the semantic meaning of a feature does not depend on its absolute position.

The conceptual diagram in Fig. 11.7 summarizes the typical organization of a convolutional neural network. Convolutional layers produce collections of feature maps, which are often followed by pooling operations that progressively reduce spatial resolution while retaining salient

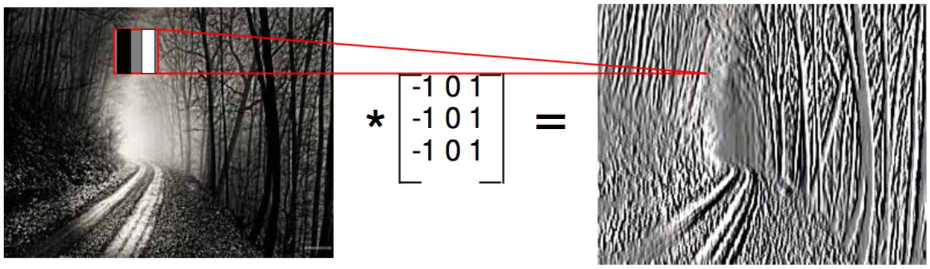


Figure 11.6: Picture credit: M. Ranzato: *Image Classification with Deep Learning*, 2015.

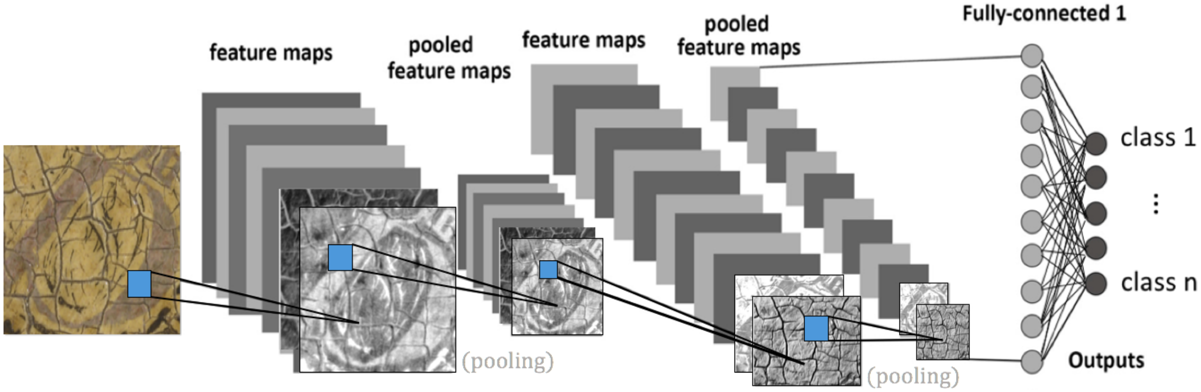


Figure 11.7: Schematic illustration of a convolutional neural network architecture, showing stacked convolutional and pooling layers followed by fully connected layers.

information. These stages are stacked to form a deep architecture, with later convolutional layers operating on increasingly abstract representations derived from earlier ones.

Although not shown explicitly in the figure, each convolutional operation is typically followed by a nonlinear activation function, such as ReLU. After several such convolutional (and pooling) stages, the resulting feature maps are flattened and passed to one or more fully connected layers, which integrate the extracted features and produce the final output, for example class scores in an image classification task. Through this combination of convolution, activation, pooling, and fully connected layers, CNNs learn hierarchical representations well suited for tasks such as image classification, detection, and segmentation.

To formalize the computation performed by a convolutional layer, consider the output of the k-th feature map in layer l at spatial location (i, j). Each feature map is produced by a filter of size $H_l \times W_l$, applied to all M feature maps (channels) of the previous layer. The activation $a_{i,j}^{l,k}$ is obtained by computing a weighted sum over a local patch of the previous feature maps, adding a bias term, and applying a nonlinear activation function g:

$$a_{i,j}^{l,k} = g\left(\sum_{m=1}^{M} \sum_{p=0}^{H_l-1} \sum_{q=0}^{W_l-1} w_{p,q}^{l,k,m}\, a_{(i+p),(j+q)}^{(l-1),m} \;+\; b^{l,k} \right). \tag{11.3}$$

This expression highlights the essential structure of convolutional layers:

- the sums over p and q traverse the spatial extent of the filter;
- the sum over m aggregates contributions from all channels of the previous layer;
- the weights $w_{p,q}^{l,k,m}$ define the filter associated with output feature map k;
- the bias $b^{l,k}$ allows the filter response to be shifted independently of the local average.

Because the same filter is applied at every spatial location (i, j), the network learns features that are meaningful regardless of where they occur in the input. Early layers typically learn edge detectors or simple texture filters, while later layers combine these primitive features into progressively more abstract templates, eventually producing representations that are selective for object parts or whole-object configurations. This hierarchical organization of learned features is a central reason for the success of convolutional architectures in computer vision.

12 Probabilistic reasoning

So far, we addressed machine learning without taking into account *uncertainty* due to unreliable or erroneous data, partial observability, lack of complete domain knowledge, non-deterministic aspects of the environment or various adversaries. Take as an example a robotic agent navigating through a partially observable environment and relying on its noisy sensor readings. It may never know for sure at which exact location it is and even less so in which state it will lend after a sequence of actions. We thus need a reasoning mechanism that supports decision making in uncertain and complex scenarios, and probabilistic reasoning provides us a consistent framework to model and reason about such uncertainty in a principled manner.

Rather than committing to a single, potentially incorrect hypothesis about the world, probabilistic models maintain distributions over latent variables, states, and outcomes. This enables us to weigh alternative explanations, update beliefs as new evidence arrives, and select actions that are robust under ambiguity. In the case of our robotic agent, a probabilistic framework allows it to represent uncertainty about its position, predict the likelihood of different future states under various action choices, and plan trajectories that minimize expected risk.

Probabilistic reasoning thus forms the foundation of a broad family of methods, ranging from Bayesian networks and hidden Markov models to to probabilistic graphical models that integrate perception, inference, and decision making. When combined with learning, these methods allow agents not only to handle uncertainty but also to learn the structure and parameters of their models from data, adapt to changes in the environment, and quantify confidence in their predictions.

In this Section, we will present basics of probabilistic reasoning. The following parts are taken from the book chapter of A. Pizurica: "Probabilistic reasoning: When the environment is uncertain" [16].

12.1 Why do we need probabilistic reasoning?

When reasoning under uncertainty, we often need a way to compare different possible courses of action to decide on a way forward. This also means being able to rank, in a consistent manner, the worthiness of actions or plans even though none of them is guaranteed to succeed. Suppose the goal is to be on time for an important meeting in a different part of the city while leaving the office as late as possible. Plan A is to drive along a ring around the city, Plan B take a shorter road through the small streets in the city center and Plan C to go by bicycle. Plan A is likely to fail if there is a road accident or a lane is closed for road works. For Plan B, a blocked street and detour through one-way streets can be detrimental while increased intensity of rain or pants caught in bike chain can be devastating for plan C. A sudden disturbance in the ionosphere caused by a geomagnetic storm can invalidate your GPS navigation system and get you off the optimal route. None of these obstacles can be excluded – no matter how unlikely they are – and many others may arise that we could not think of or did not want to bother listing explicitly. Still, we need a consistent way to rank the merits of the various plans in uncertain situations. This leads us to the concept of *probabilistic reasoning*, a.k.a. *reasoning under uncertainty.*

The importance of probabilistic reasoning is often illustrated with diagnostic tasks. Making medical diagnosis, determining the cause of a mechanical failure or a given effect in any other context involves almost always uncertainty. Think for example of diagnosing the cause of chronic fatigue. The list of possible causes is almost infinite, ranging from lack of good sleep, not enough activity or unbalanced diet to anemia or even heart problems, cancer or covid infection, just to name a few. Furthermore, in some cases one has to make a diagnosis while lacking complete knowledge of the domain (theoretical ignorance) and/or being unable to perform all the relevant tests due to various practical reasons and limitations (practical ignorance).

It is therefore often said, after [18], that probabilistic reasoning succeeds where purely logical reasoning fails due to our "laziness" (avoiding exhaustive lists of all possible causes or consequences) and ignorance (both theoretical and practical). Figure 12.1 highlights this concept.

In essence, probabilistic reasoning provides us with means to evaluate *degrees of belief* for the success of different outcomes (e.g., from a sequence of actions). This way, it also constitutes a crucial component in designing *rational agents*. Modern AI often qualifies intelligence as *acting rationally*, i.e., choosing the action that *maximizes the expected utility* given the available evidence. A decision-theoretic agent does so by combining the probability theory (to evaluate the likelihood of each outcome) and the utility theory (to express preferences regarding different outcomes), as it is higlighted in Figure 12.2. Hence, probabilistic reasoning forms also a core part of decision systems, like Markov Decision Processes (MDP) and their natural extension to reinforcement learning (RL).

The fundamental problems addressed by probabilistic reasoning often boil down to answering questions of the type *what is the probability of an event* (or a combination of events, or a sequence of states) *given some evidence*. Technically, we say that probabilistic reasoning finds the probability of a *query proposition* (like 'I'm having flue') given some evidence (e.g., symptoms like cough and fever). Similarly, we might be interested in the probability that 'it will rain tomorrow and the picnic won't be cancelled' given some meteorological data over the past days and some evidence about my friends' habits. Another type of problems that we encounter in probabilistic reasoning is that of finding an explanation or a sequence of most likely states of a given system, given a sequence of some observed features. In this context, the terms evidence, observations, observable variables, measurements, data and (observed) features have the same role in the reasoning process and are thus often used interchangeably and typically denoted generically as *evidence* in technical descriptions.

