Machine Learning: Foundations, Methodologies, and Applications

Yaochu Jin · Hangyu Zhu · Jinjin Xu · Yang Chen

Federated Learning

Fundamentals and Advances

Springer

Machine Learning: Foundations, Methodologies, and Applications

Series Editors

Kay Chen Tan, Department of Computing, Hong Kong Polytechnic University, Hong Kong, China

Dacheng Tao, University of Technology, Sydney, Australia

Yaochu Jin · Hangyu Zhu · Jinjin Xu · Yang Chen

Federated Learning

Fundamentals and Advances

Springer

Yaochu Jin
Faculty of Technology
Bielefeld University
Bielefeld, Germany

Hangyu Zhu
Department of Artificial Intelligence
and Computer Science
Jiangnan University
Wuxi, China

Jinjin Xu
Intelligent Perception and Interaction
Research Department
OPPO Research Institute
Shanghai, China

Yang Chen
School of Electrical Engineering
China University of Mining and Technology
Xuzhou, China

ISSN 2730-9908 ISSN 2730-9916 (electronic)
Machine Learning: Foundations, Methodologies, and Applications
ISBN 978-981-19-7082-5 ISBN 978-981-19-7083-2 (eBook)
https://doi.org/10.1007/978-981-19-7083-2

Preface

I heard the terminology "federated learning" for the first time when I was listening to a talk given by Dr. Catherine Huang from Intel at a workshop of the IEEE Symposium Series on Computational Intelligence in December 2016 in Athens, Greece. I was fascinated by the idea of federated learning and immediately recognized the paramount importance of preserving data privacy, since deep learning had been increasingly relying on the collection of a large amount of data, either from industrial processes or from human everyday life.

Federated learning has now become a popular research area in machine learning and received increased interest from both industry and government. Actually already in April 2016, the European Union adopted the General Data Protection Regulation, which is the most strict law on data protection and privacy in the European Union and the European Economic Area and took effective in May 2018. Consequently, data privacy and security has become indispensable for practical applications of machine learning and artificial intelligence, together with other importance requirements, Including explainability, fairness, accountability, robustness and reliability.

This book presents a compact yet comprehensive and updated coverage of fundamentals and recent advances in federated learning. The book is self-contained and suited for both postgraduate students, researchers, and industrial practitioners. Chapter 1 starts with an introduction to the most popular neural network models and gradient-based learning algorithms, evolutionary algorithms, evolutionary optimization, and multi-objective evolutionary learning. Then, three main privacy-preserving computing techniques, including secure multi-party computation, differential privacy, and homomorphic encryption are described. Finally, the basics of federated learning, including a category of federated learning algorithms, the vanilla federated averaging algorithm, federated transfer learning, and main challenges in federated learning over non independent and identically distributed data are presented. Chapter 2 focuses on communication efficient federated learning, which aims to reduce the communication costs in federated learning without deteriorating the learning performance. Two algorithms for reducing communication overhead in federated learning are detailed. The first algorithm takes advantage of the fact that shallow layers in a deep neural network are responsible for learning general features across different classes while deep layers take care of class-specific features. Consequently, the deep layers can be updated less frequently than the deep layers, thereby reducing the number of parameters that need to be uploaded and downloaded. The second algorithm adopts a completely different approach, which dramatically decreases the communication load by converting the weights in real numbers into ternary values. Here, the key question is how to maintain the learning performance after ternary compression of the weights, which is achieved by training a real-valued co-efficient for each layer. Interestingly, we are able to prove theoretically that model divergence can even be mitigated by introducing ternary quantization of the weights, in particular when the data is not independent and identically distributed. While layer-wise synchronous weight update and ternary compression described in Chap. 2 can effectively lower the communication cost, Chap. 3 addresses communication cost by simultaneously maximizing the learning performance and minimizing the model complexity (e.g., the number of parameters in the model) using a multi-objective evolutionary algorithm. To go a step further, a real-time multi-objective evolutionary federated learning algorithm is given, which searches for optimal neural architectures by maximizing the performance, minimizing the model complexity, and minimizing the computational performance (indicated by floating point operations per second) at the same time. To make it possible for real-time neural architecture search, a strategy that searches for subnetworks by sampling a supernet, and reducing computational costs by sampling clients is introduced. To enhance the security level of federated learning, Chap. 4 elaborates two secure federated learning algorithms by integrating homomorphic encryption and differential privacy techniques with federated learning. The first algorithm is based on a horizontal federated learning, in which a distributed encryption algorithm is applied to the weights to be uploaded on top of ternary quantization. By contrast, the second algorithm is meant for vertical federated learning based on decision trees, in which secure node split and construction are developed based on homomorphic encryption and predicted labels are aggregated on the basis of partial differential privacy. This algorithm does not assume that all labels are stored on one client, making it more practical for real-world applications. The book is concluded by Chap. 5, providing a summary of the presented algorithms, and an outlook of future research.

