https://doi.org/10.1142/9789811247323_0013

Chapter 13

Deep Learning and Its Adversarial Robustness: A Brief Introduction

Fu Wang, Chi Zhang, Peipei Xu, and Wenjie Ruan

College of Engineering, Mathematics and Physical Sciences
University of Exeter, UK

Deep learning, one of the most remarkable techniques in computational intelligence, has become increasingly popular and powerful in recent years. In this chapter, we, first of all, revisit the history of deep learning and then introduce two typical deep learning models including convolutional neural networks (CNNs) and recurrent neural networks (RNNs). After that, we present how the deep learning models are trained and introduce currently popular deep learning libraries and frameworks. Then we focus primarily on a newly emerged research direction in deep learning—adversarial robustness. Finally, we show some applications and point out some challenges of deep learning. This chapter cannot exhaustively cover every aspect of deep learning. Instead, it gives a short introduction to deep learning and its adversarial robustness, and provides a taste of what deep learning is, how to train a neural network, and why deep learning is vulnerable to adversarial attacks, and how to evaluate its robustness.

13.1. Introduction

As a branch of computational intelligence, deep learning or deep neural networks (DNNs) have achieved great success in many applications such as image analysis [1], speech recognition [2], and text understanding [3]. Deep learning has demonstrated superior capability in approximating and reducing large, complex datasets into highly accurate predictive and transformational outputs [4].

The origin of deep learning can be traced back to 1943, when Walter Pitts and Warren McCulloch built a mathematical model, the M-P model, to mimic the

biological neural networks inside the human brain [5]. The M-P model is the earliest and most basic unit of artificial neural networks. It is an over-simplification of the biological brain yet successfully simulates the fundamental function of a real neuron cell in our brain [5]. As shown in Figure 13.1, a neuron receives signals from other neurons and compares the sum of inputs with a threshold. The output of this neuron is then calculated based on a defined activation function.

In 1958, Frank Rosenblatt proposed the perceptron, one of the simple artificial neural networks [6]. Figure 13.2 shows the structure of a perceptron. A single-layer perceptron has a very limited learning capacity, which cannot even solve a XOR problem [7]. But a multilayer perceptron exhibits a better learning performance by stacking multiple hidden layers, as shown in Figure 13.2(b). Such a structure is still commonly seen in modern deep neural networks with a new name, i.e., fully connected layer, since the neurons in each layer connect every neuron in the next layer. Kunihiko Fukushima, known as the inventor of convolutional neural networks, proposed a type of neural network, called neocognitron [8]. It enables the model to easily learn the visual patterns by adopting a hierarchical and multilayered structure.

In the development of deep learning, there were several technical breakthroughs that significantly boosted its advances [9]. The first notable breakthrough was the backpropagation (BP) algorithm which is now a standard method for training neural networks, that computes the gradient of the loss function for weights in the network by the chain rule. The invention of BP can be traced back to 1970, when Seppo Linnainmaa presented a FORTRAN code for backpropagation in his master's thesis [10]. In 1985, the concept of BP was for the first time applied in neural networks [11]. Later on, Yann LeCun demonstrated the first practical BP applied in handwritten zip code recognition [12].

Another significant breakthrough in deep learning is the development of graphics processing units (GPUs), which has improved the computational speed

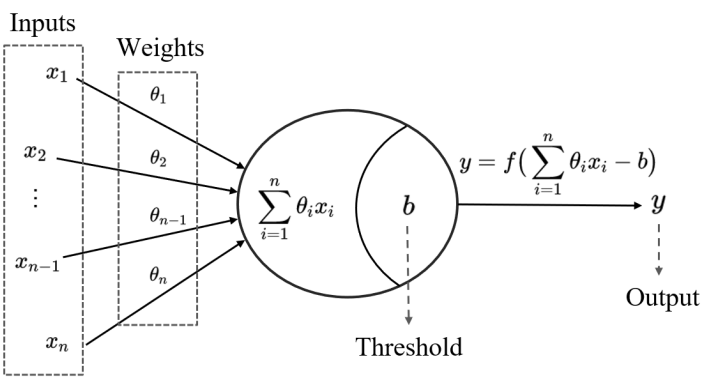


Figure 13.1: An M-P model.

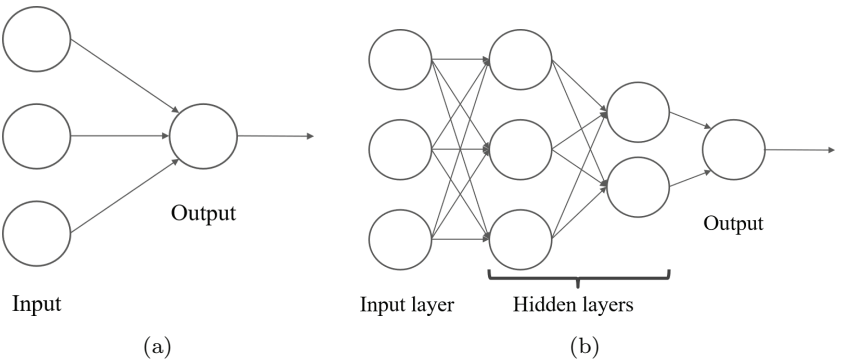


Figure 13.2: Illustrations of a single-layer perceptron and a three-layer perceptron.

by thousands of times since 1999 [13]. Deep learning has also revealed a continuously improved performance when more data are used for training. For this reason, Fei-Fei Li launched ImageNet in 2009 [1], which is an open-source database containing more than 14 million labeled images. With the computational power of GPUs and the well-maintained database ImageNet, deep learning started to demonstrate a surprisingly remarkable performance on many challenging tasks [4]. A notable example is AlexNet [14], which has won several international image competitions and demonstrated a significant improvement in the best performance in multiple image databases [14]. Since then, deep learning has become one of the most important realms in computational intelligence. A more comprehensive introduction to the development of deep learning can be seen in [15–17].

13.2. Structures of Deep Learning

In this section, we first detail the activation functions that are widely applied in deep neural networks. Then, we introduce two typical deep learning structures—convolutional neural networks and recurrent neural networks.

13.2.1. *Activation Functions*

Activation function is the most important component in deep neural networks, empowering models with a superior capacity to learn nonlinear and complex representations. Typical activation functions used in deep neural networks include Sigmoid, Hyperbolic Tangent (Tanh), and Rectified Linear Unit (ReLU).

The Sigmoid activation function was pervasively adopted at the early stage of neural networks and still plays an important role in deep learning models [18].

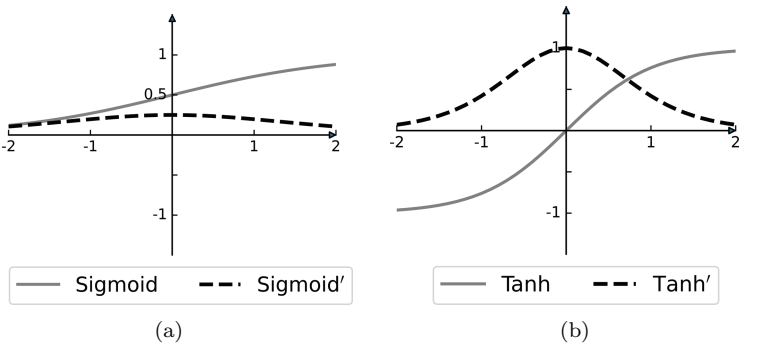


Figure 13.3: Illustration of the Sigmoid and Tanh activation functions.

Its mathematical form and derivative can be written as

$$\begin{aligned} f_{\text{sig}}(x) &= \frac{1}{1+e^{-x}}, \\ \frac{df_{\text{sig}}(x)}{dx} &= f_{\text{sig}}(x)\big(1 - f_{\text{sig}}(x)\big). \end{aligned} \tag{13.1}$$

As we can see, the derivation of the Sigmoid function is very easy to compute and its output range lies in $(0, 1)$. As Figure 13.3(a) shows, the output range of Sigmoid is not zero-centered, which might slow down the training progress [18].

The Hyperbolic Tangent activation function, on the other hand, stretches the output range of the Sigmoid function and makes it zero-centered. The Tanh function and its derivative are given by

$$\begin{aligned} f_{\text{tanh}}(x) &= \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}, \\ \frac{df_{\text{tanh}}(x)}{dx} &= 1 - f^{2}_{\text{tanh}}(x). \end{aligned} \tag{13.2}$$

As Figure 13.3 show, the gradient of Tanh is steeper than that of Sigmoid, so deep neural networks with Tanh activation functions usually converge faster in training than when using Sigmoid activation.

A common disadvantage of Sigmoid and Tanh is that they both suffer from the vanishing gradient issue. That is, after several training epochs, the gradient will be very close to zero, preventing the effective update of the model parameters in backpropagation.

The most popular activation functions adopted by deep learning are Rectified Linear Unit (ReLU) and its variants (visualized in Figure 13.4). ReLU is extremely

simple and computational-friendly and can be written as

$$\begin{aligned} f_{\mathrm{ReLU}}(x) &= \max(0, x), \\ \frac{df_{\mathrm{ReLU}}(x)}{dx} &= \begin{cases} 1, & x \geq 0, \\ 0, & x < 0. \end{cases} \end{aligned} \tag{13.3}$$

ReLU follows the biological features of neuron cells, which only fire if the inputs exceed a certain threshold. Note that ReLU is non-differentiable at zero, and all positive inputs have the same non-zero derivative, one, which could prevent the vanishing gradient problem in backpropagation.

The drawback of ReLU activation is known as dead ReLU problem, i.e., some neurons may never activate and only output zero. One way to address this issue is to give a small slope to negative inputs, which leads to parametric ReLU activation, denoted by P-ReLU. Its mathematical form is

$$\begin{aligned} f_{\mathrm{P-ReLU}}(x) &= \max(ax, x), \\ \frac{df_{\mathrm{P-ReLU}}(x)}{dx} &= \begin{cases} 1, & x \geq 0, \\ a, & x < 0. \end{cases} \end{aligned} \tag{13.4}$$

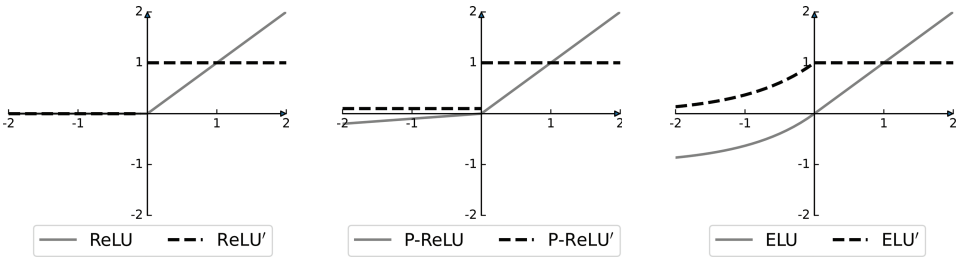
where a is a pre-defined small parameter, e.g., 0.01, or a trainable parameter. Theoretically, P-ReLU inherits ReLU's benefits and does not have the dead ReLU problem. But there is no substantial evidence that P-ReLU is always better than ReLU.

Another variant of ReLU is called Exponential Linear Units (ELU) and its mathematical form is

$$\begin{aligned} f_{\mathrm{ELU}}(x) &= \begin{cases} x, & x \geq 0, \\ a(e^x - 1), & x < 0, \end{cases} \\ \frac{df_{\mathrm{ELU}}(x)}{dx} &= \begin{cases} 1, & x \geq 0, \\ f_{\mathrm{ELU}}(x) + a, & x < 0. \end{cases} \end{aligned} \tag{13.5}$$

Compared to ReLU, ELU can utilize negative inputs and is almost zero-centered. Although empirical studies show that ELU can marginally improve the performance of deep neural networks on classification tasks, it is still unconfirmed whether it can fully surpass ReLU. Figure 13.4 shows the visualization of ReLU, P-ReLU and ELU.