12.2 What categories of problems does probabilistic reasoning solve?

The problems that are being addressed by probabilistic reasoning can be categorized in different ways:

- *Static* vs. *Dynamic* environment. A task environment is said to be static when it doesn't change while the agent is deliberating. It means that the states of all the entities of interest for a given problem remain fixed (each random variable has a fixed value) in the time span in which the agent performs inference and decides on the action. For example, the fact that the

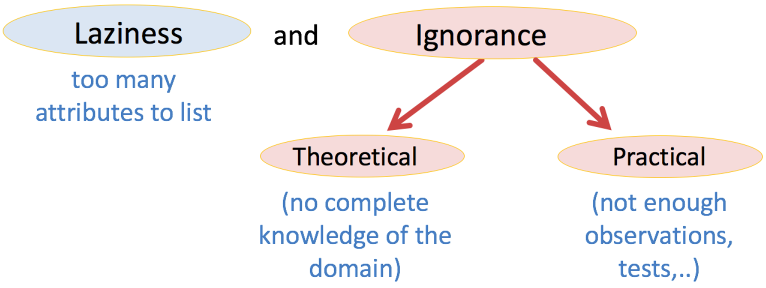

Laziness
too many attributes to list
and
Ignorance
Theoretical
(no complete knowledge of the domain)
Practical
(not enough observations, tests,..)

Figure 12.1: Sources of uncertainty that trouble purely logical and motivate probabilistic reasoning.

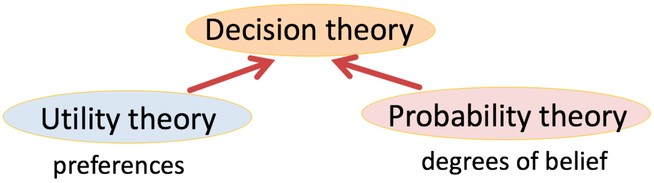


Figure 12.2: Decision-theoretic agents combine probability theory and utility theory.

alarm went off (or not) does not change while inferring the probability of burglary, and the state 'there was burglary' or 'not' also remains fixed all this time. In a dynamic environment, the states and the observations keep constantly changing while the agent is deciding about what to do next. Think of self-driving cars: the car itself and other cars keep moving, and overall, the traffic situation is changing while the driving algorithm decides on what to do next. To deal with dynamic problems we need reasoning over time, where we account for probabilities of various transitions between the states of the environment in subsequent time intervals. This type of problems will be addressed later in this chapter.

- *Causal* vs. *Non-causal* problems. Some models assume "causal" dependencies between variables, the influence of a variable over another one, represented in a graphical model by directed links. With directed graphical models we are describing the dependencies of the type "parent-child" where one influences the other in a particular direction: for example having flu causes with some probability headache, so *Flu* has a causal influence on *Headache* and this is expressed by a directed link between the two nodes in the model. Bayesian networks are often used to model this type of problems. In non-causal models, the interdependencies among the involved random variables are not directional, but are characterized symmetrically as neighbouring relations. For example, in a digital photo, neighbouring pixel intensities are statistically dependent. If pixel values are modelled as random variables, we can express their statistical dependencies with an undirected probabilistic graphical model, like Markov Random Field (MRF) or Conditional Random Field (CRF), which are covered in the subsequent sections.

- *Discrete* vs. *Continuous* (or *Hybrid*). The involved random variables can either be discrete or continuous. A discrete random variable can take only a finite number of distinct values, like being *true* or *false* or the number of free seats in a bus, while a continuous random variable follows some continuous distribution (e.g., current speed, body temperature, average price of a house etc.) In some cases, we have a mix of continuous and discrete random variables, which are then modelled by a hybrid probabilistic graphical model (e.g., a hybrid Bayesian network). Furthermore, if we are dealing with dynamic environments, the discrete/continuous characterization can also refer to how the time is handled: is it divided in some discrete steps or considered as a continuous variable.

In dynamic scenarios (*probabilistic reasoning over time*), the problems we address can be grouped in the following categories:

(i) Estimating the probability of a given state in a particular time instant, given the evidence available *before* (prediction) or *up to* (filtering) or *beyond* that time instant (smoothing). Say we don't know if the concentration of a given air pollutant, like PM2.5, in our city is above the standard but we can try to infer it based on some observations (like dense fog).

- Prediction makes the inference a step ahead. (E.g., 'Will the concentration of PM2.5 exceed the threshold *tomorrow* based on the observations until today?')
- Filtering makes the estimate for the present with all the evidence so far. ('Is PM2.5 *today* above the threshold based on the evidence so far?')
- Smoothing improves the estimate in the past based on the new evidence that arrives in the meantime ('Was PM2.5 above the threshold *yesterday* based on the evidence we received until now?').

(ii) Finding the most likely *sequence* of states given a sequence of observations – also called the *most likely explanation* (E.g., inferring for ten days in a row whether PM2.5 was above the threshold or not based on the fog observations in those ten days).

These concepts relating to reasoning in dynamic scenarios will be explained in more detail in the section "Probabilistic reasoning over time".

After introducing some basic background concepts that we need for solving any of these categories of the problems, we will first address probabilistic reasoning in a static setting. There, we will start from causal problems, which are well described by Bayesian networks and then we will turn to non-causal problems modelled by Markov Random Fields. In both of these, we will exemplify discrete and continuous or hybrid models. We will subsequently unify causal and non-causal models within a factor graph representation and we will introduce inference methods by belief propagation, Markov Chain Monte Carlo Samplers (MCMC) and briefly comment on some other inference mechanisms. Then we will turn to probabilistic reasoning over time, where we will start from basic inference problems in a dynamic setting, and explain how they are solved by belief propagation-type of methods. Then we will introduce the concepts of Hidden Markov Models, Dynamic Bayesian Networks and approximate inference by particle filtering.

12.3 Bayesian networks

Often we need to infer probability that some events arise given *causal* relationships among the involved random variables. Consider a somewhat simplified scenario from the Introduction to this Chapter, that we describe in the example below.

Example: *Late to meeting.*
Sharon has to leave for a meeting where she gives a demo. She can go either by car or by bicycle. By car, she will likely be on time unless there is huge traffic jam. It is even safer to arrive timely by bicycle, unless there is sudden heavy rain in which case she will have to look for a shelter on the way or at least do something about her hair before entering the meeting. The chairman typically decides herself to wait a while until everyone is present, but not always. Sharon's boss might decide to give the demo instead of her even if she is on time, and almost for sure if Sharon is late.

A possible way to model this problem is to use the Bayesian network given in Figure 12.3 together with some reasonable prior and conditional probability tables (CPTs). All the involved random variables here are discrete, and moreover Boolean. $P(C = true)$ models the probability to take the car, $P(J = true)$ models the probability of a traffic jam, $P(R = true)$ the probability of raining, while $P(L = true)$ models the probability that Sharon arrives late. Observe that random variables C, J and R influence the value of L, while the opposite is not true. Hence, C, J and R are *parents* of L, i.e., they provide *causal support* for L. Similarly, L is the parent of W and D.

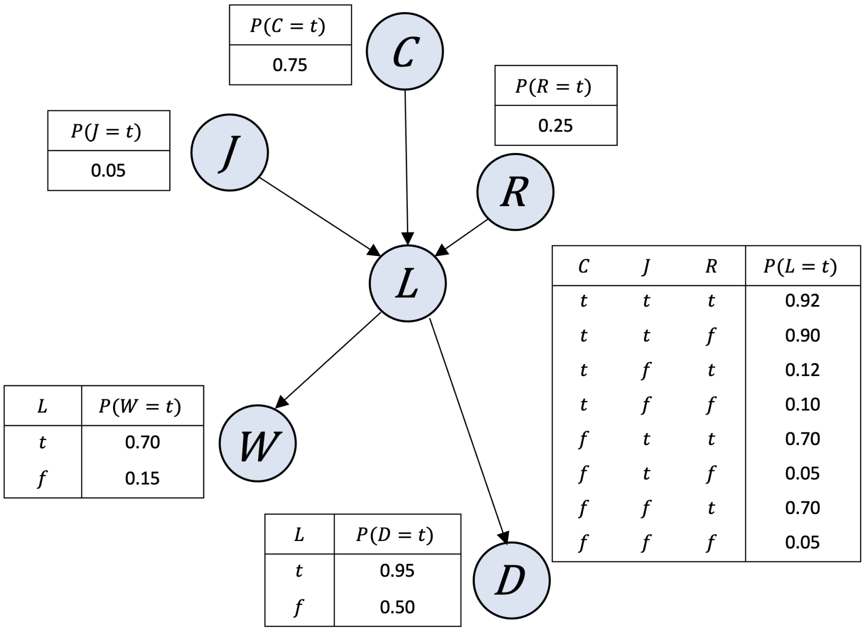


Figure 12.3: Bayesian network for the *Late to Meeting* problem. Random variable C denotes choice to go by car (if the value is true (t) or by bicycle if the value is false (f)). J denotes traffic jam, R heavy rain, L being late for the meeting, W that chairman suggests to wait and D that the boss decides to give the demo.

In constructing a Bayesian network for a particular problem, we need to make some reasonable assumptions about the independence and conditional independence relationships between the different variables in the model, in order to simplify the probabilistic representations of the environment. For example, in our model we assume that traffic jam and rain are statistically independent, but we could devise a more complex model stating that rain might impact the probability of traffic jam.

