Part III

Deep Learning

10
Introduction to Deep Learning

Deep learning is currently the most successful machine learning approach, with notable successes in object recognition, speech and language understanding, self-driving cars, automated Go playing, etc. It is not easy to give a single definition to such a broad and influential field; nevertheless here is a recent definition by Chris Manning:[1]

> *Deep Learning is the use of large multi-layer (artificial) neural networks that compute with continuous (real number) representations, a little like the hierarchically-organized neurons in human brains. It is currently the most successful ML approach, usable for all types of ML, with better generalization from small data and better scaling to big data and compute budgets.*

[1] Source: https://hai.stanford.edu/sites/default/files/2020-09/AI-Definitions-HAI.pdf.

Deep learning does not represent a specific a model per se, but rather categorizes a group of models called (artificial) neural networks (NNs) (or *deep nets*) which involve several computational layers. Linear models studied in earlier chapters, such as logistic regression in Section 4.2, can be seen as special sub-cases involving only a single layer. The main difference, however, is that general deep nets employ nonlinearity in between each layer, which allows a much broader scale of expressivity. Also, the multiple layers in a neural net can be viewed as computing "intermediate representations" of the data, or "high level features" before arriving at its final answer. By contrast, a linear model works only with the data representation it was given.

Deep nets come in various types, including Feed-Forward NNs (FFNNs), Convolutional NNs (CNNs), Recurrent NNs (RNNs), Residual Nets, and Transformers. [2] Training uses a variant of *Gradient Descent*, and the gradient of the loss is computed using an algorithm called *backpropagation*.

[2] Interestingly, a technique called *Neural Architecture Search* uses deep learning to design custom deep learning architectures for a given task.

Due to the immense popularity of deep learning, a variety of software environments such as Tensorflow and PyTorch allow quick implementation of deep learning models. You will encounter them in the homework.

10.1 A Brief History

Neural networks are inspired by the biological processes present within the brain. The concept of an artificial neuron was first outlined by Warren MuCulloch and Walter Pitts in the 1940s. [3] Later in 1986, backpropagation was discovered and provided a way for efficiently applying gradient-based training methods to these models. [4] The basic frameworks for CNNs and modern training soon followed in the late 1980s. [5] However, by the 21st century deep learning had gone out of fashion. This changed in 2012, when Krizhevsky, Sutskever, and Hinton leveraged deep learning techniques through their *AlexNet* model and set new standards for performance on the ImageNet dataset. [6] Deep learning has since begun a resurgence throughout the last decade, boosted by some key factors:

- Hardware, such as GPU and TPU (Tensor Processing Unit, specifically developed for neural network machine learning) technology have made training faster.
- The development of novel neural network architecutres as well as better algorithms for training neural networks.
- A vast amount of data collection, boosted by the spread of the internet, have augmented the performance of NN models.
- Popular frameworks, such Tensorflow and PyTorch, have made it easier to prototype and deploy NN architectures.
- Commercial payoff has caused tech corporations to invest more financial resources.

Each of the reasons listed above have interfaced in a positively reinforcing cycle, causing the acceleration of this technology into the foreseeable future.

[3] Paper: https://www.cs.cmu.edu/~./epxing/Class/10715/reading/McCulloch.and.Pitts.pdf.

[4] Paper: https://www.nature.com/articles/323533a0.

[5] Paper: https://link.springer.com/article/10.1007/BF00344251.

[6] Paper: https://papers.nips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

10.2 Anatomy of a Neural Network

10.2.1 Artificial Neuron

An *artificial neuron*, or a *node*, is the main component of a neural network. Artificial neurons were inspired by early work on neurons in animal brains, with the analogies in Table 10.1.

Formally, a node is a computational unit which receives m scalar inputs and outputs 1 scalar. This scalar output can be used as an input for a different neuron.

Consider the vector $\vec{\mathbf{x}} = (x_1, x_2, \ldots, x_m)$ of m inputs. A neuron internally maintains a trainable weight vector $\vec{\mathbf{w}} = (w_1, w_2, \ldots, w_m)$

Biological neuron	Artificial neuron
Dendrites	Input
Cell Nucleus / Soma	Node
Axon	Output
Synapse	Interconnections

Table 10.1: A comparison between biological neurons in the brain and artificial neurons in neural networks

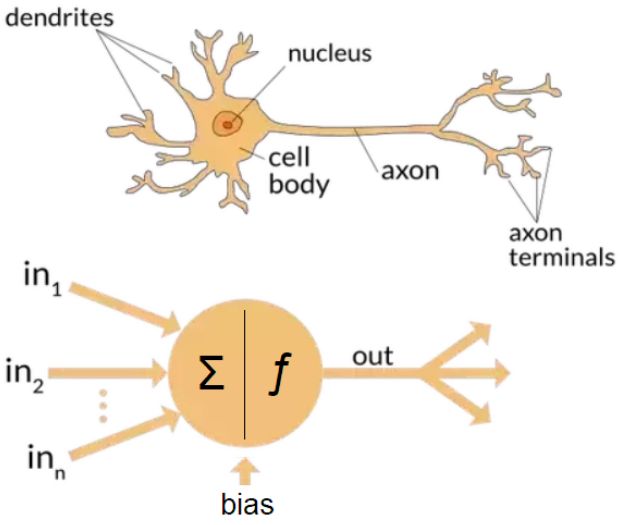


Figure 10.1: A comparison between a brain neuron and an artificial neuron.

