Chapter 6
Deep Learning for Regression and Classification

Abstract Deep learning (DL) is a subclass of machine learning methods that works in conjunction with large datasets along with many neuron layers (artificial neural networks (ANN)) (Schmidhuber, Neural Networks, 61, 85–117, 2015). An ANN is a computer system inspired by the biological neural networks in the human brain (Chen et al., Sensors, 19, 2047, 2019). The term "deep" refers to the use of multiple layers of neurons (three or more) in the ANN. Deep learning models can automatically generate features from data, and they can be designed for many structures. In this chapter, two standard structures called feed forward neural network (FFNN) and convolutional neural network (CNN) will be introduced and demonstrative examples will be presented. An application of CNN is also given. The CNN is used to read raw chest X-ray images of patent and automatically classify diseases, such as pneumonia or COVID-19. In addition, a musical instrument sound converter for changing piano musical notes to guitar musical notes will be developed using mechanistic data science. This example will demonstrate the advantage of the mechanistic data science approach compared to the standard neural networks. That is, a smaller number of training data is required to achieve a good performance.

Keywords Artificial neural networks · Feed forward neural network · Convolutional neural network · Kernel · Convolution · Padding · Stride · Pooling · Instrumental music conversion · COVID-19

6.1 Introduction

Learning is a process for acquiring knowledge and skills, so they are readily available for understanding and solving future problems and opportunities [3]. Deep learning leverages artificial neural networks to automatically find patterns in data, with the objective of predicting some target output or response. Deep

Supplementary Information: The online version of this chapter (https://doi.org/10.1007/978-3-030-87832-0_6) contains supplementary material, which is available to authorized users.

W. K. Liu et al., *Mechanistic Data Science for STEM Education and Applications*,
https://doi.org/10.1007/978-3-030-87832-0_6

Table 6.1 A comparison of supervised and unsupervised learning, in terms of methods, data, goal, and uses

	Supervised learning	Unsupervised learning
Methods	Linear regression, nonlinear regression, etc.	K-means clustering, Principal component analysis (PCA), etc.
Data	Input and output variables will be given	Only input data will be given, and the data are not labelled
Goal	To determine the relationship between inputs and outputs so that we can predict the output when a new dataset is given	To capture the hidden patterns or underlying structure in the given input data
Uses	Regression, classification, etc.	Clustering, dimension reduction, etc.

learning methods are heavily based on statistics and mathematical optimization. They can be supervised or unsupervised (or semi-supervised that is beyond the scope of this book). *Unsupervised learning* looks for previously undetected patterns in a data set with no *pre-existing labels* and minimal human supervision. It studies how systems can infer a function to describe a hidden structure from unlabeled data. In contrast, *supervised learning* maps an input to an output based on representative input-output pairs. It infers a function from labeled training data consisting of a set of training examples. Supervised machine learning algorithms can apply what has been learned in the past to new data using labeled examples to predict future events. Supervised learning plays an important role in estimating the relationships between independent variables (i.e., inputs) and dependent variables (i.e., outputs). A comparison of supervised and unsupervised learning is shown in Table 6.1.

Supervised learning attempts to learn the relationship between output variable (s) and input variable(s). If the outputs are continuous, supervised learning becomes a *regression* problem. An example of a regression problem is predicting the price of a diamond based on its properties such as carat, cut, color, and clarity (i.e., the 4Cs) as shown in Fig. 6.1. Supervised learning attempts to solve the problem of learning input-output mappings from empirical data or the training set. For example, a dataset D of n datapoints: $D = \{(\boldsymbol{x}_i, y_i) | \, i = 1, 2, \ldots, n\}$, where $\boldsymbol{x}$ is the input vector (e.g., the 4Cs in the previous example) and y is the output (e.g., the price of a diamond).

If the outputs are discrete (e.g., yes or no) instead of continuous, supervised learning turns into a *classification* problem. One example (Fig. 6.2) is recognizing if a patient has been infected with coronavirus 2019 (COVID-19) based on their chest X-ray images. COVID-19 spread rapidly around the world and became a pandemic since it first appeared in December 2019, which caused a disastrous impact on public health, daily lives, and global economy. It is very important to accurately detect the positive cases at early stage to treat patients and prevent the further spread of the pandemic. Chest X-ray imaging has critical roles in early diagnosis and treatment of COVID-19. Automated toolkits for COVID-19 diagnosis based on radiology

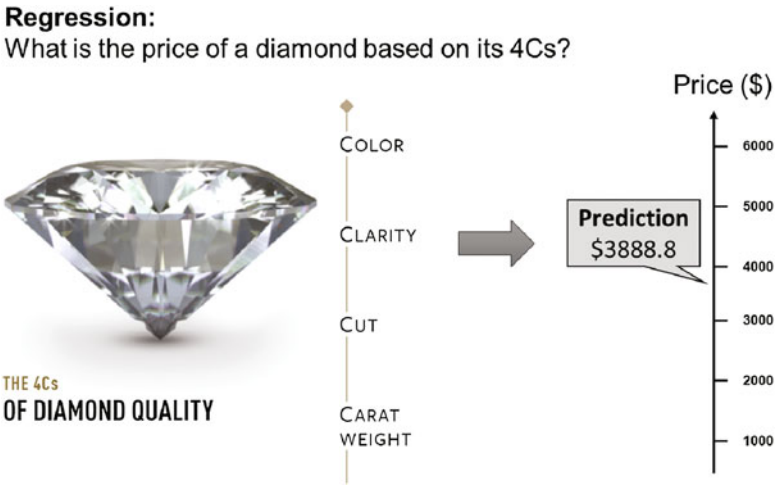

Fig. 6.1 Regression example: predicting the price of a diamond based on its properties such as carat, cut, color, and clarity (i.e., 4Cs)

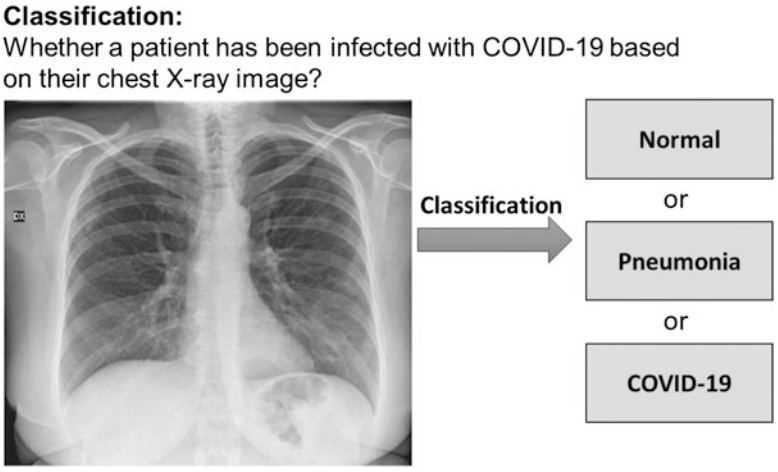

Fig. 6.2 Classification example: detect COVID-19 based on chest X-ray images [4]

imaging techniques such as X-ray imaging can overcome the issue of a lack of physician in remote villages and other underdeveloped regions. Application of artificial intelligence (AI) techniques, such as deep learning, coupled with radiological X-ray imaging, can be very helpful for the accurate and automatic detection of this disease. The classification can be binary (COVID-19 vs. Normal) and multi-class (COVID-19 vs. Normal vs. Pneumonia).

6.1.1 Artificial Neural Networks

Artificial neural networks (often called neural networks) learn (or are trained) by a set of data, which contain known inputs and outputs. The training of a neural network from a given dataset is usually conducted by determining the difference between the output of the neural networks (often a prediction) and a target output (an error). The neural network then updates its parameters (such as weights and biases) according to a learning rule based on this error value. Successive adjustments will cause the neural network to produce output which is increasingly similar to the target output. After a sufficient number of these adjustments, the training can be terminated based on certain criteria. Such neural networks "learn" to perform tasks by considering data without being programmed with task-specific rules. For example, in the previous COVID-19 image recognition, neural networks might learn to identify X-ray images that indicate COVID-19 by analyzing many images (i.e., data points) that have been manually labeled as "COVID-19" or "Normal". Neural networks do this without any prior knowledge of the pathology known by a doctor.

6.1.2 A Brief History of Deep Learning and Neural Networks

In 1976, Alexey Ivakhnenko and Lapa [5] published the first feedforward multilayer neural networks for supervised learning. The term "Deep Learning" was first introduced by Rina Dechter in 1986 [6]. As a milestone, Yann LeCun et al. [7] applied the standard backpropagation algorithm to a deep convolutional neural network (CNN) for recognizing handwritten ZIP codes on mail in 1989. In 1995, Brendan Frey and co-developer Peter Dayan and Geoffrey Hinton demonstrated that it was feasible to train a network containing six fully-connected hidden layers with several hundred neurons using a wake-sleep algorithm [8]. In 1997, a recurrent neural network (RNN) was published by Hochreiter and Schmidhuber and called long short-term memory (LSTM) [9], which avoided the longstanding vanishing gradient problem in deep learning.

In the early 2000s, the deep learning began to significantly impact industry. Industrial applications of deep learning to large-scale speech recognition started around 2010. Advances in computational hardware have driven more interest in deep learning. In 2009, Andrew Ng demonstrated that graphics processing units (GPUs) could accelerate the learning process of deep learning by more than 100 times [10].

In 2019, Yoshua Bengio, Geoffrey Hinton, and Yann LeCun was named as recipients of the 2018 Association for Computing Machinery (ACM) Turing Award (the "Nobel Prize of Computing") for conceptual and engineering breakthroughs that have made deep neural networks a critical component of computing, and the story is continuing...

6.2 Feed Forward Neural Network (FFNN)

6.2.1 A First Look at FFNN

Neural networks are based on simple building block called artificial neurons (or just neurons). A neuron (Fig. 6.3) is a computational node that takes an input value, x, adjusts that value according to a specific weight w, bias b, and activation function $\mathcal{A}(\cdot)$, to produce a new output value $y = \mathcal{A}(wx + b)$. *Weights* are values which need to be multiplied with each input value, essentially reflecting the importance of an input. *Biases* are constant values that are added to the product of inputs and weight, usually utilized to offset the result.

Activation functions $\mathcal{A}(\cdot)$ engage the neurons based on the provided input. This can typically include a nonlinear mapping of input and helps to increase the degree of freedom of ANN. Some examples of commonly used activation functions can be seen in Fig. 6.4, of which, the ReLu is the most popular due to its simplicity.

The ReLU activation function has some limitations. For example, the Dying ReLU problem [11] includes some ReLU functions that essentially "die" for all inputs and remain inactive no matter what input is supplied. This can be corrected by using Leaky ReLU or Parametric ReLU. For these ReLU functions, the slope to left of $x = 0$ is changed, causing a leak and extending the range of ReLU (see Fig. 6.5).

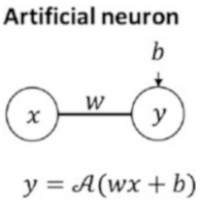

Fig. 6.3 Schematic of an artificial neuron and its functionality

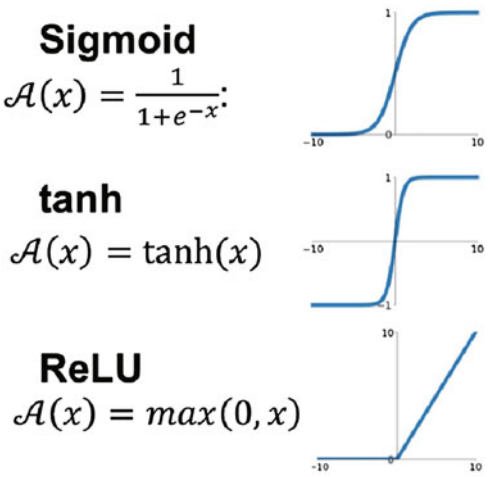

Fig. 6.4 Commonly used activation functions

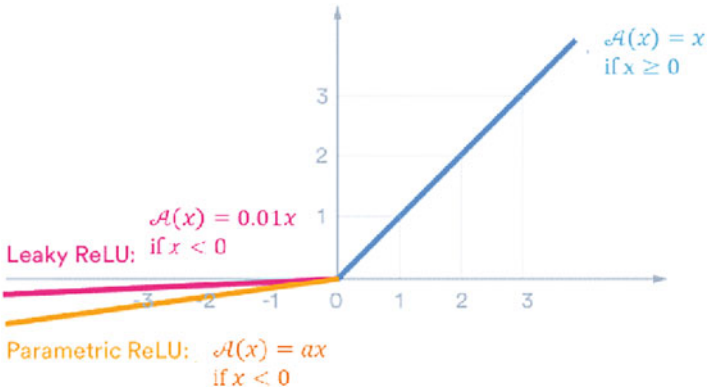

Fig. 6.5 Leaky ReLU and Parametric ReLU activation functions

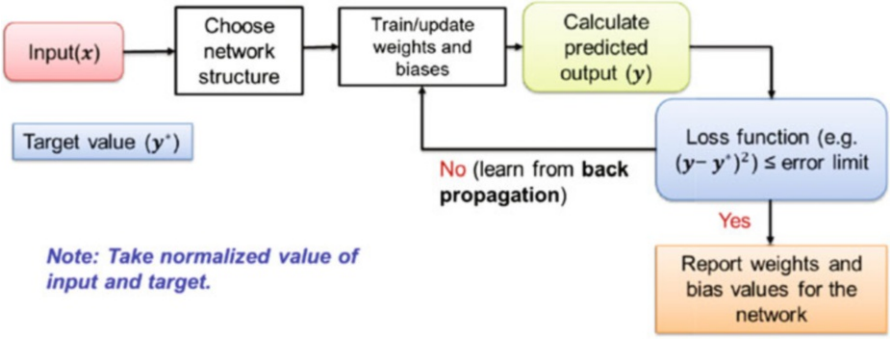

Fig. 6.6 A flowchart for training a neural network (NN)

A standard procedure for training a neural network (NN) is shown in Fig. 6.6. The goal of a neural network is to match a target value y^* given an input value of x for the network. The input and output can be either scalar or vector. Once the network structure is determined (based on prior information or just trial and error), the weights and biases of the neural network are trained/updated to make sure the predicted output, y, is close to the target value, y^*. If they are close enough, the weights and biases are reported as converged values. The closeness is quantified by a loss function, e.g., $(\boldsymbol{y} - \boldsymbol{y}^*)^2$. If the loss function is larger than a predetermined error limit, a learning algorithm, called back propagation, adjusts the weights and biases of the neural network to reduce the loss function. The back propagation algorithm is modified from the gradient decent method described in Chap. 3. Once the neural network is trained, additional new datapoints should be used to test the network, as described in Chap. 2.

