Chapter 2

The Perceptron

2.1 Perceptrons as Binary Classifiers

The perceptron is a historic artificial neural network (ANN) model invented in the late 1950s [22]. In these lecture notes, it serves as a basic example of ANN that allows to explain some of the main concepts in building ANN. Also, modified perceptrons are still of use in some applications, especially in so-called online learning systems.

The perceptron is a *binary classifier*: it maps an input $\mathbf{x}$ (a real-valued vector) to a binary output value (i.e. to 0 or 1). Binary classification problems are ubiquitous. For example, to decide whether an email is spam or not, or whether one has a certain illness or not, is a binary decision problem.

The perceptron solves the binary classification problem in the following way. Given some input $\mathbf{x}$ of length N the perceptron outputs

$$y = \sigma_{\mathbf{w}}(\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{w} \cdot \mathbf{x} + b \geq 0 \\ 0 & \text{otherwise} \end{cases} \tag{2.1}$$

where $\mathbf{w}$ is a vector of real-valued *weights* and b is the *threshold* or *bias.* The functioning of a perceptron is displayed in Fig. 2.1.

To make sense of this formula we visually inspect the case with $N = 2$, see Fig. 2.2. The expression (2.1) effectively separates the data by a line called the decision boundary: data points that satisfy $w \cdot x + b > 0$ lie on one side of the line, the other data points on the other side. The weight vector w determines the orientation of the line, the bias b shifts the line away from the origin, i.e., fixes the position of the line.

Usually, the bias b is converted to a weight $-b$, that is, the input vector is extended to a $(N + 1)$-dimensional vector x_1,...,x_N,1 and the weight vector to

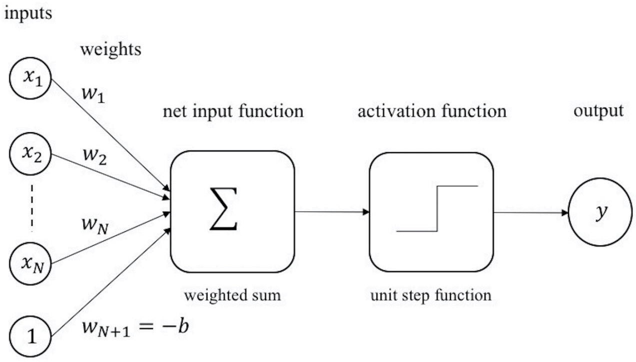

Figure 2.1: Outline of the perceptron.

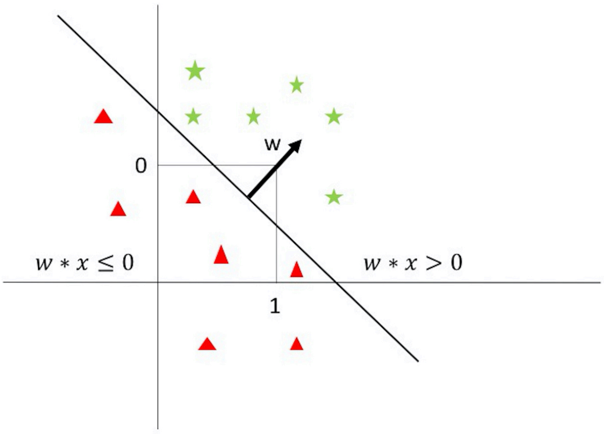

Figure 2.2: Decision boundary of a perceptron in the case N=2.

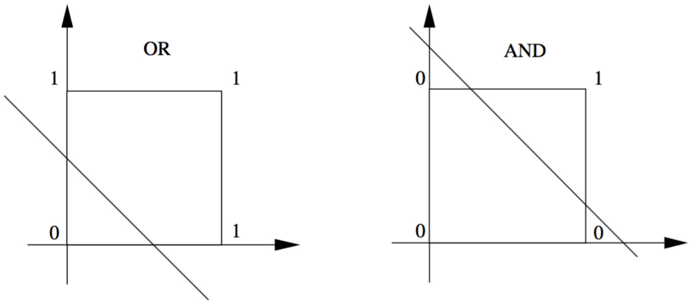

Figure 2.3: Perceptron decision boundaries for the logical operators AND and OR.

w_1,...,w_N,w_{N+1} with $w_{N+1} = b$, which turns the expression (2.1) into

$$y = \sigma_{\mathbf{w}}(\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{w} \cdot \mathbf{x} \geq 0 \\ 0 & \text{otherwise} \end{cases} \tag{2.2}$$

It is clear from Figure 2.2 that in the case $N = 2$ a perceptron can only classify the data correctly if it is possible to separate the data points by a straight line. The problems for which this is possible are called *linearly separable*.[1] Generally, a perceptron separates the input space into two half-spaces. In higher dimensions, i.e., for inputs with $N > 2$, the perceptron attempts to separate the data points by a so-called hyperplane.