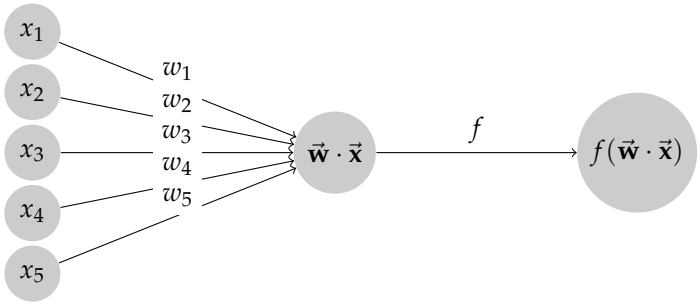


Figure 10.2: A sample artificial neuron.

and optionally a nonlinear activation function $f : \mathbb{R} \to \mathbb{R}$ and outputs the following value: [7]

$$y = f(\vec{\mathbf{w}} \cdot \vec{\mathbf{x}}) \tag{10.1}$$

We can also add a scalar bias b before applying the activation function $f(z)$ in which case the output will look like the following: [8]

$$y = f(\vec{\mathbf{w}} \cdot \vec{\mathbf{x}} + b)$$

10.2.2 *Activation Functions*

An artificial neuron can choose its nonlinear activation function $f(z)$ from a variety of options. One such choice is the sigmoid function

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{10.2}$$

[7] If no activation function is chosen, we can assume that f is an identity function $f(z) = z$.

[8] If we introduce a dummy variable for the constant bias term as in Chapter 1 we can absorb the bias term into the equation in (10.1).

Note that in this case, the neuron represents a logistic regression unit. [9] Another popular activation function is the hyperbolic tangent, which is similar to the sigmoid function:

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \tag{10.3}$$

In fact, we can rewrite the hyperbolic tangent in terms of sigmoid:

$$\tanh(z) = 2\sigma(2z) - 1 \qquad \text{(10.3 revisited)}$$

According to this expression, tanh function can be viewed as a rescaled sigmoid function. The key difference is: the range of $\sigma(z)$ is $(0, 1)$ and the range of $\tanh(z)$ is $(-1, 1)$.

Arguably the most commonly used activation function is the Rectified Linear Unit, or ReLU:

$$\text{ReLU}(z) = [z]_+ = \max\{z, 0\} \tag{10.4}$$

There are several benefits to the ReLU activation function. It is far cheaper to compute than the previous two alternatives and avoids the "vanishing gradient" problem. [10] With sigmoid and hyperbolic tangent activation functions, the vanishing gradient problem happens when $z = \vec{\mathbf{x}} \cdot \vec{\mathbf{w}}$ has high absolute values, but ReLU avoids this problem because the derivative is exactly 1 even for high values of z.

Example 10.2.1. *Consider a vector $\vec{\mathbf{x}} = (-2, -1, 0, 1, 2)$ of inputs and a neuron with the weights $\vec{\mathbf{w}} = (1, 1, 1, 1, 1)$. If the activation function of this neuron is the sigmoid, then the output will be:*

$$y = \sigma(\vec{\mathbf{w}} \cdot \vec{\mathbf{x}}) = \sigma(0) = \frac{1}{2}$$

If the activation is ReLU, it will output:

$$[\vec{\mathbf{w}} \cdot \vec{\mathbf{x}}]_+ = [0]_+ = 0$$

Problem 10.2.2. *Consider a neuron with the weights $\vec{\mathbf{w}} = (1, 1, 5, 1, 1)$ and the ReLU activation function. What will the outputs y_1 and y_2 be for the inputs $\vec{\mathbf{x}}_1 = (-2, -2, 0, 1, 2)$ and $\vec{\mathbf{x}}_2 = (2, -1, 0, 1, 2)$ respectively?*

10.2.3 Neural Network

A neural network consists of nodes connected with directed edges, where each edge has a trainable parameter called its "weight" and each node has an activation function as well as associated parameter(s). There are designated *input* nodes and *output nodes*. The input nodes are given some input values, and the rest of the network then computes as follows: each node produces its output by taking the

[9] However, in this context the output is *not* considered to be a subjective probability as in the case of standard logistic regression.

[10] The vanishing gradient problem refers to a situation where the derivative of a certain step is too close to 0, which can stall the gradient-based learning techniques common in deep learning.

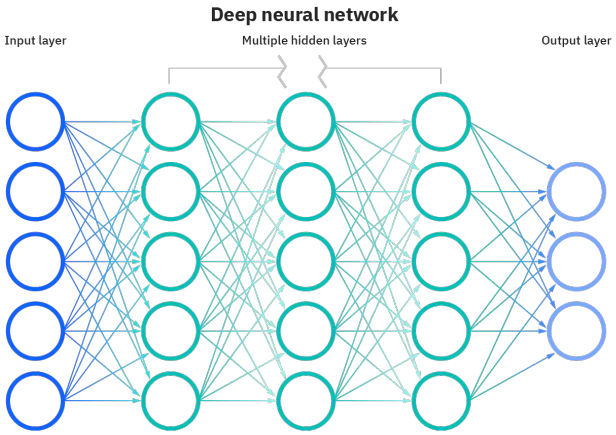


Figure 10.3: A sample neural network design. Each circle represents one artificial neuron. Two nodes being connected by an edge means that the output of the node on the left is being used as one of the inputs for the node on the right.

values produced by all nodes that have a directed edge to it. If the directed graph of connections is *acyclic* — which is the case in most popular architectures — this process of producing the values takes finite time and we end up with a unique value at each of the output nodes. [11] The term *hidden nodes* is used for nodes that are not input or output nodes.

[11] We will not study *Recurrent Neural Nets (RNNs)*, where the graph contains cycles. These used to be popular until a few years ago, and present special difficulties due to the presence of directed loops. For instance, can you come up with instances where the output is not well-defined?

10.3 Why Deep Learning?

Now that we are aware of the basic building blocks of neural networks, let's consider why we prefer these models over techniques explored in previous chapters. The key understanding is that the models previously discussed are fundamentally *linear* in nature. For instance, if we do binary classification, where the data point $\vec{\mathbf{x}}$ is mapped to a label based on $\text{sign}(\vec{\mathbf{w}} \cdot \vec{\mathbf{x}})$, then this corresponds to separating the points with label $+1$ from the points with label -1 via a linear hyperplane $\vec{\mathbf{w}} \cdot \vec{\mathbf{x}} = 0$. But such models are not a good choice for datasets which are not linearly separable. Deep learning is inherently *nonlinear* and is able to do classification in many settings where linear classification cannot work.