A neural network can include hidden neurons or hidden layers of neurons. Each layer of neurons between the input and output neurons is called hidden. Figure 6.7 shows an example of a network structure including one hidden neuron. The output of

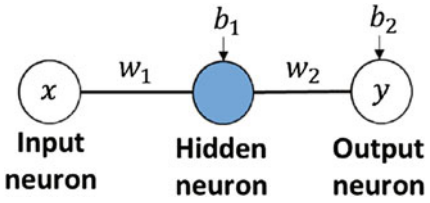

Fig. 6.7 A network structure including only one hidden neuron

the network can be computed as follows: (1) the first weight w_1 is multiplied by the input value x and the first bias b_1 is added, which is then inserted into the activation function $\mathcal{A}(\cdot)$ and the output of the hidden neuron is determined as $\mathcal{A}(wx_1 + b_1)$; (2) the output of the hidden neuron is treated as the input of the next neuron, i.e., output neuron in this case. Repeating the same operation mentioned above, the output of this network can be determined as $\mathcal{A}(w_2\mathcal{A}(w_1x + b_1) + b_2)$.

Assume the goal of a network is to fit one datapoint, such as $(x^* = 0.1, y^* = 20)$, by adjusting the unknown parameters. A good fit means if the input value $x = 0.1$, the output of this network is $y = 20$. The detailed training process will be illustrated step by step. For simplifying the problem, the biases are assumed to be zero, and a linear activation function is used. That means the output of a neuron is equal to the product of weight and input. The two unknown parameters in this case are w_1 and w_2. To train the parameters based on the datapoint, a back propagation algorithm is described below:

Step (1): initialize the weights by arbitrary values. For example,

$$w_1 = 10 \tag{6.1}$$

$$w_2 = 2 \tag{6.2}$$

Step (2): compute NN output and error:

$$y = w_2w_1x = 10 * 5 * 0.1 = 5 \tag{6.3}$$

$$y^* - y = 20 - 5 = 15 \tag{6.4}$$

Step (3): compute increments of weights based on the gradient decent (GD) explained in Chap. 3 (a learning rate of $\alpha = 0.25$ is used in this case):

$$\Delta w_1 = -\alpha\frac{\partial L}{\partial w_1} = \alpha(y^* - y)w_2x = 0.25 * 15 * 5 * 0.1 = 1.875 \tag{6.5}$$

$$\Delta w_2 = -\alpha\frac{\partial L}{\partial w_2} = \alpha(y^* - y)w_1x = 0.25 * 15 * 10 * 0.1 = 3.75 \tag{6.6}$$

Step (4): update the weights:

$$w_1 = w_1 + \Delta w_1 = 10 + 1.875 = 11.875 \tag{6.7}$$

$$w_2 = w_2 + \Delta w_2 = 5 + 3.75 = 8.75 \tag{6.8}$$

Table 6.2 Results of weights, NN output and error at each iteration

i	1	2	3	4	5
w_1	10	11.88	13.98	15.07	15.24
w_2	5	8.75	11.60	12.92	13.12
y	5	10.39	16.22	19.48	20.00
$y^* - y$	15	9.61	3.78	0.52	0.00

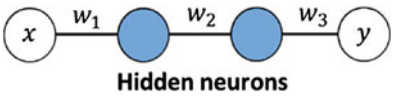

Fig. 6.8 A network structure including two hidden neurons

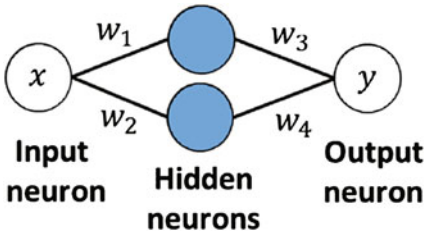

Fig. 6.9 A network structure including one hidden layer but two hidden neurons

Through step (2)–(4), the two weights of the NN have been updated based on the gradient decent of the error between NN output and target value. The step (2)–(4) can be repeated until the weights are unchanged, or the error is smaller than a criterion. Table 6.2 presents the computed weights w_1 and w_2, NN output y, and error $y^* - y$ at each iteration. As seen in Table 6.2, after five iterations the NN output reaches 19.996, which is very close to the target of 20. The result can be more accurate if more iterations are used.

Additional hidden layers can be added between the input and output neurons. For example, consider a network structure with two hidden layers, each layer with only one hidden neuron, as shown in Fig. 6.8. Now there are three unknown parameters, and the NN output can be computed based on the same rule (if the assumptions still hold): $y = w_3 w_2 w_1 x$. The same back propagation procedure can be used to train the unknown weights.

In addition, each hidden layer can have more than one neuron. For example, a NN structure with one hidden layer but two hidden neurons are shown in Fig. 6.9. In general, the input to the output neuron is equal to the sum of the two hidden neurons. Based on the operation rule, the output of this NN structure is:

$$y = \mathcal{A}[w_3 \mathcal{A}(w_1 x + b_1) + w_4 \mathcal{A}(w_2 x + b_2) + b_3] \tag{6.9}$$

Using the same assumptions that the biases are zero and activation functions are linear, the output can be rewritten as

$$y = w_1 w_3 x + w_2 w_4 x \tag{6.10}$$

To fit the same datapoint ($x^* = 0.1, y^* = 20$) by adjusting the four unknown weights, the back propagation can be used again:

Step (1): initialize the weights by arbitrary values:

$$w_1 = 10 \tag{6.11}$$

$$w_2 = 10 \tag{6.12}$$

$$w_3 = 5 \tag{6.13}$$

$$w_4 = 5 \tag{6.14}$$

Step (2): compute NN output and error. It is noted that the formula of the NN output depends on the NN structure that is different from the previous example.

$$y = w_1 w_3 x + w_2 w_4 x = 5 + 5 = 10 \tag{6.15}$$

$$y^* - y = 20 - 10 = 10 \tag{6.16}$$

Step (3): compute increments of weights based on the gradient decent (GD) (α is the learning rate that is 0.25 in this case):

$$\Delta w_1 = -\alpha \frac{\partial L}{\partial w_1} = \alpha (y^* - y) w_3 x = 0.25 * 10 * 5 * 0.1 = 1.25 \tag{6.17}$$

$$\Delta w_2 = -\alpha \frac{\partial L}{\partial w_2} = \alpha (y^* - y) w_4 x = 0.25 * 10 * 5 * 0.1 = 1.25 \tag{6.18}$$

$$\Delta w_3 = -\alpha \frac{\partial L}{\partial w_3} = \alpha (y^* - y) w_1 x = 0.25 * 10 * 10 * 0.1 = 2.5 \tag{6.19}$$

$$\Delta w_4 = -\alpha \frac{\partial L}{\partial w_4} = \alpha (y^* - y) w_2 x = 0.25 * 10 * 10 * 0.1 = 2.5 \tag{6.20}$$

Step (4): update the weights:

$$w_1 = w_1 + \Delta w_1 = 10 + 1.25 = 11.25 \tag{6.21}$$

$$w_2 = w_2 + \Delta w_2 = 10 + 1.25 = 11.25 \tag{6.22}$$

$$w_3 = w_3 + \Delta w_3 = 5 + 2.5 = 7.5 \tag{6.23}$$

$$w_4 = w_4 + \Delta w_4 = 5 + 2.5 = 7.5 \tag{6.24}$$

Repeat Step (2)–(4) until the weights are unchanged or the error is less than a criterion.

Table 6.3 presents the computed weights, NN output, and error at each iteration. As seen in the Table, this NN structure needs only four iterations (instead of five used in the previous example) to achieve a good approximation. In practice, adding more neurons or layers increases the complexity of functions that the NN can represent, but also increases the computational cost and the risk of overfitting [12]. Thus, choosing appropriate NN structure for given problem is still a challenging task.

It is interesting to note that the input and output can be generalized to vectors so the NN can handle multiple inputs and outputs. To demonstrate this, a datapoint including a 2D vector as input and a 2D vector as output $\left(\boldsymbol{x}^* = \begin{bmatrix} 1 \\ 2 \end{bmatrix}, \boldsymbol{y}^* = \begin{bmatrix} 10 \\ 20 \end{bmatrix}\right)$ will be used to train the NN with the same structure as the previous example. The same procedure can be followed but the weights of the NN are 2D vectors in the case:

Step (1): initialize the weights by arbitrary vectors:

$$\boldsymbol{w_1} = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \tag{6.25}$$

Table 6.3 Results of weights, NN output and error at each iteration

i	1	2	3	4
w_1	10	11.25	11.84	11.87
w_2	10	11.25	11.84	11.87
w_3	5	7.5	8.38	8.428
w_4	5	7.5	8.38	8.428
y	10	16.88	19.83	20.009
$y^* - y$	10	3.13	0.17	−0.009

$$\boldsymbol{w_2} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \tag{6.26}$$

$$\boldsymbol{w_3} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \tag{6.27}$$

$$\boldsymbol{w_4} = \begin{bmatrix} 2 \\ 2 \end{bmatrix} \tag{6.28}$$

Step (2): compute NN output and error:

$$\boldsymbol{y} = \boldsymbol{w_3}\left(\boldsymbol{w_1^T x^*}\right) + \boldsymbol{w_4}\left(\boldsymbol{w_2^T x^*}\right) = \begin{bmatrix} 0 \\ 1 \end{bmatrix} * 3 + \begin{bmatrix} 2 \\ 2 \end{bmatrix} * 1 = \begin{bmatrix} 2 \\ 5 \end{bmatrix} \tag{6.29}$$

$$\boldsymbol{y^* - y} = \begin{bmatrix} 10 \\ 20 \end{bmatrix} - \begin{bmatrix} 2 \\ 5 \end{bmatrix} = \begin{bmatrix} 8 \\ 15 \end{bmatrix} \tag{6.30}$$

Step (3): compute increments of weights based on the gradient decent (GD), and α is the learning rate that is 0.01 in this case. The learning rate can be predetermined by trial and error. It is noted that a too small learning rate causes a large number of iterative steps required but a too large learning rate leads to an oscillated learning process that cannot converge.

$$\boldsymbol{\Delta w_1} = -\alpha \frac{\partial L}{\partial \boldsymbol{w_1}} = \alpha(\boldsymbol{y^* - y})\boldsymbol{w_3^T x^*} = 0.01 * \begin{bmatrix} 8 \\ 15 \end{bmatrix} * 2 = \begin{bmatrix} 0.16 \\ 0.3 \end{bmatrix} \tag{6.31}$$

$$\boldsymbol{\Delta w_2} = -\alpha \frac{\partial L}{\partial \boldsymbol{w_2}} = \alpha(\boldsymbol{y^* - y})\boldsymbol{w_4^T x^*} = 0.01 * \begin{bmatrix} 8 \\ 15 \end{bmatrix} * 6 = \begin{bmatrix} 0.48 \\ 0.9 \end{bmatrix} \tag{6.32}$$

$$\boldsymbol{\Delta w_3} = -\alpha \frac{\partial L}{\partial \boldsymbol{w_3}} = \alpha(\boldsymbol{y^* - y})\boldsymbol{w_1^T x^*} = 0.01 * \begin{bmatrix} 8 \\ 15 \end{bmatrix} * 3 = \begin{bmatrix} 0.24 \\ 0.45 \end{bmatrix} \tag{6.33}$$

$$\Delta \boldsymbol{w_4} = -\alpha \frac{\partial L}{\partial \boldsymbol{w_4}} = \alpha(\boldsymbol{y}^* - \boldsymbol{y})\boldsymbol{w}_2^T \boldsymbol{x}^* = 0.01 * \begin{bmatrix} 8 \\ 15 \end{bmatrix} * 1 = \begin{bmatrix} 0.08 \\ 0.15 \end{bmatrix} \tag{6.34}$$

Step (4): update the weights:

$$\boldsymbol{w_1} = \boldsymbol{w_1} + \Delta \boldsymbol{w_1} = \begin{bmatrix} 1 \\ 1 \end{bmatrix} + \begin{bmatrix} 0.16 \\ 0.3 \end{bmatrix} = \begin{bmatrix} 1.16 \\ 1.3 \end{bmatrix} \tag{6.35}$$

$$\boldsymbol{w_2} = \boldsymbol{w_2} + \Delta \boldsymbol{w_2} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} + \begin{bmatrix} 0.48 \\ 0.9 \end{bmatrix} = \begin{bmatrix} 1.48 \\ 0.9 \end{bmatrix} \tag{6.36}$$

$$\boldsymbol{w_3} = \boldsymbol{w_3} + \Delta \boldsymbol{w_3} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} + \begin{bmatrix} 0.24 \\ 0.45 \end{bmatrix} = \begin{bmatrix} 0.24 \\ 1.45 \end{bmatrix} \tag{6.37}$$

$$\boldsymbol{w_4} = \boldsymbol{w_4} + \Delta \boldsymbol{w_4} = \begin{bmatrix} 2 \\ 2 \end{bmatrix} + \begin{bmatrix} 0.08 \\ 0.15 \end{bmatrix} = \begin{bmatrix} 2.08 \\ 2.15 \end{bmatrix} \tag{6.38}$$

Repeat Step (2)–(4) until the weights are unchanged or the error is less than a criterion. The loss function $L = \frac{1}{2}(\boldsymbol{y}^* - \boldsymbol{y})^2$ is also computed for this case. The loss function is half of the squared distance of the error $\boldsymbol{y}^* - \boldsymbol{y}$. The convergence is reached if the loss function is less than a specific criterion (0.0001 in this case). Table 6.4 presents the computed weights, NN output, and loss at each iteration.

Table 6.4 Results of weights, NN output and loss at each iteration

i	1	3	5	15
$\boldsymbol{w_1}$	$\begin{bmatrix} 1 \\ 1 \end{bmatrix}$	$\begin{bmatrix} 1.23 \\ 1.54 \end{bmatrix}$	$\begin{bmatrix} 1.16 \\ 1.63 \end{bmatrix}$	$\begin{bmatrix} 1.1 \\ 1.66 \end{bmatrix}$
$\boldsymbol{w_2}$	$\begin{bmatrix} 1 \\ 0 \end{bmatrix}$	$\begin{bmatrix} 1.62 \\ 1.38 \end{bmatrix}$	$\begin{bmatrix} 1.49 \\ 1.56 \end{bmatrix}$	$\begin{bmatrix} 1.388 \\ 1.608 \end{bmatrix}$
$\boldsymbol{w_3}$	$\begin{bmatrix} 0 \\ 1 \end{bmatrix}$	$\begin{bmatrix} 0.33 \\ 1.73 \end{bmatrix}$	$\begin{bmatrix} 0.24 \\ 1.84 \end{bmatrix}$	$\begin{bmatrix} 0.178 \\ 1.875 \end{bmatrix}$
$\boldsymbol{w_4}$	$\begin{bmatrix} 2 \\ 2 \end{bmatrix}$	$\begin{bmatrix} 2.15 \\ 2.40 \end{bmatrix}$	$\begin{bmatrix} 2.07 \\ 2.51 \end{bmatrix}$	$\begin{bmatrix} 2.002 \\ 2.542 \end{bmatrix}$
$\boldsymbol{y}$	$\begin{bmatrix} 2 \\ 5 \end{bmatrix}$	$\begin{bmatrix} 10.84 \\ 17.95 \end{bmatrix}$	$\begin{bmatrix} 10.61 \\ 19.70 \end{bmatrix}$	$\begin{bmatrix} 10.003 \\ 19.998 \end{bmatrix}$
L	144.5	2.46	0.23	$6.5e^{-6}$

Typically, the vector data is more difficult to fit as compared with scale data. In this case fifteen iterations are required to reach a convergence.