The two algorithms presented in Chap. 2 were developed by two visiting Ph.D. students I hosted at University of Surrey, Jinjin Xu and Yang Cheng. The two multi-objective evolutionary federated learning algorithms were proposed by my previous Ph.D. student, Hangyu Zhu. Finally, the two secure federated learning algorithms were designed by Hangyu Zhu, in collaboration with Dr. Kaitai Liang, and his Ph.D. student, RuiWang, who were with the Department of Computer Science, University of Surrey, and are now with the Department of Intelligent Systems, Delft University of Technology, The Netherlands. I would like to take this opportunity thank to Kaitai and Rui for their contributions to Chap. 4.

Finally, I would like to acknowledge that my research is funded by an Alexander von Humboldt Professorship for Artificial Intelligence endowed by the German Federal Ministry of Education and Research.

Bielefeld, Germany
August 2022

Yaochu Jin

Contents

Chapter 1
Introduction

Abstract This chapter introduces the background knowledge of the book, including the most widely used artificial neural network models and decision trees, gradient based learning methods, evolutionary algorithms and their applications to single- and multi-objective machine learning, traditional privacy-preserving computing methods such as multi-party secure computation, differential privacy, and homomorphic encryption, and the federated learning paradigm for privacy-preserving machine learning. An overview of horizontal and vertical federated learning, together with a description of the basic federated learning algorithm, known as federated averaging, is presented, before knowledge transfer in federated learning is briefly explained. Finally, the main challenges of federated learning over non independent and identically distributed data are discussed in detail.

1.1 Artificial Neural Networks and Deep Learning

1.1.1 A Brief History of Artificial Intelligence

The terminology of artificial intelligence was formally suggested in a proposal of a summer workshop held in Dartmouth 1955, although the idea to create a programmable machine can be traced back to the 19th century [1]. The first mathematical model proposed by McCulloch and Pitts in 1943 [2], known as the McCulloch-Pitts model, simulates the function of a single neuron. In 1949, the Hebb Law was proposed by Hebb [3], which states that neurons that fire together wire together, meaning that a connection between two neurons will become stronger if they activate simultaneously, and the connection will weaken if they activate at different times. Many unsupervised learning algorithms were based the Hebb Law in principle. The earliest functional neuron model is the perceptron proposed by Rosenblatt in 1958 [4], which can separate linearly separable patterns.

Y. Jin et al., *Federated Learning*, Machine Learning: Foundations, Methodologies, and Applications, https://doi.org/10.1007/978-981-19-7083-2_1

During 1945 and 1947, Alan Turing, a pioneer of both computer science and artificial intelligence, worked on the design of the Automatic Computing Engine, which is widely recognized as the first stored-program computer. In 1950, Turing proposed his most well known test to define if a machine is intelligent, called Turing test [5]. The basic idea behind the Turing test is that a machine can be seen as intelligent if a human interrogator is not able to distinguish a machine from a human being through conversation. The term machine learning was first proposed by Arthur Samuel in 1959, which was defined as a research field that enables computers to learn without being explicitly programmed. A large body of research on artificial intelligence was carried out after the Dartmouth workshop, including the development of the first natural language processing computer program ELIZA [6] and the first industrial robot Unimate [7]. It should be mentioned that two important research areas of artificial intelligence, namely fuzzy sets and fuzzy systems [8] that simulate human reasoning, and evolutionary computation [9] that simulates natural evolution, were also developed during the 1960s.