We illustrate the linear separability problem by the Boolean functions, i.e., the logical operators $f : \{0, 1\} \rightarrow \{0, 1\}$. Some Boolean operators can be computed by a perceptron, some cannot.[2] For example, the operators AND and OR can be computed by a perceptron, see Figure 2.3.

2.2 Perceptron Learning Rule

Given the problem to classify the elements of a set into two classes (a binary classification problem) by a perceptron, that is, to separate two sets A and B, a learning algorithm should automatically find the weights and bias necessary for the solution of the problem. The set A of input vectors in N-dimensional space must thus be separated from the set B of input vectors in such a way that a perceptron computes the binary function $\sigma_{\mathbf{w}}(\mathbf{x})$ with $\sigma_{\mathbf{w}}(x) = 1$ for $x \in A$ and

[1] Two set of points A and B in an N-dimensional space are called linearly separable if $N + 1$ real numbers w_1,...,w_N,w_{N+1} exist, such that every point $(x_1, ..., x_N) \in A$ satisfies $\sum_{i=1}^{N} w_i x_i \geq w_{N+1}$ and every point $(x_1, ..., x_N) \in B$ satisfies $\sum_{i=1}^{N} w_i x_i < w_{N+1}$.

[2] With $N = 2$, 14 out of 16 possible Boolean functions are linearly separable. With $N = 3$, 104 out of 256, with $N = 4$, 1882 out of 65'536 possible functions [21]. In general, it is not known whether there exists a formula for expressing the number of linearly separable functions as a function of N.

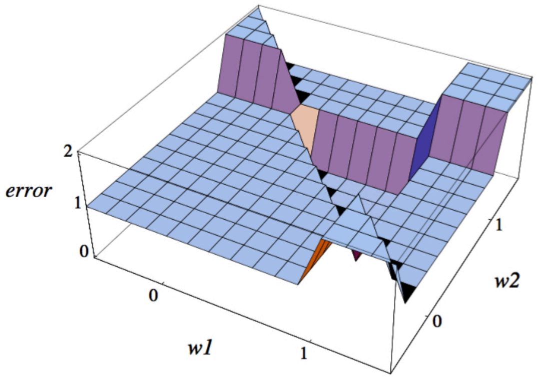

Figure 2.4: Error function for the AND function.

$\sigma_{\mathbf{w}}(\mathbf{x}) = 0$ for $\mathbf{x} \in B$. The binary function $\sigma_{\mathbf{w}}$ depends on the set $\mathbf{w}$ of weights and the threshold.

Then the error function is the number of false classifications obtained with the weight vector $\mathbf{w}$, that is,

$$E_{\mathbf{w}}(\mathbf{x}) = \sum_{\mathbf{x} \in A} (1 - \sigma_{\mathbf{w}}(\mathbf{x})) + \sum_{\mathbf{x} \in B} \sigma_{\mathbf{w}}(\mathbf{x}). \tag{2.3}$$

The error function is defined over the weight space and the aim of the perceptron learning algorithm is to minimise this function. Because $E(\mathbf{w})$ is positive or zero, the global minimum with $E(\mathbf{w}) = 0$ is to be reached.

The learning algorithm must thus minimise the error function $E(\mathbf{w})$. In weight space, the global minimum corresponds to a solution region. For example, in Figure 2.4 the error is shown graphically for a perceptron with constant bias $b = 1$ that computes the binary function AND, with the error function plotted for all combinations of the two weights between -0.5 and 1.5 in steps of 0.1. The solution region is the triangular area in the middle. The learning algorithm should thus reach from every point in the weight space the solution region. This can be achieved by descending on the error surface which corresponds to a search for an inner point of the solution region. Generally, the inner points of the solution region are the points in the interior of a convex polytope where the sides of the polytope are delimited by the planes defined by the input-output relations. In principle, for this case, the solution region can be computed.