The neural network can also handle a larger database including many data points. In the next section, the NN will be used to predict a diamond price. A python code will be introduced to implement the network.

6.2.2 *General Notations for FFNN [Advanced Topic]*

In the previous section, a few simple neural networks were trained by updating their weights and biases using the gradient descent algorithm (often called backpropagation algorithm in machine learning terminology). In this section, a general matrix-based form is explained to compute the output from a neural network with multiple layers and neurons.

To introduce the notation in an unambiguous way, Fig. 6.10 shows a three layers FFNN, and the notations which refer to weights, biases, and activation (i.e., output) in the network are also marked in the figure. The notations use $w^l_{j,k}$ to denote the weight for the connection from the k^{th} neuron in the $(l-1)^{th}$ layer to the j^{th} neuron in the l^{th} layer. For example, Fig. 6.10 shows a weight $w^3_{2,4}$ on a connection from the fourth neuron in the second layer to the second neuron in the third layer of the network. This notation appears cumbersome at first, but it will be explained below to demonstrate that this notation is easy and natural.

The similar notation can be used for the network's biases and activations, i.e., b^l_j denotes the bias of the j^{th} neuron in the l^{th} layer, and a^l_j denotes the activation (or output) of the j^{th} neuron in the l^{th} layer.

Based on these notations, the activation a^l_j of the j^{th} neuron in the l^{th} layer is related to the activations in the $(l-1)^{th}$ layer by the equation

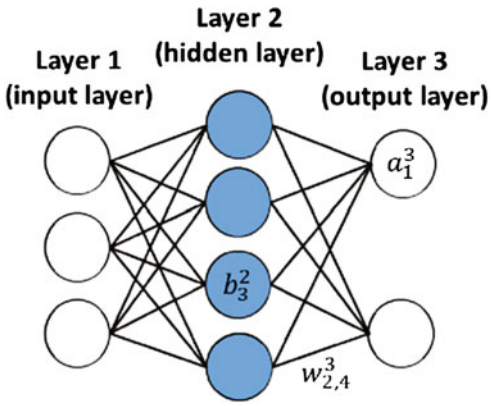

Fig. 6.10 Notations for a multilayer neural network. The blue circles indicate neurons in the hidden layer

$$a^l_j = \mathcal{A}\left(\sum_k w^l{}_{j,k} a^{l-1}_k + b^l_j\right) \tag{6.39}$$

where the sum is over all neurons k in the $(l-1)^{th}$ layer. To use a matrix form to represent this expression, a weight matrix $\boldsymbol{W^l}$ can be defined for each layer, l. The entry in the j^{th} row and k^{th} column of the weight matrix $\boldsymbol{W^l}$ is just $w^l{}_{j,k}$. Similarly, a bias vector $\boldsymbol{b^l}$ is defined, and the component of the bias vector are just the b^l_j. An activation vector $\boldsymbol{a^l}$ can be also defined, and the component are the activations a^l_j. With these matrix-formed notations the above equation can be rewritten in a compact vectorized form

$$\boldsymbol{a^l} = \mathcal{A}\left(\boldsymbol{W^l a^{l-1}} + \boldsymbol{b^l}\right) \tag{6.40}$$

Thus, in terms of mapping the data between input $\boldsymbol{x}$ and output $\boldsymbol{y}$ over M layers, the following recursive equation is derived

$$\boldsymbol{y} = \mathcal{A}_M\left(\boldsymbol{W^M}, \ldots, \mathcal{A}_3\left(\boldsymbol{W^3}\mathcal{A}_2\left(\boldsymbol{W^2}\mathcal{A}_1\left(\boldsymbol{W^1 x} + \boldsymbol{b^1}\right) + \boldsymbol{b^2}\right) + \boldsymbol{b^3}\right)\ldots\right) \tag{6.41}$$

where $\mathcal{A}_M$ is the activation function for the M^{th} layer. Based on this general form, the neural networks specifically optimize weights $\boldsymbol{W^l}$and biases $\boldsymbol{b^l}$ in a NN with M layers over a loss function as

$$\textbf{argmin}\ \boldsymbol{\mathcal{L}}\left(\boldsymbol{W^1}, \boldsymbol{W^2}, \ldots \boldsymbol{W^M}, \boldsymbol{b^1}, \boldsymbol{b^2}, \ldots, \boldsymbol{b^M}\right) \tag{6.42}$$

where the $\boldsymbol{\mathcal{L}}$ is the loss function, and for example, it can be the standard root-mean square error between NN outputs and target values

$$\underset{\boldsymbol{W^l}, \boldsymbol{b^l}}{\textbf{argmin}} \sum_{k=1}^{n} \left(\boldsymbol{y}_k - \boldsymbol{y}^*_k\right)^{\boldsymbol{2}} \tag{6.43}$$

where $\boldsymbol{y_k}$ is the NN output for the k^{th} datapoint (in total n datapoints), and $\boldsymbol{y^*_k}$ is the target value (ground truth) for the k^{th} datapoint. The $\boldsymbol{y_k}$ can be substituted by the matrix-formed NN output

$$\underset{\boldsymbol{W^l}, \boldsymbol{b^l}}{\textbf{argmin}} \sum_{k=1}^{n} \left(\mathcal{A}_M\left(\boldsymbol{W^M}, \ldots, \mathcal{A}_3\left(\boldsymbol{W^3}\mathcal{A}_2\left(\boldsymbol{W^2}\mathcal{A}_1\left(\boldsymbol{W^1 x_k} + \boldsymbol{b^1}\right) + \boldsymbol{b^2}\right) + \boldsymbol{b^3}\right)\ldots\right) - \boldsymbol{y^*_k}\right)^{\boldsymbol{2}} \tag{6.44}$$

To solve this optimization problem using gradient decent or back propagation, a lot of standard programming libraries such as Python [13], PyTorch [14], and MATLAB [15] can be used. An example will be introduced in the next section.

6.2.3 *Apply FFNN to Diamond Price Regression*

In Chap. 1, the feature-based diamond pricing problem was briefly introduced. The universal method for assessing diamond quality, regardless of location in the world, relies on the 4Cs (color, clarity, cut, carat weight), whose features and scales can be seen in Fig. 6.11. The goal of this example is to predict the price of a new diamond based on its 4Cs and other features. To achieve that goal, a feed forward neural network (FNN) will be trained based on a large database including a lot of diamonds and their information.

The data for this application was found on Kaggle, which is the world's largest data science community with powerful tools and resources to help users achieve their data science goals. The diamond dataset contains the price and features of nearly 54,000 diamonds and a portion can be seen in Fig. 6.12. The ten features and their ranges can also be seen below in Fig. 6.13.

Fig. 6.11 The 4Cs (Color, Clarity, Cut, and Carat weight) of diamond quality (https://4cs.gia.edu/en-us/)

carat	cut	color	clarity	depth	table	price	x	y	z
0.23	Ideal	E	SI2	61.5	55.0	326	3.95	3.98	2.43
0.21	Premium	E	SI1	59.8	61.0	326	3.89	3.84	2.31
0.23	Good	E	VS1	56.9	65.0	327	4.05	4.07	2.31

Fig. 6.12 Open-source diamond dataset from Kaggle (https://www.kaggle.com/)

Price: ($326--$18,823)
Carat: (0.2--5.01)
Cut: (Fair, Good, Very Good, Premium, Ideal)
Color: (J (worst) to D (best))
Clarity: (I1 (worst), SI2, SI1, VS2, VS1, VVS2, VVS1, IF (best))
Size in x direction in mm (0--10.74)
Size in y direction in mm (0--58.9)
Size in z direction in mm (0--31.8)
Depth: z / mean(x, y) = 2 * z / (x + y) (43--79) (%)
Table: width of top of diamond relative to widest point (43--95) (%)

Fig. 6.13 Diamond dataset features explained

To use this data effectively, it is important to understand how this raw data can be built into a useful model and understand machine learning datasets as well.

Given a dataset with input $\boldsymbol{X}_i^N$ and corresponding output label $\boldsymbol{y}_j^N$, where i indicates the input feature index, j indicates the output feature index, and N indicates the number of data points. The goal of the machine learning is to form a certain function with multiple parameters so that it can capture the relationship between input features and corresponding output values.

For the diamond dataset, $i = 1...9$, representing the number of independent variables, including carat, color, cut, and carat. Similarly, $j = 1$, representing the number of dependent variables: price. The dataset contains 53,940 diamonds, so $N = 53{,}940$. Machine learning aims to find the functional form $\boldsymbol{y}_j^N = f\left(\boldsymbol{X}_i^N\right)$, where $f\left(\boldsymbol{X}_i^N\right)$ correctly maps input $\boldsymbol{X}_i^N$ to output $\boldsymbol{y}_j^N$. The dataset is divided into training (70%), validation (15%), and testing (15%) sets to find the functional relationship and confirm it is the best possible fit. This process has been explained in Chap. 2.

First, inputs and outputs from the training set are fit to mapping function $f\left(\boldsymbol{X}_i^N\right)$, developing a FFNN model. The validation set tests the model after each training step. This process is iterative, meaning that the function is updated after each validation test to reduce error between the predicted and actual outputs. When error is minimal, the final functional form is established, and the training accuracy meets the required threshold, the function's performance is evaluated with the testing set. In this example, properties of diamonds with known prices are used as test model inputs. Predicted and actual prices are compared to determine model accuracy.

For this problem, there are nine different input features in $\boldsymbol{X}_i^N$ (carat, cut, color, etc.) for each of the $N = 53{,}940$ diamonds. Price is the only output feature, $\boldsymbol{y}_j^N$. The neural network attempts to build a relationship between the input features and the output feature. The overall neural network architecture can be seen in Fig. 6.14. Two hidden layers are used, with each layer including twelve hidden neurons. This NN structure is complex enough to capture the relationship between diamond features and its price.

The loss function used in the neural network for training is a mean squared error (MSE):

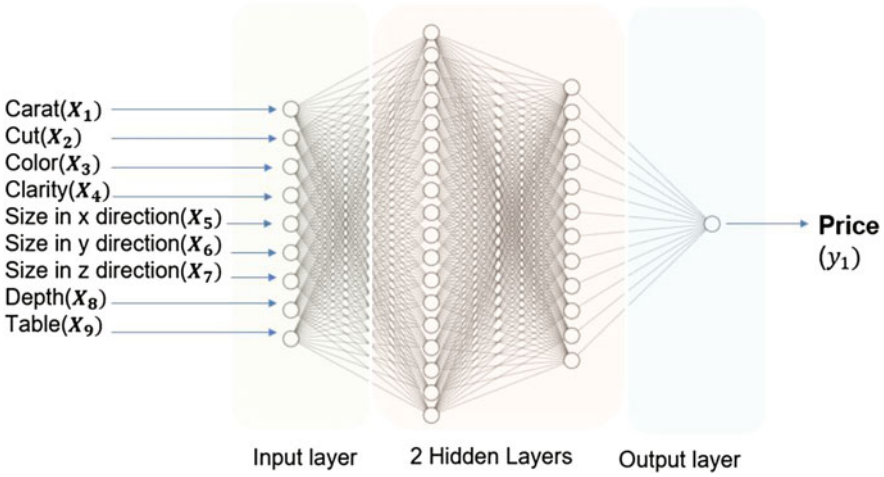

Fig. 6.14 Neural network architecture used in the diamond example

$$L = \frac{1}{N}\sum_{i=1}^{N}(y^* - y)^2 \tag{6.45}$$

An Adam optimizer [16] is used to implement back propagation algorithm. A Python code with annotations to implement this example is shown below:

```
Python code for diamond price classification:

# import necessary python library
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import RMSprop
from keras.wrappers.scikti_learn import KerasRegressor
from sklearn.model_selection import cross_val_score
from sklearn.model.selection import KFold

# data preparation and scaling
X_df = df.drop(["price"],axis=1)
y_df = df.price
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
scaler.fit(X_df)
print(X_df.shape)
display(X_df.head())
X_df = scaler.transform(X_df)
from sklearn.model_selection import train_test_split
```

(continued)

```
X_train, X_test, y_train, y_test = train_test_split(X_df,
y_df, random_state=2, test_size=0.3)

# define a neural network
Def Net():
    model=Sequential()
    model.add(Dense(input_dim=9,activation="relu",units=18))
    model.add(Dense(kernel_initializer="normal",
    activation="relu", units=12))
    model.add(Dense(kernel_initializer="normal", units=1))
    model.comile(loss="mean_squared_error", optimizer="adam")
    return model

# train the network and predict
estimator = KerasRegressor(build_fn=Net, epochs=10,
batch_size=5)
estimator.fit(X_train, y_train)
y_pred = estimator.predict(X_test)
print("Mean absolute error: {:.2f}".format(mean_absolute_error(
  y_test, y_pred)))
```

The price of the diamonds as predicted by the neural network is graphed along with the actual price of the diamonds to evaluate the accuracy of the neural network. This graph can be seen in Fig. 6.15. This model has a coefficient of determination of $R^2 = 0.93$, which shows a high level of accuracy in predicting the price of diamonds. This correlation, known as the "goodness of fit," is represented as a value between 0.0 and 1.0. A value of 1.0 indicates a perfect fit and might be thus a good model for future forecasts, while a value of 0.0 would indicate a poor fit.