The first 'winter' of artificial intelligence started in the beginning of 1970s after Minsky and Papert published a book analyzing the limitations of perceptrons [10]. Nevertheless, several prominent advances were made during the 1970 and 1980s, including the development of many successful expert systems, proposal of an early version of the error back-propagation learning algorithm for a neural network model containing one hidden layer that can solve the XOR problem [11], and genetic algorithms [12] as well as population based evolution strategies [13].

Several breakthroughs were achieved in the 1980s. In 1982, Kohonen proposed the self-organizing maps [14], which is one most important unsupervised learning algorithms, while reinforcement learning was suggested in 1983 [15]. The Hopfield neural network, which is a recurrent neural network model, remarked the start of second boom of the artificial intelligence research, which was culminated by the publication of the seminal paper [16] in 1986, in which the error back-propagating algorithm was proposed for training multiple layer perceptrons, enabling artificial neural networks containing one or two hidden layers to solve many linearly non-separable classification problems and nonlinear regression problems.

Many other learning algorithms and neural network models have also been reported during the 1980s. In 1983, the Bienenstock-Cooper-Munro (BCM) neural plasticity rule was published [17], which can be seen as an extension of the Hebbian rule and has become a powerful unsupervised learning algorithm. After more than one decade, another important unsupervised learning rule, called spike-timing dependent plasticity was developed in 1998 [18], which has become one popular unsupervised learning algorithm for spiking neural networks proposed in 1997 by Wolfgang Maas [19]. Very different from other artificial neural networks that are based on continuous analog signals, spiking neural networks use the timing and frequency of the spikes or pulses for information processing, which is biologically more plausible since all biological nervous systems, including the human brain, also rely on spikes. In the meantime, Neurocognitron [20], which was based on the architecture of the human visual system was suggested by Fukushima in 1983, and which can be seen as the predecessor of the convolutional neural network [21] proposed in 1998, which is currently the most popular neural network model for solving computer vision problems. Several other important pieces of research on artificial intelligence have also been proposed in the 1990s, e.g., the support vector machine [22], and the long short-term memory, a more powerful recurrent neural network model [23]. Interestingly, support vector machines and other statistically learning algorithms outperformed those based on artificial neural networks, resulting in the second 'winter' of the artificial intelligence research.

There were also lots of interesting new developments outside the field of neural networks and machine learning during the 1990s. Specifically, new theories and methodologies were developed in fuzzy systems, including the introduction of type-2 fuzzy sets and a wide range of successful applications of fuzzy control and fuzzy decision-making. Besides, evolutionary computation has also become a popular research field, in which not only new paradigms such as genetic programming was proposed, but several other population based meta-heuristics were also proposed, including particle swarm optimization, ant colony optimization, and differential evolution [24]. Evolutionary algorithms and other metaheuristics have been shown to be effective in solving complex optimization problems, in particular multi-objective optimization problems, dynamic optimization problems, and data-driven optimization problems.

Like in the first winter of artificial intelligence, many researchers continued working in the field of neuronal networks, fuzzy systems and evolutionary computation, which was called computational intelligence, instead of artificial intelligence, mainly because artificial intelligence became unwelcome in the research community. Many popular emerging research topics in artificial intelligence nowadays have already been studied in the 1990s and early 2000s, before the third wave of artificial intelligence. These include interpretability of trained neural networks and fuzzy systems, robust machine learning, multi-objective machine learning and structure optimization of neural networks, now popularly known as neural architecture search.

The third resurgence of artificial intelligence started in 2007 when Hinton published a paper on effective training of artificial neural networks consisting of many large hidden layers [25], now widely known as deep neural networks. The training of deep neural networks was considerably accelerated by using graphic processing units (GPUs), making it possible to effectively train deep neural networks containing tens or even hundreds of hidden layers on huge datasets such as ImageNet by taking advantage of the immense computational resources available nowadays. A revolutionary success was achieved by deep learning when a computer go player algorithm called AlphaGo on the basis of deep reinforcement learning and Monte Carlo tree search defeated a human world champion.