Multiple hidden layers with nonlinear activation functions forge the core idea of deep neural networks. Deep learning utilizes various types of layers to extract features from data automatically. In the next section, we introduce convolutional neural networks, one of the deep learning structures that have achieved human-level performance on many image classification tasks [1].



(a) (b) (c)

Figure 13.4: Illustration of different ReLU activation functions.

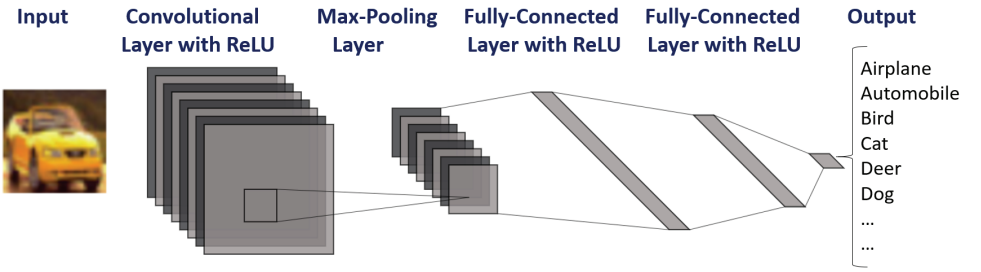


Figure 13.5: An example of convolutional neural network, comprising a convolutional layer with ReLU activation, a max-pooling layer, and two fully-connected layers with ReLU activation.

13.2.2. *Structures of Convolutional Neural Networks*

Convolutional neural networks (CNNs) are widely applied on many computer vision tasks including image classification, semantic segmentation, and object detection [19]. The connectivity patterns between hidden neurons in CNNs were actually inspired by the animal visual cortex [20–22]. Such connections based on shared weights allow CNNs to encode visual patterns and empower the network to succeed in image-focused tasks while reducing the learnable parameters in CNNs.

Typically, three types of layers are widely adopted by CNNs, i.e., convolutional layers, pooling layers, and fully connected layers. Figure 13.5 shows a simple CNN architecture for MNIST classification [23]. There are four basic components in this DNN architecture.

- **Input Layer**: It is the layer that takes in input data such as images.
- **Convolutional Layer**: It is the layer that adopts convolutional kernels to transform the image data into features in different levels. Convolutional layers also utilize activation functions, such as Sigmoid and ReLU, which are used to produce the output of each convolutional kernel.

- **Pooling Layer**: It is the layer that essentially performs a downsampling operation on its input, with an aim to reduce the parameter number in the network. The pooling operation is normally max pooling or average pooling.
- **Fully Connected Layer**: It is the layer that is the same as standard neural networks, with an aim to make the final classification or prediction.

The core idea of CNNs is to utilize a series of layer-by-layer transformations, such as convolution and downsampling, to learn the representations of the input data for various classification or regression tasks. Now we explain the convolutional layer and pooling layer, detailing their structures and operations.

13.2.2.1. *Convolutional layer*

The power of the convolutional layer lies in that it can capture the *spatial* and *temporal* dependencies on the input data (e.g., images) by convolutional kernels. A kernel usually has small dimensions (e.g., 3×3, 5×5) and is applied to convolve the entire input across the spatial dimensionality. As shown in Figure 13.6, the output of a convolutional layer on a single kernel is usually a 2D feature map. Since every convolutional kernel will produce a feature map, all feature maps are finally stacked together to form the full output of the convolutional layer.

There are three key hyper-parameters that decide the convolutional operation, namely, *depth*, *stride*, and *zero-padding*.

- **Depth** describes the 3rd (e.g., for gray images) or 4th dimension (e.g., for RGB images) of the output of the convolutional layer. It essentially measures the number of kernels applied in this convolutional layer. As a result, a small depth can significantly reduce the hidden neuron numbers, which could also compromise the representation learning capability of the neural network.

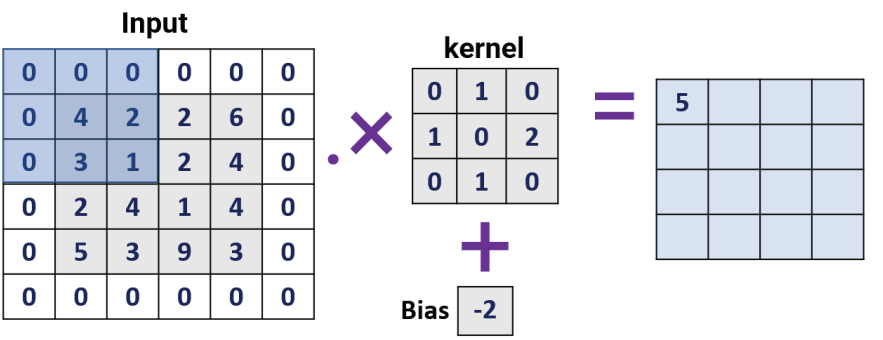


Figure 13.6: An example of convolutional operation: after the zero-padding, the kernel and the receptive field will first of all go through an element-wise multiplication, then be summed up and plus the bias. The kernel will traverse the whole input to produce all the elements in output matrix.

- **Stride** is the size of the step the convolutional kernel moves each time on the input data. A stride size is usually 1, meaning the kernel slides one pixel by one pixel. On increasing the stride size, the kernel slides over the input with a larger interval and thus has less overlap between the receptive fields.
- **Zero-padding** means that zero-value is added to surround the input with zeros, so that the feature map will not shrink. In addition to keeping the spatial sizes constant after convolution, padding also improves the performance by preserving part information of the border.

It is important to understand the hyper-parameters in convolutional layers. To visualize the convolutional operation and the impact of depth, stride, and zero-padding, you could refer to the website https://cs231n.github.io/convolutional-networks/#conv, which provides a detailed explanation and a few useful visualizations.

13.2.2.2. *Pooling layer*

The addition of a pooling layer after the convolutional layer is a common pattern within a convolutional neural network that may be repeated one or more times. The pooling operation involves sliding a two-dimensional filter over each channel of the feature map and summarizing the features lying within the region covered by the filter. Pooling layers can reduce the dimensions of the feature maps. Thus, it reduces the learnable parameter number in the network. The pooling layer also summarizes the features present in a region of the feature map generated by a convolution layer. So, further operations are performed on summarized features instead of precisely positioned features generated by the convolution layer. This makes the model more robust to variations in the position of the features in the input image. Typically, max pooling and average pooling are widely used in CNNs.

- Max pooling is a pooling operation that selects the maximum element from the region of the feature map covered by the filter. Thus, the output after the max-pooling layer would be a feature map containing the most prominent features of the previous feature map.
- Average pooling computes the average of the elements from the region of the feature map covered by the filter. Thus, average pooling gives the average of the features in a patch.

13.2.2.3. *Stacking together*

Finally, we can stack convolutional layers and pooling layers by repeating them multiple times, followed by one or a few fully connected layers. We can build a typical CNN architecture, as shown by Figure 13.5 and 13.7. We can see that,

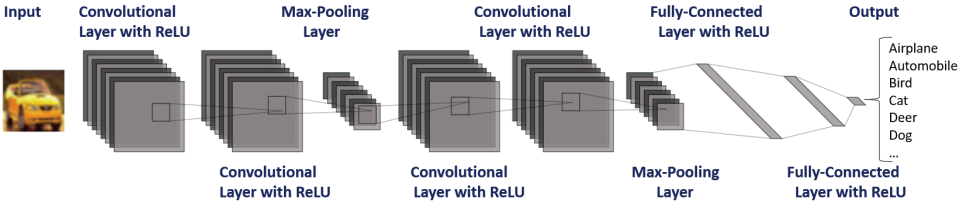


Figure 13.7: An example of complicated convolutional neural network: it contains multiple convolutional layers and max-pooling layers.

compared to a traditional neural network that only contains the fully connected layers, adopting multiple convolutional and pooling layers enables the network to learn complex visual features and further boost the classification performance.

13.2.2.4. *Summary*

We introduced the basic structures of convolutional neural networks in this section. Since AlexNet achieved a huge success in the ImageNet competition, CNNs are predominantly applied in the computer vision area, and significantly improve the state-of-the-art performances on many computer vision tasks. Meanwhile, the development of CNNs has made deep learning models more practical in solving real-world problems. For example, face recognition has been integrated into smartphones to identify their owners and secure sensitive operations, including identity verification and payment. In addition to vision tasks, convolution layers serve as a fundamental building block that can automatically extract local features. So they are essential components in a wide range of deep learning models, such as WaveNet [24] for speech synthesis and TIMAM [25] for text-to-image matching tasks.

13.2.3. *Structures of Recurrent Neural Networks*

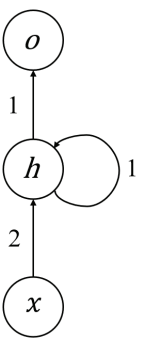
Recurrent neural networks (RNNs) refer to neural networks with loops and gates, specifying sequential processing. With further in-depth research on neural networks, scientists found out that although feed-forward neural networks, such as Multilayer Perceptron (MLP) and CNNs, can handle large grades of values well, they are insufficient for processing sequential data. When modeling human neuronal connectivity, scientists observed that human brain neurons are not one-way connected. Instead, they are connected in a dense web, namely a recurrent network. Inspired by such observations, the notion of RNNs was first built by David Everett Rumelhart in 1986 [11]. In the same year, Michael I. Jordan built a simple RNN structure, a Jordan network, which directly uses the output of the network for feedback [26]. In 1990, Jeffrey L. Elman made an improvement in the structure by

feedback of the output of the internal state, i.e., the output of hidden layers [27]. He also trained the network through backpropagation, which generated the RNN model we use today. However, these models face problems such as gradient vanishing and gradient exploding, when the networks and data volumes are large. Such problems were not alleviated until the invention of long short-term memory (LSTM) networks by Hochreiter and Schmidhuber in 1997 [28]. The LSTM structure including gates and loops, allowing for the processing of the entire sequential data, has a wide application in speech recognition, natural language procession, etc. Based on the LSTM model, Kyunghyun Cho et al. developed gated recurrent units (GRU) [29], which bear resemblance to simplified LSTM units. There are also other RNN techniques, such as bi-directional RNNs, which show a good performances when combined with LSTM or GRU. In this section, we introduce the basic structure of RNNs, analyze their working principles by unrolling the loop, and illustrate the unique structure of LSTM and GRU. We also demonstrate the specific applications of RNNs based on its structures and discuss the challenges of this technique.

13.2.3.1. *Vanilla RNN and unfolding*

Let's start by introducing a vanilla RNN, which has only one input node x, one hidden unit h, and one output node o, with only one loop. As shown in Figure 13.8, where the weights are listed besides the connection, we initialize the hidden state at the very beginning ($t = 0$) to 0 and implement no bias. Given an input sequence $x = 0.5, 1, -1, -2$ at time steps 1 to 4, the values of the hidden units and outputs are as shown in the table in Figure 13.8.

Assuming that the input sequence of a RNN is $x = x_1, \ldots, x_n$, then the output of the RNN is $o = o_1, \ldots, o_n$, with a hidden state $h = h_0, \ldots, h_n$, we present a



Time Step	x	h	o
1	0.5	1	1
2	1	3	3
3	-1	1	1
4	-2	0	0

Figure 13.8: Simple structure of RNN with three layers and ReLU activation.

more general form of RNN as follows:

$$\begin{aligned} h_t &= \delta(Ux_t + Wh_{t-1} + b_h), \\ o_t &= \phi(Vh_t + b_y), \end{aligned} \tag{13.6}$$

where $1 < i < n$, and δ and ϕ are the activation functions in nodes h and o.