In general, a Bayesian network is a *directed* and *acyclic* probabilistic graphical model, expressing causal, i.e., *parent – child* relationships among the involved random variables. The joint probability of a Bayesian network $P(X_1 = x_1, \ldots, X_n = x_n)$ denoted for brevity by $P(x_1 \ldots x_n)$ is

$$P(x_1, ..., x_n) = \prod_{i=1}^{n} P(x_i| \; parents(X_i)) \tag{12.1}$$

where $parents(X_i)$ denotes the values of the parents of the node X_i that appear in $x_1, \ldots, x_n$. Observe that a node is *conditionally independent of its non-descendants given its parents*. E.g., in our network from Figure 12.3, D is conditionally independent of W given L, and D is also conditionally independent of J, C and R given L.

If we wouldn't know anything about the conditional independences among the nodes, we could only express the full joint probability, which we can always rewrite using the *chain rule*: $P(x_1, ..., x_n) = P(x_n|x_{n-1}, ..., x_1)P(x_{n-1}|x_{n-2}, ..., x_1) \ldots P(x_2|x_1)P(x_1) = \prod_{i=1}^{n} P(x_i|x_{i-1}, ..., x_1)$. We thus make use of the conditional independence assertions to reduce the full joint probability to the expression (12.1), where we in fact reduced the set $\{x_{i-1}, ..., x_1\}$ to $parents(X_i)$ only.

For the Bayesian network from Figure 12.3 we have that

$$P(j, c, r, l, w, d) = P(j)P(c)P(r)P(l|c, j, r)P(w|l)P(d|l) \tag{12.2}$$

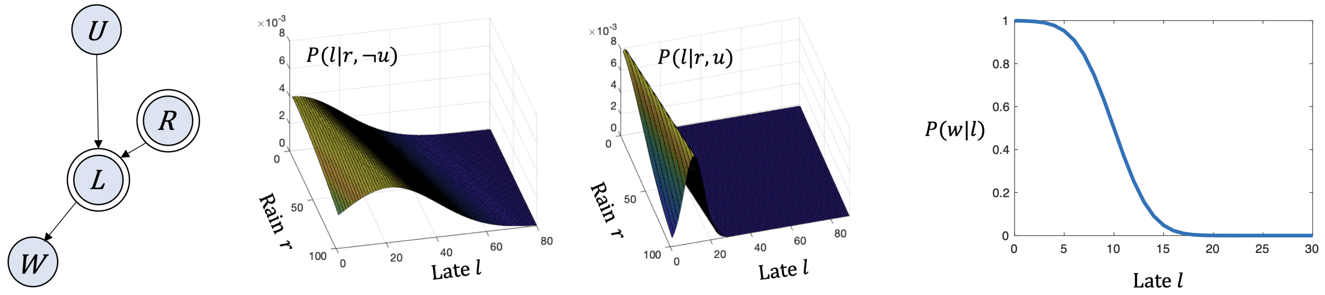


Figure 12.4: Left: A hybrid Bayesian network. Double circles denote continuous random variables. Middle: A linear Gaussian model for the probability distribution of L given its parents; Right: A probit model ($\mu = 20$; $\sigma = 3$) for the probability of W being true given the value of L.

Observe that when dealing with Boolean random variables, by convention we denote $P(J = true) = P(J = j) = P(j)$ and $P(J = false) = P(J = \neg j) = P(\neg j)$. Now if we are for example interested in the probability that Sharon went by bicycle and that she wasn't late while there was heavy traffic jam and no rain, and that chairman didn't wait for anyone and the boss gave the demo, we can calculate it easily as follows:

$$\begin{aligned} P(j, \neg c, \neg r, \neg l, \neg w, d) &= P(j)P(\neg c)P(\neg r)P(\neg l|\neg c, j, \neg r)P(\neg w|\neg l)P(d|\neg l) \\ &= 0.05 \times 0.35 \times 0.75 \times 0.95 \times 0.85 \times 0.5 = 0.053 \end{aligned} \tag{12.3}$$

We can also make *diagnostic* inference, e.g., the probability that Sharon went by car if the boss gave the demo and chairman didn't wait with opening the meeting, while it didn't rain and there was huge traffic jam: $P(c|j, \neg r, \neg w, d)$. Such probabilities can be calculated using the basic rules of probability and the given CPTs. However, for large Bayesian networks, we will need to use some efficient inference mechanism, like belief propagation [15].

We were dealing so far with the case where all the random variables were *discrete* and, in particular, Boolean. Often we have a mix of discrete and *continuous* random variables, and we are then modelling the problem with a *hybrid* Bayesian network. Let us modify our example as follows: Sharon goes on foot and she can take or not an umbrella, which is described by a Boolean random variable U. Instead of simply considering rain or not, we now model the intensity of rain by a continuous random variable R. Similarly, we replace the event of being late or not by a continuous random variable, which describes how much Sharon is late. The decision to wait or not at the beginning of the meeting naturally remains Boolean. This new situation is depicted in Fig. 12.4, where we use double circle to denote a continuous random variable. Now we have to specify two new types of the probability distributions, for a

- continuous random variable with a mix of continuous and discrete parents (node L), and a
- discrete random variable with continuous parents (node W).

The former one is typically described by a linear Gaussian model: $P(l|r, u) = \mathcal{N}(l; a_t r + b_t, \sigma_t^2)$; $P(l|r, \neg u) = \mathcal{N}(l; a_t r + b_t, \sigma_t^2)$, as in Fig. 12.4. Observe that the mean value changes linearly with the continuous parent value r and the parameters of this linear function as well as the spread of the distribution depend on the value of the discrete parent. The probability distribution of a discrete child given a continuous parent is in fact a soft-threshold function: in our example, let's say the chairman waits with high probability if the participants are late a couple of minutes and almost never if they are late more than 15 min, with some transition in between.

This is well modelled by a probit distribution:

$$P(w|Late = l) = 1 - \Phi((l - \mu)/\sigma)$$

where $\Phi(x) = \int_{-\infty}^{x} \mathcal{N}(x; 0, 1)dx$ (see Fig. 12.4, right) or alternatively by inverse logit models (using the inverse of a sigmoid function).

13 Bayesian and statistical machine learning

We studied so far machine learning and probabilistic reasoning as two important domains of AI, each with their own motivations, tasks and sets of techniques. Now we bring them together. On the one hand, we put machine learning into probabilistic reasoning in order to *learn* our probabilistic theories and models from experience (i.e., from data). On the other hand, we also bring probabilistic reasoning into machine learning to make the learning process more powerful, less susceptible to various imperfections in the data, less prone to overfit the data or base the results and decisions on a wrong, pre-selected model.

Bayesian machine learning views *learning* as a form of *uncertain reasoning from observations*. We can also say that it devises *models* to represent the *uncertain world* [18]. A key benefit that it brings to machine learning is a sound framework where we do not necessarily choose one *single* hypothesis (one single model) from a given hypothesis space (or model class) but *take each into account with its own probability.* Even if we resort to taking one hypothesis, we will do that by making use of statistical distributions over the various hypotheses and data, which will give us a more robust approach than with deterministic optimization techniques alone. Such a framework provides general solutions to dealing with noise, overfitting and ambiguities regarding what is an optimal prediction. It also copes well with the fact that an AI agent can rarely be certain about which model of the world is correct, yet it must make decisions and actions, and often in real time.

The term **Bayesian machine learning** strictly speaking refers to the class of techniques where we use priors over the models and make the prediction based on all of them. We will use the term **statistical machine learning** to refer to a wider class of approaches where we use probability distributions or statistical distributions of the data, but not necessarily making predictions based on all possible models from a given class and not necessarily using priors for these models. We adopt this terminology from [18].

As in the theory of learning, we deal with *data* and *hypotheses* but now they will be treated as *random variables* (*r.v.*s) or their realizations. The data are **evidence** – instantiations of some (or all) domain r.v.s, and hypotheses are **probabilistic models** (of how the domain "works").

13.1 Bayesian learning

Bayesian learning calculates the probability of each hypothesis, given the data. Predictions are made using *all* the hypotheses weighted by their probabilities rather than using a single "best" hypothesis. Learning becomes probabilistic inference!

Let the random variable **D** represent all the data, with observed value **d**. The (posterior) probability of the hypothesis h_j given data is:

$$\underbrace{P(h_j \mid \mathbf{d})}_{\substack{\text{posterior prob.}\\ \text{of hypotheis}}} = \alpha \underbrace{P(\mathbf{d} \mid h_j)}_{\text{likelihood}} \underbrace{P(h_j)}_{\substack{\text{hypothesis}\\ \text{prior}}} \tag{13.1}$$

The two key components are the **likelihood** of the data under each hypothesis $P(\mathbf{d} \mid h_j)$ and the **hypothesis prior** $P(h_i)$. Prediction about some unknown quantity X are now made as:

$$\mathbf{P}(X|\mathbf{d}) = \sum_j \mathbf{P}(X|h_j)P(h_j|\mathbf{d}) \tag{13.2}$$

which is a weighted average of the predictions of the individual hypotheses $\mathbf{P}(X|h_j)$, with weighting factors $P(h_j|\mathbf{d})$.

If the data are *independent identically distributed* (i.i.d), we have that

$$P(\mathbf{d}|h_j) = \prod_i P(d^{(i)}|h_j) \tag{13.3}$$

Remember that observations are i.i.d. if each example has the same prior probability distribution and is independent of other examples. In our notation, this means:

$\mathbf{P}(D^{(i)}) = \mathbf{P}(D^{(i+1)}) = \mathbf{P}(D^{(i+2)}) = \dots$ and
$\mathbf{P}(D^{(1)}, \dots, D^{(N)}) = \prod_{i=1}^{N} \mathbf{P}(D^{(i)})$

If we have a (temporal) sequence of random variables, where we are observing one sample after the other, the independence assumption can also be stated as $\mathbf{P}(D^{(i)}|D^{(i-1)}, D^{(i-2)}, \dots) = \mathbf{P}(D^{(i)})$.