Once the NN model is trained, it can be used to predict the price of a diamond if its features are known. For example, Fig. 6.16 shows four diamonds. Their features such as 4Cs are known but the prices are unknown. The trained NN model can

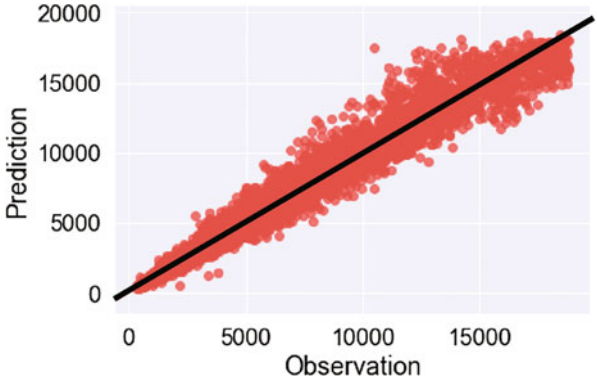

Fig. 6.15 Actual vs. predicted Price of diamonds based on a neural network

	carat	cut	color	clarity	depth	table	x	y	z
0	4.0	1	0	0	65.5	59	10.74	10.54	6.98
1	2.0	1	0	0	65.5	59	10.74	10.54	6.98
2	0.5	2	2	1	61.4	56	4.33	4.37	2.67
3	0.3	0	1	2	59.7	58	4.13	4.14	2.47

```
Prediction: [12148.8  9818.9  1573.8   503.2]
```

Fig. 6.16 Predictions of the prices for four diamonds

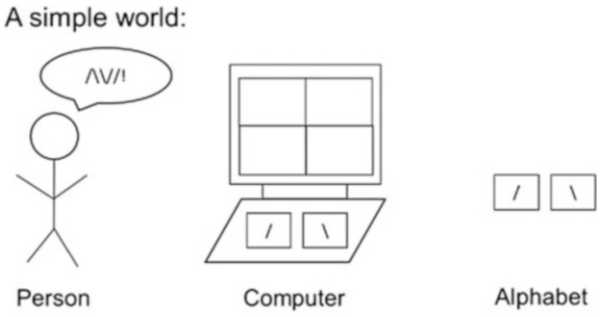

Fig. 6.17 A simple world including one person, one computer with two-by-two pixels resolution, and a two letters alphabet

predict their prices by inputting their features, which would be very helpful for evaluation the value of a diamond for a diamond vender or a person who want to buy a diamond.

6.3 Convolutional Neural Network (CNN)

6.3.1 A First Look at CNN

This section will explain how the computers recognize images using convolutional neural network (CNN). To demonstrate the basic idea of the CNN, imagine a world with only one person, and that person has one computer. The computer has a screen of resolution two-by-two pixels and it has a keyboard with two only keys. Only two keys are needed because this simple world has an alphabet with only two letters: a backward slash and a forward slash as shown in Fig. 6.17.

In this simple world, writing is very simple, and the alphabet song is also very simple—it goes like "backward slash and forward slash". What happens when the keyboard is used in this world? As shown in Fig. 6.18, if the computer key on the

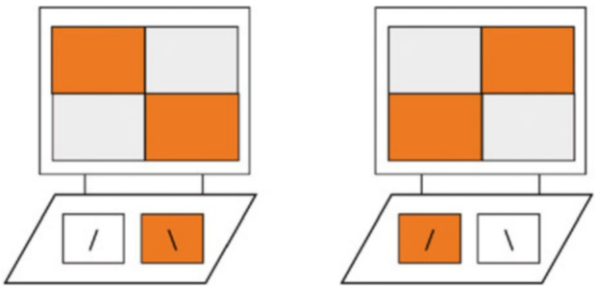

Fig. 6.18 Showing backward slash (left) and forward slash (right) on the computer screens when the keyboards are pressed

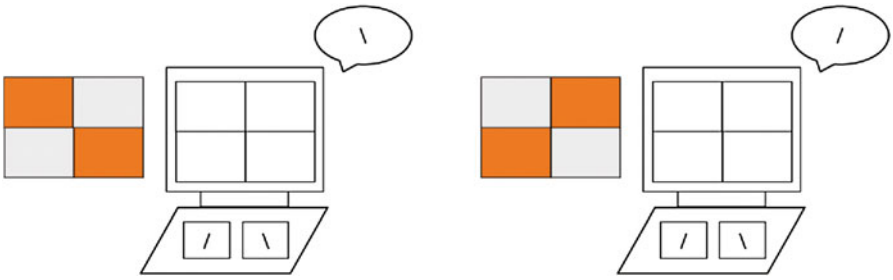

Fig. 6.19 A image recognition software that can distinguish backward slash and forward slash based on the given image

right (i.e., the backward slash) is pressed, there is a backward slash with the two-by-two pixels shown in the screen. If the computer key on the left (i.e., the forward slash) is pressed, there is a forward slash with the two-by-two pixels shown in the screen.

It is possible to add some sophistication in this world by creating an image recognition software. As shown in Fig. 6.19, if an image of a backward slash is fed into the computer, the computer should "say" that is a backward slash. If an image of a forward slash is fed into the computer, the computer should "say" that is a forward slash.

Computers do not really see things the way human see them. The computers treat pixels as numbers. Assume the orange pixel is 1 and the grey pixel is -1. Figure 6.20 shows what the computer sees the backward slash and forward slash. The computer reads data as a single line (i.e., flatted form) from left to right and bottom to top. Thus, the computer reads a backward slash as $1, -1, -1, 1$, and a forward slash as $-1, 1, 1, -1$ in this simple setting.

To differentiate these two things on the computer, some mathematical operations have to be conducted. Ideally, a certain positive value would be ouput for one and a certain negative value would be output for the other one. What operations could be used? The simplest operation that can be tried for these numbers is just adding them. If $1, -1, -1$, and 1 are added together for the backward slash, a zero (i.e., $1 - 1 - 1 + 1 = 0$) is obtained. If $-1, 1, 1$, and -1 are added for the forward slash, a

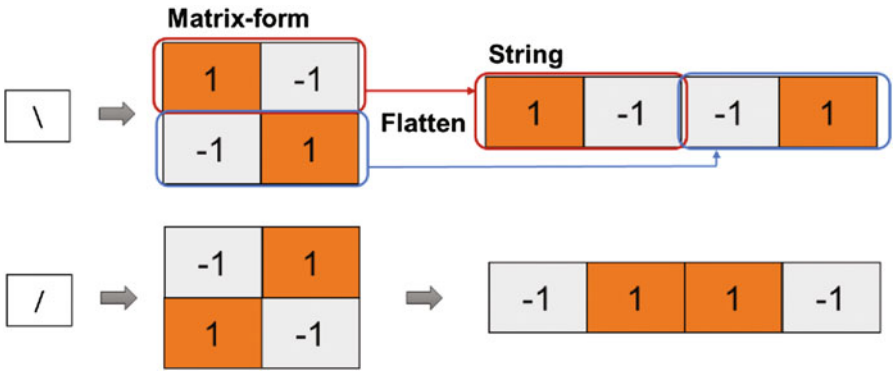

Fig. 6.20 Showing how a backward slash and a forward slash store in the computer

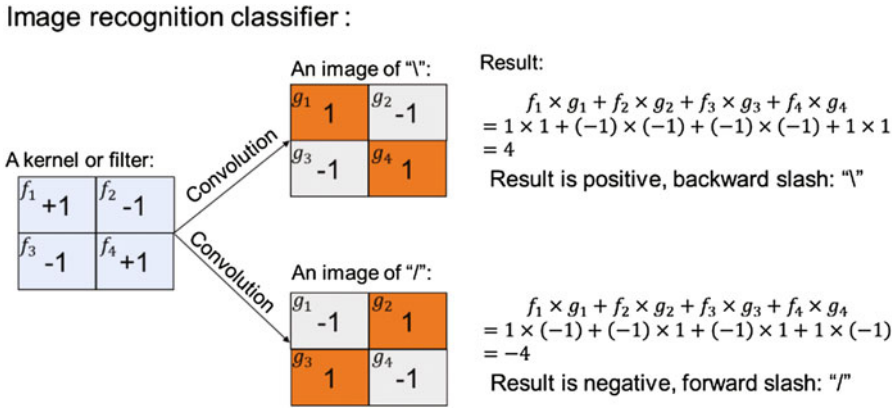

Fig. 6.21 An example showing how tell two images apart using convolution operation

zero ($-1 + 1 + 1 - 1 = 0$) is obtained again. Thus, the addition operation does not distinguish the two characters. Similarly, a multiplication operation does not effectively distinguish the characters. A mathematical operation that effectively separate these two strings of numbers is to add the first last numbers together and to subtract the sum of the two middle numbers. This operation can be defined to add the first number, subtract the second number, subtract the third number and add the fourth number of a string. For the backward slash string, this yields $1 + 1 + 1 + 1 = 4$. For the forward slash, it yields $-1 - 1 - 1 - 1 = -4$. This operation, which can distinguish the two characters, is actually a convolution operation, and its mathematical definition will be given in next section. An character recognition classifier has now been constructed in this simple world.

To make it more general, a two-by-two matrix called a kernel, or a filter, is defined as shown in Fig. 6.21. Using this kernel, the operation mentioned above can be re-described. Four symbols f_1, f_2, f_3, f_4 can be used to represent the kernel, which

is actually how this matrix is stored in the computer. They can be defined as $f_1 = 1$, $f_2 = -1, f_3 = -1, f_4 = 1$. Four different symbols g_1, g_2, g_3, g_4 are used to represent the image. The values of them are already known for a backward slash or forward slash. The operation found above is simply $f_1 \times g_1 + f_2 \times g_2 + f_3 \times g_3 + f_4 \times g_4$. If the result is positive, then the character is a backward slash. If the result is negative, the character is a forward slash.

The obvious question is how does the computer know the right values in the kernel? Actually, that is the core objective of a CNN. Humans can differentiate using the "eyeball filter". In contrast, a computer can compute very quickly, so one possibility is for a computer to check all the possibilities. There are four entities in the kernel and each entity has two choices: positive or negative. In total, there are sixteen possible kernels ($2^4 = 16$). For this method, the computer will try all the possibilities, will find that most of them work poorly, but that two of them work well (Fig. 6.22), and one of the good kernels is selected.

These 16 choices are not excessive for a world with only two letters. However, even with only two images for a computer classifier, and a computer screen with two-by-two pixels, 16 possibilities must be checked. Modern images contain many more pixels and colors to be separated apart, which results in a nearly impossibly large number of possibilities. Moreover, instead of just binary inputs, decimal values such as 0.5, 1.2, or 0.8, and any other values, can be used. So now there are almost infinite possibilities and the computer cannot check all of them.

This necessitates a smarter way to find good kernels (filters). The idea of gradient descent introduced in the previous section can be applied to CNN. Consider four random numbers (Fig. 6.23) for a filter to test performance of a classifier. The filter can be updated based on the computed error level until the error level is sufficiently small. To achieve this, the filter values are changed slightly, and a direction with reduced error can be identified. The direction is computed using an error function (or loss function). The derivatives of this error function are computed to find the gradient, or slope in any direction. The gradient can be used to find the direction of maximum and minimum error increase. This procedure can be repeated the error is minimized, resulting in a good classification filter.

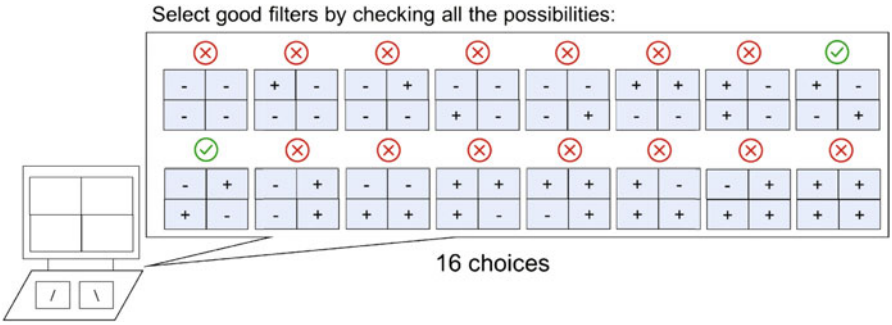

Fig. 6.22 Sixteen possible kernels to be tested

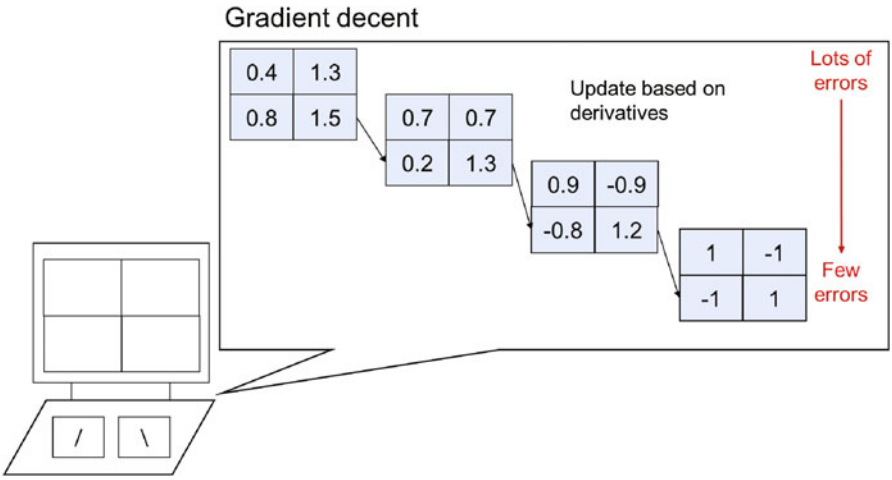

Fig. 6.23 Schematic of using gradient descent to update filter in CNN

6.3.2 Building Blocks in CNN

Before the concept of convolution neural network was proposed in the 1990's for solving image classification problem, people used other machining learning methods, such as logistic regression and support vector machine, to classify images. Those algorithms considered pixel values as features, e.g., a 36×36 image contained $36 \times 36 = 1296$ features, while a lot of spatial interactions between pixels were lost. Although it is possible to handpick features out of the image similar to what a convolution automatically does, it is very time-consuming, and the quality of those extracted features highly depends on the knowledge and experience of the domain experts. CNN uses information from adjacent pixels to down-sample the image into features by convolution and pooling and then use prediction layers (e.g., a FFNN) to predict the target values. A typical CNN structure consists of the building blocks of input, convolution, padding, stride, pooling, FFNN, and output (see Fig. 6.24). The mathematical symbols in the figure will be defined and explained in the next section.

The CNN starts from an input layer such as a signals (1D) or an image (2D). In Fig. 6.24, the input data is assumed to be one-dimensional, i.e., 1D signal or flattened image. The padded input can be obtained by adding zeros around the margin of the signal or image. Multiple convolution operations will be done using several moving kernels to extract features from the padded input. The dimensions of the convolved features can be reduced using pooling layers. Those reduced features then are used as input for a FFNN to calculate the output of the CNN. Table 6.5 shows a list of terms included in the CNN structure. Some descriptions are also provided. More detailed definitions and examples will be given in the following.