Nowadays, most of the analyses of RNNs tend to open the loop structure through an unfolding process. A RNN with a fixed input length can be transformed into an equivalent feed-forward network, which not only provides a more straightforward observation but also makes techniques designed for feed-forward networks suitable for RNNs. Figure 13.9 illustrates how a simple RNN unit is transformed into a feed-forward model. This equivalent feed-forward model shares the parameters with the original RNN, while the number of layers increases to the input sequence length. However, for RNNs only with the loop structure, the information at the very beginning may have little influence on the final output when the sequence is too long. That is when the long short-term memory (LSTM) structure takes place, which is proposed to address such problems and performs well when processing long sequential data.

13.2.3.2. *Long short-term memory*

Unlike standard RNNs only with loops, the LSTM network has its own specific structure LSTM cell. In the LSTM unit shown in Figure 13.10, the data flow is controlled by an input gate i_t, an output gate o_t, and a forget gate f_t. These gates regulate not only the input and output information between cells but also the cell states themselves. They are realized by the combination of linear transformation and activation functions, such as Sigmoid δ and Tanh.

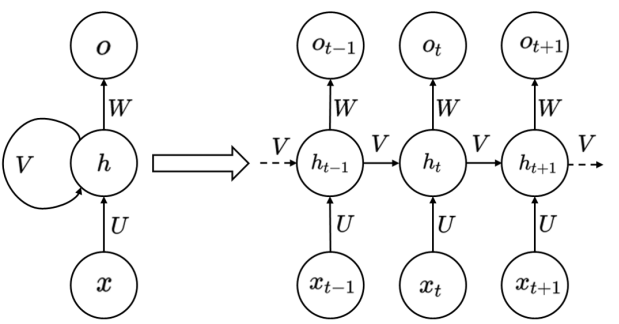


Figure 13.9: Transforming a vanilla RNN into a feed-forward model.

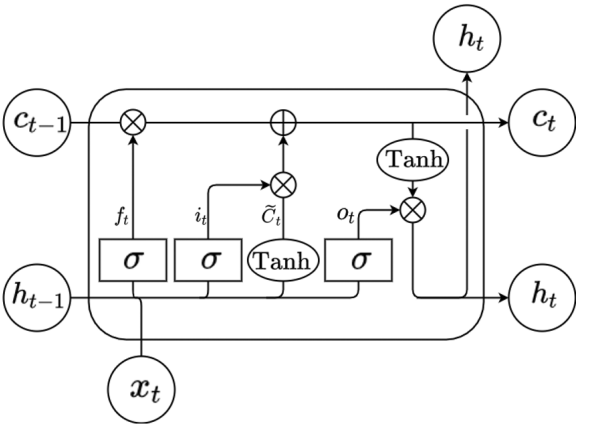


Figure 13.10: An LSTM unit.

The mathematical form of an LSTM unit can be formulated as

$$\begin{aligned} f_t &= \delta(W_f x_t + U_f h_{t-1} + b_f), \\ i_t &= \delta(W_i x_t + U_i h_{t-1} + b_i), \\ o_t &= \delta(W_o x_t + U_o h_{t-1} + b_o), \\ \widetilde{c_t} &= \tanh(W_c x_t + U c h_{t-1} + b_c), \\ c_t &= f_t \circ c_{t-1} + i_t \circ \widetilde{c_t}, \\ h_t &= o_t \circ \tanh(c_t), \end{aligned} \tag{13.7}$$

where x, h, c, and t represent input, hidden state, cell state, and time point, respectively. Both W and U indicate the weight matrix. The forget gate f_t transforms input x_t and hidden state h_{t-1}. It selects useful information in c_{t-1} and abandons the rest. The input gate i_t controls what values should be added to the cell state, and the vector $\widetilde{c_t}$ provides all the possible values for addition. i_t and $\widetilde{c_t}$ work together to update the cell state. When we have the new cell state c_t, we use tanh to scale its value to the range -1 to 1. The output gate o_t function plays the role of a filter, which selects the information from the scaled cell state and passes it to the hidden state h_t that will soon be output.

Given the specific cell and gate structure, we do not need to moderate the whole sequence of information. Through the selecting and forgetting procedure, we can moderate only a part of the data, which improves the accuracy and efficiency of processing long sequence data. There are also different variants of LSTM. For example, LSTM with peephole connections, where gates can directly access cell state information, was quite popular. In the next part, we discuss a simplified version of LSTM, i.e., the gated recurrent unit, which is more commonly used at present.

13.2.3.3. *Gated recurrent unit*

Compared with LSTM, the gated recurrent unit (GRU) contains only two gates, i.e., the update gate and the reset gate, where the output gate is omitted. This structure was first presented by Kyunghyun Cho et al. in 2014 to deal with the gradient vanishing problem when processing long sequential data via fewer gates. The structure of GRU is shown in Figure 13.11. Unlike LSTM with the cell state, GRU has only hidden state h_t for transmission between units and output delivery. Therefore, a GRU, has fewer parameters to learn during the training process and shows better performances on smaller datasets. Equation (13.8) presents the mathematical form of GRU, in which x_t represents the input data. Both W and U are weight matrices and b indicates the bias matrix. To generate the gate z_t, we first make a linear transformation of both input x_t and hidden state h_{t-1} from the last time point, and then apply a nonlinear transformation by the sigmoid function. The resulting value z_t helps decide which data to pass to next time steps. The reset gate r_t has a structure similar to that of update gate, only with different weight W_r, U_r and bias b_r parameters. The reset gate selects useful information from the hidden state h_{t-1} at the last time point and forgets the irrelevant information. Combined with a linear transformation and tanh activation, it delivers the candidate hidden state $\widetilde{h}_t$, which may also be referred to as current memory content, where useful information from the past is kept. The last step is to update the hidden state and gives out h_t. We use update gate z_t to take element-wise products with both h_{t-1} and $\widetilde{h}_t$ and deliver the new hidden state h_t, which can be written as

$$\begin{aligned} z_t &= \delta(W_z x_t + U_z h_{t-1} + b_z), \\ r_t &= \delta(W_r x_t + U_r h_{t-1} + b_r), \\ \widetilde{h}_t &= \tanh(W_h x_t + U_h(r_t \circ h_{t-1}) + b_h), \\ h_t &= (1 - z_t) \circ h_{t-1} + z_t \circ \widetilde{h}_t. \end{aligned} \tag{13.8}$$

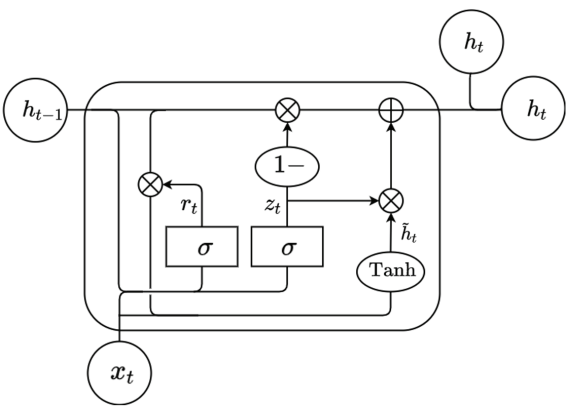


Figure 13.11: A gated recurrent unit.

The GRU model is quite popular in polyphonic music modeling, speech signal modeling, handwriting recognition, etc. Because of the relatively few parameters, it is more efficient in the training process than LSTM, especially when dealing with smaller datasets. However, it still shows limitations in machine translation and language recognition scenarios and may not wholly outperform LSTM.

13.2.3.4. *Application in natural language processing*

Unlike CNNs that commonly appear in computer vision tasks, the specific loop structures and memory function of RNNs lead to their different application scenarios. The previous input of RNNs influences the current input and output. Therefore, RNNs are widely used in processing sequential data and solving temporal problems, where many natural language processing (NLP) tasks fall into this area, e.g., machine translation, named-entity recognition, sentiment analysis, etc. Bengio et al. [30] made the first attempt at applying deep learning in NLP tasks in 2003, and Collobert et al. demonstrated that multiple-layer neural networks may be a better solution compared to traditional statistical NLP features. While the real revolution started with word embedding in 2013, the word2vec toolbox [31] showed that pre-trained word vectors can improve model performance on most downstream tasks. At that time, word embedding could not handle polysemy of human language properly, until Peters et al. [3] proposed a deep contextualized word representation, also known as embedding from language language models (ELMO), which provides an elegant solution to the polysemous problems. Note that ELMO uses BiLSTM, or bidirectional LSTM, as the feature extractor [32]. Right now, it has been demonstrated by many studies that the ability of the Transformer to extract features is far stronger than of the LSTM. ELMO and Transformer had a profound impact on subsequent research and inspired a large number of studies, such as GPT [33] and BERT [34].

13.2.3.5. *Summary*

In this section, we first introduce the basic concept of recurrent neural networks and the working principle of vanilla RNNs and then present two representative models, LSTM and GRU, which are capable of handling long sequential data. Moreover, the gate structure allows modifying only small parts of the data during long sequential data processing, avoiding the gradient vanishing situation. Apart from these two variants, there are still various models of RNN for different application scenarios, such as the bidirectional RNN, deep RNN, continuous-time recurrent neural network, and so on. Although classic RNNs are no longer the best option in NLP, various structures of RNN still enable its used in a wide range of sequence processing applications.

13.3. Training Deep Learning Models

A great number of parameters enable the fantastic representation capability of deep neural networks, but the difficulty of training neural networks grows exponentially with their scale. In this section, we briefly discuss the development of training methods and tricks.

13.3.1. *Gradient Descent and Backpropagation*

From an optimization perspective, training a network f_θ is normally formulated as a minimization problem,

$$\min_\theta L(\theta), \tag{13.9}$$

where we wish to minimize the empirical risk that is given by the expectation of a given loss function over all examples that draw i.i.d. from a distribution $\mathcal{D}$, which can be written as

$$L_{\mathcal{D}}(\theta) = \mathbb{E}_{(x,y)\sim\mathcal{D}}\ell\left(f_\theta(x), y\right). \tag{13.10}$$

However, in practice, we can only access a finite training set $S = \{(x_i, y_i)\}_{i=1}^n$, so the corresponding empirical risk or training loss is written as

$$L_S(\theta) = \frac{1}{n}\sum_{i=1}^{n}\ell\left(f_\theta\left(x_i\right), y_i\right). \tag{13.11}$$

Assume that ℓ is continuous and differentiable. The landscape of the value of ℓ is just like a mountain area that exists in a high-dimensional space, where dimension depends on the number of parameters in the trained model. Our task can be viewed as searching the lowest valley inside mountains. Imagine that we start at a random location in the mountains. All we can do is to look locally and climb down the hill step by step until we reach a local minimum point. This iterative process is achieved by gradient descent (GD), which can be written as

$$\theta^{k+1} = \theta^k - \eta^k \nabla L_S(\theta^k), \tag{13.12}$$

where $\nabla L_S(\theta)$ is the gradient of θ and shows the direction to climb down, and η^i is the step size, which is also known as the learning rate. When we use GD to train a deep learning model, the gradient of weights $\nabla L_S(\theta)$ is computed from the last layer to the first layer via the chain rule. Therefore, the model's weights are also reversely updated layer by layer, and this process is known as backpropagation. As one of the landmark findings in supervised learning, backpropagation is a foundation of deep learning.

