Chapter 2

Machine Learning Fundamentals

Francisco Câmara Pereira and Stanislav S. Borysov
Department of Management Engineering, Technical University of Denmark (DTU), Lyngby, Denmark

Chapter Outline

1 INTRODUCTION

At the time of writing this chapter, with so much buzz around concepts such as *artificial intelligence*, *big data*, *deep learning*, and *probabilistic graphical models* (*PGMs*), machine learning has gained a bit of a mystical fog around it. This excitement has more recently led to a series of dystopian visions that, among others, literally suggest the end of the world (e.g., Kurzweil, 2005; Bostrom, 2014)! We will delve into some of these aspects, below on the historical perspective, but first we want to demystify machine learning as a general discipline.

Machine learning combines statistics, optimization, and computer science. The statistics that we mention here is *exactly* the same discipline that has been taught for several centuries. No more, no less. In its vast majority, machine learning models also consist of functions that directly or indirectly end up as

$$y = f(\mathbf{x}, \boldsymbol{\beta}) + \epsilon.$$

In other words, we want to estimate (or predict) the value of a response variable, y, through a function $f(\mathbf{x}, \boldsymbol{\beta})$, where $\mathbf{x}$ is a vector with our observed (input) variables, and $\boldsymbol{\beta}$ is a vector with the parameters of our model. Since, in practice, our data (contained in $\mathbf{x}$) is not perfect, there is always a bit of noise and/or unobserved data, which is usually represented by the error term, ϵ. Because

© 2019 Elsevier Inc. All rights reserved.

we cannot really know the true value of ϵ, it is itself a *random variable*. Of course, as a consequence, y is a random variable as well.

It is almost sure that you are familiar with the most basic form of f, the linear model:

$$f(\mathbf{x},\boldsymbol{\beta}) = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_n x_n.$$

So, if you have a dataset with pairs of $(\mathbf{x}, y)$, the task is to estimate the values of $\boldsymbol{\beta}$ that best reproduce the linear relationship. You do this using optimization, sometimes called the "training" process, which is minimization of the difference between the true values of y and model's predictions, also known as a loss function. By now, you are bored? Great, let us now imagine that $f(\mathbf{x})$ is instead defined as

$$f(\mathbf{x},\boldsymbol{\beta}) = f_K(f_{K-1}(\ldots(f_1(\mathbf{x},\beta_1)\ldots),\beta_{K-1}),\beta_K),$$

where each function f_k transforms its input and gives its result to the following one. Or, before doing that, it processes its data in a distributed fashion, for example, $f_{K1}(x_1,\ldots,x_j), f_{K2}(x_{j+1},\ldots,x_k), \ldots, f_{KL}(x_{k+1},\ldots,x_r)$, where we have L subfunctions, each processing a sub-part of the data. Imagine you have really a lot (thousands!) of such functions. Congrats, you just discovered a deep neural network (DNN) (or, in fact, many other types of models, including PGMs, another popular one these days). Of course, it can become much more complicated, but the principle is precisely the same. You have a sequence of functions, each one with its own set of parameters (sometimes shared among many different functions) where the output of one is the input of the other. Due to the large number, the estimation of these parameters may require a large amount of data and computation. And that is where it becomes distinct from classical statistics.

In fact, there are several ways to categorize Machine Learning tasks. The most common way depends on whether the target variable (our y above) is present in the problem itself, leading to *supervised* and *unsupervised* learning paradigms. *Supervised learning* implies presence of the target variable we would like to predict so the model receives a feedback from the "teacher" how its prediction is close to the ground truth. It also includes semi-supervised learning (when the target values are partially missing), active learning (number of possible target values is limited so the model should decide which data samples to use first), and reinforcement learning (y is given in a form of reward for a set of actions performed by the algorithm).

Sometimes, the target variable is not present, you only have data points $\mathbf{x}$, and thus the problem formulation becomes slightly different. In such a case, the objective typically becomes to find patterns in the data or *clusters* of data points that *seem* to fit together. This is an example of *unsupervised learning*. From a statistical point of view, it is an example of generative learning of the data's joint distribution $p(\mathbf{x})$, while most well-known supervised learning algorithms are regarded as discriminative, that is, we estimate the conditional distribution $p(y \mid \mathbf{x})$. Some supervised learning algorithms exist, though, that are also generative (i.e., we estimate $p(y, \mathbf{x})$), which is the case of PGMs.

Depending on their output, machine learning models can be classified as regression (the target variable is continuous), classification (the target variable is categorical), and others like clustering (often unsupervised), probability density estimation, and dimensionality reduction (e.g., the Principal Component Analysis algorithm).

Although it would be almost impossible to provide a complete overview of this highly dynamic field, this chapter will present the several major forms of what is known today as *machine learning* and provide some tips about where to go next, if the reader wants to be part of this revolution, or simply understand more what it is about. But first, it is worthwhile to give a historical perspective.

2 A LITTLE BIT OF HISTORY

Machine learning was born as one branch within the major field of artificial Intelligence, which also includes others such as Knowledge Representation, Perception, Creativity (Russel and Norvig, 2009; Boden, 2006). The term "machine learning" was coined by Arthur Samuel, as early as 1952, who created the first program that could play and learn the checkers game (Samuel, 1959). The "learning" process here corresponded to incrementally updating a database with moves (board positions) and their score, according to probability for later success in winning or losing the game. As the computer played more, it improved its ability to win the game. This is probably the earliest version of *reinforcement learning*,[1] today an established sub-field for scenarios where the whole data set is only incrementally available and/or the label of each data point is not directly observable (e.g., the value of a game move may only be observable later in the game or even at its very end in the binary form of "win" or "lose").

During the 1960s and 1970s, many researchers were enchanted by the concept of a machine that is pure logic, and the memory and computer processing limitations were extremely tight, comparing with nowadays. More than that, the belief that human intelligence could all be represented through logic (a "computationalist" point of view) was widespread and exciting. This naturally led to an emphasis on rule-based systems, representing knowledge through logic (e.g., logical rules, facts, symbols) and natural language processing.

In parallel, other researchers believed that we should study more the neurobiology of our brain and replicate it (a "connectionist" point of view), in what became known as artificial neural networks (ANN). The first well-known example is the *perceptron* (Rosenblatt, 1957), which applies a threshold rule to a linear function to discriminate a binary output. It is depicted in Fig. 1.

1. There are records of an earlier program to have reinforcement learning capabilities, the "shopping machine", by Anthony G. Oettinger in 1951. The task was to answer the question "in what shop may article *j* be found?", and the algorithm could simulate trips to eight different shops (Moor, 2003).