13.1.1 “Surprise Candy” use case

We illustrate the concepts of statistical learning on the following example from [18]:

Our favorite surprise candy comes in two flavors: cherry (yum) and lime (ugh). The manufacturer has a peculiar sense of humor and wraps each piece of candy in the same opaque wrapper, regardless of flavor. The candy is sold in very large bags, of which there are known to be five kinds—again, indistinguishable from the outside:

h_1: 100% cherry
h_2: 75% cherry + 25% lime
h_3: 50% cherry + 50% lime
h_4: 25% cherry + 75% lime
h_5: 100% lime

Given a new bag of candy, the random variable H (hypothesis) denotes the type of the bag, with possible values $h \in \{h_1, \dots h_5\}$. As the pieces of candy are opened an inspected, data are revealed $D^{(1)}, D^{(2)}, \dots D^{(N)}$, where each $D^{(i)}$ is a random variable with possible values cherry *and* lime. *The basic task faced by the AI agent is to predict the flavor of the next piece of candy.*

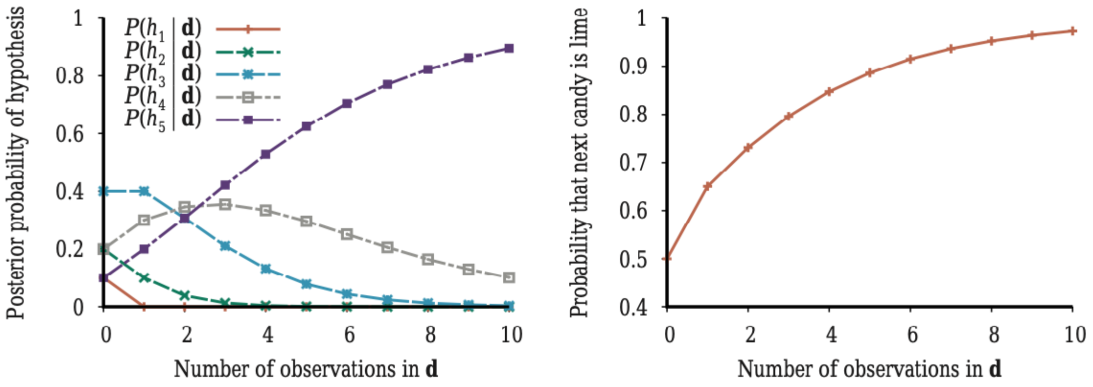


Figure 13.1: Bayesian learning example on the Surprise Candy case **Left**: Posterior probabilities $P(h_j|d^{(1)}, \dots d^{(N)})$. The number of observations N ranges from 1 to 10 and *each* observation is a *lime* candy. **Right**: Bayesian prediction $P(D^{(N+1)} = lime|d^{(1)}, \dots, d^{(N)})$. Figure from [18].

13.1.2 Bayesian learning example

We will now employ Bayesian learning to predict the flavor of the $(N + 1)$-th candy given the N opened ones. Hence, we apply (13.2), where the random variable to be predicted is now $X = D^{(N+1)}$:

$$\mathbf{P}(D^{(N+1)}|\mathbf{d}) = \sum_j \mathbf{P}(D^{(N+1)}|h_j)P(h_j|\mathbf{d}) \tag{13.4}$$

Since the candy is sold in very large bags, it is reasonable to assume that the data are i.i.d. (even if we don't re-wrap and return the opened candy back into the bag). Thus the likelihood is the product of the likelihoods for each separate candy as was given in (13.3). We still need the prior probabilities of h_j's. Suppose this prior distribution is known (e.g., was made pubic in the advertisements of the manufacturer) to be $\mathbf{P}(h_1, \dots, h_5) = \langle 0.1, 0.2, 0.4, 0.2, 0.1\rangle$.

With this, we have all what is needed to calculate the posterior probability of each h_i given the observed data **d** and, using that, also to predict the taste of the next candy. In Fig. 13.1, the posterior probabilities of the hypotheses and the probability of a given flavor (lime) of the next candy are shown for the different numbers of observations $N = 1, \dots, 10$, and assuming that each observation was *lime*.

The posterior probability plots on the left of Fig. 13.1 were obtained using Eq (13.1), where the hypothesis prior was as given above $\mathbf{P}(h_1, \dots, h_5) = \langle 0.1, 0.2, 0.4, 0.2, 0.1\rangle$, and where we used the i.i.d. assumption to express the likelihood as in Eq (13.3). For example, with $N = 10$ and all ten being *lime* observations, the data likelihood under the hypothesis h_3 is

$$P(\mathbf{d}|h_3) = \prod_{i=1}^{10} P(D^{(i)} = lime|h_3) = (0.5)^{10} \tag{13.5}$$

Observe that the initial values of the probabilities of the hypotheses (for $N = 0$) are their prior probabilities, so the probability for h_3 is initially the largest one. Already after the first candy is opened, the posterior probability of h_1 ("all cherry's") drops to zero, and h_3 is still the most probable hypothesis. After two candy's are open and both are lime, h_4 becomes the most probable, but h_5 is now only a little behind, its probability is increasing quickly. After ten all-lime candy's, h_5 is overwhelmingly the most probable hypothesis.

The probabilities of the prediction shown on the right Fig. 13.1 are calculated using Eq (13.2), (or concretely Eq (13.4), where $X = D^{(N+1)}$). There we use the already calculated posterior probabilities of the hypotheses $P(h_j|\mathbf{d})$, and $\mathbf{P}(D^{(N+1)}|h_j)$ is given by the problem description (e.g., $\mathbf{P}(D^{(N+1)}|h_1) = \langle 1, 0\rangle$ and $\mathbf{P}(D^{(N+1)}|h_2) = \langle 0.75, 0.25\rangle$). The probability plot agrees with what we would expect – the predicted probability that the next candy is lime is increasing monotonically toward 1.

13.2 Maximum a Posteriori and Maximum-Likelihood learning

The previous example illustrated a very important characteristic of the Bayesian learning: the *Bayesian prediction eventually (after sufficient number of observations) agrees with the true hypothesis*. What's nice is that this behavior does not depend on specifying a very accurate prior as long as it is reasonable enough – for any fixed prior that does not rule out the true hypothesis, the posterior probability of any false hypothesis will, under certain technical conditions, eventually vanish [18]. This is because it is not likely to generate repeatedly data that are "uncharacteristic" (they will be in negligible amounts in large datasets). Moreover, Bayesian prediction is *optimal regardless of whether the dataset is small or large*. Under a given prior, any other prediction will on the average be less correct.

While this optimality holds in theory, we might not always be able to use full Bayesian learning in practice. When the hypothesis space is very large this approach may be intractable. We then need to resort to approximate or simplified methods, and two common approaches are

- **Maximum a Posteriori** (**MAP**) learning:

$$h_{MAP} = \arg\max_{h\in\mathcal{H}} P(h|\mathbf{d}) = \arg\max_{h\in\mathcal{H}} P(\mathbf{d}|h)P(h)$$

- **Maximum-Likelihood** learning:

$$h_{ML} = \arg\max_{h\in\mathcal{H}} P(\mathbf{d}|h)$$

MAP learning (or MAP *estimation*) makes predictions based on a single most probable hypothesis. It is a commonly adopted approach in science. Maximum-likelihood approach can be seen as a simplification of the MAP learning by imposing a uniform prior on h. It is very common in statistics because many researchers distrust the subjective nature of hypothesis priors. Maximum-likelihood learning (also called maximum-likelihood estimation) is a good approximation for Bayesian and MAP learning with large data sets. This is because when the data set is large the prior is less important (evidence from data is strong enough to "swamp" the prior distribution) [18].

14 Learning with complete data

We address now the task of learning a probabilistic model from the data. This task is called **density estimation**[4] and is a form of *unsupervised learning*. In general, this problem refers to learning the entire probabilistic model (e.g., also the structure of a Bayesian network). We

[4] The term applied originally to probability density functions for continuous variables, but it is used now for discrete distributions too.

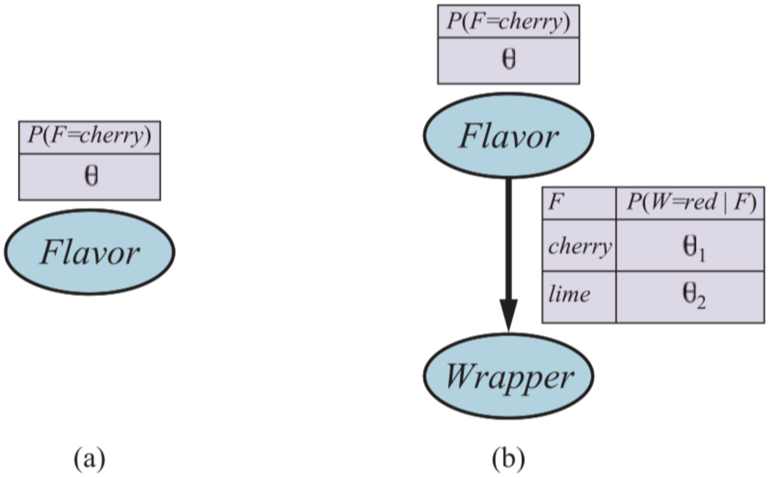


Figure 14.1: Bayesian network model for (a) **Example 1**: the case of candies with an unknown proportion of cherry and lime. (b) **Example 2**: the case where the wrapper color depends (probabilistically) on the candy flavor. Figure from [18].

will focus only on **parameter learning**, i.e., learning the values of the parameters of a given probability model whose structure is already known or fixed. In particular, we will address learning of the **conditional probability tables** in Bayesian networks (particularly focusing on the naive Bayesian model) and also learning the parameters of **continuous probability distributions** from the data. More detailed coverage, including learning Bayesian network structures and density estimation with non-parameteric models can be found compactly described in [18]. We assume that we have **complete data**, i.e., that each data point contains values for every variable in the probability model being learned. Learning with incomplete data (with hidden variables) is postponed to Section 15.