To date, deep learning has achieved tremendous successes that were unimaginable years ago. Deep learning has demonstrated unique power in solving almost all single problems in science and technology, ranging from face recognition, game playing, to healthcare, natural language processing, protein folding and drug discovery. Many very powerful deep neural network models have been proposed, such as autoencoder, variational autoencoder, generative adversarial networks, and transformer, just to name a few. In addition, new research paradigms such as transfer learning, few shots learning, and self-supervised learning have been developed to handle various problems, in particular deep learning in the presence of data paucity.

Despite the above successes deep learning has achieved, many technical, ethical, and social issues remain or have arisen. Technically, it is still a challenging issue to enable machine learning models to learn multiple tasks on small data, let alone to learn autonomously. Trustworthiness and fairness of artificial intelligence become increasingly concerning as it is more and more widely used in critical, human, and societal systems. Trustworthiness typically include explanability and or transparency, safety, reliability and robustness, privacy preservation, respect of human values, green, and accountability.

1.1.2 Multi-layer Perceptrons

A *multi-layer perceptron* (MLP) [26–28] often represent a fully connected feed-forward artificial neural network (ANN). And it consists of three types of layers: the input layer, output layer and hidden layers, each of which contains several neurons named perceptrons [29]. Before discussing about MLP neural networks, simpler perceptrons will be explained at first.

1.1.2.1 Perceptrons

The general structure of a perceptron is shown in Fig. 1.1, where $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ is a n-length input feature vector, b is the bias and $\mathbf{w} = (w_1, w_2, \ldots, w_n)$ is the corresponding weight vector of the input features.

For feed-forward propagation, a perceptron receives a weighted sum of the input features together with the bias as the input z computed by the following Eq. (1.1):

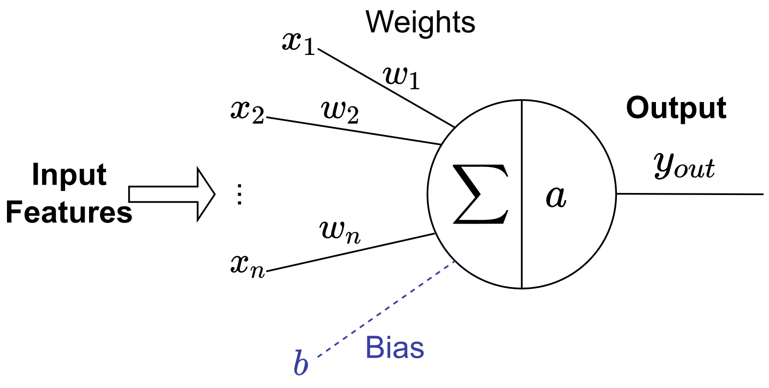


Fig. 1.1 An example of perceptron

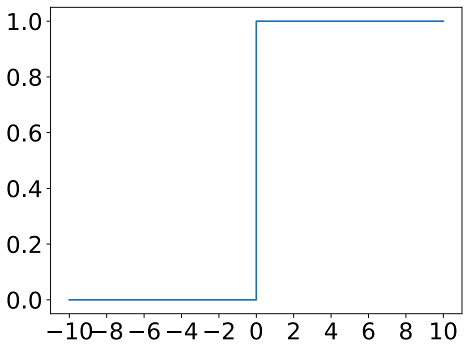


Fig. 1.2 An example of the step function when the threshold t is 0

$$z = \sum_{i=1}^{n} x_i w_i = \mathbf{x}^T \mathbf{w} \tag{1.1}$$

The computed sum z is then passed through an activation function a for nonlinear transformation, refer to Fig. 1.1. And the step function is always selected to be the activation function of the perceptrons whose outputs are calculated according to Eq. (1.2):

$$y_{out} = a(z) = \begin{cases} 1, & z > t. \\ 0, & z < t. \end{cases} \tag{1.2}$$

where t is the threshold. A simple example of the step function when $t = 0$ is shown in Fig. 1.2. If the weighted sum z is larger than the threshold t, the perceptron in Fig. 1.1 will output $y_{out} = 1$; otherwise, the output y_{out} will become 0. Note that the step function can be replaced by other activation functions or even be removed for specific learning tasks.

