1 Introduction

1.1 The basic idea behind supervised Machine Learning

Machine Learning is literally about building machines that can learn and after learning perform specific tasks. We will encounter several forms of learning, but for the most part we consider **supervised learning**. In supervised learning we show a variety of examples together with the desired response of a specific task to a learning machine from which the machine should learn to perform an appropriate response for new examples. Thus, we are trying to solve problems with computers without explicitly programming them for a specific tasks. This is desirable specifically for tasks that would be difficult to program. A typical example of applications is object recognition, and some related tasks are shown in Figure 1.1.

Fig. 1.1 Typical examples where deep learning has been instrumental in practical applications such as letter and object recognition, and image restorations. Figure from NVIDIA DLI teaching kit.

We will still need to program the learning machine, but this is somewhat more general than coding logic for a specific problem. Machine Learning might sound like a niche area in science and you might wonder why there is now so much interest in this discipline, both academically and in industry. The reason is that Machine Learning is

really about modelling data. Modelling is the basis for advanced object recognition, data mining, and ultimately intelligent systems. Machine Learning is the analytic engine in areas such as data science, big data, data analytics and to some extend to science in general in the sense of building quantitative models.

Machine Learning has a long history with traces far back in time. Alan Turing was probably one of the earliest thinkers in the field of AI. One of the first recognized exciting realizations of the promise of learning machines came in the late 1950s and early 1960s with work like Arthur Samuel's self-learning checkers program and Frank Rosenblatt's perceptron. Arthur Samuel devised a program with some form of reinforcement learning that ultimately learned to outperform its creator. Frank Rosenblatt set much of the foundation of neural networks and even started to build neural network computers, the Mark I Perceptron. Neural networks have been popularized again in the 1980s with a strong influence by David Rumelhart. Many leading figures in Deep learning have started in this era, including Geoffrey Hinton, Yoshua Bengio, Yann LeCun, Jürgen Schmidthuber to name a few. We will discuss how we are now in an era of 'Deep Learning' with important recent developments that are mostly the reason for the popularity of Machine Learning today. However, much of the progress of machine learning and their scientific embedding is due to probabilistic methods and statistical learning theories, for which we need to mention some pioneers like Vladislaw Vapnik and Judea Pearl. Indeed, the development of statistical machine learning and Bayesian networks has influenced the field strongly in the last 20 years and has been essential in much of its progress as well as in the deeper understanding of machine learning. This course will hence introduce these more general ideas for a more thorough theoretical underpinning. Finally, we will also discuss the important area of reinforcement learning where Richard Bellman has contributed important work already in the mid 1950s.

Fig. 1.2 Some pioneers of Ai and Machine Learning. From top-left to right bottom: Alan Turing, Arthur Samuel, Richard Bellman, Frank Rosenblatt, David Rumelhart, and Judea Pearl.

We will discuss how basic supervised learning resembles statistical regression. However, there are several aspects of machine learning that go beyond the scope of this traditional statistical approach. In particular, machine learning is usually concerned with high dimensional problems where many factors have to be incorporated into a model. Furthermore, a lot of emphasis is given to methods that can handle nonlinear data. Also, there are other forms of Machine Learning such as **unsupervised learning** in which the machines have to find some structure in the data. We will discuss later how unsupervised learning has been a central factor in the development of deep networks and which also underlies important application areas of machine learning such as some form of cluster analysis and dimensionality reduction.

Also, there is the area of **reinforcement learning** in which the machine has to find appropriate actions based only on some form of feedback on the value of a state reached by a series of actions. A great example of the recent progress in reinforcement learning is the ability of a computer to learn to play video games. Video games from the old Atari platform have indeed become a useful paradigm for a new class of benchmarks that go beyond classical data sets for machine learning from the UCI machine learning repository that have dominated the benchmarks in the past. Atari games are somewhat slightly simplified worlds but resemble more learning in environments that humans have to figure out. In these benchmarks only visual input is given made up of the computer frames of the video game, and feedback is only provided with how well the player did in the game. Success in this areas was also made very visible when Google DeepMind challenged the best players of the Chinese board game Go. Go was considered to be a real challenge for AI systems as it is considered to rely a lot on 'gut feelings' rather quantifiable strategies. It was hence a huge success that computers which only reached levels of an advanced beginner a few years ago would win the world championship in the spring of 2016.

There are several aspects of deep learning that make this area very exciting. One aspect is that it enables so called **representational learning**, which can be seen as the foundation of much of the recent progress. A good summary of the evolution of machine learning is shown in Fig. 1.3. While traditional rule based artificial intelligence relied on hand-designed programs such as programming specific rules for inference, classic machine learning tries to find (learn) the mapping between an input and desired output. A smart feature selection and good hand-crafted representations were essential at this time for good results and much of a machine learning course would talk about this. However, we are now seeing increasingly the emphasis on end-to-end solutions where a whole tasks are learned from sensory information which includes the discovery of appropriate hierarchical representations.