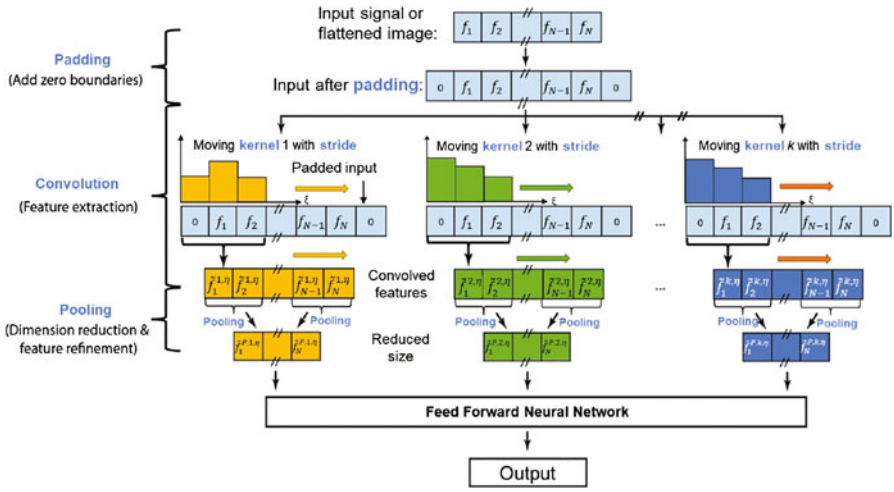

Fig. 6.24 An illustrative structure of CNN including several building blocks and concepts

Table 6.5 Terminology used in CNN and their descriptions and objectives

Terminologies	Descriptions
Convolution	A mathematical operation that does the integral of the product of functions (signals), with one of the signals slides. It can extract features from the input signals
Kernel (filter)	A function used to extract important features
Padding	A technique to simply add zeros around the margin of the signal or image to increase its dimension. Padding allows to emphasize the border values and in order lose less information
Stride	The steps of sliding the kernel during convolution. The kernel move by different stride values is designed to extract different kinds of features. The amount of stride chosen affects the size of the feature extracted
Pooling	An operation that takes maximum or average of the region from the input overlapped by a sliding kernel. The pooling layer helps reduce the spatial size of the convolved features by providing an abstracted representation of them
Fully connected layers	A FFNN in which layer nodes are connected to every node in the next layer. The fully connected layers help learn non-linear combinations of the features outputted by the convolutional layers

6.3.2.1 Convolution

Multiple convolution filters or kernels that operate over the signal or the image can be used in CNN to extract different features. The concept of convolution in machine learning stems from mathematics. Consider two univariate continuous functions: an original function, or signal, $f(t)$, and a kernel (filter) function, $\phi(t)$. The definition of the convolution is given as the integral of the product of the two functions after one is reversed and shifted:

$$(f * \phi)(t) = \int_{-\infty}^{\infty} f(t-\xi)\phi(\xi)d\xi \tag{6.46}$$

where the $*$ represents the convolution operation between the two functions, and ξ is the index sliding through the filter function. The integration interval of the convolution in mathematics is from negative infinity to positive infinity. It is typical to use multiple kernels $(\phi_1(\xi), \phi_2(\xi), \ldots, \phi_k(\xi))$ for conducting k different convolution operations in CNN. Since the signal or data in data science is finite, the definition of the convolution can be modified by limiting the integration interval from $-l$ to l

$$(f * \phi)(t) = \int_{-l}^{l} f(t-\xi)\phi(\xi)d\xi \tag{6.47}$$

where l is a real number. Since the data is stored in the computers in a discrete form, the above convolution defined for continuous functions can be modified to be a discrete convolution

$$(f * \phi)(t) = \sum_{\xi=-N}^{N} f(t-\xi)\phi(\xi)\Delta\xi \tag{6.48}$$

where N is an integer. If the $\Delta\xi$ is assumed to be 1, the discrete convolution can be written as

$$(f * \phi)(t) = \sum_{\xi=-N}^{N} f(t-\xi)\phi(\xi) \tag{6.49}$$

This is the definition used in data science where $f(t)$ is a signal or a flattened image, and $\phi(\xi)$ is a filter or kernel used to extract features from the original signal or image (in the previous example, a filter is used to classify an image is a backward slash or a forward slash).

An example is given to demonstrate how to compute a discrete convolution. Given a discrete function $f(t)$ including 12 elements $f(0)$, $f(1)$, …, $f(11)$ and a filter $\phi(\xi)$ including three elements $\phi(-1)$, $\phi(0)$, and $\phi(1)$ as shown in Fig. 6.25. The convolution is an element wise multiplication between the function $f(t)$ values and the filter $\phi(\xi)$ values, and then sum them up. Based on the definition, the first value of the convolution $(f * \phi)(1)$ can be computed as

$$\begin{aligned}(f * \phi)(1) &= f(2)\phi(-1) + f(1)\phi(0) + f(0)\phi(1) \\ &= 2 \times 6 + 7 \times 3 + 9 \times 1 = 12 + 21 + 9 = 42\end{aligned} \tag{6.50}$$

This formula is consistent with what is used in the previous backward and forward slashes example. The one dimensional case is analyzed here since the two-dimensional images are flattened into one-dimensional data and stored in the computers. Filter continues from left to right on the signal and produce the second value of the convolution as shown below (see Fig. 6.25: step 2)

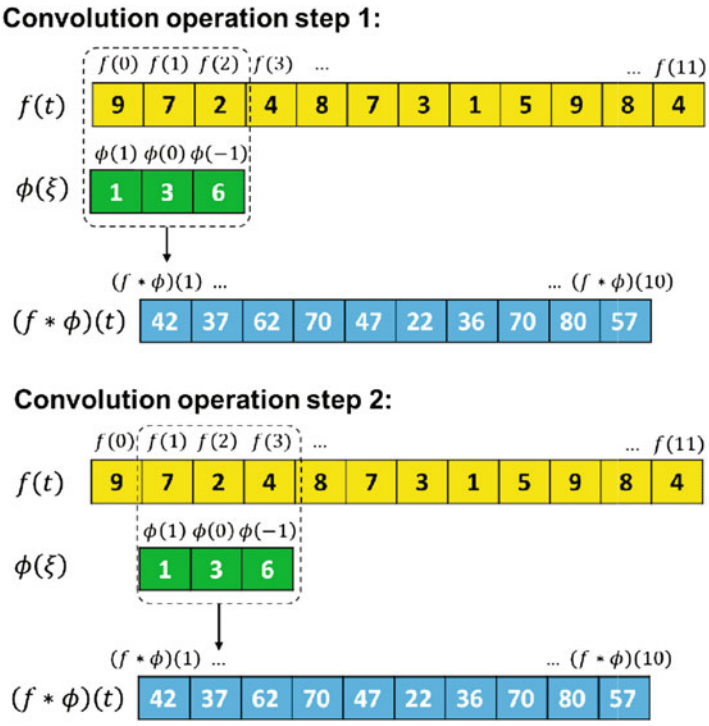

Fig. 6.25 A convolution operation example (the first two steps are shown)

$$\begin{aligned}(f*\phi)(2) &= f(3)\phi(-1)+f(2)\phi(0)+f(1)\phi(1)\\ &= 4\times 6+2\times 3+7\times 1 = 24+6+7 = 37\end{aligned} \tag{6.51}$$

The filter can be continually moved to the next position (next pixel) so all the values of the convolution $(f*\phi)(1)$, $(f*\phi)(2)$, $\ldots(f*\phi)(10)$ can be calculated.

6.3.2.2 Stride

In the above example, the filter is sliding by one position (1 pixel). This is called stride. In practice the filter can be moved by different stride values to produce the different size of convolution, and to extract different kinds of features. Figure 6.26 shows an example using a stride value of 3. The calculation of the first step is the same as the that in the previous example, so the first value of the convolution is still 42. However, for the subsequent steps the filter is moved by three pixels at a time. For example, the second value of the convolution can be computed as

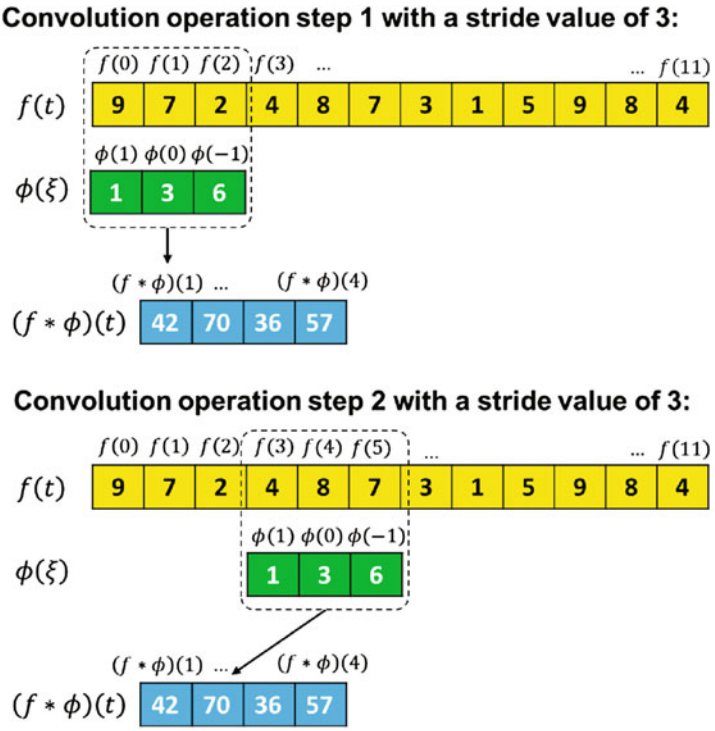

Fig. 6.26 A convolution operation with a stride value of three (the first two steps are shown)

$$\begin{aligned}(f * \phi)(2) &= f(5)\phi(-1) + f(4)\phi(0) + f(3)\phi(1) \\ &= 7 \times 6 + 8 \times 3 + 4 \times 1 = 42 + 24 + 4 = 70\end{aligned} \tag{6.52}$$

The rest of the values of the convolution can be computed based on the same procedure. As shown in Fig. 6.26, the size of the convolution $(f * \phi)(t)$ is affected by the amount of the stride. There are only four elements in the convolution if a stride value of 3 is used because some values in the original signal are skipped when the stride value is greater than 1. An equation to calculate the size of convolution for a particular filter size and stride is as follows

$$\text{Convolution size} = (\text{Signal size} - \text{Filter size})/\text{Stride} + 1 \tag{6.53}$$

This equation can be verified by putting the values for the above examples. For the example with stride 3, Convolution size $= \frac{12-3}{3} + 1 = 4$. For the example with stride 1 shown previously, Convolution size $= \frac{12-3}{1} + 1 = 10$. This equation works for both the cases.

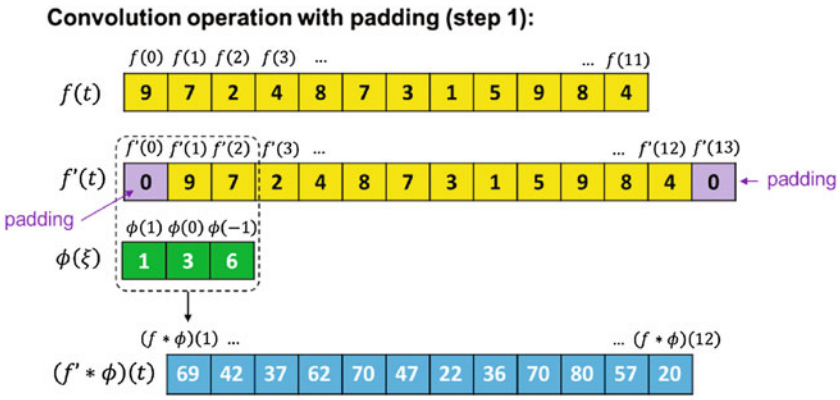

Fig. 6.27 A convolution operation with padding

6.3.2.3 Padding

From the previous examples, the convolution changes the size (or dimension) of the original signal or image. Is it possible to keep the convolution the same size as the input signal or image? Indeed, this can be achieved by padding the input. Padding is a technique to simply add zeros around the margin of the signal or image to increase the dimensions of the images before and after the convolution operation. Padding emphasizes the border pixels to reduce information loss. Figure 6.27 is an example with a 12-dimensional input $f(t)$. It can be padded to a 14-dimensional input $f'(t)$ by adding two zeros at each ends of the input. Adding these two extra dimensions results in a 12-dimensional output $(f' * \phi)(t)$ after convolution operation, which is the same as the input signal $f(t)$. The equation to calculate the dimension after convolution for a particular filter size, stride, and padding is as follows

$$\text{Convolution size} = (\text{Signal size} + 2\text{Padding size} - \text{Filter size})/\text{Stride} + 1 \tag{6.54}$$

For an image with three channels, i.e., red, green, and blue (RGB), the same operations are performed on all the three channels. In this book, only a single channel is considered. More details of the image channels can be found in the reference [17]. A CNN learns those filter values through back propagation to extract different features of the image based on training data. A CNN typically has more than one filter at each convolution layer. Those extracted features by convolution are further used to perform different tasks like classification, regression.

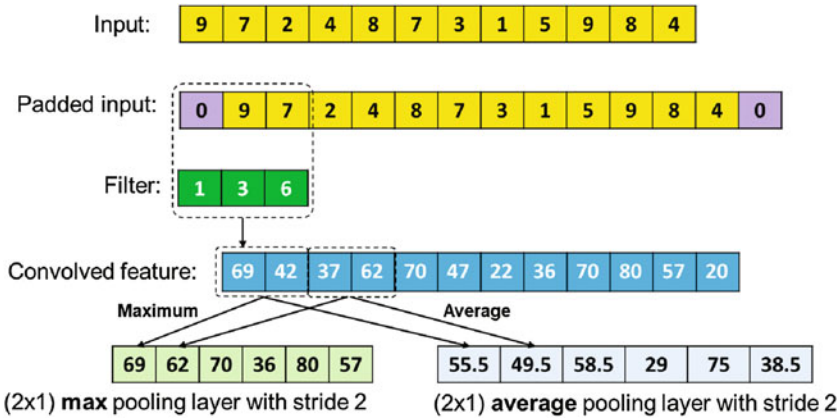

Fig. 6.28 An example showing max and average pooling layers

6.3.2.4 Pooling

Pooling layers in CNN help reduce the spatial size of the convolved features and also reduce overfitting by providing low-dimensional representations. There are two types of pooling that are widely used: max pooling and average pooling. It is similar to the convolution layer, but the pooling layer takes the maximum or average of the region from the input overlapped by the kernel (or filter). Figure 6.28 is an example showing a max pooling layer and an average pooling layer with a kernel having size of 2 and stride of 2. The max pooling operation takes the maximum of every two pixels while the average pooling operation computes the average of every two pixels. Max pooling helps reduce noise by ignoring noisy small values in the input data and hence is typically better than average pooling.

6.3.2.5 Fully Connected Networks

After the convolution and pooling layers, fully connected networks, like FFNN, are typically used with activation functions to learn complicated functional mappings between convolved features and outputs. Some activation functions have been previously introduced in the FFNN section. In the fully connected layers, neurons in a hidden layer are connected to every node in the adjacent layers. A dropout layer is normally used between two consecutive fully connective layers to reduce overfitting [18]. At the last layer the output size is decided based on the tasks. For example, one output value was sufficient for the backward and forward slashes example.