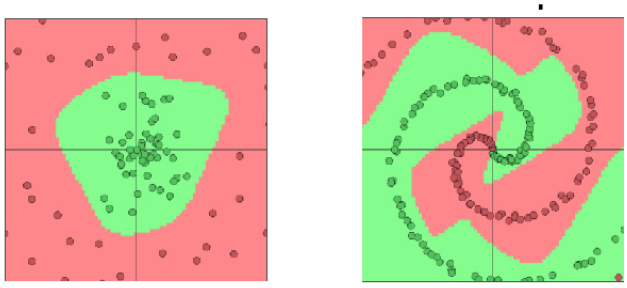


Figure 10.4: Some examples of datasets that are not linearly separable.

10.3.1 The XOR Problem

Consider the boolean function *XOR* with the truth table in Table 10.2.

x_1	x_2	**XOR**
0	0	0
0	1	1
1	0	1
1	1	0

Table 10.2: The truth table for the *XOR* Boolean function.

Let us first attempt to represent the *XOR* function with a single linear neuron. That is, consider a neuron that takes two inputs x_1, x_2 with weights w_1, w_2, a bias term b, and the following Heaveside step activation function: [12]

$$g(z) = \begin{cases} 0 & \text{if } z \leq 0 \\ 1 & \text{if } z > 0 \end{cases} \tag{10.5}$$

[12] This neuron is called a *linear perceptron*. It uses a nonlinear activation function, but the nonlinearity is strictly for the binary classification in the final step. The boundary of the classification is still linear.

Proposition 10.3.1. *There are no values of w_1, w_2, b such that the linear neuron defined by the values represent the XOR function.*

Proof. Assume to the contrary that there are such values. Let $\vec{\mathbf{x}}_1 = (0,0), \vec{\mathbf{x}}_2 = (0,1), \vec{\mathbf{x}}_3 = (1,0), \vec{\mathbf{x}}_4 = (1,1)$. Then we know that

$$g(\vec{\mathbf{w}} \cdot \vec{\mathbf{x}}_1 + b) = g(\vec{\mathbf{w}} \cdot \vec{\mathbf{x}}_4 + b) = 0$$
$$g(\vec{\mathbf{w}} \cdot \vec{\mathbf{x}}_2 + b) = g(\vec{\mathbf{w}} \cdot \vec{\mathbf{x}}_3 + b) = 1$$

which implies that

$$\vec{\mathbf{w}} \cdot \vec{\mathbf{x}}_1 + b \leq 0, \quad \vec{\mathbf{w}} \cdot \vec{\mathbf{x}}_4 + b \leq 0$$
$$\vec{\mathbf{w}} \cdot \vec{\mathbf{x}}_2 + b > 0, \quad \vec{\mathbf{w}} \cdot \vec{\mathbf{x}}_3 + b > 0$$

Now let $\vec{\mathbf{x}} = \left(\frac{1}{2}, \frac{1}{2}\right)$. Since we have $\vec{\mathbf{x}} = \frac{1}{2}\vec{\mathbf{x}}_1 + \frac{1}{2}\vec{\mathbf{x}}_4$, we should have

$$\vec{\mathbf{w}} \cdot \vec{\mathbf{x}} + b = \frac{1}{2} \cdot ((\vec{\mathbf{w}} \cdot \vec{\mathbf{x}}_1 + b) + (\vec{\mathbf{w}} \cdot \vec{\mathbf{x}}_4 + b)) \leq 0$$

since we are taking the average of two non-positive numbers. But at the same time, since $\vec{\mathbf{x}} = \frac{1}{2}\vec{\mathbf{x}}_2 + \frac{1}{2}\vec{\mathbf{x}}_3$, we should have

$$\vec{\mathbf{w}} \cdot \vec{\mathbf{x}} + b = \frac{1}{2} \cdot ((\vec{\mathbf{w}} \cdot \vec{\mathbf{x}}_2 + b) + (\vec{\mathbf{w}} \cdot \vec{\mathbf{x}}_3 + b)) > 0$$

since we are taking the average of two positive numbers. This leads to a contradiction. □

Problem 10.3.2. *Verify that the AND, OR Boolean functions can be represented by a single linear node.*

Figure 10.5 visualizes the truth table for *XOR* in the 2*D* plane. The two axes represent the two inputs x_1 and x_2; blue circles denote that $y = 1$; and white circles denote that $y = 0$. A single linear neuron can be understood as drawing a red line that can separate white points from blue points. Notice that it is possible to draw such a line for *AND* and *OR* functions, but the data points that characterize the *XOR* function are not linearly separable.

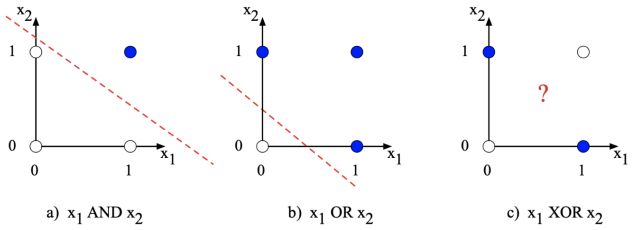


Figure 10.5: The data points that characterize the *XOR* function are not linearly separable.

Instead, we will leverage neural networks to solve this problem. Let us design an architecture with inputs x_1, x_2, a hidden layer with two nodes h_1, h_2, and a final output layer with one node y_1. We assign the *ReLU* activation function to the hidden nodes and define weights and biases as shown in Figure 10.6.