However, we want an iterative procedure to find a solution, that is, a learning algorithm. Specifically, the perceptron learning algorithm works as follows. The training set consists of two sets, A and B, in N-dimensional extended input space. We look for a vector $\mathbf{w}$ capable of separating both sets, so that all vectors in A belong to the positive half-space ($\mathbf{w} \cdot \mathbf{x} > 0$) and all vectors in B

to the negative half-space ($\mathbf{w} \cdot \mathbf{x} < 0$) of the linear separation. This is achieved by the following algorithm.

1. An initial weight vector $\mathbf{w}_0$ is selected at time $t = 0$.
2. A vector $x \in A \cup B$ is selected.
3. If $\mathbf{x} \in A$ and $\mathbf{w} \cdot \mathbf{x} > 0$, select another vector $\mathbf{x}$.
   If $\mathbf{x} \in A$ and $\mathbf{w} \cdot \mathbf{x} \leq 0$, set $\mathbf{w}_{t+1} = \mathbf{w}_t + \mathbf{x}$, $t = t + 1$ and select another vector $\mathbf{x}$.
   If $\mathbf{x} \in B$ and $\mathbf{w} \cdot \mathbf{x} < 0$, select another vector $\mathbf{x}$.
   If $\mathbf{x} \in B$ and $\mathbf{w} \cdot \mathbf{x} \geq 0$, set $\mathbf{w}_{t+1} = \mathbf{w}_t - \mathbf{x}$, $t = t + 1$ and select another vector $\mathbf{x}$.
4. If all vectors $\mathbf{x}$ are classified correctly stop. If not, define a stopping time.

The algorithm makes a correction to the weight vector each time one of the selected vectors in A or B has not been classified correctly. If the two sets A and B are linearly separable, the vector $\mathbf{w}$ is updated only a finite number of times, i.e., the algorithm converges.[3] Ideally, the algorithm stops when all vectors are classified correctly. If this is not the case, the algorithm is stopped after a certain number of *epochs*, i.e., iterations through the training set.

The perceptron learning rule can be written in a more compact way then in the scheme above. With t being the *target*, i.e., the desired output, the error is defined by

$$e = t - y. \tag{2.4}$$

Then, the weights in the extended version are updated as follows

$$\mathbf{w}^{t+1} = \mathbf{w}^t + e \cdot \mathbf{x}. \tag{2.5}$$

In the unextended version, the bias would also be updated by $b^{t+1} = b^t + e$, in the extended version the bias is included in (2.5).

Many extensions and generalisations to the perceptron exist. A first extension is to allow for multiple outputs, leading to a *multi-perceptron* that outputs a vector $\mathbf{y}$, see Fig. 2.5.

The perceptron then involves a matrix $\mathbf{W}$ with elements w_{ij} and outputs (in the extended version)

$$\mathbf{y} = \sigma(\mathbf{W} \cdot \mathbf{x}) \tag{2.6}$$

with the activation function applied to each element of the vector $\mathbf{z} = \mathbf{W} \cdot \mathbf{x}$.

A multi-perceptron allows to assign to each input vector $\mathbf{x}$ a target vector $\mathbf{t}$, resulting in an error $\mathbf{e} = \mathbf{t} - \mathbf{y}$. Thereby multiclass classification problems can be solved. The learning rule generalises to

$$\mathbf{W}^{t+1} = \mathbf{W}^t + \mathbf{e}\mathbf{x}^T, \tag{2.7}$$

where $\mathbf{x}^T$ is the transpose of $\mathbf{x}$.

[3] For a proof, see [8].

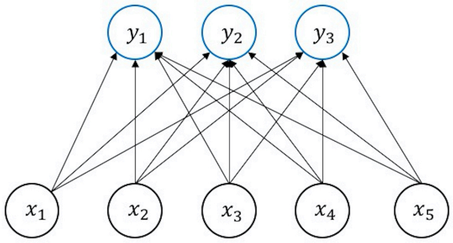

Figure 2.5: A multi-perceptron with 5 inputs and 3 outputs.