1.1.2.2 Activation Function

For a typical supervised learning problem, we redefine the outputs of the perceptron for any input vector $\mathbf{x}$ as model prediction $\hat{y}_{out}$ and the corresponding data label as y_{out}. In this case, training perceptrons can be converted into minimizing the constructed loss function representing the distance between the prediction $\hat{y}_{out}$ and the actual label y_{out}, which is conducted by optimizing the weights and bias (often called model parameters) of the perceptrons.

The gradient based optimization methods are commonly used in training perceptrons. The core idea of this kind of approach is to let the model parameters recursively subtract their corresponding product of the gradients and the learning rate until the loss function converges. And calculating the gradients of the model parameters with the chain rule [30] requires the constructed loss function to be continuous and differentiable. Therefore, the above mentioned step function is usually approximated by the sigmoid function, making it differentiable for the gradient based training.

It is clear to see that the curve of sigmoid function shown in Fig. 1.3(a) has a similar shape as that of step function shown in Fig. 1.2. These two functions also have the same output range, and the sigmoid function can be seen as the smoothed and continuous version of the step function. The sigmoid function is described in Eq. (1.3):

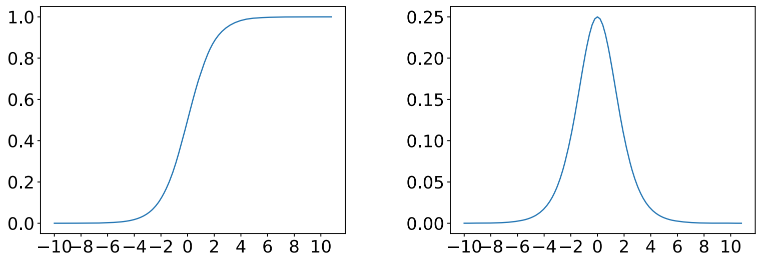
$$\hat{y}_{out} = \sigma(z) = \frac{1}{1 + e^{-z}} \tag{1.3}$$

And the derivative of the sigmoid function with respect to its input z (weighted sum of the input data features) can be easily calculated in Eq. (1.4):

$$\begin{aligned} \frac{\partial \sigma(z)}{\partial z} &= -\frac{1}{(1 + e^{-z})^2} \cdot (-e^{-z}) \\ &= \sigma(z)(1 - \sigma(z)) \end{aligned} \tag{1.4}$$

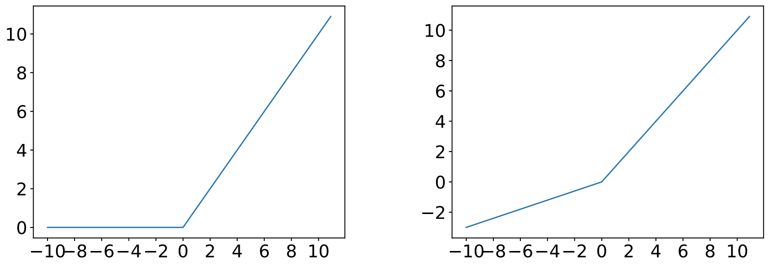
From Fig. 1.3(b), the largest gradient of the sigmoid function is located at the central part of the curve and it tends to approach 0 when the absolute value of the input z approaches to infinity. That is the reason why normalization techniques [31–33] are often used in modern machine learning tasks.