14.1 Maximum-likelihood parameter learning: Discrete models

We will explain the main concepts of the maximum-likelihood parameter learning for discrete models starting from two simple examples that build on the *Surprise Candy* use case from Section 13.1.1.

14.1.1 Example 1: A uniform prior on the hypotheses

Consider first a small generalization of the *Surprise Candy* use case as follows. We don't have any more the earlier five upfront defined hypotheses but instead the fraction of cherry is a **parameter** $\theta \in [0, 1]$, and the hypotheses are now h_θ. We assume that all proportions of the two candy flavors are equally likely *a priori* and we want to model this situation with a Bayesian network.

There is only one relevant random variable for this model: $Flavor$. The corresponding "Bayesian network" (an extreme case with one r.v.) is shown in Fig. 14.1 (a) along with the involved parameters, which is here only one parameter characterizing the prior probability $P(Flavor = cherry) = \theta$.

Let us now learn the parameter θ from data. Suppose we unwrap N candies of which c are cherry and $l = N - c$ are lime. The data likelihood given h_θ is

$$P(\mathbf{d}|h_\theta) = \prod_{i=1}^{N} P(d^{(i)}|h_\theta) = \theta^c(1-\theta)^l \tag{14.1}$$

The corresponding log-likelihood is

$$\ell(\theta) = \log P(\mathbf{d}|h_\theta) = \sum_{i=1}^{N} \log P(d^{(i)}|h_\theta) = c\log\theta + l\log(1-\theta) \tag{14.2}$$

By setting $d\ell(\theta)/d\theta = 0$ we obtain

$$\theta = \frac{c}{c+l} = \frac{c}{N} \tag{14.3}$$

i.e., $h_{ML} = h_{c/N}$. Thus, as we would expect, the maximum-likelihood learning asserts that the actual proportion of cherry is the same as the observed proportion.

14.1.2 Example 2: Generalization to two attributes

Now we build further on the previous example. Suppose there are two different wrapper colors: *red* and *green.* The wrapper color is selected for each candy depending on its flavor according to some unknown probability distribution. We want to model this new situation with a Bayesian network and to learn its parameters from the data.

First we observe that there are now two relevant r.v.s: $Flavor$ and $Wrapper$, where $Wrapper$ depends on $Flavor$, Thus we can represent this model with the Bayesian network shown in Fig. 14.1(b). Since all the r.v.s are Boolean, we have three parameters: θ, θ_1 and θ_2, where θ characterizes as before the prior probability of $Flavor$, while θ_1 and θ_2 characterize the conditional probability table (CPT) $\mathbf{P}(Wrapper|Flavor)$.

For compactness, we denote the two r.v.s with their first letters, so we the CPT is given by $P(W = red|F = cherry) = \theta_1$ and $P(W = red|F = lime) = \theta_2$. (Note that the probability of *green* given *cherry* is simply $1 - \theta_1$ and the probability of *green* given *lime* is $1 - \theta_2$).

We express the joint probability of this Bayesian network conditioned on the parameter values θ, θ_1 and θ_2. For example:

$$\begin{aligned} & P(F = cherry, W = green|h_{\theta,\theta_1,\theta_2}) \\ = \; & P(F = cherry|h_{\theta,\theta_1,\theta_2})P(W = green|F = cherry, h_{\theta,\theta_1,\theta_2}) = \theta(1-\theta_1) \end{aligned}$$

To learn the parameter values, we unwrap N candy's; c of these are cherry and l are lime. r_c of the cherry candy's have red wrappers and g_c green. Similarly, r_l of the lime candy's have red wrappers and g_l green wrappers.

The likelihood of the data is

$$P(\mathbf{d}|h_{\theta,\theta_1,\theta_2}) = \theta^c(1-\theta)^l \cdot \theta_1^{r_c}(1-\theta_1)^{g_c} \cdot \theta_2^{r_l}(1-\theta_2)^{g_l}$$

Setting the partial derivatives of the log-likelihood $\ell(\theta, \theta_1, \theta_2) = \log P(\mathbf{d}|h_{\theta,\theta_1,\theta_2})$ to zero yields

$$\theta = \frac{c}{c+l}, \quad \theta_1 = \frac{r_c}{r_c + g_c}, \quad \theta_2 = \frac{r_l}{r_l + g_l}$$

This example shows us that with complete data, the maximum-likelihood parameter learning problem for a Bayesian network decomposes into separate learning problems, one for each parameter!

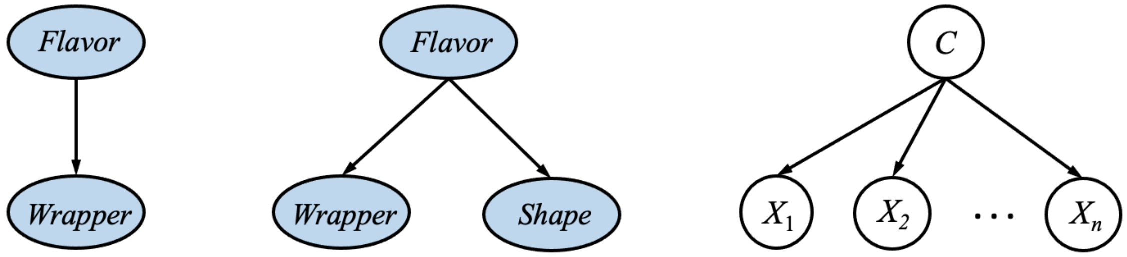


Figure 14.2: From left to right: two instances of a naive Bayes model and its general structure with a random variable C denoting some class, and n attributes.

14.1.3 Naive Bayes model

We could extend the previous example by adding another attribute, say *Shape* of the candy (e.g., being *round* or *square*), again depending on the candy's flavor. If this new attribute is conditionally independent of the wrapper color, given the flavor, we would represent the new situation with the second Bayesian network in Fig. 14.2. We can continue adding more and more attributes and as long as they are conditionally independent given $Flavor$, all the resulting models are instances of the **naive Bayes** model.

In general, in a naive Bayesian model, the **class** variable C is the root (to be predicted), and X_i are the **attributes** (**features**). We call it also **naive Bayes classifier**. Recall that we already introduced before the naive Bayesian model, when we dealt with basics of probabilistic reasoning, and at that time we referred to the root as the **cause** and the leaves as **effects** (or **symptoms**).

We call this model **naive** because of its assumption that the *attributes are conditionally independent given the class* (or put in the words of the other terminology, it's naive because of its assumption that the effects are conditionally independent given the cause).

The joint probability of a naive Bayesian model is given by

$$\mathbf{P}(C|x_1, \ldots, x_n) = \alpha \mathbf{P}(C) \prod_j \mathbf{P}(x_j|C) \tag{14.4}$$

In the case where all r.v.s are Boolean we need one parameter for the class and two per attribute:

$$\theta = P(C = 1), \quad \theta_{j1} = P(X_j = 1|C = 1), \quad \theta_{j2} = P(X_j = 1|C = 0) \tag{14.5}$$

The maximum-likelihood estimation of these parameters is exactly the same procedure as was explained in **Example 2**. Let $(\mathbf{x}^{(i)}, c^{(i)})$ be teh ith data point. We have that

$$\theta_{jk} = \frac{\sum_i \mathbb{1}[x_j^{(i)} = 1 \ \wedge \ c^{(i)} = k]}{\sum_i \mathbb{1}[c^{(i)} = k]} \tag{14.6}$$

where $\mathbb{1}[a] = 1$ if $a = true$ and 0 otherwise. In **Example 2**, we had only one attribute ($j = 1$), so the parameter θ_1 from that example is in this general notation now θ_{11} and Eq (14.6) gives us:

$$\theta_{11} = \frac{\#[W = red \ \wedge \ F = cherry]}{\#[F = cherry]} = \frac{r_c}{r_c + g_c}$$

in this particular example, as we obtained before.

Example: text classification

Let's consider now as an example a use case which is closer to real applications than the candy flavor prediction: **text classification**[5]:

The task is to classify each sentence into a category that corresponds to one of the major sections of the newspaper: news*,* sports*,* business*,* weather *or* entertainment*.*

The class variable $Category$ takes values $c \in \{news, sports, business, weather, entertainment\}$ and there are n Boolean attributes $HasWord_i$, $i = 1, \ldots, n$, where a predefined table of n words is given. The model is characterized by the prior probabilities $\mathbf{P}(Category)$ and the conditional probabilities $\mathbf{P}(HasWord_i|Category)$.

The prior probability of each category is estimated as the fraction of all previously seen documents that belong to this category (for example, if 15% of articles are about sports, we set $P(Category = sports) = 0.15$. Similarly, $P(HasWord_i|Category = sports)$ is estimated as the fraction of the documents in the sports category that contain word i (e.g., if 18% of the seen articles about sports contain word 5, "fans" we set $P(HasWord_5 = true|Category = sports) = 0.18$). Actually, we are estimating the parameters of the CPT's using Eq (14.6).

To categorize a new document, we check which key words appear in the document and then apply Eq (14.4) to obtain the posterior probability distribution over categories. If we have to predict just one category, we take the one with the highest posterior probability.