Deep learning is now a very important part of machine learning which we will focus on after introducing the basis of machine learning in more general terms. A deeper understanding of machine learning requires, to some extend, an understanding of data modelling in a wider context. This includes the importance of a probabilistic framework to formulate the problems and solutions. A course on machine learning also needs to include an overview of some traditional methods such as support vector machines, classification trees, and clustering methods. Applying machine learning methods is often easy in principle and difficult in practice. That is, there are now many tools available with which Machine Learning implementation can be programmed in

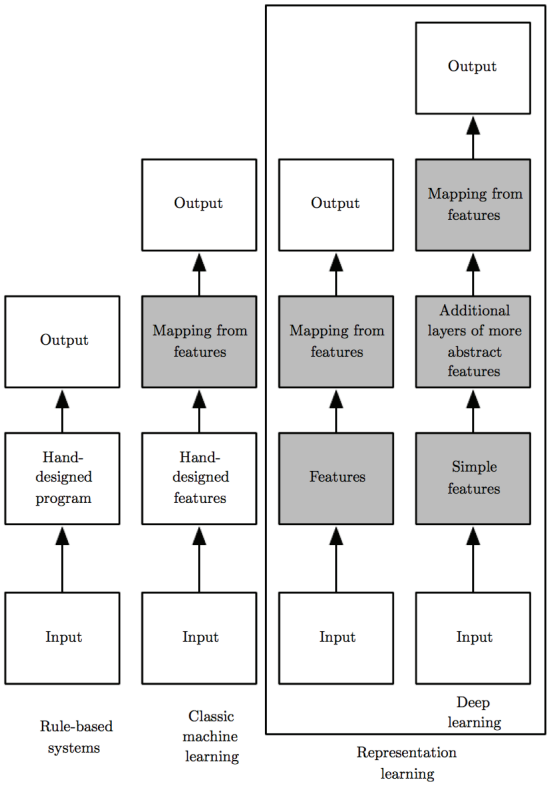

Fig. 1.3 Evolution of machine learning systems (from Goodfellow, Bengio, Courville 2015).

a few lines of code. However, the correct application of these methods in practice requires some care, experience, and a deeper understanding of the underlying issues. Therefore, our approach will be to explain some of the basic ideas of supervised learning below in this chapter. This includes the introduction of some basic concepts such as knowing what a training and validation set is, and how cross-validation can be used to optimize hyperparameters.

In the second chapter we learn how to apply such methods with some programming frameworks. Fig. 1.4 list some of the common Machine Learning frameworks. We will be using the Python programing language together with some machine learning libraries, in particular sklearn and tensorflow. The next several chapters explore the principle behind supervised learning in the form of regression and classifications. We thereby switch frequently between a functional and a probabilistic framework. A refresher on the basic probability formalism is included in the third chapter. The following chapters outline some of the fundamental ideas behind probabilistic machine learning and some important Machine Learning algorithm including supervised and unsupervised learning, and issues beyond regression and classification such as dimensionality reduction and variable selection. The second half of the course is dedicated to neural networks and deep learning. This includes convolutional networks,

Tool	Uses	Language
Scikit-Learn	Classification, Regression, Clustering	Python
Spark MLlib	Classification, Regression, Clustering	Scala, R, Java
Weka	Classification, Regression, Clustering	Java
Caffe	Neural Networks	C++, Python
TensorFlow	Neural Networks	Python

Fig. 1.4 Supervised Learning Frameworks.

autoencoders, and various recurrent networks. These subjects will be discussed with the tensorflow framework. The last section will be dedicated to (deep) reinforcement learning.

1.2 Mathematical formulation of the supervised learning problem

Much of what is currently most associated with the success of Machine Learning is supervised learning, sometimes also called predictive learning. The basic task of supervised learning is that of taking a collection of input data $\mathbf{x}$, such as the pixel values of an image, some measured medical data, or robotic sensor data, and predicting an output value $\mathbf{y}$ such as the name of an object in an image, the state of a patient's health, or the location of obstacles. It is common that each input has many components, such as many millions of pixel values in an image, and it is useful to collect these values in a mathematical structure such as a vectors in one dimension, a matrix in two dimensions, or generally in a tensor for higher dimensions. We often refer to Machine Learning problems as high-dimensional which refers in this context to the large number of components and not the dimension of the input tensor.