2.3 Activation Functions and Gradient Descent

The perceptron learning rule we have encountered is an algorithm for adjusting the network weights $\mathbf{w}$ to minimise the difference between the actual and the desired outputs. The original perceptron outputs

$$y = \sigma(z) = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{otherwise} \end{cases} \tag{2.8}$$

where $z = \mathbf{w} \cdot \mathbf{x} + b$ and the unit step function is used as the so-called *activation function.*

The original perceptron can be modified or generalised in various ways. One of the earliest modification was to allow for a different activation function. A simple choice is to have a *linear activation* function

$$y = \sigma(z) = z \tag{2.9}$$

Thus, one simply has

$$y = \mathbf{w} \cdot \mathbf{x} + b. \tag{2.10}$$

In the form (2.10) the perceptron model is equivalent to a linear regression model.

A perceptron with the activation function (2.9) results in a learning rule that is known as *Adaline*, *Widrow-Hoff* or *delta rule* which we will derive now.

Let us fix first some notation: let us assume that there are K examples (data points) of input vectors of size N and K target vectors of size M. For example, in Figure 2.5 there is $N = 5$ and $M = 3$. We then want to minimise the error or performance measure

$$E_{\mathbf{W}}(\mathbf{x}) = \frac{1}{2} \sum_{i=1,k=1}^{M,K} (t_{i,k} - y_{i,k})^2 = \frac{1}{2} \sum_{i=1,k=1}^{M,K} (t_{i,k} - \sum_{j=1}^{N} w_{ij} x_{j,k})^2. \tag{2.11}$$

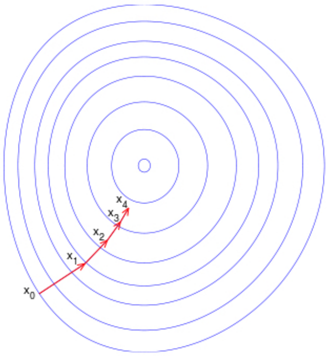

Figure 2.6: Gradient descent on error surface.

In principle, one could solve for the weights w_{ij} that minimise $E_{\mathbf{W}}$, i.e. solving $\nabla E_{\mathbf{W}} = 0$. We are however interested in setting up a learning rule. The cost function (2.11) is a function of the weights w_{ij} and we want to change the weights such that the error is reduced. This is achieved by changing each weight w_{ij} by an amount Δw_{ij} proportional to the gradient of E, that is, by $\Delta w_{ij} = -\eta \frac{\partial E}{\partial w_{ij}}$. By this we approach a local minimum of the error function is approached by taking steps proportional to the negative of the gradient of the function at the current point, see Figure 2.6.

Let us thus compute $\frac{\partial E}{\partial w_{ij}}$:

$$\begin{aligned}
\frac{\partial E}{\partial w_{ij}} &= \frac{\partial}{\partial w_{ij}} \frac{1}{2} \sum_{i,k} (t_{i,k} - y_{i,k})^2 = \frac{1}{2} \sum_k \frac{\partial}{\partial w_{ij}} (t_{i,k} - y_{i,k})^2 && (2.12) \\
&= \frac{1}{2} \sum_k 2(t_{i,k} - y_{i,k}) \frac{\partial}{\partial w_{ij}} (t_{i,k} - y_{i,k}) && (2.13) \\
&= \sum_k (t_{i,k} - y_{i,k}) \frac{\partial}{\partial w_{ij}} (t_{i,k} - \sum_j w_{ij} x_{j,k}) && (2.14) \\
&= \sum_k (t_{i,k} - y_{i,k})(-x_{j,k}). && (2.15)
\end{aligned}$$

We therefore have $\frac{\partial E}{\partial w_{ij}} = -\sum_k x_{j,k}(t_{i,k} - y_{i,k})$ and if set $\delta_{i,k} = t_{i,k} - y_{i,k}$ we have for the learning rule

$$w_{ij} = w_{ij} + \Delta w_{ij} = w_{ij} + \eta \sum_k \delta_{i,k} x_{j,k}, \tag{2.16}$$

where η is the so-called *learning rate*. The learning rate will be discussed in more detail later.

Note that if the problem is linearly separable, both the perceptron learning rule and the delta rule will find a set of weights in a finite number of iterations that solves the problem correctly.