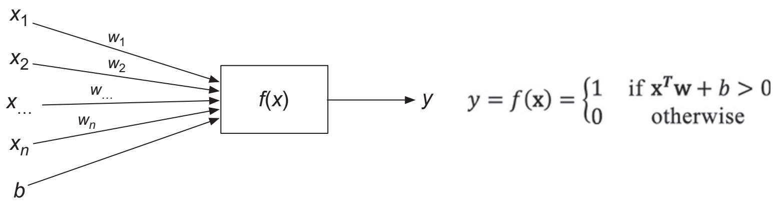

x_1
x_2
$x_{\dots}$
x_n
b
w_1
w_2
$w_{\dots}$
w_n
$f(x)$
y
$y = f(\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{x}^T\mathbf{w} + b > 0 \\ 0 & \text{otherwise} \end{cases}$

FIG. 1 Perceptron.

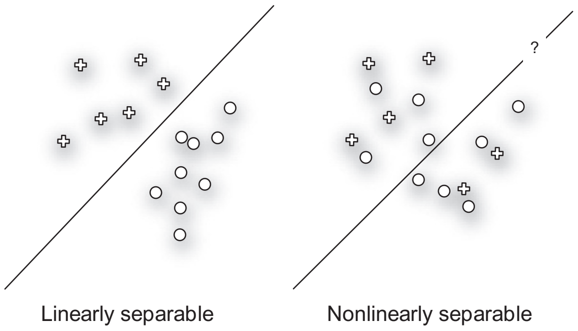


FIG. 2 Linear separation. There are two classes (+ and o), and we want a rule (a *classifier*) that discriminates between them.

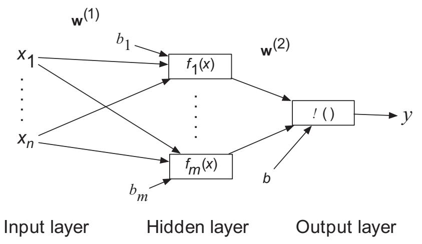
The perceptron was quite popular for a time as it can represent some logical gates (AND and OR), but not all (X-OR). Due to the latter limitation, such a simple ANN is limited to linearly separable problems (Fig. 2, left), whereas real problems quite often are inherently nonlinear in their nature (Fig. 2, right). This strong criticism pointed out by Minsky and Papert (1969) led to a decade with virtually no research on Neural Networks (NN), also known as the "first AI winter."

The third relevant research thread from the 1960s and 1970s relates to the "nearest neighbor" concept. Cover and Hart (1967) published the "Nearest neighbor pattern classification," which effectively marks the creation of the "pattern recognition" field and the birth of the well-known K-nearest neighbor algorithm (K-NN). Its underlying principle is simple: If we have a problem to solve (e.g., given an input vector $\mathbf{x}$, classify it into a class, so the target variable y becomes categorical), we can look for K most *similar* situations in our database. Of course, the key research question is how to define *similarity*, which boils down to the comparison of $\mathbf{x}$ vectors. The typical option is the Euclidean distance, but there are a lot of other metrics which are much more suitable given the nature of the data at hand (e.g., categorical, textual, temporal). Another challenge in K-NN is to define K. What would be the best number of similar examples from the data to use in the algorithm? This introduces a concept of a *hyperparameter*, which should be defined a priori, in contrast to the parameters $\boldsymbol{\beta}$, which are to be determined during the training. Of course, in those days, the complexity of data and computational power and memory were so low, that only much later we find good answers.

The most basic unsupervised clustering algorithm, known as *k*-Means, was first proposed by Lloyd (1957) at Bell Labs (but published only in 1982) and independently by Forgy (1965). Its main steps to find clusters in the data is, having *k* cluster centroids randomly initialized (*k* is another hyperparameter!), to assign each data point to a cluster with the nearest mean (step 1), and update these centroids using the data points assigned to them (step 2). Repeat these steps iteratively until convergence and the task is solved. Later, this procedure was rethought and generalized as the expectation-maximization algorithm for a Gaussian mixture model (Dempster et al., 1977).

The 1980s saw the rebirth of ANNs with enough computing power to allow for multilayer networks and nonlinear functions. Notice that, if we replace the function $f(\mathbf{x})$ in Fig. 1 for a logistic sigmoid function (as opposed to the 0/1 step function), we essentially obtain a binary logit model (or logistic regression), and this is the basis for the multilayer perceptron algorithm (MLP) summarized in Fig. 3.

Notice that now there are two sets of weights ($\mathbf{w}^{(1)}$ and $\mathbf{w}^{(2)}$). The first one associates a vector of weights ($\mathbf{w}_j^{(1)}$) to each of the f_j functions in the *hidden layer*. As mentioned, the typical form of these functions is either the logistic sigmoid (Fig. 4A) or the hyperbolic tangent. Each one of these then provides the input to the final (output) layer, weighted by the vector $\mathbf{w}^{(2)}=[w_1^{(2)}, w_2^{(2)}, \ldots, w_m^{(2)}]^T$. The function σ is typically another sigmoid (if we are doing classification) or the identity function (if we are doing regression).



$$y = \sigma\left(\sum_{j=1}^{m} w_j^{(2)} f(\mathbf{x}^T \mathbf{w}_j^{(1)} + b_j) + b\right)$$

FIG. 3 Multilayer perceptron.

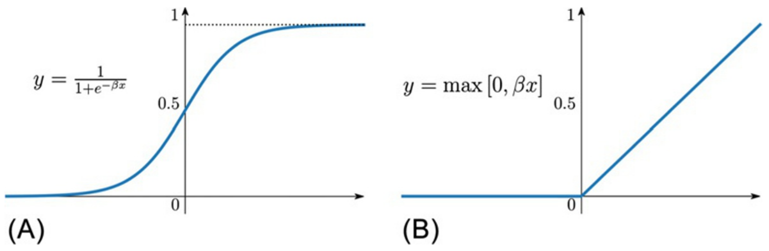


FIG. 4 (A) Sigmoid function and (B) rectified linear unit (ReLU).

From a statistical modelling perspective, an MLP is *simply* a model with *m* latent variables that are combined in the output layer. The development of the backpropagation algorithm (Rumelhart et al., 1986), which allows the MLP to be efficiently estimated from data, was fundamental for its popularity. Furthermore, Hornik et al. (1989) demonstrated that one layer of suitable nonlinear neurons followed by a linear layer can approximate any nonlinear function with arbitrary accuracy, given enough nonlinear neurons. This means that an MLP network is a universal function approximator, which motivated plenty of excitement. However, it was also verified during the 1990s that the MLP often runs into overfitting problems, and its opaqueness (the weights have no clear interpretation, neither does the hidden layer) often limits its application.