At this time we use the mathematical terms of a vector, matrix and tensor mainly to signify a data structure. In a programming context these are more commonly described as 1, 2 or 3-dimensional arrays. The difference between arrays and tensors (a vector and matrix is in some sense a special form of a tensor) is, however, that the mathematical definitions also include rules how to calculate with these data structures. This manuscript is not a course on mathematics; we are only users of mathematical notations and methods. Mathematical notation help us enormously to keep the text short while being precise. We follow here a common notation of denoting a vector, matrix or tensor with bold faced letters, whereas we use regular fonts for scalars. We usually call the input vector a feature vector as the components of this are typically a set feature values of an object. The output could also be a multi-dimensional object such as a vector or tensor itself. Mathematically we can denote the relations between the input and the output as a function

$$y = f(\mathbf{x}). \tag{1.1}$$

We consider the function above as a description of the **true underlying world**, and our task in science or engineering is to find this relation. In the above formula we considered a single output value and several input values for illustration purposes, although we see later that we can extend this readily to multiple output values.

The challenge for machine learning is to find this function or at least to approximate it sufficiently. Machine learning has several approaches to deal with this. One approach that we will predominantly follow for much of the course is to define a general parameterized function

$$\hat{y} = \hat{f}(\mathbf{x}; \mathbf{w}). \tag{1.2}$$

This formula describes that we make a parameterized hypothesis in which we specified a function $\hat{f}$ that depends on parameters $\mathbf{w}$ to approximate the desired input-output relation. This function is called a **model**:

> *A model is an approximation of a system to study specific aspects of the system and to predict novel behaviour*

This often means that not all of the underlying world has to be captured in depth. For example, a building engineer might make a model of a bridge to tests its static without including the ascetic aspects that an architect might emphasize in a model. In our context the word model is synonymous with approximation. Note that we have indicated that this model is an approximation of the desired relation by using a hat symbol above the y and the f. However, we frequently drop the hat symbol when the relation is clear from the context.

In the context of machine learning, a model typically includes parameters so that their presence is synonymous with a model. The parameters are specified in this function by including the parameters, which we often specify as vector $\mathbf{w}$, behind a semicolon in the function arguments. A more appropriate mathematical statement would be that the formula defines a set of functions in the parameter space. **Learning** is the challenge to find the values for the parameters that best describe the data. Finding these parameters is usually done with a learning algorithm. A common way of such a learning algorithm is to define a function that describes our goal of learning, such as minimizing the number of wrong classifications. We will call this function the **loss function** L, although other terms are sometimes used in the literature such as objective function, error function, or risk. A common algorithm to minimize such a loss function is to use an algorithm called **gradient descent** that is an iterative method over the training data and that changes the parameters along the gradient $\bigtriangledown\mathcal{L}$ of the loss function,

$$w_i \leftarrow w_i - \epsilon\nabla\mathcal{L} \tag{1.3}$$

where ϵ is called the learning rate and ∇ is the Nabla operator which signifies the gradient. This is a typical learning algorithm to find the parameters of a model based an example data. We elaborate on this algorithm later.

While the gradient descent can find parameters to minimize the loss of the training data, our real goal is to find the values of $\mathbf{w}$ that best predicts data that has not been seen before. Just describing the training data is somewhat more like a memory, but being able to **generalize** is the main goal of machine learning. Hence, a good solution of the machine (model) learning problem is represented by a point in the parameter space that approximates best the true underlying world. However, since we usually don't know

the true underlying world we estimate how good this model is by evaluating how good new predictions are.

In some applications of supervised learning we want to predict a continuous output variable. For example, we might want to predict the price of a house from the size information. This is called **regression**. In contrast, sometimes we want to predict discrete values such as the categories of object in a picture. This is called **classification**. The output variable y in classification is often called a **label**. It is now common to refer generally to the output of the supervised learner as label, even in the regression case where we have a continuous "label". We will see later that regression and classification are anyhow closely related; for example, binary classification can be seen as a regression problems with a discrete function $f(x)$ such as a sign function which would give us two labels, positive and negative.

We will later go one important step further by considering the more general case when we might not be able to predict an exact value but at least the probability that a certain value will occur. It is quite common that the process under investigation includes stochastic (random) factors or unknown factors that can also be treated with probabilistic methods. The true underlying world model is thus better described by a probability density function

$$p(Y = y|\mathbf{x}). \tag{1.4}$$

Formulating Machine Learning in a probabilistic (stochastic) context has been most useful and provides us with the formalization that created the most insight into this field. In the probabilistic framework we are then modelling a density function

$$p(\hat{Y} = \hat{y}|\mathbf{x}; \mathbf{w}). \tag{1.5}$$

Function approximation is in some sense a special case of density function approximation. A probabilistic framework also leads to a more general formulation of learning in that learning can be described as finding the most likely parameters given the data,

$$\mathbf{w}^* = \mathrm{argmax}_w p(\mathbf{w}|y, \mathbf{x}) \tag{1.6}$$

Of course, we still need to find the specific form of the probability function $p(\mathbf{w}|y, \mathbf{x})$, but we can then derive learning rules from this principle. The point here is to illustrate how important and useful a probabilistic formalism is in machine learning as it includes uncertainty right from the outset.