By gradient descent the weights are incremented in the negative direction to the gradient, that is, $-(-\delta x_{j,k}) = +\delta x_{j,k}$. If $(t_{i,k} - y_{i,k})$ is classified as $0, 1, -1$, one gets the perceptron learning rule with $\pm x_{j,k}$. In general, the error $\delta x_{j,k}$ is not restricted to having the values $0, 1, -1$ but may have any value.

Although the learning rule thus looks identical to the perceptron rule, there are two main differences:

- The output y is a real number and not a class label, i.e. 0 or 1, as in the perceptron learning rule.
- The weight update is calculated based on all samples in the training set (sum over k in (2.16)), which is why this approach is also called *batch gradient descent*.

One speaks of batch learning, i.e., learning by batch gradient descent, if the cost function is minimised based on the complete training data set. If we think back to the perceptron rule, we remember that it performed the weight update incrementally after each individual training sample. This approach is also called *online learning*, and in fact, this is also how the delta learning rule was first proposed.

The process of incrementally updating the weights is also called *stochastic gradient descent* since it approximates the minimisation of the cost function. Although the stochastic gradient descent approach might sound inferior to gradient descent due its stochastic nature and the approximated direction (gradient), it can have certain advantages in practice. Often, stochastic gradient descent converges much faster than gradient descent since the updates are applied immediately after each training sample; stochastic gradient descent is computationally more efficient, especially for very large datasets. Another advantage of online learning is that the weights can be immediately updated as new training data arrives, e.g., in web applications, and old training data can be discarded if needed.

Often it is also common practice to use so-called *mini-batch* gradient descent, where a number of samples is drawn from the training set to calculate the error, often resulting in smoother convergence than stochastic gradient descent.

Usually, the standard stochastic gradient descent algorithm uses sampling with replacement, which means that at each iteration, a training sample is chosen randomly from the entire training set. Sampling without replacement, which means that each training sample is evaluated exactly once in every epoch, can also be used and sometimes shows a better performance [25].

Figure 2.7 shows an example comparison of a batch, mini-bach and stochastic gradient descent on an error surface.

The delta learning rule can be derived for any linear differentiable output/activation function, whereas in the perceptron case one has the threshold output function. Since this method requires computation of the gradient of the error function at each iteration step, we must guarantee the continuity and differentiability of the error function. Obviously we have to use a kind of activation function other than the step function used in perceptrons, such as the linear

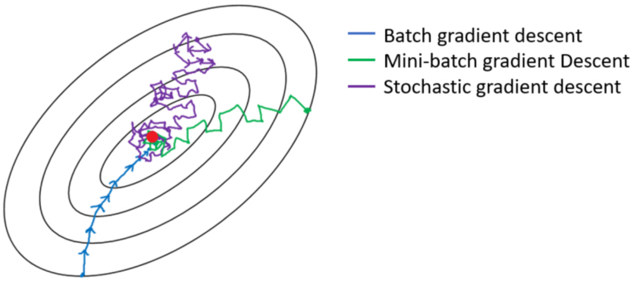

Figure 2.7: Batch, mini-bach and stochastic gradient descent on an error surface.

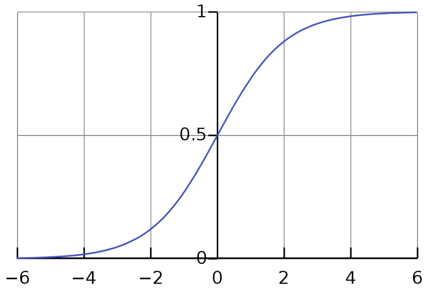

Figure 2.8: Sigmoid activation function.

activation function (2.9), because the composite function produced by interconnected perceptrons is discontinuous, and therefore the error function too.

Another often used activation function is the sigmoid activation function, a real function $\sigma : \mathbb{R} \to (0,1)$ defined by

$$\sigma(z) = \frac{1}{1+e^{-z}}, \tag{2.17}$$

see also Figure 2.8.

The sigmoid activation function leads, by gradient descent, to the batch learning rule

$$w_{ij} = w_{ij} + \Delta w_{ij} = w_{ij} + \eta \sum_k (t_{i,k} - y_{i,k}) y_{i,k} (1 - y_{i,k}) x_{j,k}. \tag{2.18}$$

Many other kinds of activation functions have been proposed and gradient descent is applicable to all of them. Any differentiable activation function makes the function computed by a neural network differentiable (assuming that the integration function at each node is just the sum of the inputs), since the network itself computes only function compositions. Thus, the error function also becomes differentiable.