13.3.2. *Stochastic Gradient Descent and Mini-Batching*

Note that in GD, we only conduct backpropagation after the model sees the whole training set, which is feasible yet not efficient. In other words, GD heads to the steepest direction at each step, while we only need to climb down in roughly the right direction. Following this intuitives, stochastic gradient descent (SGD) takes the gradient of the loss for each single training example as an estimation of the actual gradient and conducts gradient descent via backpropagation based on the stochastic estimation, as shown in Figure 13.12(b). By doing so, SGD directly speeds up GD from two perspectives. First, there might be repeated examples in the training set. For instance, we can copy an example ten times to build a training set and train a model on it, where GD is ten times slower than SGD. Besides, even though there are no redundant examples in the training set, SGD does not require the trained model to go over all examples and updates weights much more frequently. Although the stochastic estimation of the actual gradient introduces random noise into gradient descent, the training actually benefits from the noise. At the early training stage, the model has barely learned any information from the training set. Therefore, the noise is relatively small, and an SGD step is basically as good as a GD step. Furthermore, the noise in SGD steps can prevent convergence to a bad local minimum point in the rest stages. On the downside, SGD updates weights for every example, and sometimes the training effect from two extremely different examples may even cancel out. Hence, it takes more steps for SGD to converge at an ideal point than GD and cannot make the best use of parallel computation brought by the advanced graph process unit (GPU). A promising way to address these drawbacks is by calculating the estimation over a subset B of the training set, namely mini-batch. An intuitive way to partition the training set into mini-batches is to randomly shuffle the training set S and divide it into several parts as uniformly as possible. Let $S = \{B_1, B_2, \ldots, B_m\}$ represent the partition result. For any $B_j \in S$, the corresponding loss is written as

$$L_B(\theta) = \frac{1}{|B_j|} \sum_{i=1}^{|B_j|} \ell\left(f_\theta\left(x_i\right), y_i\right), \tag{13.13}$$

where $|\cdot|$ returns the size of the input batch, namely batch size. Although the estimation is calculated on multiple examples, the time consumption is usually similar to a regular SGD step when employing a GPU with enough memory. By doing so, the frequency of backpropagation has been significantly reduced, and the training effect from any two mini-batches can hardly cancel each other out. Figure 13.12 shows the differences of GD, SGD and SGD with mini-batching.

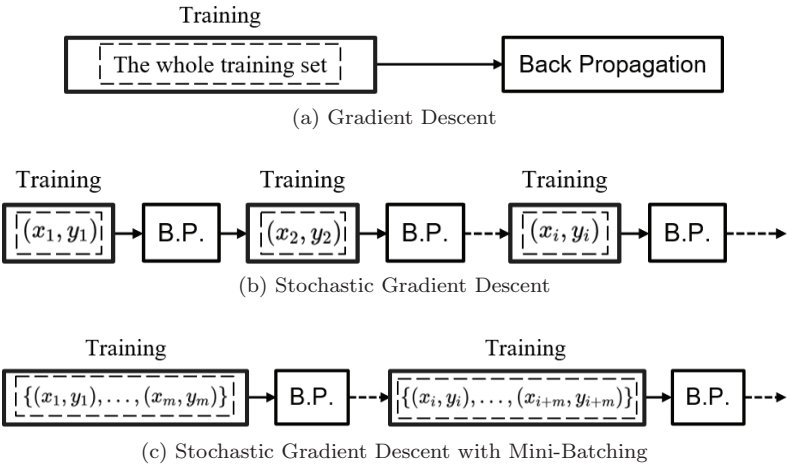


Figure 13.12: An illustration of the pipelines of GD, SGD, and SGD with mini-batching.

13.3.3. *Momentum and Adaptive Methods*

Although SGD with mini-batching outperforms regular SGD when training deep learning models under most scenarios, adding a momentum item can make it even better. The idea of this improvement is similar to the concept of momentum in physics. If SGD is a man trying to climb down a hill step by step, SGD with momentum can be imagined as a ball rolling in the same area. One mathematical form of momentum can be written as

$$\begin{aligned} p^0 &= 0, \\ p^{k+1} &= \beta p^k + \nabla L(\theta^k), \quad 0 \leq \beta \leq 1, \\ \theta^{k+1} &= \theta^k - \eta^k p^{k+1}. \end{aligned} \tag{13.14}$$

The momentum item can be viewed as a weighted average of current gradient and the momentum item from the previous step, where β is the weight factor. If we unroll (13.14) at any specific step, it can be seen that the gradient items of previous steps have accumulated at the current momentum item, where the old gradient term decays exponentially according to β, and the later gradient has a larger impact at present. This exponential decay is known as the exponential moving average, which we will see again later. Goh [35] provided a vivid visualization about the impact of momentum, where SGD with momentum makes modest moves at each step and finds a better solution than SGD eventually. The insight behind the improvement is that the historical gradient information can smooth the noise in SGD steps and dampen the oscillations.

Except for momentum, another intuitive way to enhance SGD is by adding adaptivity into the training process. In the above methods, we assume that all

parameters in the trained model are updated via a global learning rate, while adaptive methods aim to adjust the learning rate for each parameter automatically during training. The earliest attempt in this research direction is the adaptive gradient (AdaGrad) method. AdaGrad starts training with SGD at a global learning rate. For a specific parameter, AdaGrad decreases the corresponding learning rate according to the parameter's gradient value that is calculated via backpropagation. The update process can be written as

$$\begin{aligned} s^0 &= 0, \\ s^{k+1} &= s^k + \nabla L(\theta^k)^2, \\ \theta^{k+1} &= \theta^k - \frac{\eta^k}{\sqrt{s^{k+1}} + \epsilon} \nabla L(\theta^k), \end{aligned} \tag{13.15}$$

where ϵ is added to avoid dividing zero. The learning rate of a specific parameter will be enlarged if the corresponding item in s^{k+1} is smaller than 1, otherwise it will be shrunk. Note that in AdaGrad, all previous gradient information is treated equally, so most parameters' learning rates decrease during training due to the accumulation of historical gradient. One problem with this adapting strategy is that the training effect of AdaGrad at the later training stage can hardly be maintained because of the small learning rates. The root mean square propagation (RMSProp) method addresses this by taking a weighted average of the historical accumulation item and the square of current gradient, which can be formalized as

$$\begin{aligned} v^0 &= 0, \\ v^{k+1} &= m v^k + (1-m) \nabla L(\theta^k)^2, \\ \theta^{k+1} &= \theta^k - \frac{\eta^k}{\sqrt{v^{k+1}} + \epsilon} \nabla L(\theta^k), \end{aligned} \tag{13.16}$$

where $m \in [0, 1)$ is introduced to achieve an exponential moving average of past gradients. Compared to AdaGrad, the newer gradient has larger weights in RMSProp, which prevent the learning rate from continuously decreasing over training. The Adam method combines the RMSProp and momentum together and is the most widely used method at present. An Adam step requires calculating the exponential moving average of momentum and the square of past gradient, which are given by

$$\begin{aligned} v^0 = 0, \quad p^0 &= 0, \\ p^{k+1} &= \beta_1 p^k + (1-\beta_1) \nabla L(\theta^k), \\ v^{k+1} &= \beta_2 v^k + (1-\beta_2) \nabla L(\theta^k)^2. \end{aligned} \tag{13.17}$$

Then the parameters is updated via

$$\theta^{k+1} = \theta^k - \eta^k \frac{p^{k+1}}{\sqrt{v^{k+1}} + \epsilon}. \tag{13.18}$$

Although Adam normally performs better than AdaGRrad and RMSProp in practice, it has two momentum parameters that need to be tuned and uses more memory to conduct backpropagation. On the other hand, our understanding of its theory is still poor. From the practical perspective, Adam is necessary for training some networks for NLP tasks, while both SGD with momentum and Adam generally work well on training CNN models. However, using Adam to minimize the training loss may lead to worse generalization errors than SGD, especially in image problems.

13.3.4. *Challenges in Training*

Except for the high demand for computational resources, training deep neural networks is facing overfitting and gradient vanishing. Overfitting is well studied in the machine learning area, which normally occurs when the trained model is too complex or the training set is not large enough. However, in the scope of deep learning, because we do not really understand why deep learning works, there are no methods or mathematical theorems that can tell us how to quantify a specific model's capability and how many parameters and layers a model needs to handle a specific task. But existing studies show that increasing the model's size in a proper way is the best most likely way to increase the model's performance. Therefore, neural network's complexity keeps growing. In the previous section, we describe how neural networks are trained in supervised learning by default. Under this scenarios, we need a great amount of labeled data to train our models. However, generating such data is heavily dependent on human participation, which is much slower than the evolution of neural network's architecture. Thus, overfitting becomes an inevitable problem, and all we can do is to mitigate it.

As mentioned in Section 13.2.1 that multiple nesting of Sigmoid and Tanh may make the gradient close to zero and cause the gradient vanishing. Before ReLU came out, Hinton et al. [36] proposed an unsupervised layer-wise training approach, which only trains one layer at a time to avoid gradient vanishing. For a given hidden layer, this method takes the output of the previous layer as input to conduct unsupervised training and repeat the same process on the next layer. After all the layers are pre-trained, it is time for gradient descent to fine-tune the model's weights. In fact, this pre-training with the fine-tuning approach first partitions the model's weights and then finds a local optimal setting for each group. The global training starts in the combination of these pre-trained weights, which makes the training process much more efficient. Due to the advancement of computer hardware, this method is rarely

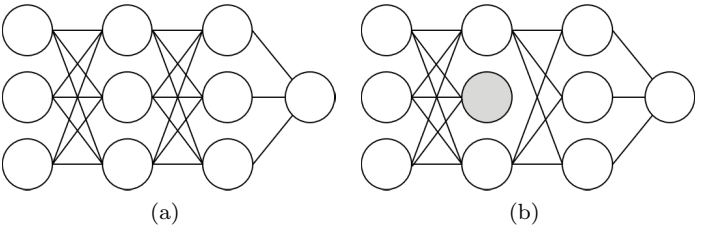


Figure 13.13: Illustration of applying dropout in a multiple-layer perceptron.

used in training small and medium size neural networks now. But this paradigm is still very useful when training very large language models.

Dropout is an interesting approach to prevent overfitting. Suppose we have a multiple-layer perceptron as shown in Figure 13.13(a). Applying dropout during training means that hidden units are randomly discarded with a given probability when performing forward propagation. The discarded unit in Figure 13.13(b) simply outputs zero no matter what its inputs are and remains unchanged at that iteration. Because every hidden unit might be deactivated, dropout introduces random noise into the output of hidden layers. Correspondingly, units in the following layer are forced to find useful patterns and make predictions independently because any specific dimension of their inputs is not reliable. Note that dropout is only applied in the training process, and a sub-model that inherits parameters from the original model is sampled at each training iteration. While at the inference procedure, all these trained sub-models are integrated together to give predictions, which implies the idea of ensemble learning.

Batch normalization is one of the recent milestone works in deep learning that can mitigate both gradient vanishing and overfitting. It applies as the so-called batch normalization layer and plays an important role in improving the training and convergence speed. Neural networks are supposed to learn the distribution of the training data. If the distribution keeps changing, learning will be difficult. Unfortunately, for a specific hidden layer, the distribution of its previous layer's output does change if the activation function is not zero-centered after each weight update. Ioffe et al. [37] used the term *internal covariate shift* to describe this phenomenon.

A straightforward idea to address the internal covariate shift is by normalizing the output of each layer of the neural network. Suppose that we normalize those outputs to 0 mean, 1 variance, which satisfies the normal distribution. However, when the data distribution of each layer is the standard normal distribution, a neural network is completely unable to learn any useful features of the training data. Therefore, it is obviously unreasonable to directly normalize each layer.

To address this, batch normalization introduces two trainable parameters to translate and scale the normalized results to retain information. Generally, the translate and scale parameters are calculated on the training set and directly used to process data at testing. Although we still do not understand why batch normalization is helpful, including it in our neural networks allows training to be more radical. Specifically, batch normalization reduces sensitivity to weight initialization and makes the networks easier to optimize, where we can use larger learning rates to speed up the training process. On the other hand, the translate and scale parameters can be viewed as extra noise because each mini-batch is randomly sampled. Like in dropout and SGD, extra noise enables better generalization and prevents overfitting. However, there might be a difference in the distribution of training data and test data, and the pre-calculated normalization cannot suit the test data perfectly, which leads to inconsistency in the model's performance at training and testing.