Formulating specific probabilistic models for problems with many stochastic factors is demanding. However, there is the important area of **causal learning** that tries to provide specific probabilistic models of the components that provide the necessary foundations of the inference engine. Inference here means that the system can be used to 'argue' about a solution in a probabilistic sense. Such systems fall generally in the domain of Bayesian networks, and we will include some introduction to this important domain in this course. Figure 1.5 shows a famous example form Judea Pearl, one of the inventors of this important modelling framework. We will also give an example of a modelling tool in this domain.

Fig. 1.5 An example of a graphical representation of a causal model.

1.3 Applied Learning: Training, validating and testing

A linear model in low dimensions is excellent way to demonstrate the principle mechanisms of machine learning. In the following example we consider at first only one input feature x. Supervised machine learning in this example is equivalent to linear regression, and area that has been studied for centuries. What makes machine learning different today is that we are usually considering high-dimensional non-linear problems, and that we have computers (machines) to help us with this task. For now we will follow the function approximation formalization, but we will return to the probabilistic framework later.

Coming up with the right parameterized approximation function is the hard problem in machine learning, and we will later discuss several choices. There are also methods to systematically develop the approximation function from the data, generally called non-parametric methods. At this point we assume that we have a parameterized approximation function. To illustrate this with we chose here an example where we assume we have a single input feature, x, and we hypotheses that the output y is linearly related to x. Mathematically we write this linear model as

$$\hat{y} = ax + b, \tag{1.7}$$

where a is the slope of the linear function and b is the y-axis intercept or bias of this function. Using the training data to determine good parameters is called linear regression.

The question is then how we determine good parameters. This is where the learning process comes in. In supervised learning we must be given some examples of input-output relation from which we learn. We can think about these examples as given by a teacher. The teacher data called the **training set** are used to directly determine the parameters of the model. We can denote this training set as

$$\{\mathbf{x}^{(i)}, y^{(i)}\}, \tag{1.8}$$

where the superscript i labels the specific training example. These indices are enclosed in brackets to not confuse them with exponents. An example of a training set with 4 example points are shown in the left table in Fir. 1.6.

There are several possible training algorithm to determine the parameters of a model. One way to determine the two parameters in this example is to use some

x	y
0	1.80
1	6.72
2	7.56
3	10.15

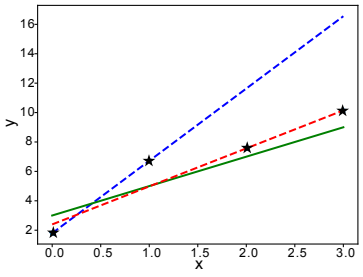

Fig. 1.6 A form of linear regression of data and cross-validation.

training data to analytically calculate the two unknowns. As we have a linear equation with only two unknowns, we only need two data points to determine their values. Using the two first data points as **training set**, we get

$$a = \frac{y^{(2)} - y^{(1)}}{x^{(2)} - x^{(1)}} = 4.9 \tag{1.9}$$

$$b = y^{(1)} - ax^{(1)} = 1.8 \tag{1.10}$$

This regression line is shown as blue dotted line in Fig. 1.6. To quantify the goodness of fit we need to define an evaluation function that is commonly called the **loss function** or **error function** E. We chose here to evaluate the goodness of the model with the mean square error (MSE) function,

$$E = \frac{1}{2N} \sum_{i=1}^{N} (\hat{y} - y)^2, \tag{1.11}$$

where N is the number of data points used to calculate the loss. If we use the training data themselves to calculate the **training error** results in a zero loss, $E_{\text{training}} = 0$. We will later use more general applicable learning rules that do not always lead to zero training error with limited training, but we need to keep in mind that the training error can typically be made small and even zero. In this specific example this happens when the number of parameters in the model is large compared to the number of training data. Hence, the training error is not always a good indicator of the performance of the model. What is more telling, and our principle aim in supervised learning, is how the models performs in predicting labels of new data points. Data that has not been used in any way in the learning process is called the **testing set**. There are sometimes competition for machine learning algorithms, and testing data for these competitions are usually withheld from the participants so that they cannot be used to develop the model. Also, it is common that some data from the training set are withheld by the model developer to 'test' the model. The test on data that have not been used in the learning process can give us an estimate about the expected performance on unseen data. If we calculate the error function from the test data we call this the **generalization error** E_{g} since this measure gives us an indication of how the model prediction performs to unseen data. The generalization error of the model with parameters calculated from the first two points in the training set is

$$E_{\mathrm{g}}(x^{(1)}, y^{(1)}, x^{(2)}, y^{(2)}) = 21.8. \tag{1.12}$$

Of course, it would be much better to have more test data so that we can not only give a better estimate on the mean, but even provide an estimate of the range such as a variance, or in general the distribution of the generalization error. We will discuss this point further on examples later in this course.