2.4 Tasks, Performance and Experience

We have introduced in this chapter a first learning algorithm, the perceptron learning algorithm. It solves a specific task, namely a binary classification task according to an error or performance measure relying on some feedback mechanism based on the data given. This kind of approach can be generalised to all kinds of learning tasks. In this section we want to introduce a general framework for learning algorithms.[4]

Generally, one may say to have a learning algorithm, if *the algorithm learns from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E* [7].

We use this framework, that is, the concepts of a task, performance measure and experience, to deal with and classify any kind of learning algorithm. Let us give a few intuitive examples of the different kinds of tasks, performance measures and experiences that can be used to construct learning algorithms.

Tasks Solving tasks is the raison d'être of machine learning. Machine learning is so successful because it allows to tackle tasks that are too difficult to be solved by writing and designing fixed programs. Learning is thereby the means of attaining the ability to perform the task. For example, if we want a robot to be able to walk, then walking is the task. We could program the robot to learn to walk, or we could attempt to write a program that directly specifies how it should walk.

Usually, machine learning tasks are described in terms of how the machine learning system should process an example. An example is a collection of features that have been measured from some object or event. An example is usually represented as a vector $\mathbf{x} \in \mathbb{R}^n$ where each entry x_i of the vector is a *feature*. For example, the features of an image are usually the values of the pixels in the image.

- **Classification** In this type of task, one is asked to specify which of k categories some input belongs to. To solve this task, the learning algorithm is usually asked to produce a function $f : \mathbb{R}^n \to \{1, ..., k\}$. When $y = f(x)$, the model assigns an input described by vector $\mathbf{x}$ to a category identified by numeric code y.
  There are other variants of the classification task, for example, where f outputs a probability distribution over classes.
  An example of a classification task is object recognition, where the input is an image (usually described as a set of pixel brightness values) and the output is a numeric code identifying the object in the image.

- **Regression** This type of task asks to predict a numerical value given some input. That is, the learning algorithm is asked to output a function $f : \mathbb{R}^n \to \mathbb{R}$. This type of task is similar to classification, except that the

[4] This section is based on [7].

format of output is different.
For example, prediction of the number of sales in an online shop over time is a regression task.

- **Transcription** This type of task asks to observe a relatively unstructured representation of some kind of data and transcribe the information into discrete textual form.
For example, reading addresses by autonomous cars is such a task.

- **Machine translation** In a machine translation task, the input consists of a sequence of symbols in some language, and the algorithm must convert this into a sequence of symbols in another language.
This is for example applied to natural languages such as translating from English to German.

- **Structured output** Transcription and translation tasks can be subsumed under the structured output tasks, that is, tasks that have as output a vector (or other data structure containing multiple values) with relationships between the different elements.
For example, an image caption task is such a task.

- **Anomaly detection** In this type of task, the computer program looks through a set of events or objects and flags some of them as being unusual or atypical.
For example, recognising from sensor data when a fridge has a problem is such a task.

- **Synthesis and sampling** In this type of task, the machine learning algorithm is asked to generate new examples that are similar to those in the training data. Synthesis and sampling via machine learning can be used in media applications when generating large volumes of content by hand would be expensive or require too much time.
This technique is for example used in game programming.

- **Finding of missing values** In this type of task, one is given examples $\mathbf{x} \in \mathbb{R}^n$ but with some entries x_i missing. The algorithm must provide a prediction of the values of the missing values.

- **Denoising** In this type of task, the machine learning algorithm is given as input an example $\tilde{\mathbf{x}}$ corrupted by an unknown noise process. The task is to recover the clean example $\mathbf{x} \in \mathbb{R}^n$.