Thus, after the MLP, we saw a relatively low investment on NN research, marking what was known (guess what…) as the "second AI winter," until the boom of DNN models since 2010. The key ingredients for its success were new sophisticated architectures, computational resources and amounts of data available. Differently to an MLP, which always has one layer (Hastie et al., 2009), a DNN may have multiple layers, sometimes dozens or hundreds. While the MLP is always fully connected (all elements, or neurons, in one layer are connected with all elements of the subsequent layer), a DNN is often very selective, with different sets of neurons connected to different parts of the subsequent layer. The most popular activation function is not anymore the sigmoid; instead it is the rectified linear unit (ReLU), shown in Fig. 4B. The ReLU function became popular as it helps to mitigate the vanishing gradient problem (exponential decrease of a feedback signal when the errors propagate to the bottom layers), which hindered training of NNs with many layers.

Back to the 1980s, it was the golden age of the *Expert Systems*. These usually relied on explicit representation of domain knowledge in the form of logical rules and facts, obtained by interviewing experts. The new profession of *Knowledge Engineer* would typically combine the ability to do such interviews and code the knowledge into systems such as FRED (Freeway Realtime Expert System Demonstration) or FASTBRID (Fatigue Assessment of Steel Bridges), just to mention in the transportation domain. For a notion of expectations and opportunities of such systems for transportation, from the eyes of the early 1990s, the paper from Wentworth (1993) is an interesting read.

The major problem with *Expert Systems*, and related rule-based paradigms, has always been the costly process of collecting and translating such knowledge into a machine. To make things more complicated, such knowledge is often not deterministic or objective, which eventually led to the development of *fuzzy logic*, as a way to improve the Expert Systems potential, which were still huge collections of hand-coded rules. On the other hand, one should ask is the machine *learning* at all? Aren't the Expert Systems instead an attempt to mechanize human knowledge, explicitly extracted and hand-crafted by humans (the coders)?

A different approach to this problem comes from probability theory. Between 1982 and 1985, Judea Pearl created the concept of *Bayesian Network*, where domain knowledge is also important to structure relationships between variables, but not anymore as long complex rules. Instead, he proposed to decompose knowledge into individual relationships, inspired by the concepts of causality and evidence. More importantly, he developed the *belief propagation* (BP) method for learning (or making inference) from data (Pearl, 1982, 1985). Such networks were successful, particularly in classification problems, and later, the challenge of the automatically generation of their structure from data (as opposed to domain expertise) was proposed by Spirtes and Glymour (1991), and remains a relevant topic until today.

But perhaps the hottest trend in the 1980s and 1990s were in fact the kernel methods, and the support vector machines (SVM) in particular. We get back to the problem of linear separation (Fig. 2), where reality tends to be nonlinearly separable. If we can find a function mapping that, when applied to a data set, makes it linearly separable, then we solve the problem. Typically, this implies mapping data to a higher dimension. Fig. 5 shows an example. On the left, we have the original data set (class "o" has points $\{-7, -5, -4, 2, 3, 4\}$, while class "+" is $\{-2, -1, 0, 1\}$). This is one-dimensional, so let us add a new dimension by simply squaring the values. Each point is now $(\mathbf{x}, \mathbf{x}^2)$. On the right, we plot those points. Here it goes, we can now draw a straight line to separate the classes!

Vapnik and others discovered that there is always one (higher) dimension where a data set is linearly separable. More importantly, they found a process, called the *kernel trick*, where such mapping is done without the need to explicitly project to the higher dimension (Boser et al., 1992) as we did above. Using convex optimization to train prediction models on data, the SVM obtains global optimality guarantees for the model/parameters learned. The SVM algorithm learns the separating line (a straight line in our example, or a hyperplane in higher dimensions) that has the largest distance to the nearest training-data point

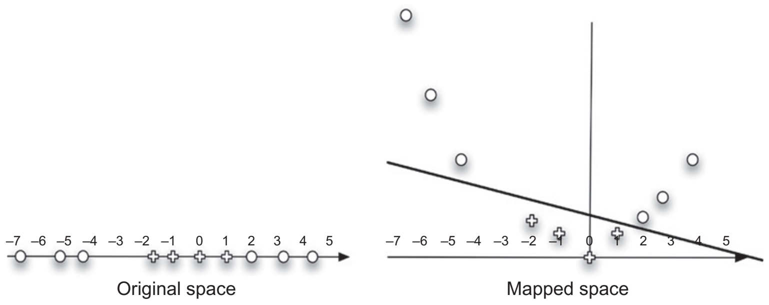


FIG. 5 Mapping 1-D data points, which are nonseparable linearly, to a higher dimensional (2-D) space makes them linearly separable.

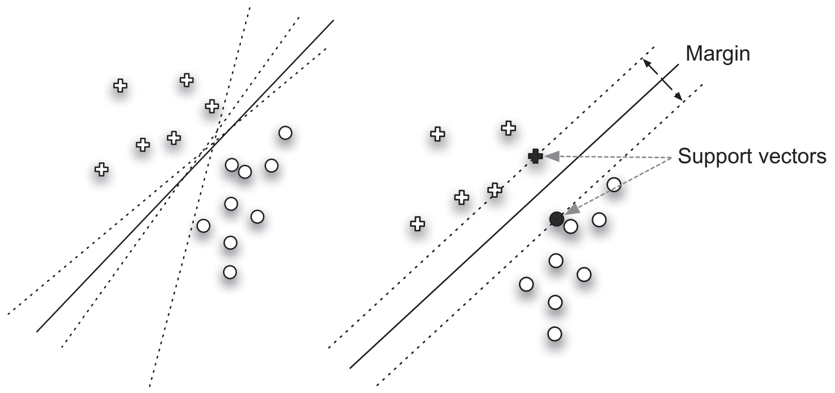


FIG. 6 SVM finding the best hyperplane. There are many possible choices of linear separation (left). Support vectors are found that maximize the margin (right).

of any class. In other words, it finds the dividing line where each side has the maximum margin to the closest data points. The larger this margin, the lower the error. Fig. 6 illustrates the concept. Notice that the maximum margin always coincides with vectors from each class. These are the closest data points, the *support vectors*. Of course, all this happens at a potentially very high dimensionality and the 2-D example is only provided for illustration purposes.