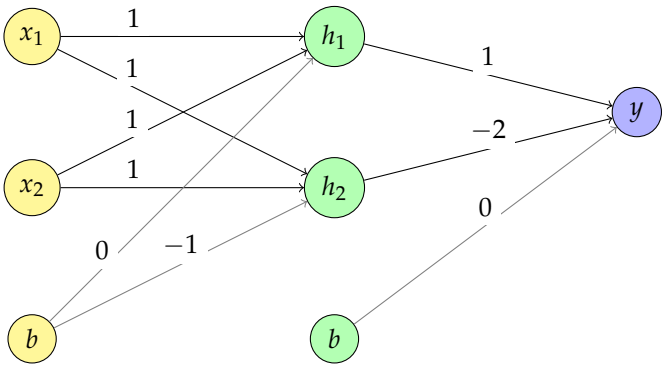


Figure 10.6: A sample neural network which computes the *XOR* of its inputs x_1 and x_2. The weights for inputs are shown by black arrows, while bias terms are shown by grey arrows.

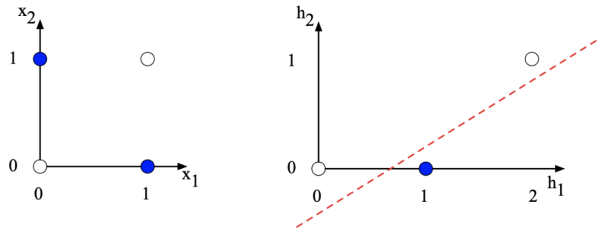
To be more explicit, the neural network is defined by the following three neurons:

$$h_1 = ReLU(x_1 + x_2)$$
$$h_2 = ReLU(x_1 + x_2 - 1)$$
$$y_1 = ReLU(h_1 - 2h_2)$$

Problem 10.3.3. *Verify that the model in Figure 10.6 represents the XOR function by constructing a truth table.*

The main difference between the single linear neuron approach and the neural network for the *XOR* function is that the network now

has two layers of neurons. If we only focus on the final layer of the neural network, we expect the boundary of the binary classification to be linear to the values of h_1, h_2. However, the values of h_1, h_2 are *not* linear to the input values x_1, x_2 because the hidden nodes utilize a *nonlinear* activation function. Hence the boundary of the classification is also *not* linear to the input values x_1, x_2. The nonlinear activation function transforms the input space into a space where the *XOR* operation is *linearly separable*. As shown in Figure 10.7, the h space is quite clearly linearly separable in contrast to the original x space.



a) The original x space b) The new (linearly separable) h space

Figure 10.7: Unlike the x space, after applying the nonlinear *ReLU* activation function, the mapped h space is linearly separable.

10.4 Multi-class Classification

Neural networks, like multi-class regression in Chapter 4, can be used for classification tasks where the number of possible labels is larger than 2. Real-life scenarios include hand-written digit recognition on the MNIST dataset, where the model designer could use ten different classes to correspond to each possible digit. Another possible example is an speech recognition language model, where the model is trained to distinguish between sounds of $|V|$ vocabularies.

It turns out that such functionality can be added by simply including as many output neurons as desired classes in the output layer. Then, the values of the output neurons will be converted into a probability distribution $\Pr[y = k]$ over the number of classes.

10.4.1 Softmax Function

Just as in Chapter 4, we use the softmax function for the purpose of the multi-class classification. See Chapter 19 for the definition of the softmax function.

Example 10.4.1. *Say $\vec{\mathbf{o}} = (3, 0, 1)$ are the values of the output neurons of a neural network before applying the activation function. If we decide to apply the softmax function as the activation function, the final outputs of the*

network will be $softmax(\vec{\mathbf{o}}) \simeq (0.84, 0.04, 0.11)$. *If the network was trying to solve a multi-class classification task, we can understand that the given input is most likely to be of class* 1, *with probability* 0.84 *according to the model.*

One notable property of the softmax function is that the output of the function is the same if all coordinates of the input vector is shifted by the same amount; that is $softmax(\vec{\mathbf{z}}) = softmax(\vec{\mathbf{z}} + c \cdot \vec{\mathbf{1}})$ for any $c \in \mathbb{R}$, where $\vec{\mathbf{1}} = (1, 1, \ldots, 1)$ is a vector of all ones.

Example 10.4.2. *Consider two vectors* $\vec{\mathbf{z}}_1 = (5, 2, 3)$ *and* $\vec{\mathbf{z}}_2 = (3, 0, 1)$. *Then* $softmax(\vec{\mathbf{z}}_1) = softmax(\vec{\mathbf{z}}_2)$ *because* $\vec{\mathbf{z}}_2 = \vec{\mathbf{z}}_1 - (2, 2, 2)$.

Problem 10.4.3. *Prove the property that* $softmax(\vec{\mathbf{z}}) = softmax(\vec{\mathbf{z}} + c \cdot \vec{\mathbf{1}})$ *for any* $c \in \mathbb{R}$. *(Hint: multiply both the numerator and the denominator of* $softmax(\vec{\mathbf{z}})_k$ *by* $\exp(c)$.*)*

11
Feedforward Neural Network and Backpropagation

Feedforward Neural Networks (FFNNs) are perhaps the simplest kind of deep nets and are characterized by the following properties:

- There are nodes connected with no cycles.
- Nodes are partitioned into layers numbered 1 to k for some k. The nodes in the first layer receive input of the model and output some values. Then the nodes in layer $i+1$ receive output of the nodes in layer i as their input and output some values. The output of the model can be computed with the output of the nodes in layer k.
- No outputs are passed back to lower layers.

Now, we only consider fully-connected layers — a special case of a layer in feedforward neural networks.

Definition 11.0.1 (Fully-Connected Layer)**.** *A **fully-connected layer** is a neural network layer in which all the nodes from one layer are fully connected to every node of the next layer.*

Note that not all layers of feedforward neural networks are necessarily fully-connected (a typical case is a Convolutional Neural Network, which we will explore in Chapter 12). However, feedforward neural networks with fully-connected layers are very common and also easy to implement.

11.1 Forward Propagation: An Example

Forward propagation refers to how the network converts a specific input to the output, specifically the calculation and storage of intermediate variables from the input layer to the output layer. In this section, we use concrete examples to motivate the topic. We will provide a more general formula in the next section. Readers who have a stronger background in math may feel to skip this section altogether.