- **Density estimation** Many of the tasks described above require the learning algorithm to capture the structure of a probability distribution. Density estimation explicitly captures that distribution. In the density estimation problem, the machine learning algorithm is asked to learn a function $p : \mathbb{R}^n \to \mathbb{R}$ where $p(\mathbf{x})$ can be interpreted as a probability density function (if $\mathbf{x}$ is continuous) or a probability mass function (if $\mathbf{x}$ is discrete) on the space that the examples were drawn from. In principle, we can then perform computations on that distribution to solve the other tasks as well. For example, if we have performed density estimation to obtain a probability distribution $p(\mathbf{x})$ we can use that distribution to solve the finding missing value task. However, in practice, density estimation does not always enable us to solve all these related tasks, because in many cases the required operations on $p(\mathbf{x})$ are computationally intractable.

Performance Measures To evaluate the abilities of a learning algorithm, we must design a quantitative measure of its performance. Usually this performance measure P is specific to the task T being carried out by the system.

For tasks such as classification, classification with missing inputs, and transcription, we often measure the accuracy of the model. Another measure introduced so far is the error function (2.11) used in the delta learning rule.

It is clear that for tasks such as density estimation, it does not make sense to measure accuracy or any of the other classification performance measures. Instead, we must use a different performance metric that gives the model a continuous-valued score for each example. The most common approach is to report the average log-probability the model assigns to some examples, see the next chapter.

As said, for gradient descent or the backpropagation algorithm introduced later on, the error function or performance measure must be differentiable.

It is often difficult to choose a performance measure that corresponds well to the desired behaviour of the system. In some cases, this is because it is difficult to decide what should be measured. In other cases, we know the quantity we would like to measure, but measuring it is impractical. In these cases, one must design an alternative criterion that still corresponds to the design objectives, or design a good approximation to the desired criterion. For these reasons a major research effort in ANN & DL theory and applications is to find “good” performance measures.

Usually we are interested in how well the machine learning algorithm performs on data that it has not seen before, since this determines how well it will work in the real world. We therefore evaluate the performance measures using a *test set* of data that is separated from the data used for training the system, see also the next section.

Experience Learning algorithms learn by experience. In the ANN & DL context this usually means to experience a dataset. A dataset is a collection of examples also called data points.

We will deal with a variety of data sets. One of the oldest datasets studied in statistics and machine learning, which we will also encounter, is the *Iris dataset*. It is a collection of measurements of different parts of 150 iris plants. Each individual plant corresponds to one example. The features within each example are the measurements of each part of the plant: the sepal length, sepal width, petal length and petal width. The dataset also records which species each plant belonged to: Three different species are represented in the dataset.

A dataset is conveniently organised with a *design matrix.* A design matrix is a matrix containing a different example in each row. Each column of the matrix corresponds to a different feature. For instance, the Iris dataset contains 150 examples with four features for each example. This means we can represent the dataset with a design matrix $\mathbf{X} \in \mathbb{R}^{150 \cdot 4}$, where $X_{i,1}$ is the sepal lenght of plant i, $X_{i,2}$ is the sepal width of plant i etc..

With tabular data, one usually operates with design matrices. Of course, to describe a dataset as a design matrix, it must be possible to describe each example as a vector, and each of these vectors must be the same size. This is not always possible. For example, if you have a collection of photographs with different widths and heights, then different photographs will contain different numbers of pixels, so not all the photographs may be described with the same length of vector.

Machine learning algorithms can be categorised by the kind of experience they are allowed to have during the learning process. We have already encountered the broad categories: unsupervised, supervised and reinforcement learning algorithms.

Often however, in practice, one deals with a mixture of unsupervised learning and supervised learning. Also, some machine learning algorithms do not experience a fixed dataset. For example, reinforcement learning algorithms interact with a changing environment, so there is a feedback loop between the learning system and its experiences.

In general, we do not want a learning algorithm just to learn well a training set but to be able to generalise, that is, to perform on hitherto unseen examples. For this reason, the data set is usually split into a *training set* and a *test set.* The training set is used to fit the model, i.e., to learn the weights of the neural network. The test set is used to assess the generalisation error. It is clear that the two sets must be strictly separated.

Often, however, there is an additional step, involving so-called *hyperparameters.* The hyperparameters are not learned by the learning algorithm itself. Rather they are used to control the algorithm's behaviour. For example, the learning rate introduced earlier is such a hyperparameter.