The concept of kernel is useful much beyond SVM. Intuitively, a kernel function defines similarity between two data points. The concept is illustrated in Fig. 7 with one-dimensional kernels. Imagine that you want to compare two

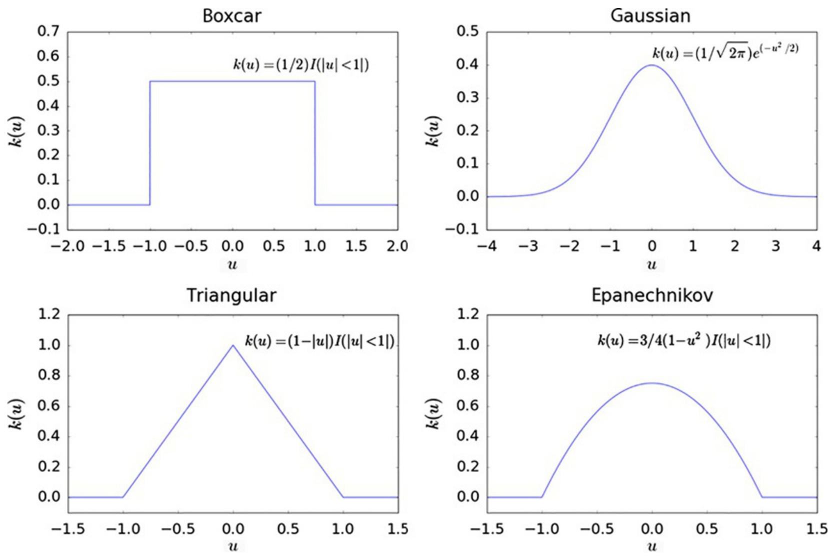


FIG. 7 Four different kernels.

data points, $x_1$and x_2. The parameter u is simply the scaled difference between them, $u=(x_1-x_2)/h$. The vertical axis represents their similarity. Naturally, in all cases, when $u=0$, the similarity is always maximum, but the way it decays is considerably different from kernel to kernel.

Take the K-NN, created in the 1960s, now we can use kernels to define similarity in a different way. The similarity of a vector $\mathbf{x}$ to other vectors $\mathbf{x}'$ is simply determined by the value of the kernel function, which depends on the distance between them and hence has a higher value for neighbors. We can even use different distance definitions for different dimensions of $\mathbf{x}$, and, more importantly, we do not need to define K anymore. However, hyperparameters of a kernel, such as the characteristic length h, now come into play!

By the early 2000s, the four general trends in Machine Learning had been formed:

1. *Rule-based systems:* These include hand-crafted rules à lá expert systems, decision trees, decision tables, logic programming. They are the most interpretable machine learning systems. From a laymen standpoint, they are intuitive to implement and debug, and often have competitive accuracies.
2. *Kernel-based algorithms:* These are typically based on the concept of neighborhood, and include a wide range, from K-NN to SVM and their derivatives. They demand a careful definition of similarity (or *kernel*), and do not require the definition of the model function form, as opposed, for example, to linear or polynomial regression, where we define the general function form and estimate their parameters from the data. For these aspects, these models are generally called *nonparametric*. To run a kernel-based algorithm, one has to retain the data set (or part of it) for use whenever a prediction is made. Another weakness of such methods is poor scaling with the data set size due to the computationally costly data matrix inversion involved into the training. These issues have contributed to the revival of NNs, which also support *online learning*, avoiding reuse of the entire data set to incorporate new data points.
3. *DNN:* A rebirth of the NN view of machine learning, DNNs, are reaching the news in almost every corner of the world. They have hit fantastic benchmarks, particularly in the areas where plenty of high-dimensional data are present such as computer vision, automatic translation, robotics, computer games, and speech recognition. Such successes are behind the resurgence of artificial intelligence as a hot topic for at least the next few years. These NNs are called "deep" because they rely on many layers of neurons, sometimes hundreds, stacked together allowing for processing the data in a hierarchical manner. Often having thousands or even millions of parameters to tune, they usually depend on massive amounts of data and computing power. Curiously, advanced computer

graphics cards (initially designed for gaming!) became an essential piece of hardware for deep learning methods, because they can manipulate extremely large matrices in parallel (and you can put many such cards in a single computer now). A major drawback of this approach remains its low interpretability.[2] A trained DNN ends up with thousands of parameters that have hardly a meaning for a human modeler. Also, deep learning algorithms still lack a sufficiently established set of theoretical foundations, being mainly empirical and ad hoc.

4. *Bayesian statistics:* Another current trend builds on Bayesian probability theory to make inference on data, that is, predict the distribution of a variable of interest and model's parameters given a set of observations. Typically, such models require a definition of their structure (how the different variables relate to each other, and with the observations), either manually or through automatic processes (a.k.a. Bayesian Network structure learning). A particular advantage of these models is that they allow the combination of domain-based formulations (e.g., well-known domain laws, with clear function forms) with data-driven methods (e.g., unknown function form). This area has been extremely prolific in the 2010s, with the emergence of "probabilistic programming languages"[3] and the general family of "Probabilistic Graphical Models." In contrast to other methods, the Bayesian approach allows do define the full distribution of the model's parameters instead of point estimates found, for instance, by the backpropagation algorithm in NNs.

To conclude this brief historical overview, it is worth saying that despite many striking advances in Machine Learning we witness in 2010s (e.g., Generative adversarial networks can generate photorealistic pictures of nonexisting humans, and deep reinforcement learning algorithms can easily beat the best Go players), we achieved artificial intelligence only in a narrow (or *weak*) sense, where it outperforms human capabilities in very specific tasks. We are still quite far away from artificial *General* intelligence, which can be applied to any problem on a par with humans; not mentioning artificial *Super* intelligence, which is even difficult to define except that it should be qualitatively smarter than any human being.

At the time of writing, we can identify two major trends of new research in the machine learning community. So, in the remainder, we will focus on DNN and Bayesian models, highlighting their weakness and strengths. To complete our short introduction into the subject, we will briefly touch basics of machine learning experiments such as model testing and comparison.

2. However, various indirect approaches can be applied to address this issue to some extent, for instance, see Ribeiro et al. (2016).
3. http://probabilistic-programming.org/.

3 DEEP NEURAL NETWORKS AND OPTIMIZATION

As we already saw, a DNN is a straightforward extension of an MLP with multiple layers of artificial neurons (Fig. 8).

Its *loss function*, L, measures the difference between the model's predictions and ground truth. For instance, it can be mean squared error (MSE) for continuous outputs (regression) or entropy loss for categorical outputs (classification). Since it is a function of the parameters (weights and biases), in order to minimize it, the *gradient descent* algorithm is usually applied. All we need is to calculate gradients of the loss function with respect to the parameters and update them in the direction of a loss function minimum:

$$w_{ijk}^{\text{new}} = w_{ijk}^{\text{old}} - \eta \frac{\partial L}{\partial w_{ijk}},$$

where η is the *learning rate*, which defines how big the steps should be. The famous backpropagation algorithm is basically a chain rule applied to the loss function differentiation to calculate the required gradients. In a nutshell, it says that each neuron contributes to the loss function proportionally to the weights of its connections to the following neurons. We can simply use this fact to calculate these contributions starting from the output layer and propagate them backward in the network using its weights and the derivative of the activation function f.