11.1.1 One Output Node

We start with the network in Figure 11.1 as an example. The network receives three inputs x_1, x_2, x_3 and has a first hidden layer with two nodes $h_1^{(1)}, h_2^{(1)}$, a second hidden layer with two nodes $h_1^{(2)}, h_2^{(2)}$, and a final output layer with one node o. We assign the *ReLU* activation function to the hidden units, and define weights as shown in Figure 11.1.

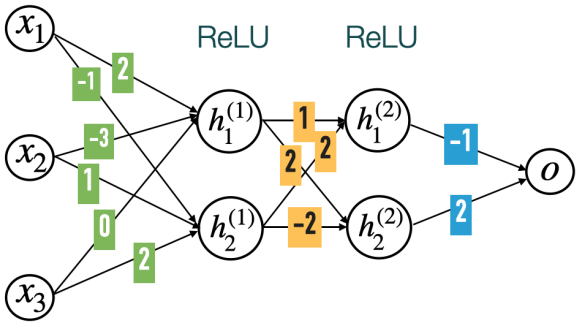


Figure 11.1: A sample feedforward neural network with two hidden layers and one output node.

The two hidden nodes in the first hidden layer are characterized by the following equations:

$$\begin{aligned} h_1^{(1)} &= ReLU(2x_1 - 3x_2) \\ h_2^{(1)} &= ReLU(-x_1 + x_2 + 2x_3) \end{aligned} \tag{11.1}$$

and the two hidden nodes in the second hidden layer are characterized by the following equations:

$$\begin{aligned} h_1^{(2)} &= ReLU(h_1^{(1)} + 2h_2^{(1)}) \\ h_2^{(2)} &= ReLU(2h_1^{(1)} - 2h_2^{(1)}) \end{aligned} \tag{11.2}$$

and the output node is characterized by the following equation:

$$o = -h_1^{(2)} + 2h_2^{(2)}$$

Therefore, if we know the input values x_1, x_2, x_3, we can first calculate the values $h_1^{(1)}, h_2^{(1)}$, then using these values, calculate $h_1^{(2)}, h_2^{(2)}$, and finally using these values, we can calculate the output o of the network.

Example 11.1.1. *If the provided input vector to the neural network in Figure 11.1 is $\vec{\mathbf{x}} = (1, 1, 1)$, we can calculate the first hidden layer as*

$$\begin{aligned} h_1^{(1)} &= ReLU(2 - 3) = 0 \\ h_2^{(1)} &= ReLU(-1 + 1 + 2) = 2 \end{aligned}$$

and the second hidden layer as

$$h_1^{(2)} = ReLU(0 + 2 \cdot 2) = 4$$
$$h_2^{(2)} = ReLU(0 - 2 \cdot 2) = 0$$

and the output as

$$o = -4 + 0 = -4$$

11.1.2 *Multiple Output Nodes*

Networks can have more than one output node. An example is the network in Figure 11.2.

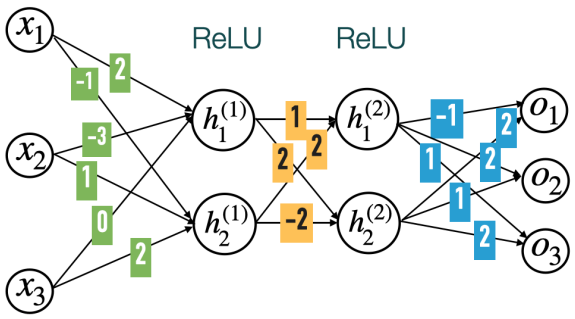


Figure 11.2: A sample feedforward neural network with two hidden layers and three output nodes.

The networks in Figure 11.1 and Figure 11.2 are the same except for the output layer; the former has one output node, while the latter has three output nodes. Now the output values of the network in Figure 11.2 can be calculated as:

$$\begin{aligned} o_1 &= -h_1^{(2)} + 2h_2^{(2)} \\ o_2 &= 2h_1^{(2)} + h_2^{(2)} \\ o_3 &= h_1^{(2)} + 2h_2^{(2)} \end{aligned} \tag{11.3}$$

Recall from the previous Chapter 10 that a FFNN with multiple output nodes is used for multi-class classfication. After the naive output values are calculated, the output nodes will use the softmax activation function to transform the values into the probabilities for each of the three classes. That is, the probability for predicting each class will be calculated as:

$$\begin{aligned} \hat{o}_1 &= softmax(o_1, o_2, o_3)_1 \\ \hat{o}_2 &= softmax(o_1, o_2, o_3)_2 \\ \hat{o}_3 &= softmax(o_1, o_2, o_3)_3 \end{aligned} \tag{11.4}$$

Example 11.1.2. *If the provided input vector to the neural network in Figure 11.1 is* $\vec{\mathbf{x}} = (1, 1, 1)$*, we can calculate the output layer as*

$$\begin{aligned} o_1 &= -4 + 0 = -4 \\ o_2 &= 2 \cdot 4 + 0 = 8 \\ o_3 &= 4 + 0 = 4 \end{aligned}$$

and the probabilities of each class as

$$\begin{aligned} \widehat{o}_1 &= softmax(-4, 8, 4)_1 = \frac{e^{-4}}{e^{-4} + e^{8} + e^{4}} \simeq 0.00 \\ \widehat{o}_2 &= softmax(-4, 8, 4)_2 = \frac{e^{8}}{e^{-4} + e^{8} + e^{4}} \simeq 0.98 \\ \widehat{o}_3 &= softmax(-4, 8, 4)_3 = \frac{e^{4}}{e^{-4} + e^{8} + e^{4}} \simeq 0.02 \end{aligned}$$

11.1.3 Matrix Notation

Let $w_{i,j}^{(1)}$ be the weight between the i-th node $h_i^{(1)}$ in the first hidden layer and the j-th input x_j. Then (11.1) can be rewritten as

$$\begin{aligned} h_1^{(1)} &= ReLU(w_{1,1}^{(1)} x_1 + w_{1,2}^{(1)} x_2 + w_{1,3}^{(1)} x_3) \\ h_2^{(1)} &= ReLU(w_{2,1}^{(1)} x_1 + w_{2,2}^{(1)} x_2 + w_{2,3}^{(1)} x_3) \end{aligned}$$