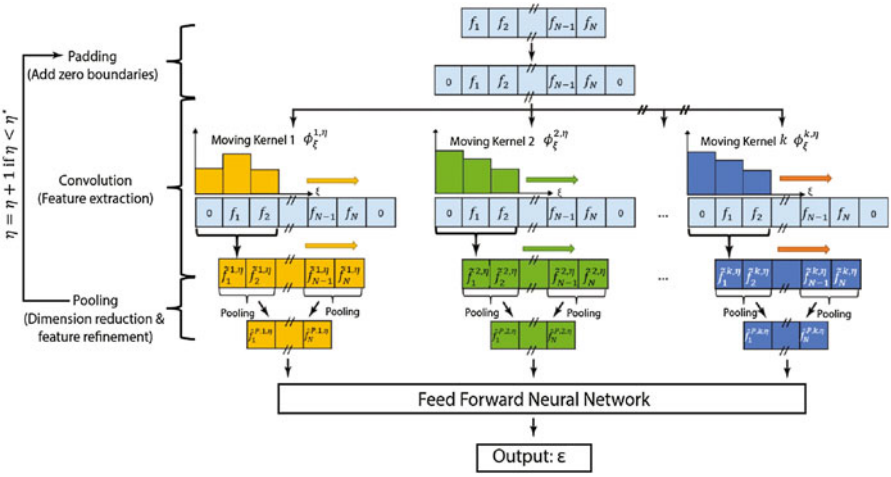

Fig. 6.29 Illustration of one-dimensional CNN with the following setup: padding, convolution, pooling, and a FFNN for regression analysis. The first three steps may be repeated [20]

6.3.3 *General Notations for CNN [Advanced Topic]*

A CNN model consists of four basic unit operations: (1) padding, (2) convolution, (3) pooling, and (4) a feed forward neural network (FFNN). A one-dimensional CNN model is shown in Fig. 6.29. It begins with the input of a series of N values f_1, f_2, …, f_N. The CNN consists of several loops of padding, convolution, and pooling. As shown in Fig. 6.29, for each loop iteration η, a padding procedure adds zeros around boundaries to ensure that the post-convolution dimension is the same as the input dimension.

After padding, kernel functions will be used to approximate the discrete convolution operator $\widetilde{f}_x^{\kappa,\eta}$ given by

$$f_x^{\kappa,\eta} = \sum_{\xi=-(L_{conv}-1)/2}^{(L_{conv}-1)/2} \phi_\xi^{\kappa,\eta} f_{x+\xi}^{padded,\eta} + b^{\kappa,\eta} \tag{6.55}$$

where $f_{x+\xi}^{padded,\eta}$ is the padded input, x is the counting index for location within the signal, ξ is the counting index for a location within the kernel, $\phi_\xi^{\kappa,\eta}$ is the κ^{th} kernel function, and $b^{\kappa,\eta}$ is the bias for η^{th} convolution process ($\eta = 1, 2, \ldots, N_{conv}$). The total number of iterations is N_{conv}. The size of the kernel function is L_{conv}. A pooling layer is used after convolution to reduce the dimensions of data and extract features from the convolved data. A one-dimensional max pooling layer is formulated by

$$\widehat{f}_{\alpha}^{P,\kappa,\eta} = \text{MAX}\left(\widetilde{f}_{\xi}^{\kappa,\eta}, \xi \in \left[(\alpha-1)L_{pooling}+1, \alpha L_{pooling}\right]\right) \tag{6.56}$$

where $\widehat{f}_{\alpha}^{P,\kappa,\eta}$ is the output value after the max pooling, $\widetilde{f}_{\xi}^{\kappa,\eta}$ is the value before the max pooling, α is the counting index for location within output after pooling ($\alpha = 1, 2, .., N_{pooling}^{\eta}$), $N_{pooling}^{\eta}$ is the size of the output after pooling for a loop iteration η, MAX is the function which returns the largest value in a given list of arguments, and $L_{pooling}$ is the length of the pooling window. Padding, convolution, and pooling are repeated N_{conv} times to extract important patterns and features. The output of the pooling layers is transferred to a fully connected FFNN which is illustrate in the Sect. 6.2.2.

The output of the CNN is represented by the vector $\boldsymbol{\varepsilon}$. The CNN training process can be written as an optimization problem, which finds the filter values in the convolution layers and weights and biases in the FFNN by minimizing the loss function (e.g., the mean square error (MSE)) representing the distance between training data $\boldsymbol{\varepsilon}^*$ and CNN output $\boldsymbol{\varepsilon}$

$$\min \text{loss function} : \text{MSE} = \frac{1}{N}\sum_{i=1}^{N}\left(\boldsymbol{\varepsilon}^i + \boldsymbol{\varepsilon}^{*i}\right)^2 \tag{6.57}$$

where N is the number of data points in the training set, $\boldsymbol{\varepsilon}^i$ is the CNN output of the *ith* data point, and $\boldsymbol{\varepsilon}^{*i}$ is the labeled output of the *ith* data point. Since $\boldsymbol{\varepsilon}^i$ is a function of the filter values in the convolution layers and weights and biases in the FFNN, those parameters in CNN can be iteratively updated by minimizing loss function based on the back-propagation algorithm.

6.3.4 *COVID-19 Detection from X-Ray Images of Patients [Advanced Topic]*

In this section a CNN model for automatically detect COVID-19 by classifying raw chest x-ray images is presented. Coronavirus 2019 (COVID-19) first appeared in December 2019 and caused a worldwide pandemic. Part of the effects of the virus is that infect lungs and airways and cause inflammation. As the inflammation progresses, a dry or barking cough would results, followed by tightness in the chest and deep pain when breathing. X-rays of the chests of COVID-19 patients show a progression that differs from healthy patients or patients with pneumonia.

The CNN model is developed to provide COVID-19 diagnosis for multi-class classification, i.e., COVID-19 vs. Pneumonia vs. No-Findings. The X-ray images of a COVID-19 patient's chest reveal several important features, which are critical for diagnosing inflammation caused by COVID-19. For example, Fig. 6.30 shows chest X-ray images taken at days 1, 4, 5 and 7 for a 50-year-old COVID-19 patient. At day 1, the lungs are clear and there are no significant findings. At days 4 and 5, ill-defined

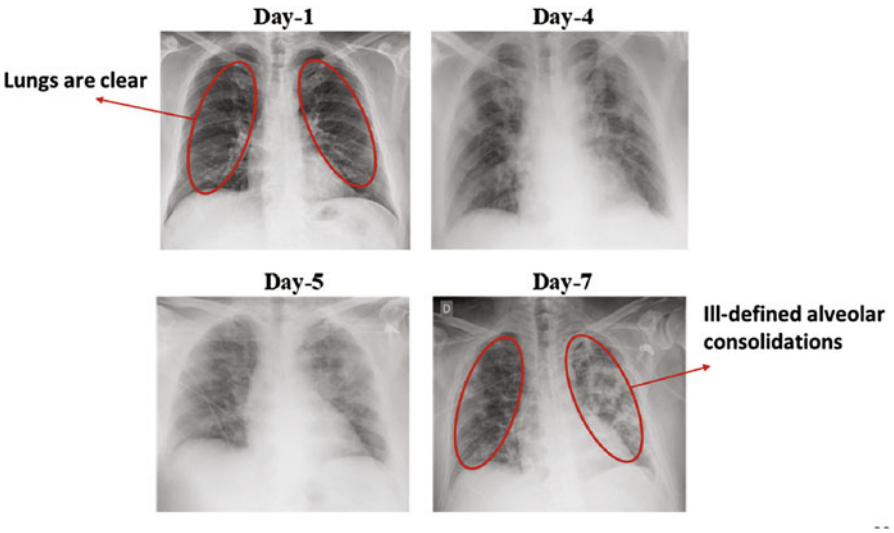

Fig. 6.30 Chest X-ray images of a 50-year-old COVID-19 patient over a week [22]

alveolar consolidations can be observed on the X-ray images. At day 7, the radiological condition has worsened, with typical findings of Acute Respiratory Distress Syndrome (ARDS [20]). Machine learning-based automatic diagnosis of COVID-19 based on chest X-ray images provide an end-to-end architecture that can automatically extract important features (such as alveolar consolidations) from images to assist clinicians in making accurate diagnoses (Fig. 6.31).

The X-ray images obtained from two sources were used for the diagnosis of COVID-19. The first COVID-19 X-ray image database was generated and collected by Cohen JP [22]. Another chest X-ray image database was provided by Wang et al. [23]. These two X-ray databases are combined for a total of 125 X-ray images of COVID-19 patients (43 female, 82 male, average age of approximately 55 years), 500 X-ray images pneumonia patients, and 500 X-ray images of patients with no-findings. The grey scale X-ray images are obtained from institute's PACS system [24]. Each pixel in the images has an intensity ranging from 0 to 255. Nine-hundred X-ray images are used for training and 225 images for validation (including 28 COVID-19 cases, 88 pneumonia cases, and 109 no-finding cases). The five-fold cross-validation is used to evaluate the model performance (see Chap. 2 for more details of cross-validation).

A typical CNN structure has many convolution layers that can extract features and produce feature maps from the input with the applied filters, subsequent pooling layers to reduce the size of the feature maps, and fully connected layers (i.e., a FFNN). The trainable internal parameters in CNN are adjusted to accomplish a classification or regression task. The developed CNN structure described in this section is inspired by the Darknet-19 model [25], which is a well-tested classifier for

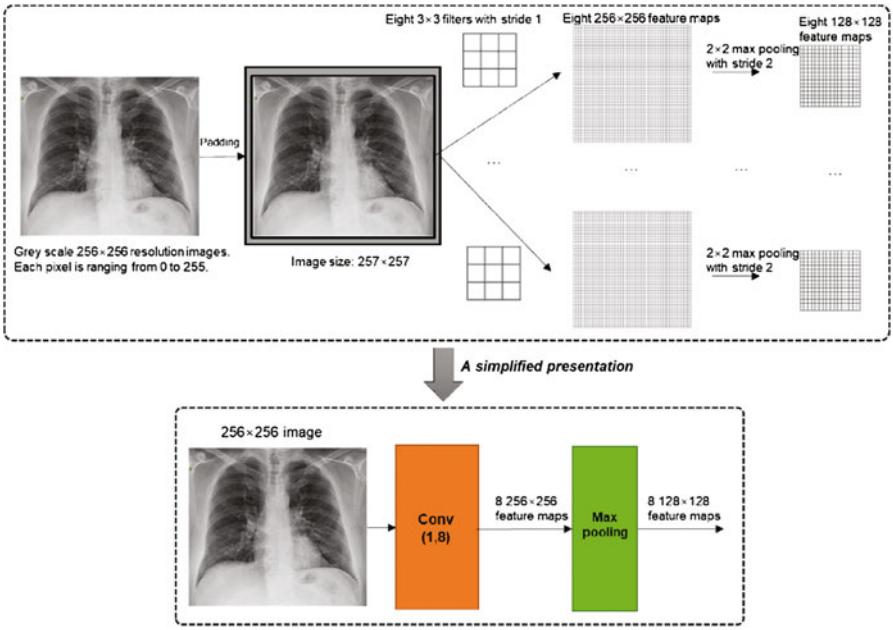

Fig. 6.31 A schematic presentation of the first convolution layer and max pooling layer

many real-time object detection systems. Leaky rectified linear unit (leaky ReLU) is used as an activation function in the CNN.

The CNN structure consists of 17 convolutional layers and 5 max pooling layers. These layers are typical CNN layers with different filter numbers, sizes, and stride values. A schematic presentation for the first convolution layer and max pooling layer is given in Fig. 6.31. After padding operation introduced in the Sect. 6.3.1, the input image with 256 × 256 resolution increase its size to 257 × 257. Eight 3 × 3 filters with stride 1 are then used to produce eight 256 × 256 feature maps. This method allows different features from the input image to be extracted. The values in the filters will be obtained during the data training process. To reduce the size of the feature maps, eight 2 × 2 max pooling operators with stride 2 are used so that the size of feature maps can be reduced to 128 × 128. To present the whole structure of the CNN in a compact manner, a simplified presentation of the first convolution and max pooling layer is also shown in Fig. 6.31.

Figure 6.32 shows the CNN structure with 17 convolutional layers and 5 max pooling layers. Each convolution block layer has one convolutional layer followed by LeakyReLU activation functions (see Sect. 6.2.1). ReLU activation function has zero value in the negative part of their derivatives, but LeakyReLU has a small value that can overcome the dying neuron problem [26]. The CNN model performs the COVID-19 detection task to determine the labels of the input chest X-ray images. COVID-19 is represented by a vector $[1 \quad 0 \quad 0]^T$, pneumonia is represented by a

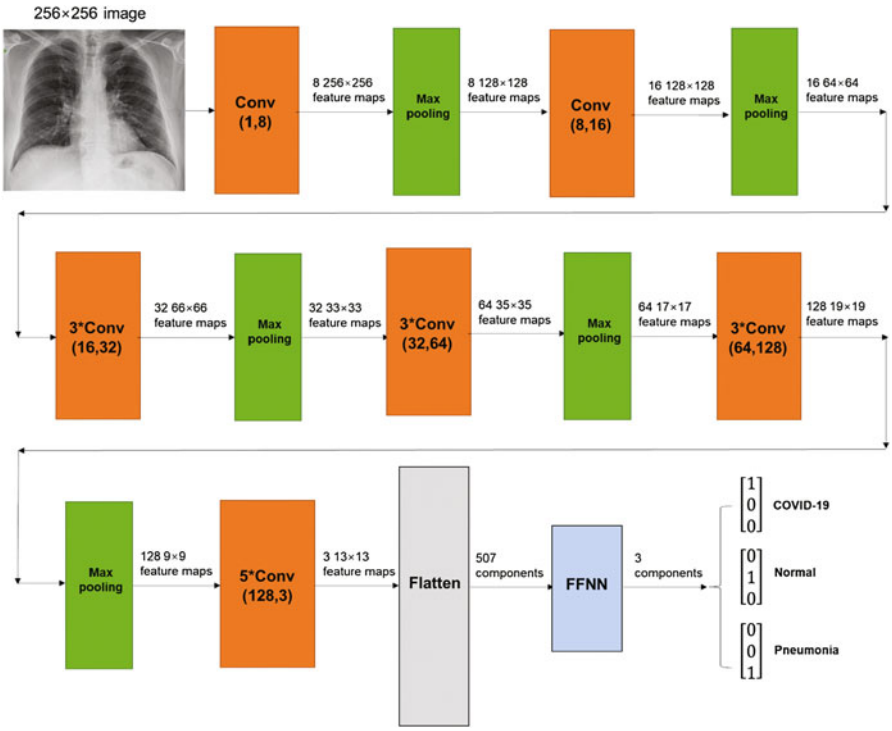

Fig. 6.32 The architecture of the CNN model

vector $[0 \quad 1 \quad 0]^T$, a normal (i.e., No-Findings) represented by $[0 \quad 0 \quad 1]^T$. Finally, the layer details and layer parameters of the model are given in the Python code included with the book. The developed deep learning model consists of 1,164,434 parameters. The Adam optimizer [27] is used for updating the weights and loss functions with a selected learning rate of 3×10^{-3} (Supplementary Data 6.1).