Properties

Naive Bayes is a commonly used model in machine learning due to its nice properties:

- scales well to very large problems (e.g., with n Boolean attributes, only $=2n+1$ parameters);
- deals well with noisy or missing data;
- gives both probabilistic and deterministic prediction (by choosing the most likely class);
- it is intuitive, simple to implement and performs well in a wide range of applications;
- its boosted version is one the most effective general-purpose learning algorithms.

Next to all these advantages, there are also some downsides. The main drawback is that the conditional independence assumption is seldom accurate, although in practice it is often a good approximation. When this approximation is not well justified, the model can lead to wrong predictions with overconfident probabilities (close to 0 or 1), especially when the number of attributes is large.

14.1.4 Generative and discriminative models

Two kinds of machine learning models are used for classifiers

- **Generative models** – model the probability distribution of each class. From these distributions we can compute the **joint probability** and (by sampling from this joint distribution) we can **generate new examples** that are representative of each class. A representative is the **naive Bayes** model.

[5] Example from [18], chapter 12.

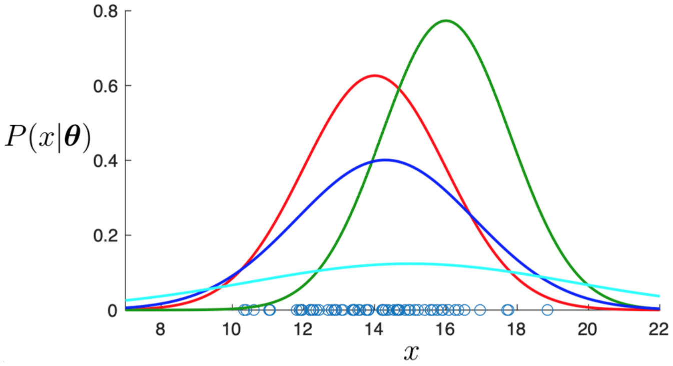


Figure 14.3: An illustration of the maximum-likelihood parameter estimation. Circles on the horizontal axis are the measurements and four Gaussian distributions, with different parameters $\boldsymbol{\theta} = (\mu, \sigma^2)$, are shown as candidates to fit the true distribution. (The data were actually generated from the distribution with $\mu = 14$ and $\sigma = 2$, which corresponds to the curve shown in red, so the maximum-likelihood estimation should identify this distribution as the best fit.)

- **Discriminative models** – learn the decision boundary between classes. A discriminative model can learn to classify a new input to the correct class, but cannot generate new examples from that class. Representatives are **logistic regression**, **decision trees**, and **support vector machines**.

To understand well the distinction between the two types of models and why the first one can generate the new examples and the other not, consider the task of text categorization described in Section 14.1.3. The naive Bayes classifier creates a separate model for each possible category of text, given by the prior probability, e.g. $P(Category = weather)$ and the conditional probabilities $\mathbf{P}(Inputs|Category = weather)$. From these, we can compute the joint probability, e.g., $\mathbf{P}(Inputs, Category = weather)$ and we can generate a random selection of words that is representative of texts in the weather category [18]. This makes the model *generative.* In the same text categorization task, a discriminative model would learn directly the posterior probability of the class given the inputs, that is, $\mathbf{P}(Category|Inputs)$, which serves directly the classification task but from which we cannot generate new example inputs.

Discriminative models tend to perform better in the classification tasks on very large data sets but on very small data sets generative models often do better.

14.2 Maximum-likelihood learning: Continuous models

Now we address the task of learning the parameters $\boldsymbol{\theta}$ of some continuous probability distribution $P(\mathbf{x}|\boldsymbol{\theta})$ from the observed data $\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(N)}$. Consider first the simplest case, illustrated in Fig. 14.3, where data are one-dimensional and the probability distribution is Gaussian:

$$P(x|\boldsymbol{\theta}) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

To learn the parameters $\boldsymbol{\theta} = \{\mu, \sigma^2\}$, we express the log-likelihood:

$$\ell(\mu, \sigma) = -\frac{N}{2}\log(2\pi) - \frac{N}{2}\log(\sigma^2) - \frac{1}{2\sigma^2}\sum_{i=1}^{N}(x_i^{(i)} - \mu)^2$$

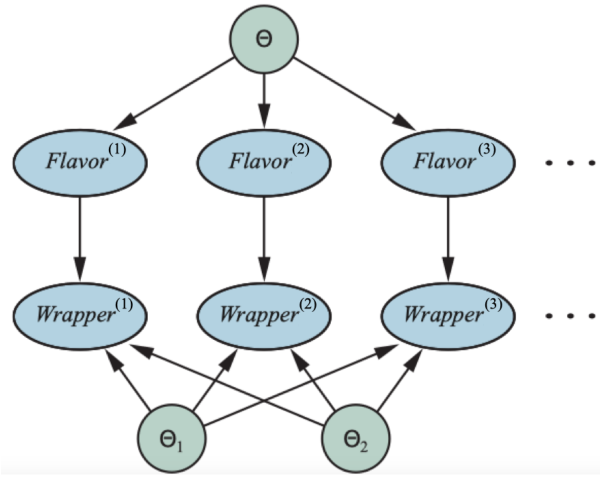


Figure 14.4: A Bayesian network that corresponds to a Bayesian learning process. Posterior distributions for the parameter variables Θ, Θ_1 and Θ_2 can be inferred from their prior distributions and the evidence in $Flavor^{(i)}$ and $Wrapper^{(i)}$. Figure from [18] with adapted notation.

From $\partial\ell/\partial\mu = 0$ and $\partial\ell/\partial\sigma = 0$ we obtain:

$$\hat{\mu} = \frac{1}{N}\sum_{i=1}^{N} x^{(i)} \quad \text{and} \quad \hat{\sigma}^2 = \frac{1}{N}\sum_{i=1}^{N}(x^{(i)} - \hat{\mu})^2$$

which are exactly the sample mean and the sample variance.

We can generalize this procedure to the case where our observations are vectors $\mathbf{x}^{(i)}$ (i.e., each measurement is not a single number but consists of d components $\mathbf{x}^{(i)} = [x_1^{(i)}, \dots x_d^{(i)}]$) modelled by a *multivariate* normal distribution $\mathbf{x} \sim \mathcal{N}(\mathbf{x}; \boldsymbol{\mu}, \boldsymbol{\Sigma})$, with mean $\boldsymbol{\mu}$ and the covariance matrix $\boldsymbol{\Sigma}$. Applying the same procedure as above, we obtain the maximum-likelihood estimates of the parameters as

$$\hat{\boldsymbol{\mu}} = \frac{1}{N}\sum_{i=1}^{N} \mathbf{x}^{(i)}, \quad \text{and} \quad \hat{\boldsymbol{\Sigma}} = \frac{1}{N}\sum_{i=1}^{N}(\mathbf{x}^{(i)} - \hat{\boldsymbol{\mu}})(\mathbf{x}^{(i)} - \hat{\boldsymbol{\mu}})^\top \tag{14.7}$$

These simple examples with the normal distribution are easy to interpret, but we can apply the same procedure to estimate the parameters of other, arbitrary statistical models.

14.3 Bayesian parameter learning

Maximum-likelihood learning is not reliable with small data sets. The Bayesian approach to parameter learning starts with a prior distribution for the hypotheses and **updates this distribution as data arrive**.

In the candy case, the parameter θ was representing the probability that a randomly selected piece of candy is of cherry flavor. In the Bayesian view, θ is a realization of a random variable Θ that defines the hypothesis space. The hypothesis prior is the prior distribution over $P(\Theta)$. Thus, $P(\Theta = \theta)$ is the prior probability that the bag has a fraction θ of cherry candies.

In **Example 1** and **Example 2** in Section 14.1, we simply used a uniform prior for θ, i.e., $P(\theta) = Uniform(\theta; 0, 1)$. This is because we reasoned that we don't know anything about the

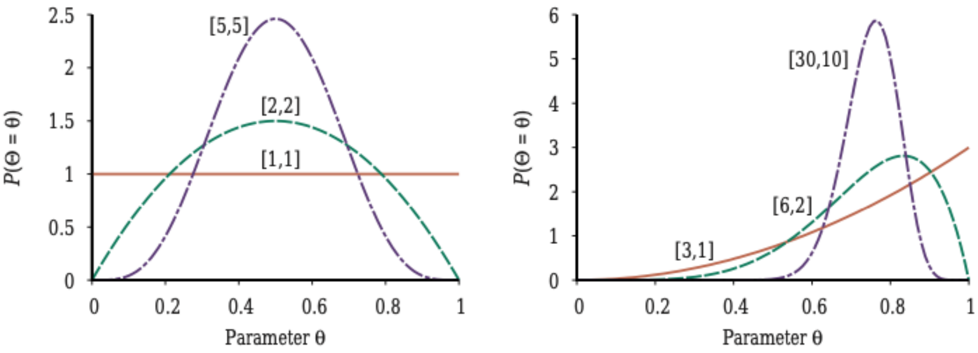


Figure 14.5: Examples of the $Beta(a, b)$ distribution for different values of (a, b). Figure from [18].

possible values of θ, so we let them all be equally likely. Now we rather assume some more general, parametrized distribution that is flexible enough and whose parameters can be updated based on the new data. This will allow us to continuously adjust the prior distribution with new data. Similarly, we will do for all the involved parameters that define the hypotheses. This process is illustrated in Fig. 14.4.

A very convenient distribution for this purpose is the **Beta** distribution:

$$Beta(\theta; a, b) = \alpha\ \theta^{(a-1)}(1-\theta)^{(b-1)}$$

for θ in the range $[0, 1]$. Its mean value is $a/(a+b)$, so larger values of a suggest a belief that Θ is closer to 1 than to 0. Larger values of $a+b$ make the distribution more peaked, suggesting greater certainty about the value of Θ. Fig. 14.5 illustrates the shape of this distribution with different parameters. Note that $Beta(1, 1)$ is a uniform distribution on the interval $[0, 1]$.