Notice that if we set $\vec{\mathbf{x}} = (x_1, x_2, x_3) \in \mathbb{R}^3$ and $\vec{\mathbf{h}}^{(1)} = (h_1^{(1)}, h_2^{(1)}) \in \mathbb{R}^2$ and define a matrix $\mathbf{W}^{(1)} \in \mathbb{R}^{2\times 3}$ where its (i, j) entry is $w_{i,j}^{(1)}$, then we can further rewrite (11.1) as [1]

$$\vec{\mathbf{h}}^{(1)} = ReLU\left(\mathbf{W}^{(1)} \vec{\mathbf{x}}\right) \tag{11.5}$$

[1] Here we interpret the vectors $\vec{\mathbf{x}}$, $\vec{\mathbf{h}}^{(1)}$ as a column vector, or equivalently a 3×1 matrix and a 2×1 matrix respectively. This will be a convention throughout this chapter.

where the $ReLU$ function is applied element-wise.

Similarly, if we let $w_{i,j}^{(2)}$ be the weight between the i-th node $h_i^{(2)}$ in the second hidden layer and the j-th node $h_j^{(1)}$ in the first hidden layer, (11.2) can be rewritten as

$$\begin{aligned} h_1^{(2)} &= ReLU(w_{1,1}^{(2)} h_1^{(1)} + w_{1,2}^{(2)} h_2^{(1)}) \\ h_2^{(2)} &= ReLU(w_{2,1}^{(2)} h_1^{(1)} + w_{2,2}^{(2)} h_2^{(1)}) \end{aligned}$$

or in a matrix notation as

$$\vec{\mathbf{h}}^{(2)} = ReLU\left(\mathbf{W}^{(2)} \vec{\mathbf{h}}^{(1)}\right) \tag{11.6}$$

where $\vec{\mathbf{h}}^{(2)} = (h_1^{(2)}, h_2^{(2)}) \in \mathbb{R}^2$ and $\mathbf{W}^{(2)} \in \mathbb{R}^{2\times 2}$ is a matrix whose (i, j) entry is $w_{i,j}^{(2)}$.

Next, if we let $w_{i,j}^{(o)}$ be the weight between the i-th output node o_i (before softmax) and the j-th node $h_j^{(2)}$ in the second hidden layer, (11.3) can be rewritten as

$$\begin{aligned} o_1 &= w_{1,1}^{(o)} h_1^{(2)} + w_{1,2}^{(o)} h_2^{(2)} \\ o_2 &= w_{2,1}^{(o)} h_1^{(2)} + w_{2,2}^{(o)} h_2^{(2)} \\ o_3 &= w_{3,1}^{(o)} h_1^{(2)} + w_{3,2}^{(o)} h_2^{(2)} \end{aligned}$$

or in a matrix notation as

$$\vec{\mathbf{o}} = \mathbf{W}^{(o)} \vec{\mathbf{h}}^{(2)} \tag{11.7}$$

where $\vec{\mathbf{o}} = (o_1, o_2, o_3) \in \mathbb{R}^3$ and $\mathbf{W}^{(o)} \in \mathbb{R}^{3\times 2}$ is a matrix whose (i, j) entry is $w_{i,j}^{(o)}$.

Finally, if we let $\hat{\vec{\mathbf{o}}} = (\hat{o}_1, \hat{o}_2, \hat{o}_3) \in \mathbb{R}^3$, then (11.4) can be rewritten as

$$\hat{\vec{\mathbf{o}}} = softmax(\vec{\mathbf{o}}) \tag{11.8}$$

We summarize the results above into the following matrix equations

$$\begin{aligned} \vec{\mathbf{h}}^{(1)} &= ReLU\left(\mathbf{W}^{(1)} \vec{\mathbf{x}}\right) \\ \vec{\mathbf{h}}^{(2)} &= ReLU\left(\mathbf{W}^{(2)} \vec{\mathbf{h}}^{(1)}\right) \\ \vec{\mathbf{o}} &= \mathbf{W}^{(o)} \vec{\mathbf{h}}^{(2)} \\ \hat{\vec{\mathbf{o}}} &= softmax(\vec{\mathbf{o}}) \end{aligned} \tag{11.9}$$

Example 11.1.3. *If the provided input vector to the neural network in Figure 11.2 is $\vec{\mathbf{x}} = (1, 1, 1)$, we can calculate the first hidden layer as*

$$\vec{\mathbf{h}}^{(1)} = \mathbf{W}^{(1)} \vec{\mathbf{x}} = ReLU\left(\begin{bmatrix} 2 & -3 & 0 \\ -1 & 1 & 2 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}\right) = \begin{bmatrix} 0 \\ 2 \end{bmatrix}$$

and the second hidden layer as

$$\vec{\mathbf{h}}^{(2)} = \mathbf{W}^{(2)} \vec{\mathbf{h}}^{(1)} = ReLU\left(\begin{bmatrix} 1 & 2 \\ 2 & -2 \end{bmatrix} \begin{bmatrix} 0 \\ 2 \end{bmatrix}\right) = \begin{bmatrix} 4 \\ 0 \end{bmatrix}$$

and the output layer $\vec{\mathbf{o}}$ (before the softmax) as

$$\vec{\mathbf{o}} = \mathbf{W}^{(o)} \vec{\mathbf{h}}^{(2)} = \begin{bmatrix} -1 & 2 \\ 2 & 1 \\ 1 & 2 \end{bmatrix} \begin{bmatrix} 4 \\ 0 \end{bmatrix} = \begin{bmatrix} -4 \\ 8 \\ 4 \end{bmatrix}$$