As shown in Figure 13.14, batch normalization works on the batch dimension. We can naturally conduct normalization along different dimensions, which motivates layer, instance, and group normalization methods. The first two methods are straightforward, while for group normalization it is relatively hard to get the intuition behind it. Recall that there are many convolution kernels in each layer of a CNN model. The features learned by these kernels are not completely independent. Some of them should have the same distribution and thus can be grouped. In the visual system, there are many physical factors that can cause grouping, such as frequency, shape, brightness, texture, etc. Similarly, in the biological visual system, the responses of neuron cells would also be normalized.

So far, we have only briefly described a few tricks and methods that can make training neural networks easier. Deep learning is a very active field, and there are scores of well-developed methods we have not mentioned. If readers are interested in more, they are advised to further study the existing literature according to specific needs.

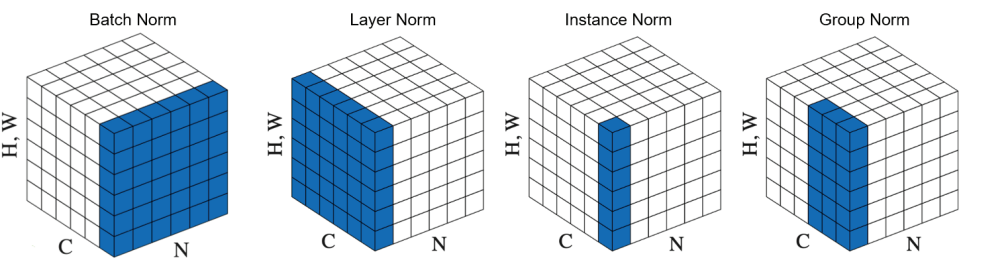


Figure 13.14: Illustration of different types of normalization layers [38]. C, N, and H, W are shorthands for channels, instances in a batch, and spatial location.

13.3.5. *Frameworks*

At the end of this section, we want to give a quick review of deep learning frameworks that greatly boost the development of this area. Although deep learning benefits a lot from advanced GPUs and other hardware, it is not easy for most users and researchers who are not majoring in computer science to make use of such improvements. Deep learning frameworks enable people to implement and customize their neural networks in their projects. Theano is the first framework that focuses on deep learning study. It was developed by Mila, headed by Yoshua Bengio, back in 2008. Caffe is also an early open-source deep learning framework, which was started by Yangqing Jia during his PhD at UC Berkeley. About roughly the same period, Microsoft Research Asia presented a framework called Minerva, which may be the first one that introduced the concept of the computational graph into deep learning implementation. In a computational graph, all components, e.g., input examples, model weights, and even operations, are traded as graph nodes. During the forward propagation, the data flow from the input layer to the output layer, where computation is done at each graph node. Meanwhile, backpropagation can directly utilize the useful intermediate results kept by the computational graph to avoid repeated calculations when updating model weights. Published by Google Brain, TensorFlow further develops the computational graph and is now the most popular framework in the industry. It includes most deep learning algorithm models and is supported by an active open source community. Keras is an advanced framework on top of other frameworks. It provides a unified API to utilize different frameworks and is probably the best option for new programmers. The latest version of TensorFlow also adopts Keras's API to simplify its usage and is now more friendly for beginners. Meanwhile, in the research area, Pytorch is also a popular option. This framework originated from Torch, a Lua-based machine learning library, and was improved by Facebook. Pytorch dynamically builds the corresponding computational graph when performing deep learning tasks, allowing users to customize models and verify their ideas easily. There are many more deep learning frameworks available now, such as MXNet, PaddlePaddle, MegEngine, Mindspore, and so on, which benefit a wide range of users. Meanwhile, it can be seen on GitHub that a growing number of third-party libraries have been developed to enhance these frameworks. Overall, the rapid development of deep learning is inseparable from these open-source frameworks. There is no such thing as the best deep learning frameworks, and we should appreciate those developers, engineers, and scientists who have contributed to such projects.

13.4. The Robustness of Deep Learning

Given the prospect of broad deployment of DNNs in a wide range of applications, it becomes critical that a deep learning model obtain satisfactory yet trustworthy

performance. Unfortunately, a large number of recent research studies have demonstrated that deep learning models are actually too fragile or extremely non-robust to be applied in high-stakes applications. Such vulnerability has raised concerns about the safety and trustworthiness of modern deep learning systems, especially those on safety-critical tasks. So, how to enhance the robustness of deep learning models to adversarial attacks is an important topic. Recently, although a significant amount of research has emerged to tackle this challenge [39], it remains an open challenge to researchers in computational intelligence. In this section, we first describe the progress in discovering and understanding such a threat and then introduce how to quantify the deep learning model's robustness via verification techniques. The mainstream defense methods will be summarized at the end of this section.

13.4.1. *Adversarial Threat*

Due to the lack of understanding about the basic properties behind these sophisticated deep learning models, adversarial examples, i.e., a special kind of inputs that are indistinguishable from nature examples but can completely fool the model have become the major security concern.

Adversarial attacks, the procedure that generates adversarial examples, are conceptually similar to the so-called evasion attacks in machine learning [40]. The fact that DNNs are vulnerable to such attacks was first identified by Szegedy et al. back in 2013 [41]. They found that well-trained image classifiers can be misled by trivial perturbations δ generated by maximizing the loss function, which can be formulated as

$$\max_{\delta} \ell(f_\theta(x+\delta), y), \text{ s.t. } \|\delta\| \leq \epsilon,$$

where $\|\cdot\|$ is a pre-defined metric and ϵ is the adversarial budget. Later, Goodfellow et al. proposed the fast gradient sign method (FGSM) [42], which is one of the earliest adversarial attack methods. Along this direction, both basic iterative method [43] and projected gradient descent (PGD) [44] conduct multiple FGSM steps to search strong adversarial examples. PGD also takes advantage of random initialization and re-start, so it performs much better in practice and is adopted by recent works as a robustness testing method. Besides, not only is the most popular adversarial attack at present but it also plays an essential role in adversarial training, a promising defense technique that we will discuss later.

Generally, adversarial attacks can be divided into two types depending on the attacker's goals. Non-targeted attacks want to fool a given classifier to produce wrong predictions, and they do not care about what these predictions exactly are.

In classification tasks, the goal of non-targeted attacks can be described as

$$f_\theta(x+\delta) \neq f_\theta(x), \quad \text{where } f_\theta(x) = y.$$

One step further, targeted attacks aim to manipulate classifiers and make them give a specific prediction, i.e.,

$$f_\theta(x+\delta) = y_t, \quad \text{where } f_\theta(x) = y \quad \text{and} \quad y \neq y_t.$$

Conducting targeted attacks is more difficult than non-targeted attacks, and they are more threatening for real-world applications. While non-targeted attacks have a higher success rate, they are commonly used to test the robustness of deep learning models under experimental environments.

On the other hand, adversarial attacks are also classified by the ability of the attackers. FGSM and PGD are both white-box attacks, which assume that adversaries can access all information about their victims, especially the gradients of the model during backpropagation. However, conducting black-box attacks is much more difficult and complex than white-box attacks. In this case, the details of deep learning models are hidden from attackers, but they can pretend to be regular users, which can get the model's predictions with respect to input examples. Based on the query results, attackers can either fine-tune a surrogate model to produce transferable adversarial examples or estimate the gradient information to generate malicious perturbation as in white-box scenarios. Because of the inherent high dimensionality of deep learning tasks, attacks usually need thousands of query results to generate one usable adversarial example [45]. The significant amount of queries required by black-box attacks makes them computationally expensive and hard to achieve in real-world tasks. Thus, making full use of the information obtained in the query process to produce efficient adversarial attacks is still an open problem. In addition to the additive perturbation, conducting small spatial transformations such as translation and rotation can also generate adversarial examples [46]. This indicates that l_p norm is not an ideal metric of adversarial threat. Besides, universal adversarial perturbation, i.e., a single perturbation that works on most input examples simultaneously and causes misclassification, is also an interesting and practical technique. Along this direction, we found that combining spatial and additional perturbations can significantly boost the performance of universal adversarial perturbation [47]. Note that adversarial attacks can be easily generalized to a wide range of deep learning tasks beyond classification. As shown in Figure 13.15, our previous work [48] provides an example of how to attack an object detection system via PGD with a limited number of perturbed pixels.

Deep learning models being deceived by adversarial examples under white-box scenarios is almost inevitable at the current state, and existing defense methods can only mitigate the security issues but cannot address them adequately. The existence

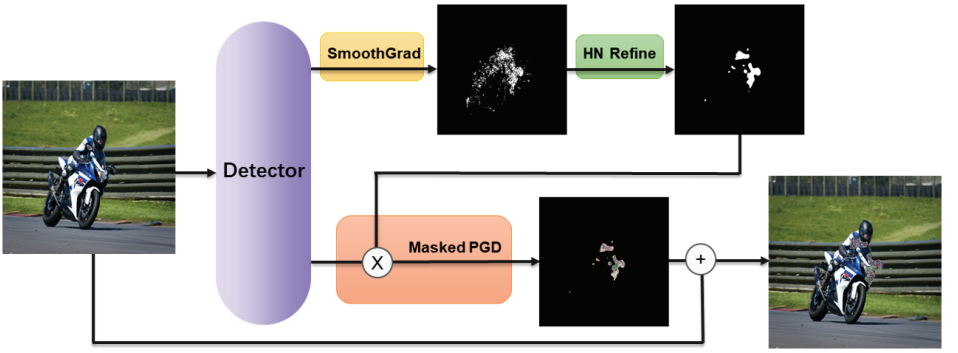


Figure 13.15: Generating an adversarial example toward an object detection system with a limited number of perturbed pixels [48]. Here, SmoothGrad finds the sensitive pixels within an input example, and Half-Neighbor refinement selects the actual perturbed area for PGD attack accordingly.

of adversarial examples demonstrated that small perturbations of the input (normally under l_p metrics) are capable of producing a large distortion at the ending layers [41]. Therefore, in addition to security, enabling adversarial robustness would also help us to better understand the fundamental properties of deep learning and computational intelligence.

13.4.2. *Robustness Verification*

Although DNNs have achieved significant success in many domains, there are serious concerns when applying them to real-world safety-critical systems, such as self-driving cars and medical diagnosis systems. Even though adversarial attacks can evaluate the robustness of DNNs by crafting adversarial examples, these approaches can only falsify robustness claims yet cannot verify them because there is no theoretical guarantee provided on their results. Safety verification can provide the provable guarantee, i.e., when using verification tools to check DNN models, if there is no counterexample (adversarial example), such verification techniques can provide a robustness guarantee. To better explain verification techniques on adversarial robustness, this section explains the relationship between the verification and adversarial threat, then defines verification properties, and further presents various verification techniques.

To verify that the outputs of DNNs are invariant against a small probation, we introduce the maximal safe norm ball for local robustness evaluation, as shown in Figure 13.16. Given a genuine image x_0, we use the l_p norm ball to define the perturbation against the original image, i.e., $\|x - x_0\|_p \leq r$. When the norm ball is small and far away from the decision boundary, DNN gives the invariant output, whereas an intersection with the decision boundary appears on enlarging the norm ball, which means that the perturbation is large enough to change the output.