The multi-class classification performance of the CNN model can be evaluated using the error matrix shown in Fig. 6.33. The error matrix allows visualization of the general performance of the CNN model. A total of 225 images in the test set are used to evaluate the performance of the model. The classification accuracy for the COVID-19 category is $24/28 = 85.7\%$, the accuracy for the No-finding category is $102/109 = 94.6\%$, and the accuracy for the Pneumonia category is $75/88 = 85.2\%$. The CNN model achieved an average classification accuracy of 88.5% for these categories.

During the COVID-19 pandemic, X-ray imaging is a very important assisted tool to the diagnostic tests for the early diagnosis. Deep learning models such as the one introduced in this section provide high accuracy rate in diagnosis and thus are particularly useful in identifying early stages of COVID-19 patients. The deep learning model can potentially be used in healthcare centers for an early diagnosis

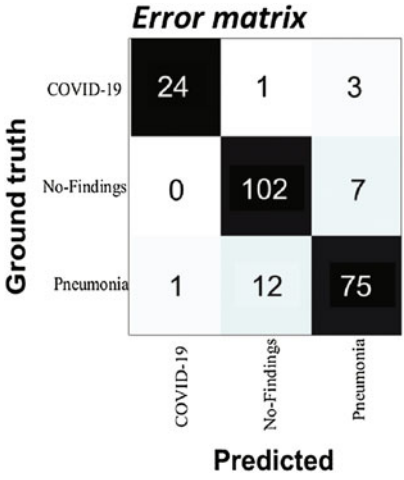

Fig. 6.33 The error matrix results of the multi-class classification COVID-19 task

or a second "opinion". However, in this case of CNN, more than one million parameters are involved. A large number of training datapoints are required to train those parameters. In the next section, a mechanistic data science approach will be introduced. It significantly reduces the model parameters by using mechanistic knowledge, which can achieve a good model performance using a small number of datapoints.

6.4 Musical Instrument Sound Conversion Using Mechanistic Data Science

6.4.1 Problem Statement and Solutions

A machine learning model that can change any piano sound/music to a guitar sound/music will be used to demonstrate mechanistic data science. Pianos and guitars can perform the same music, but a piano and a guitar have quite different instrumental structures and sounds. The challenge is to use mechanistic data science to convert piano sounds to guitar sounds. The first step in the process is to change a single piano note to the same note that sounds like a guitar (Fig. 6.34).

The input consists of eight pairs of notes (notes A4, A5, B5, C5, C6, D5, E5, G5) performed by a piano and by a guitar. For this example, signals from an open-source database [30] are used for the training dataset. Figure 6.35 shows the time-amplitude signal (sound signal) for the A4 note for piano and guitar. The two signals look different, but they have the same fundamental frequency (same pitch). A Fourier analysis can be conducted to obtain the fundamental frequency and harmonics of the sound (see Chap. 4). It should be noted that the dimension of each curve is very high since the sampling rate was 44,100 Hz.

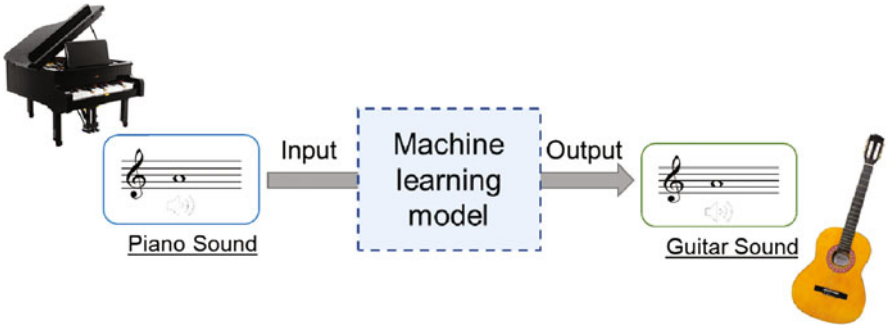

Fig. 6.34 A schematic of changing a piano sound to a guitar sound suing machine learning. Two images in the figure come from the Internet [29, 30]

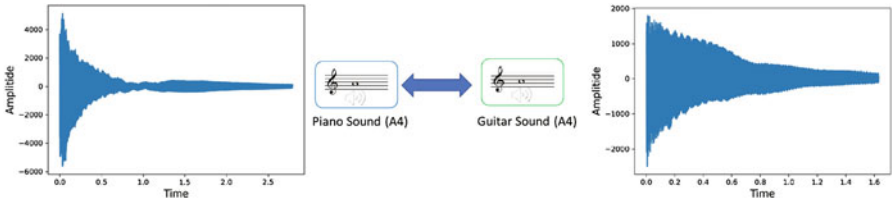

Fig. 6.35 A pair of A4 time-amplitude curves for piano and guitar. Audios are available in the E-book (Supplementary Audio 6.1)

Given the training data paired sounds from the piano and the guitar, two strategies can be used to train a machine learning model: (1) a pure CNN analysis and (2) a mechanistic data science analysis (see Fig. 6.36). The first strategy uses the deep CNN architecture introduced in the previous section. For this strategy, piano sounds are used as input and guitar sounds are used as output. Both are high-dimensional time-amplitude curves. Convolution layers and pooling layers can be used repeatedly to extract features from the piano sound signals and a low-dimensional representation (deep features shown in Fig. 6.36) of the original sound curve can be obtained after the CNN. These deep features then are mapped to another set of deep features, which can be converted to guitar sounds using another CNN structure for feature reconstruction. This flexible deep learning structure can be applied to many regression and classification applications with various data structures. However, a drawback of this strategy is the high number of trainable parameters involved in the CNN and FFNN structures. Both filter values and hyperparameters (weights and biases) need to be determined by the training dataset. The number of hyperparameters in a CNN can be thousands, millions, or even larger, depending on the size of the network. For some applications where the amount of data is small or the quality of data is low, the performance of the CNN is limited. To overcome

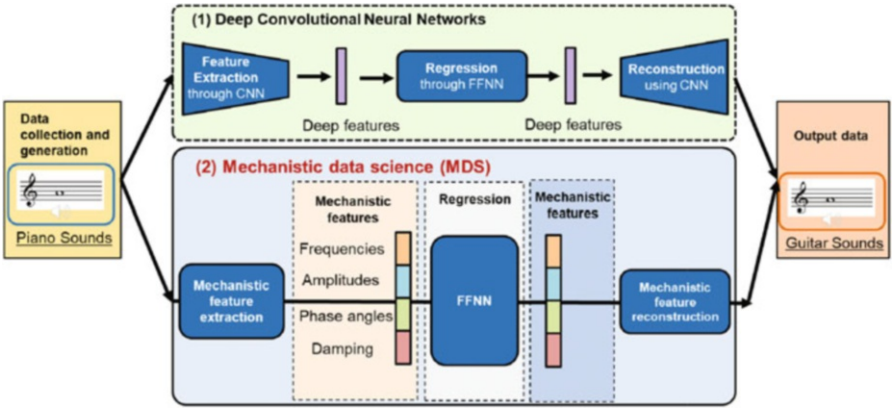

Fig. 6.36 A schematic of two solutions for changing a piano sound to a guitar sound: pure CNN approach and mechanistic data science

this drawback of the standard CNN, another strategy called mechanistic data science is proposed (Fig. 6.36). Instead of extracting deep features by CNN, mechanistic data science extracts mechanistic features based on the underlying scientific principles. In this specific problem, a low-dimensional set of mechanistic features can be extracted from each piano signal and guitar signal. Each signal can be simplified to a set of sine functions, where the mechanistic features are the frequencies, damping coefficients, amplitudes, and phase angles (see Chap. 4 for more details). The damping coefficients describe a decreasing of the amplitude of the sound wave due to frictional drag or other resistive forces. These mechanistic features can be obtained by Short Time Fourier Transform (STFT) and a regression to fit a mechanistic model, such as a spring-mass-damper model introduced in Chap. 4. In this way, a high-dimensional sound curve can be represented by a set of mechanistic features with a physical meaning. The hyperparameters involved in the model can be significantly decreased from thousands to dozens in this manner, which reduces the amount of training data required. The computer codes for conducting CNN and MDS are included in the E-book (Supplementary Data 6.2).

Figure 6.37 shows the values of the CNN and the MDS loss functions at each iteration step of training. The goal of the training is to minimize the loss function, and the MDS (right in Fig. 6.37) provides a better solution than the CNN (left in Fig. 6.37). The loss function of the CNN is volatile and does not converged to zero. The high number of parameters in the CNN and the amount of training data available in this case is insufficient to enable the CNN to find appropriate values of all the parameters. Training a CNN model requires the number of data points to be larger than the dimension of input signal or image. In this case, the dimension of input is a million, so a million training data points are needed to successfully train the CNN model. As the results show, the eight training data are not enough. In contrast, the

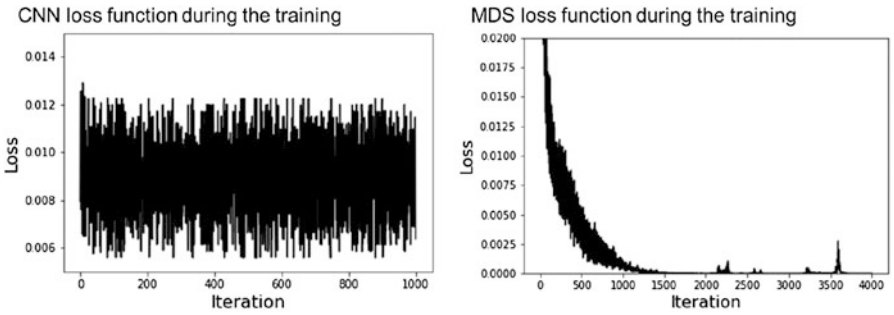

Fig. 6.37 Values of CNN loss function (left) and MDS loss function (right) at each iteration step during the training

loss function of the MDS converges to zero if sufficient iterations steps are used. MDS reduces the number of parameters significantly using mechanistic features, which provides an efficient manner to solve this kind of problem with a relatively small amount of data.

6.4.2 Mechanistic Data Science Model for Changing Instrumental Music [Advanced Topic]

A mechanistic data science model is presented which converts music from one instrumental sound to another. In particular, a piano sound will be converted to a guitar sound.

The training data for the analysis consisted of eight pairs of piano and guitar sound files, with signal durations ranging from 1.5 to 3.0 s. The notes used are A4, A5, B5, C5, C6, D5, E5, and G5. A representative pair of the time-amplitude curves is shown in Fig. 6.35. The sampling rate used is 44.1 *kHz*. Recorded duration for the piano sounds is 2.8 s and for the guitar sounds is 1.6 s. Thus, the dimension of a piano sound is 120,000 (44.1 kHz $\times$ 2.8 s), and the dimension of the guitar sound is 72,000 (44.1 kHz $\times$ 1.6 s). The high dimension of the input signal necessitates a mechanistic feature extraction to efficiently perform the analysis.

Mechanistic feature were extracted from the signals to enhance the mapping from the piano to the guitar. Short Time Fourier Transform (STFT) is used to reveal the frequency, amplitude, damping and phase angle, as shown in Chap. 4 (see Fig. 6.38).

A mechanistic model of the system is introduced in the form of a spring-mass-damper system. The STFT and a least-squares optimization are performed to determine the parameters for the mechanistic model, as shown in Chap. 4. The coefficients of the reduced order model for the A4 piano and guitar signals are shown in Tables 6.6 and 6.7, respectively. Data for the other piano and guitar sounds are included in the E-book (Supplementary Data 6.2).

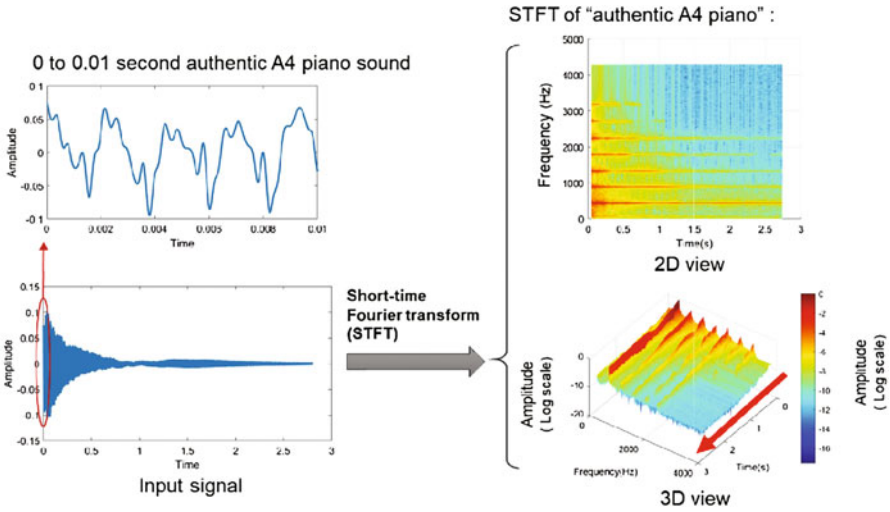

Fig. 6.38 An A4 piano sound signal and its STFT result (2D and 3D)

Table 6.6 Optimal coefficients to represent the authentic A4 piano sound

Type	Frequency (Hz)	Initial amplitudes	Damping coefficients	Phase angle (rad)
Fundamental	4.410E+02	1.034E-01	3.309E+00	6.954E-01
Harmonics	8.820E+02	1.119E-02	1.844E+00	7.202E-01
	1.323E+03	6.285E-03	5.052E+00	3.469E-01
	1.764E+03	7.715E-04	2.484E+00	5.170E-01
	2.205E+03	1.455E-03	8.602E+00	5.567E-01
	2.646E+03	5.130E-04	1.198E+01	1.565E-01
	3.087E+03	1.899E-04	8.108E+00	5.621E-01
	3.528E+03	3.891E-05	3.282E+00	6.948E-01

Table 6.7 Optimal coefficients to represent the authentic A4 guitar sound

Type	Frequency (Hz)	Initial amplitudes	Damping coefficients	Phase angle (rad)
Fundamental	4.400E+02	1.649E-02	1.287E+00	9.798E-01
Harmonics	8.800E+02	8.022E-03	1.865E+00	2.848E-01
	1.320E+03	2.551E-03	2.176E+00	5.950E-01
	1.760E+03	5.454E-03	1.100E+00	9.622E-01
	2.200E+03	5.523E-03	3.346E+00	1.858E-01
	2.640E+03	6.742E-03	2.504E+00	1.930E-01
	3.080E+03	7.643E-04	1.666E+00	3.416E-01
	3.520E+03	9.748E-04	2.609E+00	9.329E-01

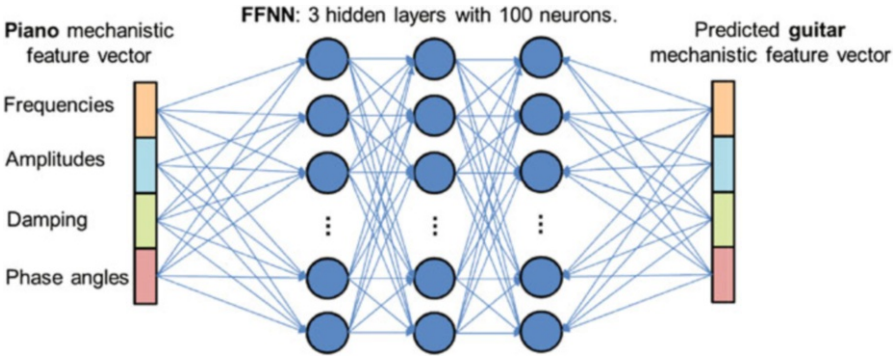

Fig. 6.39 FFNN structure with reduced mechanistic features

Deep learning for regression is performed using the reduced mechanistic features as input and output. A fully-connected FFNN is used to map the relationships between piano and guitar sounds. Figure 6.39 shows the FFNN structure with the reduced order mechanistic model. Three hidden layers with 100 neurons are used for this FFNN. The *tanh* function is used as the activation function. Standard mean squared error (MSE) is used as the loss function. The mathematical description of the optimization process of the FFNN training can be found in Sect. 6.2.2. The code for implement MDS and FFNN is attached to the E-book. Note that the generation of the guitar sound from the piano sound is possible with a significantly smaller dimension (i.e., 4(sets) $\times$ 8 (features) $=$ 32) and only 8 data points are sufficient to train the MDS model (Supplementary Data 6.2).