Furthermore, the sigmoid function may cause gradient vanishing especially for MLPs with a large number of hidden layers. This is because the gradients of MLP neural networks are calculated using the error back propagation (to be discussed later in detail) by computing the derivative from the last output layer to the input layer. And according to the chain rule, the gradients of shallower layers (closer to the input layer) are derived by multiplying the calculated derivatives of deeper layers (closer to the output layer). Consequently, when calculating, for example, the gradients of the shallower layer for a n-hidden-layer MLP using the sigmoid activation function, n small derivatives (the maximum value is 0.25 shown in Fig. 1.3(b) are multiplied together, making the resulting gradients extremely small. In this case, the gradients



(a) Sigmoid function (b) Derivative of the sigmoid function

Fig. 1.3 An example of the sigmoid function and its corresponding derivative



(a) Relu function (b) Leaky relu function

Fig. 1.4 ReLU and leaky ReLU functions

are too small to have negligible influence on the parameter updates during the training period.

To overcome this issue, rectified linear unit (ReLU) or leaky ReLU is instead selected to be the activation function as described in Eq. (1.5):

$$\hat{y}_{out} = a(z) = \begin{cases} z, & z \geq 0. \\ az, & z < 0. \end{cases} \tag{1.5}$$

where a is real-valued hyperparameter with a range of [0, 1), and above equation becomes ReLU function if $a = 0$; otherwise it called is leaky ReLU. And the corresponding plot is also shown in Fig. 1.4.

It is easy to find that the derivatives of both ReLU and leaky ReLU are 1 if the input $z > 0$, thus, effectively avoiding the gradient vanishing issue of multiplying derivatives layer by layer compared to the sigmoid function. In addition, the ReLU function is computationally efficient, where only comparison, addition and multiplication operations are involved during model training. Therefore, ReLU has become the most popular activation function for modern deep neural networks [34].

1.1.2.3 Model Structure

A perceptron without any connection (the node represented by a big circle in Fig. 1.1) is acted as a basic building element called a neuron in MLP neural networks. Roughly speaking, MLPs are a brunch of neurons connected together with a multi-layer structure as shown in Fig. 1.5, where circles in solid lines are neurons containing the activation function and circles in dashed lines represent biases with weight connections equal to 1.

When a training data passes through the MLP model from the input layer to the output layer, exactly the same forward computation is performed as a perceptron does for each neuron of each layer. And the mathematical formulation for the output of each layer is described as follows:

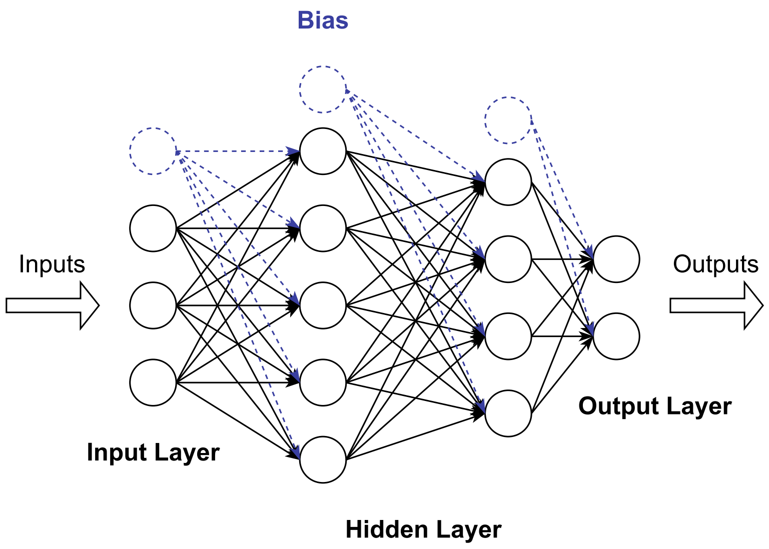


Fig. 1.5 An illustrative example of MLP

$$y^l = a(y^{l-1} W_{l-1,l} + \mathbf{b}_l) \tag{1.6}$$

where y^l is the output of the l-th layer, $W_{l-1,l}$ are the weights between layer $l-1$ and layer l, and $\mathbf{b}_l$ is the bias vector of the lth layer. Note that y^0 is in fact the input feature vector when $l = 1$. Both weights and biases are required to be initialized with random numbers (normally small real numbers from –1 to 1). Common initialization methods include Gaussian initialization, Xavier initialization, and Kaiming initialization [35], among others.