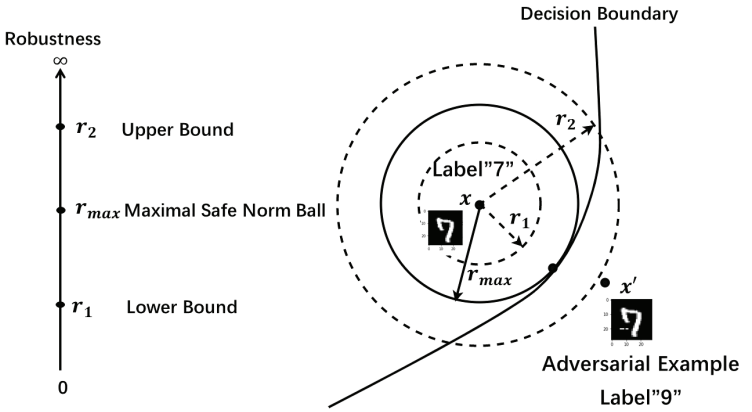


Figure 13.16: Visualizing the decision boundary and different safety bounds.

In this process, a critical situation exists when the norm ball is tangent to the boundary, generating the maximal safe norm ball $r_{\max}$. No matter what kind of adversarial attacks are used, the model is safe and robust as long as the perturbation is within the range of $r_{\max}$. Otherwise, the model is unsafe because there exists at least one adversarial example. Ideally, we only need to determine the maximal safe radius $r_{\max}$ for the robustness analysis. However, in practice, $r_{\max}$ is not easy to calculate, we thus estimate the upper bound r_2 and lower bound r_1 of $r_{\max}$, corresponding to the adversarial threat and robustness verification.

13.4.2.1. *Robustness Properties*

Here, we summarize four fundamental definitions that support different kinds of verification techniques.

Definition 1 (Local Robustness Property). Given a neural network $f : \mathbb{R}^n \to \mathbb{R}^m$ and an input x_0, the local robustness of f on x_0 is defined as

$$Robust(f, r) \triangleq \forall \|x - x_0\|_p \leq r, \exists i, j \in \{1, \ldots, m\}, \quad f_l(x) \geq f_j(x). \quad (13.19)$$

For targeted local robustness of f on x_0 and label l, it is defined as

$$Robust_l(f, r) \triangleq \forall \|x - x_0\|_p \leq r, \exists j \in \{1, \ldots, m\}, \quad f_l(x) \geq f_j(x). \quad (13.20)$$

From Definition 1, the local robustness means the decision of a DNN does not change within the region $r_{\max}$. For example, there exists a label l such that, for all inputs x in region $r_{\max}$, and other labels j, the DNN believes that x is more possible to be in class l than in any class j.

To compute the set of outputs with respect to predefined input space, we have the following definition of output reachability.

Definition 2 (Output Reachability Property). Given a neural network f, and an input region $\|x - x_0\|_p \leq r$, the reachable set of f and r is a set $Reach(f, r)$ such that

$$Reach(f, r) \triangleq \{f(x) \mid x \in \|x - x_0\|_p \leq r\}. \tag{13.21}$$

Due to the region r covering a large or infinite number of entities within the input space, there is no realistic algorithm to check their classifications. In addition to the highly nonlinear (or black-box) neural network f, the output reachability problem is highly non-trivial. Based on this, the verification of output reachability is to determine whether all inputs x with distance r can map onto a given output set $\mathcal{Y}$, and whether all outputs in $\mathcal{Y}$ have a corresponding input x within r, that is, check whether the equation $Reach(f, r) = \mathcal{Y}$ is satisfied or not. Normally, the largest or smallest values of a specific dimension of the set $Reach(f, r)$ are much more useful in reachability problem analysis.

Definition 3 (Interval Property). Given a neural network $f : \mathbb{R}^n \to \mathbb{R}^m$, an input x_0, and $\|x - x_0\|_p \leq r$, the interval property of f and r is a convex set $Interval(f, r)$ such that

$$Interval(f, r) \supseteq \{f(x) \mid x \in \|x - x_0\|_p \leq r\}. \tag{13.22}$$

If the set $Interval(f, r)$ is the smallest convex set of points in $\{f(x) \mid x \in \|x - x_0\|_p \leq r\}$, we call this set a convex hull, which means $Interval(f, r)$ is the closest to $\{f(x) \mid x \in \|x - x_0\|_p \leq r\}$.

Unlike output reachability property, the interval property calculates a convex over-approximation of output reachable set. Based on Definition 3, the interval is also seen as an over-approximation of output reachability. Similarly, for f, r and output region $\mathcal{Y}$, we can define the verification of interval property as $\mathcal{Y} \supseteq \{f(x) \mid x \in \|x - x_0\|_p \leq r\}$, which means that all inputs in r are mapped onto $\mathcal{Y}$. The computing process has to check whether the given convex set $\mathcal{Y}$ is an interval satisfying (13.22). Similar to the output reachability property, there exist useful and simple problems, e.g., determining whether a given real number d is a valid upper bound for a specific dimension of $\{f(x) \mid x \in \|x - x_0\|_p \leq r\}$.

Definition 4 (Lipschitzian Property). Given a neural network $f : \mathbb{R}^n \to \mathbb{R}^m$, an input x_0, and $\eta(x_0, l_p, r) = \{x | \|x - x_0\|_p \leq r\}$,

$$Lips(f, \eta, l_p) \equiv \sup_{x \in \eta} \frac{|f(x) - f(x_0)|}{||x - x_0||_p}. \tag{13.23}$$

Inspired by the Lipschitz optimization [49], $Lips(f, \eta, l_p)$ is a Lipschitzian metric of f, η, and l_p, which is to quantify the changing rate of neural networks within η. Intuitively, the greatest changing rate of this metric is the best Lipschitz constant. Based on the Lipschitzian metric and a real value $d \in \mathbb{R}$, the verification of Lipschitzian property is to find out whether $Lips(f, \eta, l_p) \leq d$. Usually, the value of this metric is not easy to compute, and d is the upper bound for this value.

There are some relationships between the definitions of the above four properties. For example, when the set $Reach(f, r)$ is computed exactly, we can easily evaluate the inclusion of the set in another convex set $Interval(f, r)$. Similarly, based on the Lipschitzian metric $Lips(f, \eta, l_p)$, we can evaluate the interval property and verify this property. Please refer to our papers [39, 50] for more details.

13.4.2.2. *Verification Approaches*

Current verification techniques mainly include constraint solving, search-based approach, optimization, and over-approximation. To make these techniques efficient, one of these categories can also be combined with another category. The most significant aspect of verification techniques is that they can provide a guarantee for the result. Based on the type of guarantee, verification methods can be divided into four categories. They are methods with deterministic guarantee, one-side guarantee, guarantee converging bounds, and statistical guarantee.

Methods with deterministic guarantee state exactly whether a property holds. They transform the verification problem into a set of constraints, further solving them with a constraint solver. As a result, the solvers return a deterministic answer to a query, i.e., either satisfiable or unsatisfiable. Representative solvers used in this area are Boolean satisfiability (SAT) solvers, satisfiability modulo theories (SMT) solvers, linear programming (LP) solvers, and mixed-integer linear programming (MILP) solvers. The SAT method determines if, given a Boolean formula, there exists an assignment to the Boolean variables such that the formula is satisfiable. The prerequisite for the employment is to abstract a network into a set of Boolean combinations, whose example is given in [51]. The SMT method takes one step further and determines the satisfiability of logical formulas with respect to combinations of background theories expressed in classical first-order logic with equality. Classical SMT methods are Reluplex [52] and Planet [53]. The LP method optimizes the linear objective function subject to linear equality or linear inequality constraints. Given the requirement that some of the variables are integral, the LP problem becomes a MILP problem. For instance, a hidden layer

$z_{i+1} = \text{ReLU}(W_i z_i + b_i)$ can be described with the following MILP conditions:

$$W_i z_i + b_i \leq z_{i+1} \leq W_i z_i + b_i + M t_{i+1},$$
$$0 \leq z_{i+1} \leq M(1 - t_{i+1}),$$

where t_{i+1} has value 0 or 1, representing the activated situation, and $M > 0$ is a large constant that can be treated as ∞. The details of the encoding neural network to a MILP problem can be found in [54]. To improve the verification efficiency, extensions are made to the simple MILP method, achieving successful variant methods such as [55] and Sherlock in [56].

Approaches with a one-side guarantee only compute an approximated bound, which can claim the sufficiency of achieving properties. While these methods provide only a bounded estimation with high efficiency, they are applicable for analyzing large models. Some approaches adopt abstract interpretation, using abstract domains, e.g., boxes, zonotopes, and polyhedra, to over-approximate the computation of a set of inputs. Its application has been explored in a few approaches, including AI2 [57] and [58], which can verify interval property but cannot verify reachability property. Because the non-convex property of neural networks partially causes difficulty of verification, methods such as [59] tend to make a convex outer over-approximation of the sets of reachable activation and apply the convex optimization. In the approach ReluVal [60], the interval arithmetic is leveraged to compute rigorous bounds on the DNN outputs, i.e., interval property. The key idea of the interval analysis approach is that given the ranges of operands, an over-estimated range of the output can be computed by using only the lower and upper bounds of the operands. In addition, one-side guarantee approaches also include [61] dealing with output reachability estimation and FastLin/FastLip in [62] using linear approximation for ReLU networks.

Approaches aiming to achieve statistical guarantees on their results usually make claims such as the satisfiability of a property or a value is a lower bound of another value with a certain probability. Typical methods are CLEVER [63] working with Lipschitz property [64], which proposes the robustness estimation to measure the frequency and the severity of adversarial examples. Approaches to compute the converge bounds can be applied on large-scale real world systems and they can work with both output reachability property and interval property. Huang et al. [65] made a layer-to-layer analysis of the network, guaranteeing a misclassification being found if it exists. DeepGame [66] reduces the verification problem to a two-player turn-based game, studying two variants of point-wise robustness.

Unfortunately, these verification algorithms have some weaknesses. Algorithms based on constraint solvers, such as SMT/SAT and MILP/LP, need to encode

the entire network layer by layer, limiting them to only work on small-scale or quasi-linear neural networks. Although approaches based on linear approximation, convex optimization, and abstract interpretation can work on nonlinear activation functions, they are essentially a type of over-approximations algorithm, i.e., they generally calculate the lower bound of the maximal safe radius $r_{\max}$. Besides, these algorithms depend on the specific structure of the networks to design different abstract interpretations or linear approximations. Hence, they cannot provide a generic framework for different neural networks.

To circumvent such shortcomings, approaches based on global optimization are developed. DeepGo [50] firstly proves that most neural network layers are Lipschitz continuous and uses Lipschitz optimization to solve the problem. Since the only requirement is Lipschitz continuity, a number of studies have been done to expand this method to various neural network structures. DeepTRE [67] focuses on the Hamming distance, proposing an approach to iteratively generate lower and upper bounds on the network's robustness. The tensor-based approach returns intermediate bounds, allowing for efficient GPU computation. Furthermore, we also proposed a generic framework, DeepQuant [68], to quantify the safety risks on networks based on a Mesh Adaptive Direct Search algorithm.

In summary, the maximal safe radius $r_{\max}$ provides the local robustness information of the neural network. The adversarial attacks focus on estimating the upper bound of the radius and searching for adversarial examples outside the upper bound of the safe norm ball, while verification studies the lower bound of $r_{\max}$, ensuring that inputs inside this norm ball are safe. Both analysis methods only concern evaluating existing neural networks. How to further improve the neural network robustness will be discussed in the section on adversarial defense.

13.4.3. *Defense*

While verification aims to provide theoretical guarantees on the robustness and reliability of DNNs, defense methods attempt to enhance such properties from a more practical perspective.