Figure 6.40 shows the result of reconstructing an A4 guitar note from an input piano key. Audios in the figure are available in the E-book. In this figure, the mechanistic features (i.e., frequencies, amplitudes, damping coefficients, and phase angles) obtained from ground truth piano sound, ground truth guitar sound, and MDS generated guitar sound are compared. As shown in the Fig. 6.40, the features of ground truth guitar sound (marked in orange) and MDS generated guitar sound (marked in blue) are very similar, and they are different from the features of ground truth piano sound (marked in green).

Figure 6.41 shows the time-amplitude curves (i.e., sound waves) reconstructing an A4 guitar note from an input piano key. Figure 6.41b, d present the magnification plots of the sound waves ranging from 0 to 0.01 s. The MDS generated guitar sound wave is similar to the authentic one (i.e., ground truth), which is quite different from the input piano sound wave as shown in Fig. 6.41e, f. The enveloped shapes of the sound waves are controlled by the damping coefficients. The detailed wave shapes are affected by the frequencies and their amplitudes. Figure 6.41 demonstrates that the MDS generated guitar sound captures the key features compared to the authentic guitar sound.

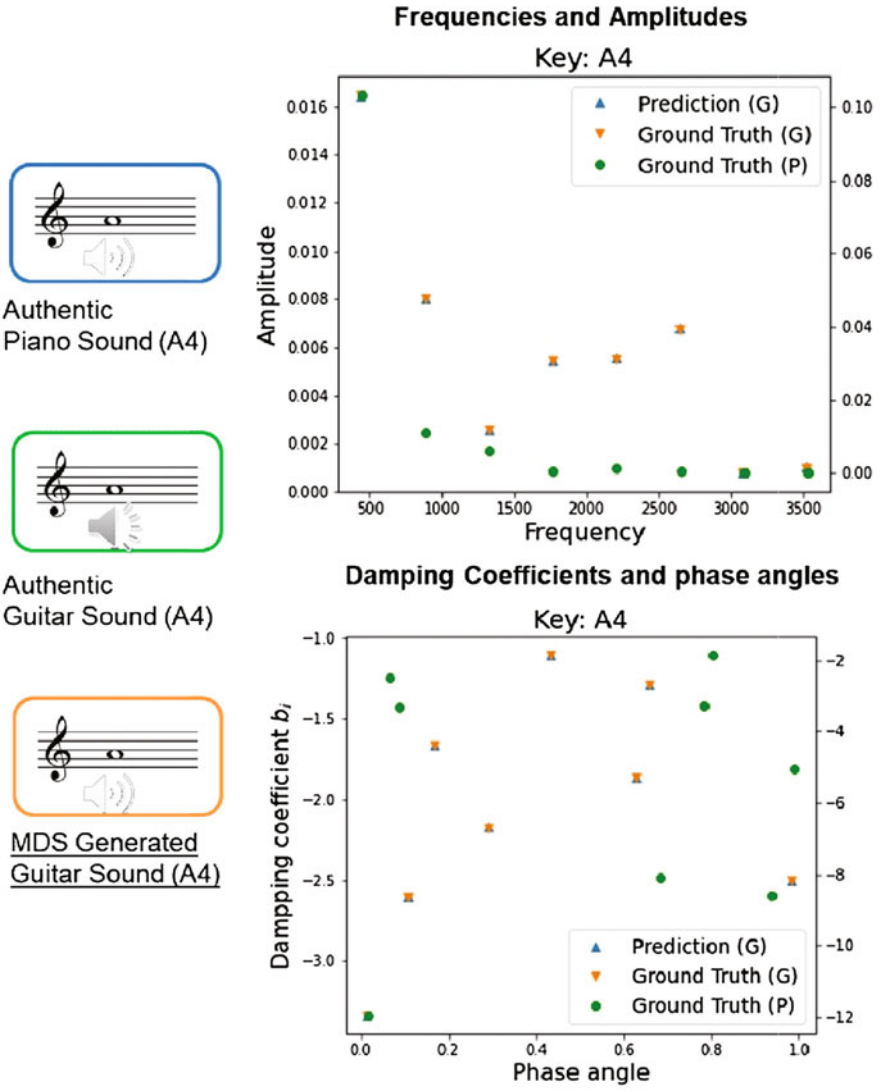

Fig. 6.40 Results of reconstructing a single guitar key A4 from a piano key as input. Audios are available in the E-book (Supplementary Audio 6.2)

6.5 Conclusion

Two major variants of Deep learning neural networks (i.e., FFNN and CNN) are presented in this Chapter and their ability is demonstrated through examples. The neural networks can be combined with mechanistic data science to simplify and enable solutions with relatively small amount of data. A musical sound conversion

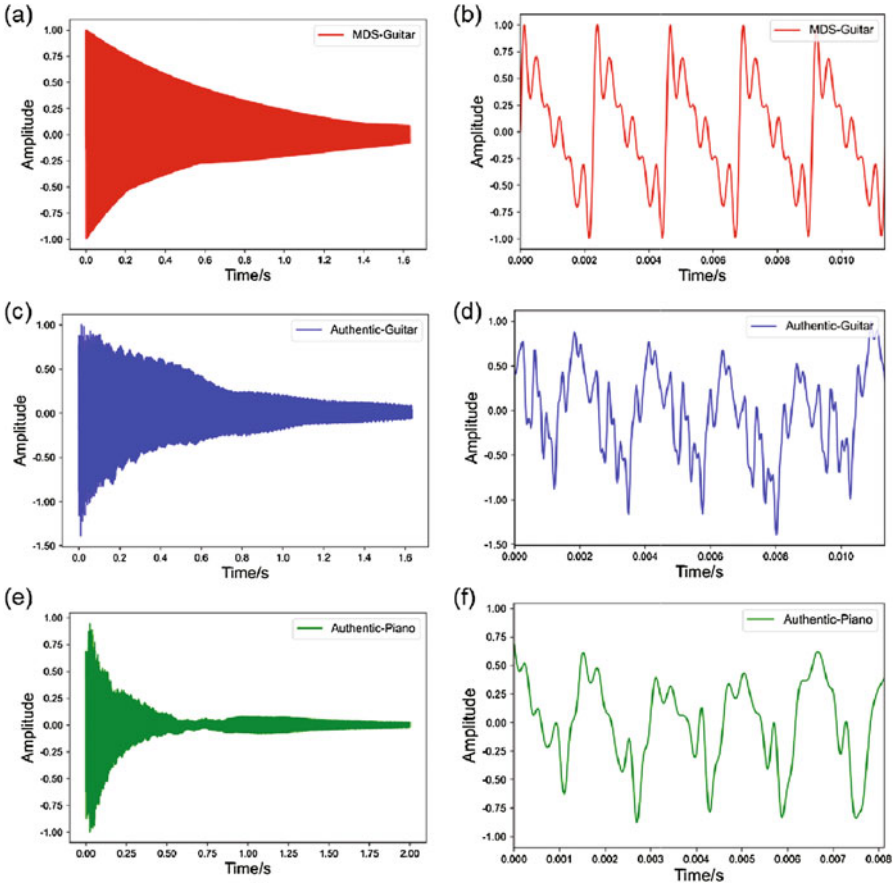

Fig. 6.41 Time-amplitude curves (sound waves) of reconstructing a single guitar key A4 from a piano key as input. (**a**) MDS generated guitar sound. (**b**) Magnification plot of MDS generated guitar sound ranging from 0 to 0.01 s, to highlight the detailed wave shape. (**c**) Authentic guitar sound. (**d**) Magnification plot of Authentic guitar sound ranging from 0 to 0.01 s, to highlight the detailed wave shape. (**e**) Authentic piano sound. (**f**) Magnification plot of Authentic piano sound ranging from 0 to 0.01 s, to highlight the detailed wave shape

from piano to guitar demonstrated this capability. Four sets of mechanistic features replaced the CNN, and these features are used as the input layer to the FFNN for the training of the neural network. Incorporating mechanistic knowledge to perform dimension reduction opens a new avenue to other scientific methods. Researchers have demonstrated new approaches to solve the partial differential equation by application dimension reduction using mechanistic deep learning [31]. This allows the solution of scientific problems with limited data and limited understanding of the relevant physics. This is highly advantageous for predicting biomechanical process such as progression of scoliosis [32] and optimizing additive manufacturing processes by discovering dimensionless parameters [33].

References

1. Schmidhuber J (2015) Deep Learning in neural networks: an overview. Neural Netw 61:85–117
2. Chen Y-Y, Lin Y-H, Kung C-C, Chung M-H, Yen I-H (2019) Design and implementation of cloud analytics-assisted smart power meters considering advanced artificial intelligence as edge analytics in demand-side management for smart homes. Sensors 19(9):2047
3. Smith JK, Brown PC, Roediger III HL, McDaniel MA (2014) Make it stick. The science of successful learning (2015):346–346
4. Cohen JP (2020) COVID-19 image data collection. https://github.com/ieee8023/COVID-chestxray-dataset
5. Ivakhnenko AG, Lapa VG (1967) Cybernetics and forecasting techniques. American Elsevier, New York
6. Dechter R (1986) Learning while searching in constraint-satisfaction problems. University of California, Computer Science Department, Cognitive Systems Laboratory, Los Angeles
7. LeCun et al (1989) Backpropagation applied to handwritten zip code recognition. Neural Comput 1:541–551
8. Hinton GE, Dayan P, Frey BJ, Neal R (1995) The wake-sleep algorithm for unsupervised neural networks. Science 268(5214):1158–1161
9. Hochreiter S, Schmidhuber J (1997) Long short-term memory. Neural Comput 9(8):1735–1780
10. Nvidia CEO bets big on deep learning and VR. Venture Beat, 5 April 2016
11. Maas AL, Hannun AY, Ng AY (2013) Rectifier nonlinearities improve neural network acoustic models. Proc icml 30(1)
12. https://machinelearningmastery.com/overfitting-and-underfitting-with-machine-learning-algorithms/
13. https://scikit-learn.org/stable/modules/neural_networks_supervised.html
14. https://pytorch.org/
15. https://www.mathworks.com/help/deeplearning/ref/trainnetwork.html
16. https://arxiv.org/abs/1412.6980
17. https://machinelearningmastery.com/introduction-to-1x1-convolutions-to-reduce-the-complexity-of-convolutional-neural-networks/
18. https://machinelearningmastery.com/dropout-for-regularizing-deep-neural-networks/
19. Li H, Kafka OL, Gao J, Yu C, Nie Y, Zhang L et al (2019) Clustering discretization methods for generation of material performance databases in machine learning and design optimization. Comput Mech 64(2):281–305
20. https://www.mayoclinic.org/diseases-conditions/ards/symptoms-causes/syc-20355576
21. https://radiopaedia.org/cases/COVID-19-pneumonia-evolution-over-a-week-1?lang=us
22. Cohen JP (2020) COVID-19 image data collection. https://github.com/ieee8023/COVID-chestxray-dataset
23. Wang X, Peng Y, Lu L, Lu Z, Bagheri M, R.M. (2017) Summers Chest x-ray8: hospital-scale chest x-ray database and benchmarks on weakly-supervised classification and localization of common thorax diseases. In: Proceedings of the IEEE conference on computer vision and pattern recognition, pp 2097–2106
24. Choplin R (1992) Picture archiving and communication systems: an overview. Radiographics 12:127–129
25. Redmon J, Farhadi A (2017) YOLO9000: better, faster, stronger. In: Proceedings of the IEEE conference on computer vision and pattern recognition
26. https://medium.com/@shubham.deshmukh705/dying-relu-problem-879cec7a687f
27. Kingma DP, Ba J (2014) Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980
28. https://www.brothers-brick.com/2020/07/23/lego-ideas-21323-grand-piano-makes-music-starting-aug-1st-news/
29. https://www.dawsons.co.uk/blog/a-guide-to-the-different-types-of-guitar

30. https://www.apronus.com/
31. Zhang L, Cheng L, Li H, Gao J, Yu C, Domel R, Yang Y, Tang S, Liu WK (2021) Hierarchical deep-learning neural networks: finite elements and beyond. Comput Mech 67:207–230
32. Tajdari M, Pawar A, Li H, Tajdari F, Maqsood A, Cleary E, Saha S, Zhang YJ, Sarwark JF, Liu WK (2021) Image-based modelling for adolescent idiopathic scoliosis: mechanistic machine learning analysis and prediction. Comput Methods Appl Mech 374:113590
33. Saha S, Gan Z, Cheng L, Gao J, Kafka OL, Xie X, Li H, Tajdari M, Kim HA, Liu WK (2021) Hierarchical Deep Learning Neural Network (HiDeNN): an artificial intelligence (AI) framework for computational science and engineering. Comput Methods Appl Mech 373, p 113452