The beta family has a very useful property for the learning process: it is **closed under update**, meaning that the distribution updated with the new data remains the *Beta* distribution, just with the updated parameters.

To see how this updating of the hypothesis prior works, suppose we start from the distribution $P(\theta) = Beta(\theta; a, b)$ with some chosen values of a and b, and we unwrap the first candy. If this first candy is cherry, the update will be as follows:

$$\begin{aligned}P(\theta \mid D^{(1)} = cherry) &= \alpha P(D^{(1)} = cherry \mid \theta)P(\theta)\\ &= \alpha\ \theta \cdot Beta(\theta; a, b) = \alpha'\theta \cdot \theta^{(a-1)}(1-\theta)^{(b-1)}\\ &= \alpha'\theta^{a}(1-\theta)^{(b-1)} = \alpha' Beta(\theta; a+1, b)\end{aligned}$$

Thus, seeing a cherry candy leads to incrementing the a parameter of the distribution. Effectively, this increases the belief that θ, which represents the portion of cherry, is closer to one than to zero. Conversely, observing a lime candy will lead to incrementing the b parameter and this way shifting our belief towards smaller values of θ. Let's show this by repeating the calculation above for the case where our first unwrapped candy was lime:

$$\begin{aligned}P(\theta \mid D^{(1)} = lime) &= \alpha P(D^{(1)} = lime \mid \theta)P(\theta)\\ &= \alpha\ (1-\theta) \cdot Beta(\theta; a, b) = \alpha'(1-\theta) \cdot \theta^{(a-1)}(1-\theta)^{(b-1)}\\ &= \alpha'\theta^{a-1}(1-\theta)^{b} = \alpha' Beta(\theta; a, b+1)\end{aligned}$$

We learn this way the distribution parameters from the observed data. The process also has a nice interpretation. We can view the hyperparameters a and b as "virtual counts": $Beta(a, b)$, with some particular values for a and b, would be obtained if we had started from a uniform prior $Beta(1, 1)$ and gradually updated it after observing $a - 1$ cherry candies and $b - 1$ lime candies.

15 Learning with hidden variables

In practice, data entries are often missing resulting in incomplete information to specify a likelihood. This complicates the learning of the model parameters. We make a differentiation between the data that are observable but just not available in some data points (e.g., due to a particular sampling pattern or sensor errors etc.) and data that are never directly observable – these correspond to the so called **hidden** or **latent variables**.

Latent variables are random variables that are essential for the model description but never directly observed. The following definition can be found on Wikipedia:
In statistics, latent variables (from Latin: present participle of lateo, "lie hidden") are variables that can only be inferred indirectly through a mathematical model from other observable variables that can be directly observed or measured [5].

For example, the underlying physics of a model may contain latent processes which are essential to describe the model, but cannot be directly measured. Remember also that we sometimes introduce hidden variables to **sparsify** the structure of a Bayesian network. This way we can dramatically reduce the number of parameters required to specify a Bayesian network. We will focus on a particular type of latent variable models: mixture models that we address next.

15.1 Clustering by learning Gaussian mixture models

In general, statistical models that represent distributions of observable variables using latent (or hidden) variables are called **latent variable models**. An important class of these models are the so-called **mixture models**, where the probability distribution is a mixture of k **components**:

$$P(\mathbf{x}) = \sum_{i=1}^{k} \underbrace{P(C = i)}_{\pi_i} P(\mathbf{x}|C = i) \tag{15.1}$$

Here, C is a random variable that denotes the component, with values $1, \ldots, k$ and $\pi_i = P(C = i)$ is the weight given to the ith component in the mixture. A mixture model naturally represents (soft) clustering of data. We thus also say that mixture models make use of latent variables to model different parameters for different groups (or **clusters**) of data points

When all the components in the mixture model are Gaussian, we have a mixture of Gaussians, also called the **Gaussian mixture model** (**GMM**):

$$P(\mathbf{x}) = \sum_{i=1}^{k} \pi_i \, \mathcal{N}(\mathbf{x}; \mu_i, \Sigma_i) \tag{15.2}$$

Fig. 15.1 illustrates data generated from a GMM model, and also fitting the model to data with an algorithm that will be explained in Section 15.2.

Mixture models are very useful in practice. Consider this example from [6]:

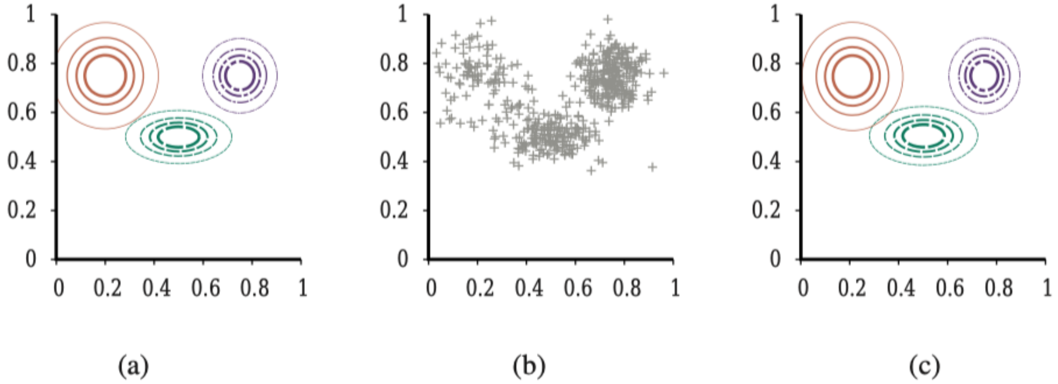


Figure 15.1: A Gaussian mixture model with three components; the weights (left-to right) are 0.2, 0.3, and 0.5. (b) 500 data points sampled from the model in (a). (c) The model reconstructed by EM from the data in (b). Figure from [18].

We need to build a probabilistic language model of news articles, which assigns probabilities to sequences of words $\mathbf{x}_1, \ldots, \mathbf{x}_n$*. Each article typically focuses on a specific topic e.g., finance, sports, politics. Using this prior knowledge, we may build a more accurate model* $P(\mathbf{x}|c)P(c)$*, in which we have introduced an additional, unobserved random variable* C*. This model can be more accurate, because we can now learn a separate* $P(\mathbf{x}|c)$ *for each topic, rather than trying to model everything with one (very complex)* $P(\mathbf{x})$*.*

Note certain similarities with the description of the naive Bayes use case for text classification in Section 14.1.3. However, since c is now unobserved, we cannot directly use the learning methods that we have studied in Section 14 but we will introduce a new learning algorithm that we will explain particularly for learning the mixtures of Gaussians.

15.2 EM algorithm for the mixtures of Gaussians

Fitting a GMM model to data like in Fig. 15.1 is in essence a **clustering problem**. Each cluster will be modelled by a distribution (in this particular case, a Gaussian of given mean and covariance) and we will obtain the probabilities describing how likely it is that the data point belongs to a given cluster. This is a form of **soft clustering** but we can obtain easily from it also **hard clustering** (e.g., simply by assigning the data point to the component (cluster) for which the probability is the largest.

The parameters of this model are effectively learned by the so-called **expectation-maximization** (**EM**) algorithm. The basic idea of EM is to start from some initial parameters of the model and to iterate two steps: (1) infer the probability that each data point belongs to each component. (2) refit the components to the data, where each component is fitted to the entire data set with each point weighted by the probability that it belongs to that component.

Concretely, the EM algorithm for GMMs iterates the following two steps [18]:

- **E-step**: *Compute probabilities* $p_{ij} = P(C = i|\mathbf{x}^{(j)})$
  - By Bayes' rule:
    $p_{ij} = \alpha P(\mathbf{x}^{(j)}|C = i)P(C = i)$

  - Define n_i as the effective number of data points assigned to component i:
    $n_i = \sum_j p_{ij}$

- **M-step**: *Compute the new mean, covariance and weights*
  - Means:
    $\mu_i \leftarrow \sum_j p_{ij}\mathbf{x}^{(j)}/n_i$
  - Covariance matrices:
    $\Sigma_i \leftarrow \sum_j p_{ij}(\mathbf{x}^{(j)} - \mu_i)(\mathbf{x}^{(j)} - \mu_i)^\top/n_i$
  - Weights:
    $\pi_i \leftarrow n_i/N$

GMM-based clustering can be seen as a probabilistic version of the deterministic clustering method known as **K-means**.

K-means algorithm initializes (randomly) the **centroids** $\mu_1, \ldots, \mu_k$ and iterates two steps:

1. Assign each point to the nearest (in terms of L_2 distance) centroid:

$$\forall j, \quad c^{(j)} = \arg\max_{i=1,\ldots,k} \|\mathbf{x}^{(j)} - \mu_i\|^2$$

2. Recompute the centroids based on the assigned points: $i = 1, \ldots, k$:

$$\mu_i \leftarrow \frac{1}{|\{j : c^{(j)} = i\}|} \sum_{j:c^{(j)}=i} \mathbf{x}^{(j)}$$

K-means is sometimes employed to initialize GMM-based clustering.

16 Appendix: Basic concepts in probabilistic reasoning

In probabilistic reasoning, we are dealing with randomness, arising because the process (or a "trial") in whose outcomes we are interested *didn't happen yet* (e.g., tomorrow's match) or the *already existing outcome is uncertain* (e.g., due to various measurement imprecisions). Here we review briefly the basic principles of probability and we explain the notation that we will use.