At this point we want to ask if the choice of using the first two points was good or if we should have used to other points to build the model. Indeed, if we use the last two data points for training we get the following parameter values,

$$a = \frac{y^{(4)} - y^{(3)}}{x^{(4)} - x^{(3)}} = 2.6 \tag{1.13}$$

$$b = y^{(3)} - ax^{(3)} = 2.4, \tag{1.14}$$

which is shown as red dotted line in Fig. 1.6. So which one is better, the blue dotted line or the red dotted line? It seems that the second fits better all the points, though it also becomes now handy that we quantified the goodness of the fit with the loss function on the test data, which gives in this case the value

$$E_{\mathrm{g}}(x^{(3)}, y^{(3)}, x^{(4)}, y^{(4)}) = 1.7 \tag{1.15}$$

We can even check how well different pairs of training data can predict the other data not used in training. Using different data from the original data set for training and testing is called **cross-validation**. If we calculate the generalization error of the different training/test sets we can choose as our predictive model the parameters with the smallest cross-validation error. What we did here is that we used the originally withheld data to ultimately tune a parameter of an algorithm, here the algorithm which tells us which data points to use in the training set. This is again a form of training, we determine some parameters from example data. We will call these parameters **hyperparameters** and denote them here with the symbol θ. We will later encounter different algorithm parameters such as learning rates, momentum terms or maximum number of iterations, or which data points to use in the training set to determine w as in the above example.

If we are using the original test data to tune or train the hyperparameters, than this data that we originally called the test set is really the training set for the hyperparameter. To distinguish them we call this specific training set that is used to validate the w training in order to train the hyperparameters the **validation set**. This can be sometimes confusing and some literature even uses the term validation set and cross validation in situations of testing. In principle we could view the complete procedure, our mathematical model together with the training procedures as the hypermodel $f(\mathbf{x}; \mathbf{w}, \theta)$. If we want to test this hypermodel then we need to hold out data that are neither used in the determination of $\mathbf{w}$ nor θ. Hence in this case we need really three data sets, the training set to determine w, the validation set to determine θ, and the test set to evaluate the resulting model. It is of utmost importance not to use data in any learning process if we want to estimate the performance of the model on unseen data. Using test data in training, or even any derived information of test data in training can lead to a drastic underestimation of the generalization error. This is sometimes

called **information contamination**, and information contamination can completely invalidate the results.

Note that the generalization error is still only an estimate of the true error which we usually never really know. However, since I was the one who generated the example data I can tell you how I chose them. I actually derived them from the world model

$$y = 2x + 3 + \eta, \tag{1.16}$$

where η is a normal distributed random variable. I added this random number to the perfect linear model to include a typical challenge in machine learning, that of having imprecise measurements and hence noisy training data. The other way to interpret the model is actually to accept the 'world' as stochastic and hence we are considering a stochastic model. In any case, from this model we chose some data points by sampling, though the true parameters of the world model are $a = 2$ and $b = 3$.

1.3.1 Performance measures

We used above an objective function to evaluate the performance of the model, specifically in this case the MSE. This is often a start to look at this measure, but this should not be where your investigations ends for several reasons. We will discuss later that the MSE is not always appropriate and we will learn that this should really only be used with linear models that contain Gaussian noise. Secondly, when using a test set it can be very useful and informative to look beyond this average measure and see how individual examples do or at least what the distribution of performances is. This is specifically useful when tuning the hyperparameters as this can give us some pointers to potential problems.

While we mentioned above that we deem the MSE as not always appropriate, it is also important to acknowledge important that the loss function is ultimately the choice of the user. The loss function specifies what the user wants to achieve, and this is ultimately a personal choice. However, there are generally some guidelines that this function must obey, and some others that are useful to observe. Since a loss function measures the distance between a desired point in the parameter space, let's call this $\mathbf{a}$ and the point described by the current model in the parameter space, let's call this $\mathbf{b}$, it should be based on an appropriate distance measure. That is, a loss function should be zero if these points are the same, and strictly larger and monotonously increasing otherwise,

$$E(\mathbf{a}, \mathbf{b}) = \begin{cases} = 0 & \text{for } \mathbf{a} \neq \mathbf{b} \\ > 0 & \text{else} \end{cases} \tag{1.17}$$

This still leaves a lot of options, such as the absolute function, or a logarithmic function.