1.1.2.4 Input Layer

The input layer is actually the (preprocessed) input data where each neuron represents one data attribute. And table-format data like credit card [36] and bank marketing [37] are intrinsically well suited to MLP neural networks. Furthermore, other types of data can also be processed and converted to fit the structure of MLPs. For instance, image data can be easily transformed into the input vector whose elements are just pixel values, and time-series data can be partitioned along the sequence direction to several input vectors. However, the MLP neural network, for instance, is not able to extract spatial and time sequence information for image data and time-series data, respectively.

1.1.2.5 Hidden Layer

Layers after the input layer except the last layer are hidden layers. The width of MLPs means the number of neurons of hidden layers and the depth represents the number of hidden layers. And deep neural networks often refer to those networks having many hidden layers. By contrast, wide neural networks represent a network with a large number of neurons per layer.

1.1.2.6 Output Layer

The last layer of MLPs is called the output layer that outputs a scalar or a vector corresponding to the requirement of the learning task. And both the activation function and the number of neurons for the output layer is constrained by the modeling type.

For a regression problem, the output layer may contain only one neuron with no activation function (which can be seen as a linear activation function). Similarly, for a simple binary classification problem, the output layer may also have one neuron using the sigmoid function to output a value between 0 and 1, representing the probability of predicting class 1. In addition, a multi-class classification problem may have multiple neurons with softmax activation function in the output layer and each neuron outputs the probability of predicting one class value.

1.1.2.7 Loss Function

For a typical supervised learning task, the loss function $\ell(\hat{y}_{out}, y_{out})$ representing the distance between the desired label y_{out} and the real prediction $\hat{y}_{out}$ should be constructed before parameter optimization. Two most commonly used loss function for both classification and regression problems will be introduced below.

The first widely used loss function is the cross entropy loss [38] for multi-class classification problems as shown in Eq. (1.7):

$$\ell(\hat{y}_{out}, y_{out}) = -\sum_{c=1}^{M} y_{out,c} \log(\hat{y}_{out,c}) \tag{1.7}$$

where c is the class index and M is the total number of classes. It should be emphasized that the data label y_{out} requires to be converted into a M-length binary one-hot code [39] where only the corresponding position of label class is filled with 1. For instance, for a 4-class classification problem, the label 2 is converted into a binary code 0100 as shown in Fig. 1.6. Furthermore, the prediction value $\hat{y}_{out,c}$ of each class c follows a discrete probability distribution satisfying $\sum_{c=1}^{M} \hat{y}_{out,c} = 1$.

Intuitively, the cross entropy loss aims to make two vectors $\hat{y}_{out}$ and y_{out} become similar. Since y_{out} is one-hot encoded, it is easy to find that the formula

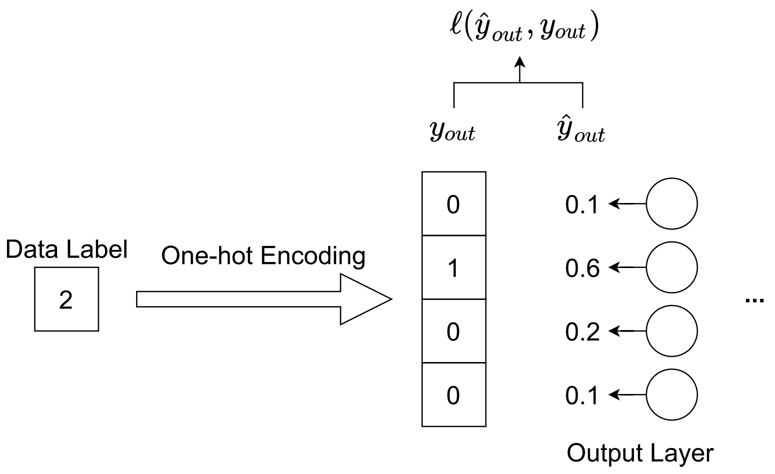


Fig. 1.6 An example of cross entropy loss

$y_{out,c}\log(\hat{y}_{out,c})$ is 0 if c is not equal to the label class. Therefore, the original cross entropy loss can be simplified into Eq. (1.8):

$$\begin{aligned}\ell(\hat{y}_{out}, y_{out}) &= -y_{out,c'}\log(\hat{y}_{out,c'}) \\ &= -\log(\hat{y}_{out,c'})\end{aligned} \tag{1.8}$$

where $c^{'}$ represents the label class number. Considering that log function is monotonically increasing (monotonically decreasing for negative log function) and $\hat{y}_{out,c'}$ ranges from 0 to 1, minimizing $\ell(\hat{y}_{out}, y_{out})$ is in fact letting $\hat{y}_{out,c'}$ approach to the true label element $y_{out,c'} = 1$.