The problem is that the plain gradient descent will almost surely get stuck in a local minimum or saddle point (Fig. 9). Many different optimization techniques have been elaborated to tackle this issue. The most simple and popular one is stochastic gradient descent, where the gradients of the errors are calculated over small subsets of the data, called *mini-batches*. In fact, it introduces some noise to the estimated gradients, so the parameters have a chance to escape

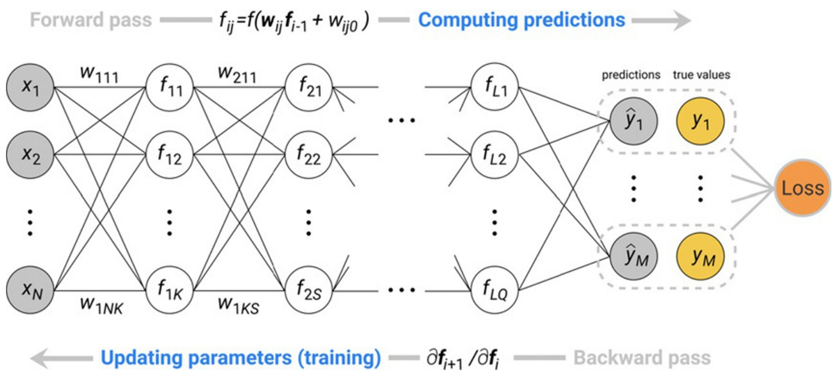


FIG. 8 A generic deep neural network architecture. To compute predictions, output from a layer passed to the next layer using the network's parameters. During the training step, parameters of each layer are updated using the loss function gradients propagating in the opposite direction ("backpropagation").

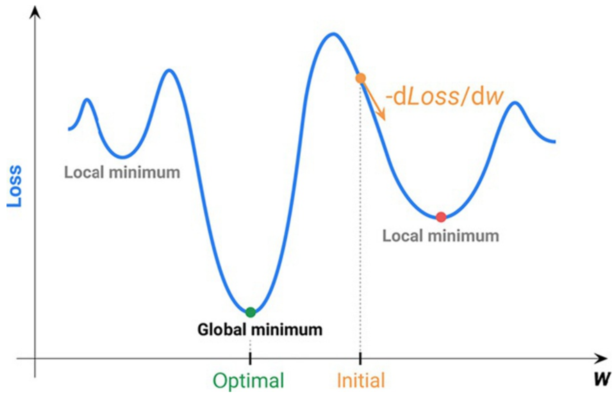


FIG. 9 Loss function is almost always a nonconvex function with a lot of local minima, so the plain gradient descend algorithm will almost surely converge to one of them.

from unwanted regions of the loss function. Many modifications to the parameters' update rule have been also proposed, to name a few, momentum, Nesterov, Adagard, Adadelta, RMSprop, Adam, AdaMax, and many others. These modifications usually incorporate gradients from the previous learning steps into the update rule.

Being quite general and flexible, a DNN has become a very popular and powerful model with a wide range of applications. One of its main strengths is the ability to process information in a hierarchical way, automatically capturing new levels of abstraction in the data, effectively dealing with the *curse of dimensionality*. This problem arises because, with growth of dimensionality (i.e., number of features) of the data, the volume of a unit sphere in this space growths exponentially. As a consequence, to explore this volume and provide reasonable statistical estimations, a model requires exponentially more data samples. One of the known approaches to tackle this problem is dimensionality reduction, which assumes that there are only few important dimensions, which can be represented as (often nonlinear) combinations of the initial ones. A DNN is believed to do it automatically, where each following layer learns new low-dimensional representations of the data.

For example, consider a document topic classification problem given scanned handwritten text as an input. It would be almost impossible to perform this semantic analysis directly on the raw pixel intensities. Instead, a DNN detects strokes and curves first, then tries to identify letters, which in turn constitute words. This can be achieved thanks to the layered structure that *theoretically* can learn any complex mapping. We stress the word "theoretically" because it appears not so trivial to do it in practice. The main challenge is training of a DNN: Fitting its parameters essentially boils down to finding a minimum of a nonconvex function in a highly (thousand- or even million-) dimensional space. Therefore, plain fully connected DNNs are rarely used alone. Many different hybrid architectures of DNNs tailored for different purposes have been proposed. Below, we discuss a convolutional neural network (CNN) and a recurrent neural network (RNN)—the two most fundamental prototypes which can be found in almost all modern deep learning models.

A CNN architecture is believed to resemble a visual system of a brain. It consists of several convolutional-pooling layer pairs followed by a fully connected network (Fig. 10). Neurons in a convolutional layer, which share their weights, scan through the input and produce multiple outputs of the same size as the input,[4] also known as *feature maps*. The layer is called convolutional because it literarily performs convolutions of the input using adaptive kernels represented by an NN. A pooling layer downsamples these feature maps, using, for example, the *max* function over small contiguous regions, effectively reducing their dimensionality. These layers preserve spatial correlations and are able to capture the before-mentioned hierarchical features, such as strokes, basic geometric shapes, and so on. They are followed by a fully connected NN with a small number of hidden layers which is now able to solve the initially complex task (e.g., classification of cats vs. noncats). Originally, a CNN was designed to process images (Lecun et al., 1998) but, in principle, it can be applied to any correlated data of a fixed length such as text or time series. For image-related tasks, a CNN can be pretrained on a large set of images for further reuse, where the fully connected layer is then replaced and trained from scratch for the task at hand (the approach known as *transfer learning*).

An RNN has a quite simple underlying idea appeared to be very powerful in practice. It is capable of storing state of the network so it has some notion of memory (Fig. 11). Output of an RNN depends not only on the current input but also on the all previous inputs. RNNs are used to process sequences of a variable length with particular emphasis on natural language processing applications. However, it turned out that the plain RNN implementation was not practical due to the limited ability to maintain its memory for a long time.

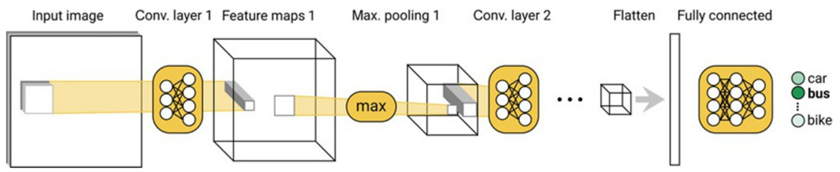


FIG. 10 An example of a convolutional neural network. A sliding window over the input image (or its all channels such as RGB) is used as an input to a convolutional layer (which is basically a fully connected NN with two layers) which produces a number of outputs (feature maps). These feature maps are then downsampled using the max function in a pooling layer. The output can be passed to the next convolutional layer and the whole procedure repeats several times. The final output is flattened into a vector and passed to fully connected layers, which compute predictions.