We then have an additional "training set" called the *validation set.* The test examples should not be used in any way to make choices about the model, including its hyperparameters. Therefore, the validation set is drawn from the training data. Specifically, the training data is split into two disjoint subsets. One of these subsets is used to learn the parameters. The other subset is the validation set, used to estimate the generalisation error during or after training, allowing for the hyperparameters to be tuned. Typically, one uses about 80 percent of the training data for training and 20 percent for validation. Since the validation set is used to "train" the hyperparameters, the validation set error will underestimate the generalisation error. Once the hyperparameter optimisation is complete, the generalisation error is evaluated using the test set.

2.5 Classification Tasks

The perceptron learning algorithm deals with binary classification problems. Binary classification problems fall within the broader class of classification problems. Also, we would like to measure how good we are in our classification. For this purpose so-called *performance measures* are introduced.

In a **binary classification** task we want to classify the elements of a given set into two groups and to predict which group each element belongs to. To establish how good the classification or the prediction of the binary classifier is a *contingency table* is introduced, see Table 2.1.

		Predicted class		
		p	**n**	**total**
Actual class	**p′**	True Positive	False Negative	P′
	n′	False Positive	True Negative	N′
	total	P	N	

Table 2.1: Contingency table.

		Predicted class		
		yes	**no**	**total**
Actual class	**yes**	100	5	105
	no	10	50	60
	total	110	55	

Table 2.2: Contingency table example.

Table 2.2 shows a concrete example: a classifier predicted for 165 day whether is is dry the next day ("yes") or not ("no"). There was a total of 110 dry days predicted and 55 days with some precipitation. Actually, it was dry on 105 days and there was some rain on 60 days.

Based on this table various performance measures can be computed.

- **Accuracy**: the accuracy measures how often the classifier is correct, that is, $\frac{tp+tn}{N} = \frac{100+50}{165} = 0.91$.
- **Misclassification Rate**: the misclassification rate states how often the classifier was wrong, that is, $\frac{fp+fn}{N} = \frac{10+5}{165} = 0.09$.
- **Precision**: precision displays how often the classifier was correct in the cases it predicted a "yes", that is, $\frac{tp}{tp+fp} = \frac{100}{100+10} = 0.91$.
- **Recall** or **Sensitivity** or **True Positive Rate**: recall shows how often the classifier predicted "yes" when it actually was "yes", that is, $\frac{tp}{tp+fn} = \frac{100}{100+5} = 0.95$.
- **False Positive Rate**: the false positive rate shows how often the classifier predicted "yes" when it actually was "no", that is, $\frac{fp}{fp+tn} = \frac{10}{10+50} = 0.17$.
- **Specificity**: specificity tells you how often the classifier predicted "no" when it actually was "no", that is, $\frac{tn}{fp+tn} = \frac{50}{10+50} = 0.83$.
- **F-Score**: the F-score gives the harmonic mean of precision and recall, that is, $F_1 = 2 \cdot \frac{precision \cdot recall}{precision + recall}$.

Which performance measure to use depends on the case at hand.

In a **multiclass classification** problem we want to classify and predict which elements of a given set belong to one of three or more classes. In the multiclass case, one has a *confusion matrix* as the basis for performance measures. For example, if in a classification problem one has three classes coded by $\{0, 1, 2\}$ and the true classes are $(2, 0, 2, 2, 0, 1)$ and the predicted classes are $(0, 0, 2, 2, 0, 2)$, the confusion matrix is

$$\mathbf{C} = \left.\left(\underbrace{\begin{matrix} 2 & 0 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 2 \end{matrix}}_{\text{predicted}}\right)\right\} \text{actual}$$

That is, there are 2 matches on regard to the first class and two matches in regard to the third class. These counts are on the diagonal. There is also a case where the true class was 1 but class 2 was predicted and a case where the true class was 2 but class 0 was predicted. These counts are on the off-diagonal.

The measures for the binary case can be generalized to the multiclass classification case [26]. These performance measures are listed in the appendix A.2.

In a **multi-label classification** task instances are associated with more than one class, that is, there can be multiple classes or labels attached to an example. For example, data on a patient in a hospital usually includes multiple labels such as blood pressure, temperature, etc. Formally, a multi-label classification maps inputs $\mathbf{x}$ to binary vectors $\mathbf{y}$ (assigning a value of 0 or 1 for each element (label) in $\mathbf{y}$). For the multi-label classification task similar metrics as in the multiclass case can be defined [27]. These metrics are also listed in the appendix A.2.