Generally, the defense strategy can be divided into four categories: pre-processing, model enhancement, adversarial training, and certified defenses. Specifically, example pre-processing methods are probably the most straightforward defense. In this strategy, input examples are processed via methods such as median smoothing, bit depth compression, and JPEG compression [69] before feeding them to the deep learning system. Pre-processing can reduce the impact of perturbation and keep enough information within the processed examples for the downstream tasks. Because most kinds of adversarial perturbation are essentially additional noise, pre-processing defenses are highly effective in vision and audio applications. Besides,

this pre-processing defense does not require to re-train the protected models and can be deployed directly. Although it might cause a trivial problem if the attackers are aware of the pre-processing procedure, they can correspondingly update the generation of adversarial examples to break or bypass such defense. This means that pre-processing defenses are not reliable in white-box or "gray-box" (attackers have limited knowledge about their target model) environments.

In parallel with pre-processing, model enhancement focuses on deep learning models themselves. The core of this type of method is to improve the robustness by modifying the model architecture, activation functions, and loss functions. A typical defense along this direction is defensive distillation [70], which follows the teacher–student model [71]. Defensive distillation first produces a regular network (teacher) to generate soft labels instead of the one-hot encoding of the true labels (hard labels) to train a student model. This defense had significantly boosted the model's robustness on several benchmarks, but it was soon broken by the C&W attack proposed by Carlini and Wagner [72]. In addition to processing perturbed examples correctly, model enhancement sometimes achieves the defense effect via adversarial detection, i.e., identifying adversarial examples and refusing to give corresponding outputs. Although many model enhancement methods eventually failed [73], this direction is still a valuable and active research direction.

The idea of adversarial training appeared with the FGSM attack [42] and has been significantly generalized by later works [44]. In these pioneering works, adversarial examples were regarded as an augmentation of the original training dataset, and adversarial training required both original examples and adversarial examples to train a robust model. Madry et al. formalized adversarial training as a Min–Max optimization problem:

$$\min_{\theta} \mathbb{E}_{(x,y)\sim\mathcal{D}}\Big(\max_{\delta\in B_\epsilon} L(F_\theta(x+\delta))\Big). \tag{13.24}$$

Here, in the inner maximization problem, adversarial examples are generated by maximizing the classification loss, while the outer minimization updates model parameters by minimizing the loss on adversarial examples from the inner maximization. Equation (13.24) provides a principled connection between the robustness of neural networks and the adversarial attacks. This indicates that a qualified adversarial attack method could help train robust models. At the current stage, PGD adversarial training is still one of the most effective defenses [73, 74], although there are quite incremental works aiming for improvement. Nevertheless, all these PGD-based adversarial trainings take multiple backpropagations to generate adversarial perturbations, which significantly raises the computational cost. As a result, conducting adversarial training, especially on large-scale networks, is prohibitively

time-consuming. On the way to improve its efficiency, recent studies have two interesting findings. On the one hand, Wong et al. [75] demonstrated that high-quality or strong adversarial examples might not be necessary for conducting successful adversarial training. They empirically showed that FGSM with uniform initialization could adversarially train models with considerable robustness more efficiently than PGD adversarial training. On the other hand, YOPO adversarial training [76] simplifies the backpropagation for updating training adversarial examples, i.e., it treats the first layer and others separately during training and generates adversarial examples toward the first layer instead of the whole trained model. This successful attempt suggests that we might need to adversarially train only the first few layers to produce robust deep learning models. Moreover, inspired by recent studies, our work [77] reveals that the connection of the lower bound of Lipschitz constant of a given network and the magnitude of its partial derivative toward adversarial examples. Supported by this theoretical finding, we utilize the gradient's magnitude to quantify the effectiveness of adversarial training and determine the timing to adjust the training procedure. In addition to robustness, adversarial training also benefits the deep learning study in terms of interpretability [78] and data augmentation. Such a technology is getting increasing attention in applications beyond security, such as text-to-image tasks [25, 79] and nature language processing [80]. Therefore, we believe that adversarial training will still be an interesting and promising research topic in the next few years.

The previous defenses are all heuristic strategies and cannot provide theoretical guarantees. Like the defensive distillation method, most defenses are likely to be broken by emerging adversarial attacks. To end such an "arms race", certified defenses aim to theoretically guarantee that the outputs of deep neural networks would not change within a certain disturbance range around inputs. Similar to some verification techniques, certified defenses come out with strong theoretical guarantees. However, they are usually computationally expensive and difficult to be extended on large-scale networks or high-dimensional datasets [50]. A promising method in this direction is randomized smoothing [81], which introduces random noise to smooth the learned feature during training. This technique demonstrates that it is feasible to use randomized perturbation to improve the trained model's robustness concretely. However, it can only work and provide guarantees under the l_2 constraint. Thus, how to generalize it to other threat models is an urgent need of research.

Overall, defending against adversarial attacks is still a work in progress, and we are far from solving this problem. We believe that preventing adversarial threats and interpreting DNNs are two sides of the same coin, and any useful finding on one side would boost our progress on the other side. Although all current defense strategies are not perfect and might only work temporarily, we would gain a better

understanding of the fundamental mechanism of deep learning models if we keep digging along this direction.

13.5. Further Reading

After more than 10 years of exploration, deep learning is still a jungle full of secrets and challenges. This chapter only provides a brief yet general introduction, and we are certainly not experts in most sub-areas of this gigantic and fast-developing research topic. So, at the end of this chapter, we provide a simple overview on two emerging deep learning techniques and existing challenges.

13.5.1. *Emerging Deep Learning Models*

One of the most noticeable threads is the generative adversarial networks (GANs) [82]. Beyond learning from data, GAN models can generate new information by utilizing learned features and patterns, which gives birth to many exciting applications. For instance, convolutional neural networks can decouple the semantic content and art style in images and rebuild images with the same semantics but different style [83]. Coloring grayscale images without manual adjustment was a difficult task for computers, which is can be done by a deep learning algorithm called deep colorization [84]. In the audio processing area, Google AI utilized the GAN model to generate high-fidelity raw audio waveforms [85], and the latest deep learning model [86] can even decompose audio into fundamental components, such as loudness, pitch, timbre, and frequency, and manipulate them independently. Various GANs would start new revolutions in a lot of applications, such as Photoshop, one of the popular graphics editors, which includes a neural network-based toolbox to help users advance their arts. Applications such as deepfake and deepnude also raise concern about how to prevent such powerful techniques from being abused.

As we mentioned before, Transformer [32] has greatly boosted the raw performance of DNNs on natural language processing tasks. The most famous deep learning models built upon Transformer are GPT [33] and BERT [34]. The latest version of GPT in 2020, namely GPT-3, is the largest deep neural network, which can generate high-quality text that sometimes cannot be distinguished from that written by a real person. BERT is the most popular method and has been widely used across the NLP industry. One difference between these two models is that GPT has only one direction, while BERT is a bidirectional model. Therefore, BERT usually has a better performance but is incompetent for text generation tasks, which are GPT's specialty. Nowadays, the application of Transformer is no longer limited to the NLP area. Recent studies [87] show that this powerful architecture is capable of vision tasks.

13.5.2. *Challenges*

Despite the exciting new technologies and applications, the difficulties associated with deep learning have not changed significantly.

In addition to adversarial threats and the high demand for computational power, the lack of interpretability is also a big challenge in deep learning. A modern deep neural network usually has billions of weights, intricate architecture, and tons of hyper-parameters such as learning rate, dropout rate, etc. We do not yet fully understand the inner mechanism of these models, especially why and how a deep learning model makes certain decisions. Currently, the training of deep learning models is still largely based on trial and error and lacks a principled way to guide the training. Taking the LeNet-5 model as an example, it has two conventional layers and two fully connected layers [88]. We know that if we train it properly on the MNIST dataset, it can achieve approximately 99% classification accuracy on the test set. However, we cannot know how the accuracy changes without experiments after adding one more conventional layer into the model because we do not have a mathematical tool to formulate the represent capability of deep neural network models. Without such a theoretical guide, designing neural network architectures for a specific task has become a challenging task. This leads us to the neural architecture search (NAS) technology, which aims to find the best network architecture under limited computing resources in an automatic way. However, at the current stage, research on understanding deep learning is still in its infancy.

The lack of labeled data is also a barrier, which significantly limits the applicability of deep learning in some scenarios where only limited data are available. Training a deep learning model to a satisfactory performance usually requires a huge amount of data. For example, ImageNet, the dataset that has significantly accelerated the development of deep learning, contains over 14 million labeled images. While being data-hungry is not a problem for consumer applications where large amounts of data are easily available, copious amounts of labeled training data are rarely available in most industrial applications. So, how to improve the applicability of deep learning models with limited training data is still one of the key challenges today. Emerging techniques such as transfer learning, meta learning, and GANs show some promise with regard to overcoming this challenge.

References

[1] O. Russakovsky, J. Deng, H. Su, et al., Imagenet large scale visual recognition challenge, *Int. J. Comput. Vision*, **115**(3), 211–252 (2015).

[2] D. Amodei, S. Ananthanarayanan, R. Anubhai, J. Bai, E. Battenberg, C. Case, J. Casper, B. Catanzaro, Q. Cheng, G. Chen, et al., Deep speech 2: end-to-end speech recognition in english and mandarin. In *International Conference on Machine Learning*, pp. 173–182. PMLR (2016).

[3] M. E. Peters, M. Neumann, M. Iyyer, et al., Deep contextualized word representations, *arXiv preprint arXiv:1802.05365* (2018).
[4] Y. LeCun, Y. Bengio, and G. Hinton, Deep learning, *Nature*, **521**(7553), 436–444 (2015).
[5] W. S. McCulloch and W. Pitts, A logical calculus of the ideas immanent in nervous activity, *Bull. Math. Biophys*., **5**(4), 115–133 (1943).
[6] F. Rosenblatt and S. Papert, The perceptron, A perceiving and recognizing automation, Cornell Aeronautical Laboratory Report. pp. 85–460 (1957).
[7] M. Minsky and S. A. Papert, *Perceptrons: An Introduction to Computational Geometry* (MIT Press, 2017).
[8] K. Fukushima, A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position, *Biol. Cybern.,* **36**, 193–202 (1980).
[9] T. Gonsalves, The summers and winters of artificial intelligence. In *Advanced Methodologies and Technologies in Artificial Intelligence, Computer Simulation, and Human-Computer Interaction*, pp. 168–179. IGI Global (2019).
[10] S. Linnainmaa, The representation of the cumulative rounding error of an algorithm as a Taylor expansion of the local rounding errors, Master's Thesis (in Finnish), Univ. Helsinki. pp. 6–7 (1970).
[11] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, Learning representations by back-propagating errors, *Nature*, **323**(6088), 533–536 (1986).
[12] Y. LeCun, B. Boser, J. S. Denker, et al., Backpropagation applied to handwritten zip code recognition, *Neural Comput.,* **1**(4), 541–551 (1989).
[13] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, GPU computing, *Proc. IEEE*, **96**(5), 879–899 (2008).
[14] A. Krizhevsky, I. Sutskever, and G. E. Hinton, Imagenet classification with deep convolutional neural networks, *Commun. ACM*, **60**(6), 84–90 (2017).
[15] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio, *Deep Learning* (MIT Press Cambridge, 2016).
[16] J. Schmidhuber, Deep learning in neural networks: an overview, *Neural Networks*, **61**, 85–117 (2015).
[17] A. Zhang, Z. C. Lipton, M. Li, and A. J. Smola, *Dive into Deep Learning* (2020). https://d2l.ai.
[18] C. Nwankpa, W. Ijomah, A. Gachagan, and S. Marshall, Activation functions: comparison of trends in practice and research for deep learning, *arXiv preprint arXiv:1811.03378* (2018).
[19] W. Zhang, et al., Shift-invariant pattern recognition neural network and its optical architecture. In *Annual Conference of the Japan Society of Applied Physics* (1988).
[20] D. H. Hubel and T. N. Wiesel, Receptive fields and functional architecture of monkey striate cortex, *J. Physiol*., **195**(1), 215–243 (1968).
[21] K. Fukushima and S. Miyake, Neocognitron: a self-organizing neural network model for a mechanism of visual pattern recognition. In *Competition and Cooperation in Neural Nets*, pp. 267–285. Springer (1982).
[22] M. Matsugu, K. Mori, Y. Mitari, and Y. Kaneda, Subject independent facial expression recognition with robust face detection using a convolutional neural network, *Neural Networks*, **16**(5–6), 555–559 (2003).
[23] K. O'Shea and R. Nash, An introduction to convolutional neural networks, *arXiv preprint arXiv:1511.08458* (2015).
[24] A. V. D. Oord, S. Dieleman, H. Zen, et al., Wavenet: a generative model for raw audio, *arXiv preprint arXiv:1609.03499* (2016).
[25] N. Sarafianos, X. Xu, and I. A. Kakadiaris, Adversarial representation learning for text-to-image matching. In *International Conference on Computer Vision* (2019).
[26] M. I. Jordan, Serial order: a parallel distributed processing approach. In *Advances in Psychology*, vol. 121, pp. 471–495. Elsevier (1997).