4. Or slightly smaller, taking into account the parameters of the sliding window.

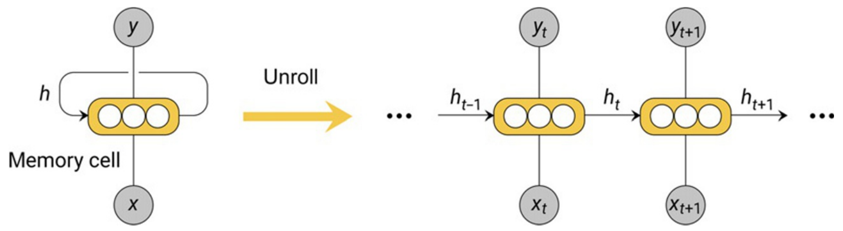


FIG. 11 An example of a recurrent neural network. The RNN computes the output, y, using not only the input variables, $\mathbf{x}$, but also its hidden state, h (left). The hidden state usually comes from the previous inputs to the RNN, so for a sequence of inputs, the RNN can be "unrolled" in time (right). The memory cells can be either plain fully connected NN or contain special internal gates to control their memory in a better way.

To address this issue, many recurrent neuron types were proposed. The most popular ones are the long short-term memory (LSTM) cell (Hochreiter and Schmidhuber, 1997) and Gated Recurrent Unit (GRU) (Cho et al., 2014), which contain special gates to control and propagate their internal (or *hidden*) state in a more efficient way. Similar ideas were also applied to regular feed-forward NNs leading to various network architectures such as deep residual networks and highway networks, which are capable of propagating information (both forward and backward) through hundreds of layers. These networks become highly efficient for image-processing tasks.

There are countless ways to modify and combine various DNN architectures due to the modular structure of DNNs equipped with end-to-end training via backpropagation. For instance, for a video frame prediction task, a CNN can be used to encode each frame as a low-dimensional vector (*embedding*) using, for example, output of the top hidden layer of its fully connected part, while an RNN is then applied for sequential prediction of these vectors. Finally, a *deconvolutional* NN can be used to decode these representations back to the images. Another example can be given by a caption-generation task, where RNN is used to process textual input and CNN to process images. Interestingly, a single joint loss function which accounts for both CNN and RNN can be used in an unsupervised manner to encode images and text in the same low-dimensional vector space. In this case, a picture of a cat and the word "cat" will end up being very close in this space of embeddings.

Some would say that deep learning has been so prominent by the end of 2010s that it overshadows almost everything. While it has been behind the rebirth of *Artificial Intelligence* as a hot topic, supported by gigantic teams in Google, Facebook, or Amazon, we would recommend caution in this interpretation. Deep learning has shown incredible results in image and sound, but it has a long way to go in domains such as transport, which often involve human behavior modelling and simulation. The main reasons are low interpretability of DNNs, difficulty to incorporate prior and domain knowledge, stability issues, and poor ability to provide statistical properties for the estimates. In this regard, Bayesian approaches for PGMs may have a strong say.

4 BAYESIAN MODELS

Machine learning can be viewed from a statistical point of view as estimation of probability distributions, and a very powerful way of representing such distributions can be achieved through a *PGM*.

A PGM is primarily a graphical representation framework. In a PGM, each node corresponds to a variable and each edge to a dependence between a pair of variables. Whenever a PGM is a directed graph representing conditional dependencies, it is called a Bayesian network, which is what we illustrate here. Fig. 12A shows a linear regression model. Notice that the response variable, y, is dependent on X (inputs), β (parameters), and σ (standard deviation of the error). A PGM represents a *factorization* of the joint probability of all random variables. In Fig. 12A, we have $p(y,X,\beta,\sigma)=p(y|X,\beta,\sigma)p(\beta)p(\sigma)$. Notice that X is observed data (constant), which is why it is shaded. The likelihood $p(y|X,\beta,\sigma)$ is specified as $\mathcal{N}\left(\beta^T X, \sigma^2\right)$.

PGMs are in fact more than just a representation. They are a machine learning model in their own right, having a variety of tools for model inference, one of which is called BP. BP is based on the idea of *message passing*, whereby inference for each random variable follows a specific sequence, based on what is *known* at each moment. To give an example, let us estimate the parameters β for the linear model in Fig. 12A. We will add a trivial but useful extension, two prior distributions for β and σ (e.g., from domain knowledge, or previous data), determined by parameters ϕ and γ, that is, we have $p(\beta|\phi)$ and $p(\sigma|\gamma)$, respectively. Imagine we have received a batch of data (matrix X is a set of input values, and vector, $\mathbf{y}$, a set of y values). A possible sequence of messages to be passed is illustrated in Fig. 12B, with numbered arrows.[5] Both priors propagate their distributions initially (1 and 2). We now know the distribution of σ, to be further transmitted (3). β is what we want to estimate, so it will *not* transmit

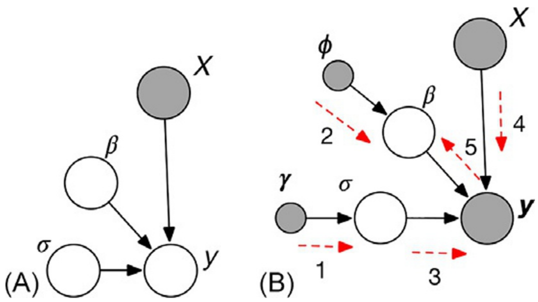


FIG. 12 A graphical model for regression (A) and belief propagation example (B) calculating the value (marginal) of β.

5. Factor graphs, a specific variant of PGMs, are known as a more appropriate representation for belief propagation, but we keep a simpler representation for the sake of an intuitive explanation.

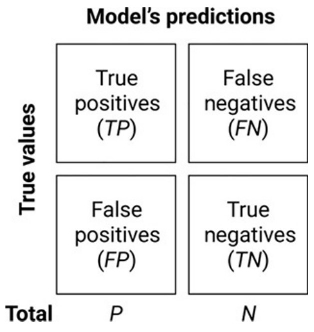
anything; it will only receive. On the other side, input data X is observed and also transmitted as point-mass (i.e., as constant) (4). Together with the received data $\mathbf{y}$, we now have all the ingredients to propagate to β the message 5, as $p(\mathbf{y}|\beta) = \int p(\mathbf{y}|\beta, \mathrm{X}, \sigma)p(\sigma|\gamma)\mathrm{d}\sigma$. Notice that we are marginalizing out σ (we do not marginalize $\mathbf{y}$, X, and γ because they are constants), and so message 5 is exclusively a function of β, and works like a likelihood function. Effectively, the marginal distribution of β is determined by multiplying (2) and (5), that is, $p(\beta|\,\mathbf{y}) = \frac{1}{Z}p(\beta|\,\phi)p(\mathbf{y}|\,\beta)$, where Z is the normalizing constant. We could also simultaneously estimate σ and β, in which case we would have an iterative process until convergence (β would also send a message).