A lot of performance measures have been defined and are frequently reported in machine learning papers. In binary classification it is common to report the **confusion matrix**. In binary classification it is common to call one class 'positive' and the other the 'negative'. This nomenclature comes from diagnostics such as trying to decide if a person has some disease based on some clinical tests. We can then define define the following four performance indicators,

- True Positive Rate (TP): Correctly identified example of positive class

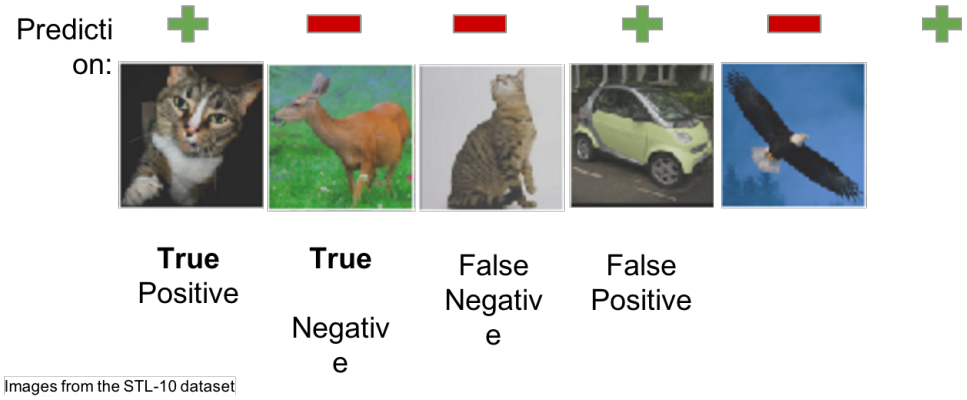

Fig. 1.7 Figure from NVIDIA DLI teaching kit.

- True Negative (TN): Correctly identified as example of negatove class
- False Positive (FP): Incorrectly identified example of positive class
- False Negative (FN): Incorrectly identified example of negative class

A graphical illustration of the meaning of these measures is give in Fig. **??**. From these it is common define the **True positive rate** or **precision** as the percentage of true positive among the positive labeled examples, and **recall** as the percentage of positive examples that are correctly labeled,

$$\text{Precision} = \frac{\#\text{ True Positives}}{\#\text{ True Positives} + \#\text{ False Positives}} \tag{1.18}$$

$$\text{Recall} = \frac{\#\text{ True Positives}}{\#\text{ True Positives} + \#\text{ False Negative}} \tag{1.19}$$

These quantities are useful to characterize the performance in a way that is relevant for some application. In particular, most algorithms have a hyperparameter which tunes the decision point, such as a decision threshold. The algorithm can thus be tuned to implement a certain form of trade-off between how reliable a prediction of the positive class is (precision), also called the **true positive rate TPR**, versus how many positive examples are wrongly predicted, called the **false positive rate FPR**. This trade-off is often visualized as a **Receiver Operating Characteristic (ROC) curve**. An example is shown in Fig. 1.8a. Ideally we want the TPR to be one and the FPR to be zero, which corresponds to a point in the upper left corner. While this is not typically not always possible, the next best is that the TPR is as close to one for all possible values of FPR. In contrast a random binary classification corresponds to the diagonal in this plot, which has a value of 0.5 as the area under this curve. Hence, when comparing two algorithm we generally prefer an algorithm that has a larger area under the ROC curve, or an area that is close to one. For many applications we have curves that are somewhere in between.

While focusing on a positive class is common in diagnostics, we sometimes are equally interested in classifying different classes, in which case we use an accuracy

measure that averages over all classes, or the positive and negative predictions in the case of binary classification,

$$\text{Accuracy} = \frac{\#\text{True Positives} + \#\text{True Negatives}}{\#\text{Samples}} \tag{1.20}$$

This measure is useful when we place equal weight on the prediction of all classes. There are many more measures defined in the literature such as placing different weight to specific prediction. A summary of some of the definitions are shown in Fig. 1.8b. Note that the application of these measures encapsulate the importance that a user places onto specific characteristics. This is similar of discussing which car is better. Some might find that larger horsepowers are good, while others want a car to consume as little gas as possible. Hence, there is not a simple best measure.

b)

sensitivity, recall, hit rate, or true positive rate (TPR)

$$\text{TPR} = \frac{\text{TP}}{P} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

specificity or true negative rate (TNR)

$$\text{TNR} = \frac{\text{TN}}{N} = \frac{\text{TN}}{\text{TN} + \text{FP}}$$

precision or positive predictive value (PPV)

$$\text{PPV} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

negative predictive value (NPV)

$$\text{NPV} = \frac{\text{TN}}{\text{TN} + \text{FN}}$$

miss rate or false negative rate (FNR)

$$\text{FNR} = \frac{\text{FN}}{P} = \frac{\text{FN}}{\text{FN} + \text{TP}} = 1 - \text{TPR}$$

fall-out or false positive rate (FPR)

$$\text{FPR} = \frac{\text{FP}}{N} = \frac{\text{FP}}{\text{FP} + \text{TN}} = 1 - \text{TNR}$$

false discovery rate (FDR)

$$\text{FDR} = \frac{\text{FP}}{\text{FP} + \text{TP}} = 1 - \text{PPV}$$

false omission rate (FOR)

$$\text{FOR} = \frac{\text{FN}}{\text{FN} + \text{TN}} = 1 - \text{NPV}$$

accuracy (ACC)

$$\text{ACC} = \frac{\text{TP} + \text{TN}}{P + N} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}$$