[27] J. L. Elman, Finding structure in time, *Cognit. Sci*., **14**(2), 179–211 (1990).
[28] S. Hochreiter and J. Schmidhuber, Long short-term memory, *Neural Comput.,* **9**(8), 1735–1780 (1997).
[29] K. Cho, B. van Merrienboer, Ç. Gülçehre, et al., Learning phrase representations using RNN encoder-decoder for statistical machine translation. In *Conference on Empirical Methods in Natural Language Processing* (2014).
[30] Y. Bengio, R. Ducharme, P. Vincent, and C. Jauvin, A neural probabilistic language model, *J. Mach. Learn. Res.,* **3**, 1137–1155 (2003).
[31] T. Mikolov, I. Sutskever, K. Chen, et al., Distributed representations of words and phrases and their compositionality. In *Proceedings of the Advances in Neural Information Processing Systems* (2013).
[32] A. Vaswani, N. Shazeer, N. Parmar, et al., Attention is all you need. In *Proceedings of the Advances in Neural Information Processing Systems* (2017).
[33] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever, Improving language understanding by generative pre-training (2018).
[34] J. Devlin, M. Chang, K. Lee, and K. Toutanova, BERT: pre-training of deep bidirectional transformers for language understanding. In *North American Chapter of the Association for Computational Linguistics: Human Language Technologies* (2019).
[35] G. Goh, Why momentum really works, *Distill* (2017). doi: 10.23915/distill.00006. URL http://distill.pub/2017/momentum.
[36] G. E. Hinton, S. Osindero, and Y.-W. Teh, A fast learning algorithm for deep belief nets, *Neural Comput.,* **18**(7), 1527–1554 (2006).
[37] S. Ioffe and C. Szegedy, Batch normalization: accelerating deep network training by reducing internal covariate shift, *arXiv preprint arXiv:1502.03167* (2015).
[38] Y. Wu and K. He, Group normalization. In *European Conference on Computer Vision* (2018).
[39] X. Huang, D. Kroening, W. Ruan, et al., A survey of safety and trustworthiness of deep neural networks: verification, testing, adversarial attack and defence, and interpretability, *Comput. Sci. Rev*., **37**, 100270 (2020).
[40] B. Biggio and F. Roli, Wild patterns: ten years after the rise of adversarial machine learning, *Pattern Recognit.,* **84**, 317–331 (2018).
[41] C. Szegedy, W. Zaremba, I. Sutskever, et al., Intriguing properties of neural networks. In *International Conference on Learning Representations* (2014).
[42] I. J. Goodfellow, J. Shlens, and C. Szegedy, Explaining and harnessing adversarial examples. In *International Conference on Learning Representations* (2015).
[43] A. Kurakin, I. J. Goodfellow, and S. Bengio, Adversarial examples in the physical world. In *International Conference on Learning Representations, Workshop* (2017).
[44] A. Madry, A. Makelov, L. Schmidt, et al., Towards deep learning models resistant to adversarial attacks. In *International Conference on Learning Representations* (2018).
[45] J. Uesato, B. O'Donoghue, P. Kohli, et al., Adversarial risk and the dangers of evaluating against weak attacks. In *International Conference on Machine Learning* (2018).
[46] C. Xiao, J.-Y. Zhu, B. Li, et al., Spatially transformed adversarial examples. In *International Conference on Learning Representations* (2018).
[47] Y. Zhang, W. Ruan, F. Wang, et al., Generalizing universal adversarial attacks beyond additive perturbations. In *IEEE International Conference on Data Mining* (2020).
[48] Y. Zhang, F. Wang, and W. Ruan, Fooling object detectors: adversarial attacks by half-neighbor masks, *arXiv preprint arXiv:2101.00989* (2021).
[49] D. R. Jones, C. D. Perttunen, and B. E. Stuckman, Lipschitzian optimization without the lipschitz constant, *J. Optim. Theory Appl*., **79**(1), 157–181 (1993).
[50] W. Ruan, X. Huang, and M. Kwiatkowska, Reachability analysis of deep neural networks with provable guarantees. In *International Joint Conference on Artificial Intelligence* (2018).

[51] L. Pulina and A. Tacchella, An abstraction-refinement approach to verification of artificial neural networks. In *International Conference on Computer Aided Verification* (2010).
[52] G. Katz, C. Barrett, D. L. Dill, et al., Reluplex: an efficient SMT solver for verifying deep neural networks. In *International Conference on Computer Aided Verification* (2017).
[53] R. Ehlers, Formal verification of piece-wise linear feed-forward neural networks. In *International Symposium on Automated Technology for Verification and Analysis* (2017).
[54] A. Lomuscio and L. Maganti, An approach to reachability analysis for feed-forward ReLU neural networks, *arXiv preprint arXiv:1706.07351* (2017).
[55] C.-H. Cheng, G. Nührenberg, and H. Ruess, Maximum resilience of artificial neural networks. In *International Symposium on Automated Technology for Verification and Analysis* (2017).
[56] S. Dutta, S. Jha, S. Sanakaranarayanan, et al., Output range analysis for deep neural networks, *arXiv preprint arXiv:1709.09130* (2018).
[57] T. Gehr, M. Mirman, D. Drachsler-Cohen, et al., AI2: safety and robustness certification of neural networks with abstract interpretation. In *IEEE Symposium on Security and Privacy* (2018).
[58] M. Mirman, T. Gehr, and M. Vechev, Differentiable abstract interpretation for provably robust neural networks. In *International Conference on Machine Learning* (2018).
[59] E. Wong and Z. Kolter, Provable defenses against adversarial examples via the convex outer adversarial polytope. In *International Conference on Machine Learning* (2018).
[60] S. Wang, K. Pei, J. Whitehouse, et al., Formal security analysis of neural networks using symbolic intervals. In *USENIX Security Symposium*. USENIX Association (2018).
[61] W. Xiang, H.-D. Tran, and T. T. Johnson, Output reachable set estimation and verification for multi-layer neural networks, *IEEE Trans. Neural Networks Learn. Syst*., **29**, 5777–5783 (2018).
[62] L. Weng, H. Zhang, H. Chen, et al., Towards fast computation of certified robustness for ReLU networks. In *International Conference on Machine Learning* (2018).
[63] T.-W. Weng, H. Zhang, P.-Y. Chen, et al., Evaluating the robustness of neural networks: an extreme value theory approach, *arXiv preprint arXiv:1801.10578* (2018).
[64] O. Bastani, Y. Ioannou, L. Lampropoulos, et al., Measuring neural net robustness with constraints, *arXiv preprint arXiv:1605.07262* (2016).
[65] X. Huang, M. Kwiatkowska, S. Wang, and M. Wu, Safety verification of deep neural networks. In *International Conference on Computer Aided Verification* (2017).
[66] M. Wu, M. Wicker, W. Ruan, et al., A game-based approximate verification of deep neural networks with provable guarantees, *Theor. Comput. Sci.*, **807**, 298–329 (2020).
[67] W. Ruan, M. Wu, Y. Sun, et al., Global robustness evaluation of deep neural networks with provable guarantees for the Hamming distance. In *International Joint Conference on Artificial Intelligence* (2019).
[68] P. Xu, W. Ruan, and X. Huang, Towards the quantification of safety risks in deep neural networks, *arXiv preprint arXiv:2009.06114* (2020).
[69] R. Shin and D. Song, JPEG-resistant adversarial images. In *Annual Network and Distributed System Security Symposium* (2017).
[70] N. Papernot, P. D. McDaniel, X. Wu, et al., Distillation as a defense to adversarial perturbations against deep neural networks. In *IEEE Symposium on Security and Privacy* (2016).
[71] G. E. Hinton, O. Vinyals, and J. Dean, Distilling the knowledge in a neural network, *arXiv preprint arXiv:1503.02531* (2015).
[72] N. Carlini and D. A. Wagner, Towards evaluating the robustness of neural networks. In *IEEE Symposium on Security and Privacy* (2017).
[73] A. Athalye, N. Carlini, and D. A. Wagner, Obfuscated gradients give a false sense of security: circumventing defenses to adversarial examples. In *International Conference on Machine Learning* (2018).
[74] L. Rice, E. Wong, and J. Z. Kolter, Overfitting in adversarially robust deep learning. In *International Conference on Machine Learning* (2021).

[75] E. Wong, L. Rice, and J. Z. Kolter, Fast is better than free: revisiting adversarial training. In *International Conference on Learning Representations* (2020).
[76] D. Zhang, T. Zhang, Y. Lu, et al., You only propagate once: accelerating adversarial training via maximal principle. In *Proceedings of the Advances in Neural Information Processing Systems* (2019).
[77] F. Wang, Y. Zhang, Y. Zheng, and W. Ruan, Gradient-guided dynamic efficient adversarial training, *arXiv preprint arXiv:2103.03076* (2021).
[78] T. Zhang and Z. Zhu, Interpreting adversarially trained convolutional neural networks. In *International Conference on Machine Learning* (2019).
[79] T. Chen, Y. Liao, C. Chuang, et al., Show, adapt and tell: adversarial training of cross-domain image captioner. In *International Conference on Computer Vision* (2019).
[80] D. Wang, C. Gong, and Q. Liu, Improving neural language modeling via adversarial training. In *International Conference on Machine Learning* (2019).
[81] J. M. Cohen, E. Rosenfeld, and J. Z. Kolter, Certified adversarial robustness via randomized smoothing. In *International Conference on Machine Learning* (2019).
[82] I. Goodfellow, J. Pouget-Abadie, M. Mirza, et al., Generative adversarial nets. In *Proceedings of the Advances in Neural Information Processing Systems* (2014).
[83] L. A. Gatys, A. S. Ecker, and M. Bethge, Image style transfer using convolutional neural networks. In *IEEE Conference on Computer Vision and Pattern Recognition* (2016).
[84] Z. Cheng, Q. Yang, and B. Sheng, Deep colorization. In *International Conference on Computer Aided Verification* (2015).
[85] J. Engel, K. K. Agrawal, S. Chen, et al., Gansynth: adversarial neural audio synthesis, *arXiv preprint arXiv:1902.08710* (2019).
[86] J. Engel, L. Hantrakul, C. Gu, and A. Roberts, DDSP: differentiable digital signal processing, *arXiv preprint arXiv:2001.04643* (2020).
[87] A. Dosovitskiy, L. Beyer, A. Kolesnikov, et al., An image is worth 16 x 16 words: transformers for image recognition at scale. In *International Conference on Learning Representations* (2021).
[88] Y. LeCun, C. Cortes, and C. Burges, MNIST handwritten digit database, ***AT&T Labs*, 2** (2010).