Moreover, some or all of these messages may have nontractable forms and need to be approximated by well-known methods such as Markov Chain Monte Carlo (MCMC) or Variational Bayes. BP is a very intuitive and flexible method for Bayesian inference that builds on the modularity of a PGM. Directionality and message sequence are dependent on what we *know* and what we *want to know* at each moment, so in practice any nonobserved variable or parameter can be estimated through this process, as we did for β, and this includes, obviously, the response variable, y.

Finally, advances in inference algorithms and growth of computational power opened a path toward deep Bayesian models similar to DNNs, capable of dealing with numerous latent variables in a hierarchical way. Applications of Bayesian approaches in Deep Learning and combinations of PGMs and DNNs are other areas of research, where a Variational Autoencoder (Doersch, 2016) can be given as a typical example.

5 BASICS OF MACHINE LEARNING EXPERIMENTS

Having the model $f(\mathbf{x},\boldsymbol{\beta})$ trained, a natural question to ask is how to estimate its performance? Well, if our target variable y is continuous, common choices of performance metrics are MSE and its root (RMSE), mean absolute error (which is less sensitive to outliers), or the Pearson correlation coefficient. For classification problems, a *confusion matrix* and a whole zoo of metrics derived on its base are used. For binary classification, the confusion matrix has the following form.



The most popular metrics are accuracy = (TP + TN)/(P + N), precision = TP/(TP + FP), recall (or sensitivity) = TP/(TP + FN) and F1 score, which is simply a harmonic mean of the last two = 2 * precision * recall/(precision + recall). It is worth noting that there is no single "golden standard" in the world of metrics. Each metric should be considered in connection with modelling objectives. For example, if it is important not to miss any document in an informational retrieval task, priority might be given to recall over precision. Or, in medicine, cost of a False Negative error might be much higher than cost of False Positive.

But the most important question to ask (which basically the whole field is about) is the following: How well can the model *generalize*, that is, make reliable predictions for new, unseen data? Of course, machine learning models are not oracles and cannot guarantee the exact value of something which has not been observed yet. Nevertheless, they can still provide some useful statistical intuition about the expected future performance using some tricks with the data available. With this aim, the data are divided into train and test sets,[6] pretending that the latter is hidden from the model. The division ratio is arbitrary, but usually 70%–80% of the data is used for training and remaining 20%–30% for test purposes. So, we simply use the train set to fit parameters $\boldsymbol{\beta}$, but evaluate the model's performance using the data from the test set. Be careful about i.i.d. assumptions about the data samples, whether they can be randomly shuffled or not. Another important point to remember is that machine learning models are mainly designed for interpolation so it is difficult to expect fair performance of the model on any data outside boundaries of the train set. In other words, a test set should normally possess the same statistical properties as train data. Also, a closer attention should be paid to various data set problems, especially when dealing with imbalanced data sets, and statistical techniques to address them (e.g., boosting and bagging).

Using this trick, we can also then compare different models, $f_1, \ldots, f_n$, to select the best one. Here, the importance of baseline models should be stressed. It is a rule of thumb to try simpler models first. A mean predictor—the simplest baseline you can imagine, which always predicts the mean of the train set for regression or the most frequent class for classification independently on the input $\mathbf{x}$—or a linear model are always a good choice to start with. It gives a reference point for your fancy model and provides better understanding of the data. Then, you can try to beat more complex state-of-the-art models. Given the same performance of two models, you should be always in favor of a less complex one (the famous Occam's razor). However, sometimes the choice might be made towards a more interpretable "white-box" model.

The concept of generalization can be also illustrated in the following way. Consider the training dataset consisting of N data points which come from the true (but unknown to us) distribution $y = x^2 + \epsilon$, where ϵ is a normally distributed

6. In-sample and out-of-sample data in statistics terminology.

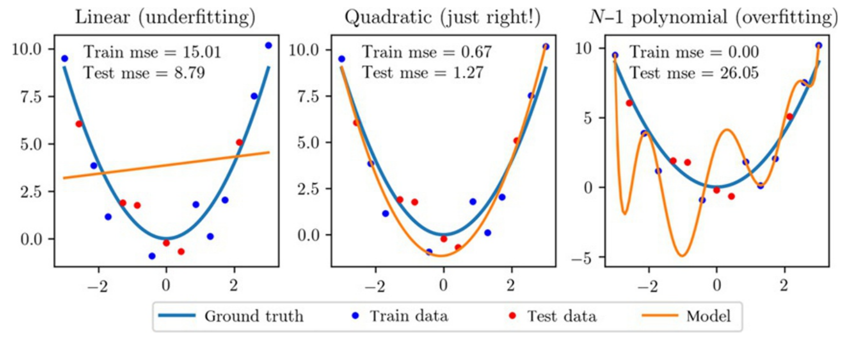


FIG. 13 Underfitting and overfitting.

noise term with zero mean and small variance. If we choose our model in this quadratic from, we will unsurprisingly get low errors both on train and test sets (Fig. 13, middle). But suppose that we want to try a linear model first. In this case, the model will have higher errors on both train and test sets, or, in other words, *underfit* or have *high bias* (Fig. 13, left). If we go to another extreme and consider a polynomial model of $N-1$ degree, $y=\sum_{i=0}^{N-1}\beta_i x^i$, it will go through every point in the train data. The train error in this case will be exactly zero but the model's performance on the test set will still be poor (Fig. 13, right). In this case, the model *overfits* or has *high variance*. Under/overfitting is also called a bias-variance tradeoff. The problem of finding the best model is to find a sweet spot somewhere in between. Here, plotting of the *learning curves* might be really helpful (Fig. 14).

If the data set is not huge, a straightforward extension of the train/test partition, called *crossvalidation* (CV), can be employed. In the conventional k-fold CV approach, the data set is divided into k bins. Then, the data from $k-1$ bins are used for training and the remaining kth bin is used for testing. The procedure

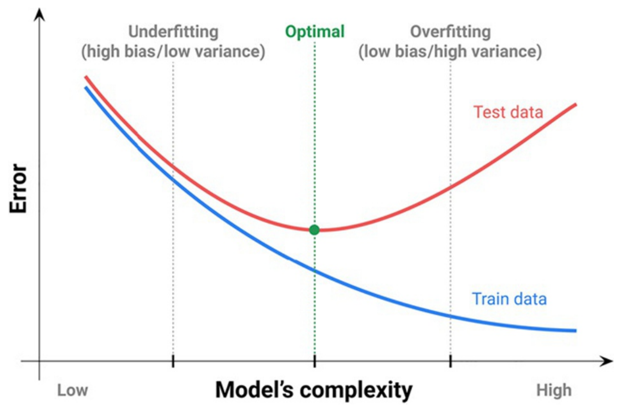


FIG. 14 Learning curves illustrate prediction errors for train and test data depending on the model's complexity.

is repeated k times so the model's performance is averaged over the all bins. The extreme (and the most statistically precise) case is leave-one-out CV, when k equals to the size of the data set. In practice, training of a model can be a very computationally demanding task, so small k (≤ 10) is often used.