F1 score

is the harmonic mean of precision and sensitivity

$$F_1 = 2 \cdot \frac{\text{PPV} \cdot \text{TPR}}{\text{PPV} + \text{TPR}} = \frac{2\text{TP}}{2\text{TP} + \text{FP} + \text{FN}}$$

Matthews correlation coefficient (MCC)

$$\text{MCC} = \frac{\text{TP} \times \text{TN} - \text{FP} \times \text{FN}}{\sqrt{(\text{TP} + \text{FP})(\text{TP} + \text{FN})(\text{TN} + \text{FP})(\text{TN} + \text{FN})}}$$

Informedness or Bookmaker Informedness (BM)

$$\text{BM} = \text{TPR} + \text{TNR} - 1$$

Markedness (MK)

$$\text{MK} = \text{PPV} + \text{NPV} - 1$$

Sources: Fawcett (2006), Powers (2011), and Ting (2011) [1] [2] [3]

Fig. 1.8 a) Example of ROC curve. The ideal classifier is in the upper left corner. b) ummary of evaluation measures for binary classification (from Wikipedia)

1.4 Non-linear regression and high-dimensionalility

Above we have discussed a case where we assumed a linear function, but regression with more general non-linear functions brings another level of challenges. An example

of data that do not follow a linear trend is shown in Fig.1.9A. There, the number of transistors of microprocessors is plotted against the year each processor was introduced. This plot includes a line showing a linear regression, which is of course not very good. It is however interesting to note that this linear approximation shows some systematic deviation in some regional under and over estimation of the data. This systematic deviation or **bias** suggest that we have to take more complex functions into account. Finding the right function is one of the most difficult tasks, and there is not a simple algorithm that can give us the answer. This task is therefore an important area where experience, a good understanding of the problem domain, and a good understanding of scientific methods are required.

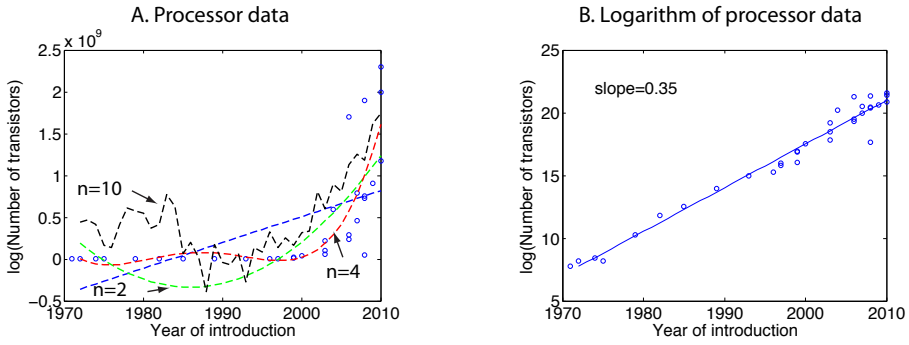

Fig. 1.9 Data showing the number of transistors in microprocessors plotted against the year they were introduced. (A) Data and some linear and polynomial fits of the data. (B) Logarithm of the data and linear fit of these data.

It is often a good idea to visualize data in various ways since the human mind seems good in 'seeing' trends and patterns. Domain-knowledge can also be valuable as specialists in the area from which the data are collected can give important advice or they might have specific hypothesis that can be investigated. It is often helpful to know common mechanisms that might influence processes. For example, the rate of change in basic growth processes is often proportional to the size of the system itself. Such an situation leads to exponential growth. (Think about why this is the case). Such situations can be revealed by plotting the functions on a logarithmic scale or the logarithm of the function as shown in Fig.1.9B. A linear fit of the logarithmic values is also shown, confirming that the average growth of the number of transistors in microprocessors is exponential, which is known as Moore's law.

But how about more general functions. For example, we can consider a polynomial of order n, that can be written as

$$y = \mathbf{w}_0 x^0 + \mathbf{w}_1 x^1 + \mathbf{w}_2 x^2 + ... + \mathbf{w}_n x^n. \tag{1.21}$$

We would usually even consider different offsets for each term which we neglected for simplicity. Given this new hypothesis in the form of a non-linear parametric function, we can again use a regression method to determine the parameters from the data by minimizing the LMS error function between the hypothesis and the data. The LMS

regression of the transistor data to polynomials for orders $n = 2, 4, 10$ are shown in Fig.1.9A as dashed lines.