The probability distribution $\hat{\vec{\mathbf{o}}}$ of the three classes can then be calculated as

$$\hat{\vec{\mathbf{o}}} = softmax(\vec{\mathbf{o}}) = \left(\frac{e^{-4}}{e^{-4} + e^{8} + e^{4}}, \frac{e^{8}}{e^{-4} + e^{8} + e^{4}}, \frac{e^{4}}{e^{-4} + e^{8} + e^{4}}\right)$$

11.2 Forward Propagation: The General Case

We now consider an arbitrary feedforward neural network with $L \geq 1$ layers. Let $\vec{\mathbf{x}} \in \mathbb{R}^{d_0}$ be the vector of d_0 *inputs* to the network. For $k = 1, 2, \ldots, L$, let $\vec{\mathbf{h}}^{(k)} = (h_1^{(k)}, h_2^{(k)}, \ldots, h_{d_k}^{(k)}) \in \mathbb{R}^{d_k}$ represent the d_k nodes of the *k-th hidden layer*. The L-th hidden layer is also known as the *output layer*, and we alternatively denote $d_0 = d_{in}$ and $d_L = d_{out}$ to emphasize that they are respectively the number of inputs and the number of output nodes.

Additionally, we consider $\mathbf{W}^{(k)} \in \mathbb{R}^{d_k \times d_{k-1}}$ to represent the weights for the k-th hidden layer. Its (i, j) entry is the weight between the i-th node $h_i^{(k)}$ of the k-th hidden layer and the j-th node $h_j^{(k-1)}$ of the $(k-1)$-th hidden layer. We also alternatively denote $\mathbf{W}^{(L)} = \mathbf{W}^{(o)}$ to emphasize that it represents the weights for the output layer.

Finally, let $f^{(k)}$ be the nonlinear activation function for layer k. For instance, consider the output layer. If $d_{out} = 1$ (*i.e.*, there is one output node), we can assume that $f^{(L)}$ is the identity function. On the other hand, if $d_{out} > 1$ (*i.e.*, there are multiple output nodes), we can assume that $f^{(L)}$ is the softmax function. It is also possible to use different activation functions for each layer.

With all these new notations in mind, we can express the nodes of layer k as:

$$\vec{\mathbf{h}}^{(k)} = f^{(k)}(\mathbf{W}^{(k)}\vec{\mathbf{h}}^{(k-1)})$$

for each $k = 1, 2, \ldots, L$.

If $d_{out} = 1$, we let $o = \mathbf{W}^{(L)}\vec{\mathbf{h}}^{L-1}$ denote the final output of the model. If $d_{out} > 1$, we let $\vec{\mathbf{o}} = \mathbf{W}^{(L)}\vec{\mathbf{h}}^{L-1}$ denote the output layer before the softmax and $\vec{\hat{\mathbf{o}}} = f^{(L)}(\vec{\mathbf{o}})$ denote the output layer after the softmax.

11.2.1 Number of Weights

We now briefly consider the number of weights in a feedforward network. There are $d_{in} \cdot d_1$ weights (or variables) for the first hidden layer. Similarly, there are $d_1 \cdot d_2$ weights for the second hidden layer. In total, the number of weights is $\sum_{i=0}^{L-1} d_i \cdot d_{i+1}$.

Example 11.2.1. *The number of weights in the model in Figure 11.2 can be calculated as*

$$3 \times 2 + 2 \times 2 + 2 \times 3 = 16$$

11.2.2 What If We Remove Nonlinearity?

If we removed the nonlinear activation function *ReLU* in our model from (11.9), we would have the following forward propagation equa-

tions:

$$\vec{\mathbf{h}}^{(1)} = \mathbf{W}^{(1)}\vec{\mathbf{x}}$$
$$\vec{\mathbf{h}}^{(2)} = \mathbf{W}^{(2)}\vec{\mathbf{h}}^{(1)}$$
$$\vec{\mathbf{o}} = \mathbf{W}^{(o)}\vec{\mathbf{h}}^{(2)}$$
$$\vec{\hat{\mathbf{o}}} = softmax(\vec{\mathbf{o}})$$

Notice that if we set $\mathbf{W}' = \mathbf{W}^{(o)}\mathbf{W}^{(2)}\mathbf{W}^{(1)} \in \mathbb{R}^{3\times 3}$, then

$$\vec{\hat{\mathbf{o}}} = softmax(\mathbf{W}'\vec{\mathbf{x}})$$

We have thus reduced our neural net to a standard multi-classification logistic regression model! As we have discussed the limitation of linear models earlier, this indicates the importance of having nonlinear activation functions between layers.

11.2.3 Training Loss

Just like we have defined a loss function for ML models so far, we also define an appropriate loss function for neural networks, where the objective of the network becomes finding a set of weights that minimize the loss. While there are many different definitions of loss functions, here we present two — one that is more appropriate when there is a single output node, and another that is more appropriate for multi-class classification.

When there is only one scalar node in the output layer (*i.e.*, $d_{out} = 1$), we can use a *squared error loss*, similar to the least squares loss from (1.4):

$$\sum_{(\vec{\mathbf{x}},y)\in\mathcal{D}} (y - o)^2 \qquad \text{(Squared Error Loss)}$$

where $\vec{\mathbf{x}} \in \mathbb{R}^{d_{in}}$ is the input vector, y is its gold value (*i.e.*, actual value in the training data), and $o = \mathbf{W}^{(o)}\vec{\mathbf{h}}^{(L-1)}$ is the final output (*i.e.*, prediction) of the neural network.