The train data can be further subdivided to extract a separate validation set, which is to be used to fit hyperparameters. As mentioned before, hyperparameters are the parameters that should be defined a priori, before fitting model's parameters. They can also control model's *complexity*, and examples can be given by the highest power in polynomial regression or a number of layers and neurons in an NN (or even parameters of the training algorithm, such as the learning rate, itself!). In the same spirit, we can compare models with different hyperparameters on this validation set and select the best one to finally estimate its performance on the test set. *Regularization* is another way to control model's complexity. The main idea behind it is to introduce a complexity-dependent penalty term in the loss function (or use a special prior in the Bayesian setting), for example, requiring that the most of the inferred parameters $\boldsymbol{\beta}$ should be close to zero.

6 CONCLUDING REMARKS

At time of writing, artificial intelligence and machine learning, are two hot-boiling areas where a lot of exciting news are coming almost every day. The main intention of this chapter was to provide you with a quick overview of this dynamic field and introduce some of its basic concepts. We have described a small subset of models and approaches, mainly from supervised learning. To deepen your knowledge in this direction we recommend to read also about kernel regression, Gaussian processes, decision trees, and ensemble methods (random forests, gradient boosting). You might also want to become familiar with unsupervised algorithms such as clustering (hierarchical clustering, DBSCAN), dimensionality reduction (PCA, autoencoders), and generative models (variational autoencoders, generative adversarial networks) and a few basic probabilistic models such as naïve Bayes, Gaussian mixtures, and latent Dirichlet allocation.

For a keen reader, this chapter will rise far more questions than provide answers. Given the intense dynamics of the field, the best advice is that you use your favorite search engine and look for the concepts we talk about here. In any case, we encourage you to read some classic textbooks covering both machine learning and statistics. The book by Bishop (2006) might be a good starting point for understanding machine learning from a statistical point of view. If you would like to focus more on deep learning, Goodfellow et al. (2016) might be another good read. If you are interested in the PGMs, a pretty smooth introduction is the MBML e-book (http://www.mbmlbook.com), but for a more technical one, we recommend Koller and Friedman (2009). We also encourage you to check countless online resources, such as courses, video lectures, tutorials, blog posts, talks and presentations. Stay tuned!

REFERENCES

Bishop, C., 2006. Pattern Recognition and Machine Learning. Springer-Verlag, New York.

Boden, M., 2006. Mind As Machine. Oxford University Press, Oxford.

Boser, B.E., Guyon, I.M., Vapnik, V.N., 1992. In: A training algorithm for optimal margin classifiers. Proceedings of the fifth annual workshop on Computational learning theory—COLT'92, p. 144.

Bostrom, N., 2014. Superintelligence: Paths, dangers, strategies. Oxford University Press, Oxford.

Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., Bengio, Y., 2014. Learning Phrase Representations Using RNN Encoder-Decoder for Statistical Machine Translation. arXiv preprint arXiv: 1406.1078.

Cover, T., Hart, P., 1967. Nearest neighbor pattern classification. IEEE Trans. Inf. Theory. 13 (1).

Dempster, A.P., Laird, N.M., Rubin, D.B., 1977. Maximum likelihood from incomplete data via the EM algorithm. J. R. Stat Soc Series B (Methodological) 1–38.

Doersch, C., 2016. Tutorial on Variational Autoencoders. arXiv preprint arXiv:1606.05908.

Forgy, E.W., 1965. Cluster analysis of multivariate data: efficiency versus interpretability of classifications. Biometrics 21, 768–769.

Goodfellow, I., Bengio, Y., Courville, A., 2016. Deep Learning. MIT Press, Cambridge, Massachusetts, USA.

Hastie, T., Tibshirani, R., Friedman, J., 2009. The Elements of Statistical Learning: Data Mining, Inference, and Prediction. Springer, New York, NY.

Hochreiter, S., Schmidhuber, J., 1997. Long short-term memory. Neural Comput 9 (8), 1735–1780.

Hornik, K., Stinchcombe, M., White, H., 1989. Multilayer feedforward networks are universal approximators. Neural Netw 2 (5).

Koller, D., Friedman, N., 2009. Probabilistic Graphical Models: Principles and Techniques. MIT Press, Cambridge, Massachusetts, USA.

Kurzweil, R., 2005. The Singularity Is Near. Viking Books, New York.

Lecun, Y., Bottou, L., Bengio, Y., Haffner, P., 1998. Gradient-based learning applied to document recognition. Proc. IEEE 86 (11), 2278–2324.

Lloyd, S. P. (1957). *Least square quantization in PCM*. Bell Telephone Laboratories Paper. Published in journal: Lloyd., S. P. (1982).

Minsky, M., Papert, S., 1969. Perceptrons: An Introduction to Computational Geometry. MIT Press, Cambridge, Massachusetts, USA.

Moor, J. (Ed.), 2003. The Turing Test: The Elusive Standard of Artificial Intelligence. Springer Science & Business Media, Dordrecht, Netherlands.

Pearl, J., 1982. Reverend Bayes on inference engines: A distributed hierarchical approach. Cognitive Systems Laboratory, School of Engineering and Applied Science, University of California, Los Angeles, pp. 133–136.

Pearl, J., 1985. In: Bayesian networks: a model of self-activated memory for evidential reasoning. Proceedings of the 7th Conference of the Cognitive Science Society, pp. 329–334.

Rosenblatt, F., 1957. The Perceptron—a perceiving and recognizing automaton, Cornell Aeronautical Laboratory Report 85-460-1.

Rumelhart, D., Hinton, G., Williams, R., 1986. Learning representations by back-propagating errors. Lett Nat 323.

Russell, S.J., Norvig, P., 2009. Artificial Intelligence: A Modern Approach, third ed. Prentice Hall, Upper Saddle River, NJ.

Samuel, A.L., 1959. Some studies in machine learning using the game of checkers. IBM J Res Dev. 3 (3).

Spirtes, P., Glymour, C., 1991. An algorithm for fast recovery of sparse causal graphs. Soc Sci Comput Rev 9 (1), 62–72.

Wentworth, J. A. (1993). Expert systems in transportation. AAAI Technical Report WS-93-04.

FURTHER READING

Ribeiro, M.T., Singh, S., Guestrin, C., 2016. In: Why should I trust you? Explaining the predictions of any classifier.Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Miningpp. 1135–1144.