Random events and axioms of probability. The space of all possible outcomes of a given trial is called the *sample space.* Let Ω be this sample space and $\omega \in \Omega$ its elements. A random **event** is any subset of the sample space $\theta \subset \Omega$ and its elements ω are *atomic events.* Take as an example rolling a die. In this case, $\Omega = \{1, 2, 3, 4, 5, 6\}$. An event "die roll > 4" is a subset $\{5, 6\} \subset \Omega$, which contains atomic events 5 and 6. The basic principles underpinning the probability theory are: *the probability of each outcome lies between* 0 *and* 1: $0 \leq P(\omega) \leq 1$, and *the probabilities of all the possible outcomes sum up to* 1: $\sum_{\omega\in\Omega} P(\omega) = 1$. Furthermore, *the probability of any event is the sum of the atomic events where this event is true*: $P(\theta) = \sum_{\omega\in\theta} P(\omega)$. E.g., assuming a fair die, the probability of each outcome is $1/6$, their sum is 1, and $P(\text{die roll} > 4) = P(5) + P(6) = 1/6 + 1/6 = 1/3$. These three principles are the **axioms of probability**. From these three axioms, all other rules that hold in the probability theory can be derived, including the one which says that the probability of a union of two events is the sum of their probabilities minus the probability of their intersection: $P(a \vee b) = P(a) + P(b) - P(a \wedge b)$.

Random variables versus events. What is the difference between events and random variables? An event, as was defined above, corresponds to the term *proposition* in logical reasoning, and informally it can be described as any meaningful statement about the experiment we are interested in (e.g., "die roll > 4" or "die roll is an even number", etc.) Thus, an event happens or not with some probability, while a **random variable** is a variable whose value is affected by some random phenomenon. Typically, the values of random variables are *real numbers* but in some cases also Boolean values $\{true, false\}$. In the die roll experiment, we can define the event "die roll > 4" as a random variable X whose possible values are *true* or $false$, or if we define X as the outcome of rolling, its possible values are $\{1, 2, 3, 4, 5, 6\}$. In general, we will denote by a capital letter a random variable, e.g., X and by the corresponding small letter its particular value that we call *realization*, e.g., x. We use boldface letters to denote vectors of random variables $\mathbf{X} = (X_1, \ldots X_n)$ and their realizations $\mathbf{x} = (x_1, \ldots x_n)$. We also say that $X = x$ is an event where the random variable X takes the value x.

Discrete random variables. In the examples above, we illustrated *discrete* random variables, which may take on only a countable (finite or infinite) number of distinct values. Another example of a discrete random variable is a class label, where x is one of the possible classes. The probability that X takes the value x is commonly denoted by $P(X = x)$ or by $P_X(x)$. Also, for compactness, we will use sometimes only $P(x)$ when there can be no confusion to what this refers. We apply the same convention to the vectors of random variables $P(\mathbf{X} = \mathbf{x}) = P_{\mathbf{X}}(\mathbf{x})$, and for short by $P(\mathbf{x})$ when no confusion is possible. Boldface $\mathbf{P}$ in $\mathbf{P}(X)$ (or in $\mathbf{P}(\mathbf{X})$) denotes a vector where each entry is the probability of a particular realization of X (or $\mathbf{X}$).

Continuous random variables. A *continuous* random variable can take infinitely many values. Examples are height or weight of a person, air temperature, distance covered or blood sugar level. Continuous random variables are characterized by the **probability density function** (pdf), also called *density*, $p_X(x)$. The pdf integrates to 1: $\int_{-\infty}^{\infty} p_X(x) = 1$, and the *cumulative* function is

$$F_X(x) = P(X \leq x) = \int_{-\infty}^{x} p_X(x)dx \tag{16.1}$$

In the problems that we addressed in this chapter, the data distribution is typically a continuous random variable and the labels associated with each data sample are discrete random variables.

Prior and conditional probabilities. Prior probabilities express a priori belief (about the value of some random variable), without any (new) evidence (i.e., without any measurements or observations). For example, let R be the random variable expressing "it will rain tomorrow", with two possible values $r \in \{true, false\}$. Then the prior probability $P(R = true) = 0.3$ is reasonable in July in Antwerp (30% of days are rainy), and $P(R = true) = 0.04$ in Lisabon. *Conditional* probabilities express *belief given some evidence.* For example, we know the current value of barometric air pressure B (in mbar) in Antwerp, and we refine the probability of rain as: $P(R = true|B = 992.21) = 0.43$. Both for prior and conditional probabilities, we often use the more compact representation, same as we explained above, e.g., $P(R = r|B = b) = P_{R|B}(r|b)$, and for a conditional probability density function $p_{B|R}(b|r)$ (since B is a continuous random variable), and when no confusion possible we also use a shorter notation, like $P(r|b)$ or $p(b|r)$. Formally, we define the conditional probability of x given y as

$$P(x|y) = \frac{P(x \wedge y)}{P(b)} \quad \text{if} \quad P(y) \neq 0 \tag{16.2}$$

The *product rule* gives an alternative formulation: $P(x \wedge y) = P(x|y)P(y) = P(y|x)P(x)$, and successive application of the product rule gives the well-known **chain rule**:

$$P(x_1, \ldots x_n) = P(x_n|x_1, \ldots, x_{n-1})P(x_{n-1}|x_1, \ldots, x_{n-2}) \ldots P(x_1) = \prod_{i=1}^{n} P(x_i|x_1, \ldots, x_{i-1}) \tag{16.3}$$

For example, $P(x_1, x_2, x_3) = P(x_1)P(x_2|x_1)P(x_3|x_2, x_1)$. This chain rule has many practical uses and we will see later on how its special case applies to Bayesian networks.

Statistical independence & conditional independence. Random variables X and Y are statistically independent if $P(x|y) = P(x)$ or equivalently $P(y|x) = P(y)$ or equivalently $P(x \wedge y) = P(x)P(y)$. The statistical independence, when it holds, simplifies a lot the probabilistic modelling and inference: if we have n binary random variables, 2^n parameters are needed in general to describe their joint probability distribution, and if they are all statistically independent, this amount is only n. Hence, statistical independence reduces the amount of the parameters exponentially. Unfortunately, strict statistical independence among the random variables of interest may not hold in practice. However, *conditional* independence often applies to some subsets of the involved random variables. For example, low barometric pressure increases the chance of rain and headache, but the probability that it will rain is conditionally independent of whether one has headache or not, given that the air pressure is known: $P(rain|lowpressure, headache) = P(rain|lowpressure)$.

Bayes' rule. In probabilistic reasoning, the *Bayes'* rule (also known as the Bayes' theorem), named after Thomas Bayes (1701–1761), has a central place as it connects the *diagnostic* inference (inferring the cause given the effects) to the *causal* inference (the probability of effects given the cause). Formally, the Bayes's rule is:

$$P(x|y) = \frac{P(y|x)P(x)}{P(y)} \tag{16.4}$$

Observe that this expression follows directly from the expression for the conditional probability (16.2) and the product rule that was written below it. The true value of the Bayes' rule is that it facilitates the inference. Think of medical diagnosis, where $P(x|y)$ is the probability of disease x given symptoms y. Inferring $P(x|y)$ directly is very difficult because the same symptoms can arise due to various diseases (e.g. symptoms like headache or fever can point to numerous diseases, from benign to very serious ones). The Bayes' rule allows us to solve this problem by inferring instead the probability of symptoms given the disease $P(y|x)$, and the prior probabilities of the disease $P(x)$ and symptoms $P(y)$ (e.g., in a population of interest). Typically, these can be readily estimated based on experience, collected data and earlier statistical analyses.

References

[1] H. Blockeel. *Machine Learning.* KU Leuven, 2024.

[2] L. Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

[3] M. Charikar and S. Koyejo. (CS221): Artificial intelligence: Principles and techniques. *Stanford.*

[4] M. Charikar and S. Koyejo. *Artificial Intelligence: Principles and Techniques (CS221).* Stanford University, 2024.

[5] Y. Dodge. *The Oxford Dictionary of Statistical Terms.* 2003.

[6] S. Ermon. *Probabilistic Graphical Models. (CS228).* Stanford University, 2024.

[7] Tin Kam Ho. Random decision forests. In *Proceedings of the 3rd International Conference on Document Analysis and Recognition (ICDAR)*, pages 278–282. IEEE, 1995.

[8] T.K. Ho. The random subspace method for constructing decision forests. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(8):832–844, 1998.

[9] K. Hornik. Multilayer feedforward networks are universal approximators. *Neural Networks*, 1989.

[10] Y. Hu, C. Li, Meng K., J. Qin, and X. Yang. Group sparse optimization via $\ell_{p,q}$ regularization. *Journal of Machine Learning Research*, 18, 2017.

[11] D. Klein and P. Abbeel. *Intro to AI (ai.berkeley.edu).* UC Berkeley, 2023.

[12] Y. LeCun, Y. Bengio, and Y. Hinton. Deep learning. *Nature*, 521:436–444, 2015.

[13] N. Narasimhan. Multiple linear regression-an intuitive approach. *Medium*, 2020.

[14] A. Ng and T. Ma. *Machine Learning CS229: Lecture Notes.* Stanford University, 2023.

[15] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference.* Morgan Kaufmann, 1988.

[16] A. Pizurica. Probabilistic reasoning: When the environment is uncertain. In Emmanuel Gillain, editor, *Demystifying Artificial Intelligence Symbolic, Data-Driven, Statistical and Ethical AI*, pages 181–236. De Gruyter, 2024.

[17] T Poggio and S. Smale. The mathematics of learning: Dealing with data. *Notices of the AMS*, 50(5):537–544, 2003.

[18] S. Russel and P. Norvig. *Artificial Intelligence: A Modern Approach, 4th Edition.* Pearson, 2021.