Using a polynomial as a nonlinear function is only one possible choice of many. We will later consider mainly functions that that have been termed Artificial Neural Networks. These functions can be represented graphically as shown in Fig.1.10. Each node in such a graph is also called a neuron as it resembles to some extend the basic functionality of a biological neuron in the brain. Such an artificial neuron weights each individual input with an adjustable parameter, sums this weighted input, and then applies a nonlinear function such as a $\tanh$ function on this summed input. The output of each node is hence,

$$y_j = \tanh(\sum_i w_{ji} x_i). \tag{1.22}$$

This output can be the input to another node, and we can in such a way build elaborate functions with graphs of such nodes as shown in Fig.1.10B. We will later elaborate on specific network architectures that represent specific classes of non-linear functions that will be useful for specific applications. We will specifically explore how networks with many layers of neurons have advanced the capabilities of learning machines considerably which is now known as **deep learning**.

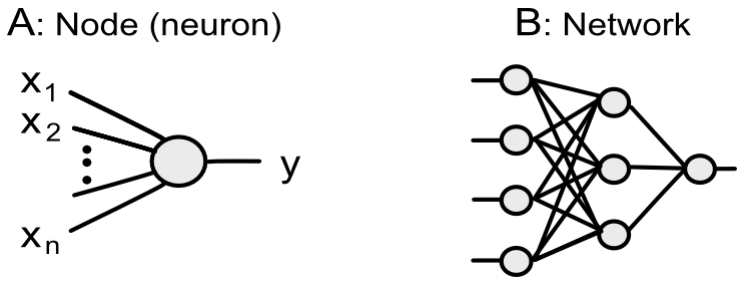

Fig. 1.10 Basic elements of an Artificial Neural Network (ANN). Each node represents an operation of summing weighted inputs and applying an nonlinear transfer functions f to this net input. The output of each node can become the input of another node or represent the output of the networks.

A major question when fitting data with fairly general non-linear functions is the complexity of the function in terms of the number of parameters such as the order of the polynomial or the number of nodes. The polynomial of order $n = 4$ seem to somewhat fit the transistor data also shown in Fig.1.9B. However, notice there are systematic deviations between the curve and the data points. For example, all the data between years 1980 and 1995 are below the fitted curve, while earlier data are all above the fitted curve. Such a systematic **bias** is typical when the order of the model is too low. However when we increase the order, then we usually get large fluctuations, or **variance**, in the curves. This fact is also called **overfitting** the data since we have typically too many parameters compared to the number of data points so that our model starts describing individual data points with their fluctuations that we earlier assumed to be due to some noise in the system. This difficulty to find the right balance between these two effects is also called the **bias-variance trade-off**.

The bias-variance trade-off is quite important in practical applications of machine learning because the complexity of the underlying problem is often not know. It then becomes quite important to study the performance of the learned solutions in some detail. A schematic figure showing the bias-variance trade-off is shown in Fig.1.11. The plot shows the error rate as evaluated by the training data (dashed line) and validation curve (solid line) when considering models with different complexities. When the model complexity is lower than the true complexity of the problem, then it is common to have a large error both in the training set and in the evaluation due to some systematic bias. In the case when the complexity of the model is larger than the generative model of the data, then it is common to have a small error on the training data but a large error on the generalization data since the predictions are becoming too much focused on the individual examples. Thus varying the complexity of the data, and performing experiments such as training the system on different number of data sets or for different training parameters or iterations can reveal some of the problems of the models.

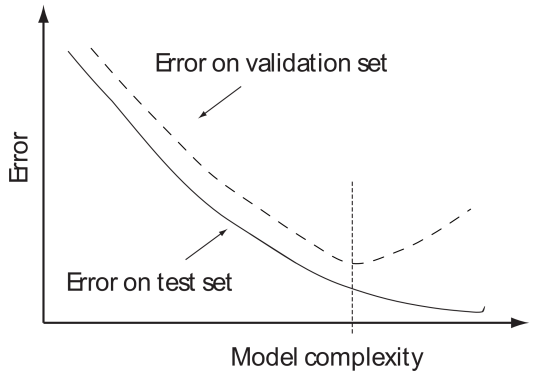

Fig. 1.11 Illustration of bias-variance trade-off.

Deep neural networks are a form of high dimensional non-linear fitting function, and preventing overfitting is therefore a very important component in deep learning. Deep networks have many free parameters, and large data sets (big data) has therefor been important for the recent progress in this area in combination with other techniques to prevent such as a technique called **dropout** that we will discuss later. In general one can think about techniques to prevent overfitting by restricting the possible range of the parameters. Indeed, learning from data already represents providing information about the values of the parameters, and restricting such ranges further is a key element in machine learning. This area is generally discussed under the heading of **regularization**. In a probabilistic framework this can be incorporated with a **prior**, a probability density function of our prior belief or restrictions in the parameters.

In the next section we will see that basic implementation of Machine learning methods are not difficult when using application programs that implement these techniques. This is good news. However, a deeper understanding of the methods is necessary to make these applications and their conclusion appropriate. The machine learning algorithms will come up with some predictions, but if these predictions are sensible is

important to comprehend and evaluate. Machine learning education needs therefore to go beyond learning how to run an application program, and this course aims to find a balance between practical applications and their theoretical foundation.