Besides, the cross entropy loss function can be modified into a binary cross entropy loss to fit the scope of binary classification problems as shown in Eq. (1.9). In this case, the output layer contains one neuron only and y_{out} is a binary number with a value of either 0 or 1.

$$\ell(\hat{y}_{out}, y_{out}) = -y_{out}\log(\hat{y}_{out}) - (1 - y_{out})\log(1 - \hat{y}_{out}) \tag{1.9}$$

The other commonly used loss function for regression problems is the squared error shown in Eq. (1.10):

$$\ell(\hat{y}_{out}, y_{out}) = \frac{1}{2}(y_{out} - \hat{y}_{out})^2 \tag{1.10}$$

where both $\hat{y}_{out}$ and y_{out} are real-valued scalars. Since Eq. (1.10) is a quadratic function achieving the minimum loss value if $\hat{y}_{out} = y_{out}$, thus, minimizing the squared error function will make the prediction $\hat{y}_{out}$ close to the actual label y_{out}.

The above introduced loss functions are constructed by a single data entry. In most scenarios, however, multiple training data are required to be simultaneously imported into the MLP neural network. And the averaged loss function is often adopted to deal with this situation. For convenience, θ is used here to represent both weights and biases, and the averaged loss function for N data samples can be reformulated as Eq. (1.11):

$$L(\theta, \mathbf{X}) = \frac{1}{N} \sum_i \ell(\theta, \mathbf{x}_i) \quad \mathbf{x}_i \in \{\mathbf{x}_1, \mathbf{x}_2 \ldots, \mathbf{x}_N\} \tag{1.11}$$

where $\mathbf{x}_i$ is the ith training data vector and N is the data size. The objective of model training is to find suitable model parameters θ to minimize the expected loss of N data entries.

1.1.2.8 Gradient-Based Optimization Methods

The gradient based parameter optimization method [40] is the most popular MLP training algorithm used during back-propagation due to its efficiency and fast convergence.

Given that the total training data size is N, the batch size is B and the entire data are evenly divided into $\frac{N}{B}$ mini-batches, a typical mini-batch gradient descent (GD) algorithm in each iteration is performed in Eq. (1.12):

$$\begin{aligned} g_t &= \frac{1}{B} \nabla_\theta \, L(\theta, \mathbf{x}_{t:t+B}) \\ \theta_{t+1} &= \theta_t - \eta g_t \end{aligned} \tag{1.12}$$

where g_t is the expected gradients of a B-size mini-batch data at the t-th iteration and η is the learning rate controlling the training footsteps. In each iteration of the MLP training, the averaged model gradients of a randomly selected (without replacement) min-batch data are computed layer by layer. After that, the model parameters θ_{t+1} for the next iteration can be updated by subtracting the product ηg_t.

Note that if the batch size $B = 1$, Eq. (1.12) becomes the standard stochastic gradient descent (SGD) in which the model parameters θ are immediately upgraded once the gradients for a random data sample are computed. However, SGD may cause instability in training. By contrast, the standard GD is another extreme compared to SGD, where the batch size B is equal to the entire data size N, and thus the average gradients for all N training are calculated and subtracted at each iteration. The weakness of the GD is that the computational consumption in each iteration is intensive when the training data is huge and it is more likely to be trapped into the local minimum.

Consequently, to strike a good balance between SGD and GD, mini-batch SGD is usually selected as the training algorithm by setting $1 < B < N$. And nowadays, the usage of these terms are not very strict, and 'GD', 'SGD' and 'mini-batch SGD' all