Example 11.2.2. *If the provided input vector to the neural network in Figure 11.1 is $\vec{\mathbf{x}} = (1, 1, 1)$ and the training output is $y = 0$, we can calculate the squared error loss as*

$$(y - o)^2 = (0 - (-4))^2 = 16$$

When there are multiple nodes in the output layer (*e.g.*, for multi-class classification), we can use a loss function that is similar to the logistic loss. Recall that in logistic regression, we defined the logistic loss as a sum of log loss over a set of data points:

$$\sum_{(\vec{\mathbf{x}},y)\in\mathcal{D}} -\log \Pr[\text{label } y \text{ on input } \vec{\mathbf{x}}] \qquad \text{(4.5 revisited)}$$

where $y \in \{-1, 1\}$ denotes the gold label. For neural networks, we can analogously define the *cross-entropy loss*:

$$\sum_{(\vec{\mathbf{x}},y)\in\mathcal{D}} -\log \widehat{o}_y \qquad \text{(Cross-Entropy Loss)}$$

where $y \in \{1, \ldots, d_{out}\}$ denotes the gold label, and $\widehat{o}_y$ denotes the probability that the model assigns to class y — that is, the y-th coordinate of the output vector $\vec{\widehat{\mathbf{o}}} = softmax(\vec{\mathbf{o}})$ after applying the softmax function.

Example 11.2.3. *If the provided input vector to the neural network in Figure 11.2 is $\vec{\mathbf{x}} = (1, 1, 1)$ and the training output is $y = 2$, we can calculate the cross-entropy loss for this data point as*

$$-\log \widehat{o}_y = -\log \frac{e^8}{e^{-4} + e^8 + e^4} \simeq 4.02$$

11.3 Backpropagation: An Example

Just like in previous ML models we have learned, the process of training a neural network model involves three steps:

1. Defining an an appropriate loss function.
2. Calculating the gradient of the loss with respect to the training data and the model parameters.
3. Iteratively updating the parameters via the gradient descent algorithm.

But once a neural network grows in size, the second step of calculating the gradients starts to become a problem. Naively trying to calculate each of the gradients separately becomes inefficient. Instead, *Backpropagation* [2] is an efficient way to calculate the gradients with respect to the network parameters such that the number of steps for the computation is *linear* in the size of the nueral network.

The key idea is to perform the computation in a very specific sequence — from the output layer to the input layer. By the Chain Rule, we can use the already computed gradient value of a node in a higher layer in the computation of the gradient of a node in a lower layer. This way, the gradient values *propagate* back from the top layer to the bottom layer.

[2] Reference: https://www.nature.com/articles/323533a0

11.3.1 The Delta Method: Reasoning from First Principles

The main goal of backpropagation is to compute the partial derivative $\partial f / \partial w$ where f is the loss and w is the weight of an edge. This

will allow us to apply the gradient descent algorithm and appropriately update the weight w. Students often find backpropagation a confusing idea, but it is actually just a simple application of Chain Rule in multivariate calculus.

In this subsection, we motivate the topic with the *Delta Method*, which is an intuitive way to compute $\partial f/\partial w$. We perturb a weight w by a small amount Δ and measure how much the output changes. In doing so, we also measure how the rest of the network changes. As we will see later, the process of computing the partial derivative of the form $\partial f/\partial w$ requires us to also compute the partial derivative of the form $\partial f/\partial h$ where h is the value at a node.

Readers who are familiar with Chain Rule can quickly browse through the rest of this subsection.

Example 11.3.1. *Consider the model from Figure 11.2 but now with the inputs $\vec{\mathbf{x}} = (3, 1, 2)$. We use the same notation for the nodes and the weights that we used throughout Section 11.1. The goal of this simple example is to illustrate what the derivatives mean.*

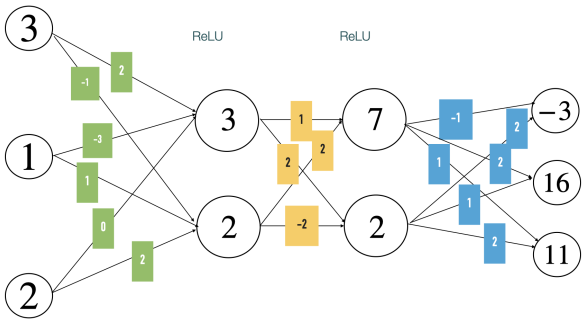


Figure 11.3: The model from Figure 11.2 with inputs $\vec{\mathbf{x}} = (3, 1, 2)$.

Suppose we want to take partial derivative of first output node o_1 with respect to the weight $w^{(1)}_{2,2}$ (i.e., the weight on the edge between the second input x_2 and the second node $h^{(1)}_2$ of first hidden layer). This is denoted as $\partial o_1/\partial w^{(1)}_{2,2}$. Its definition involves considering how changing $w^{(1)}_{2,2}$ by an infinitesimal amount Δ changes o_1, whose current value is -3.

Adding Δ to $w^{(1)}_{2,2}$ will change the values of the first hidden layer to

$$h^{(1)}_1 = ReLU(2 \cdot 3 + (-3) \cdot 1 + 0 \cdot 2) = 3$$
$$h^{(1)}_2 = ReLU((-1) \cdot 3 + (1 + \Delta) \cdot 1 + 2 \cdot 2) = 2 + \Delta$$

Letting $\Delta \to 0$, we see that the rate of change of $h^{(1)}_1$ and $h^{(1)}_2$ with respect to change of $w^{(1)}_{2,2}$ is 0 and 1 respectively. In more formal terms, $\partial h^{(1)}_1/\partial w^{(1)}_{2,2} = 0$ and $\partial h^{(1)}_2/\partial w^{(1)}_{2,2} = 1$.

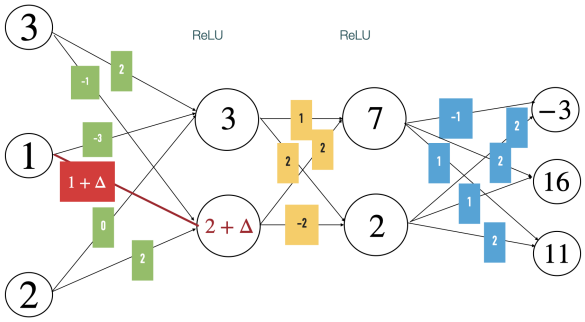


Figure 11.4: The model from Figure 11.2 with inputs $\vec{x} = (3, 1, 2)$. When $w^{(1)}_{2,2}$ is changed by a small number Δ, the first hidden layer is also affected.

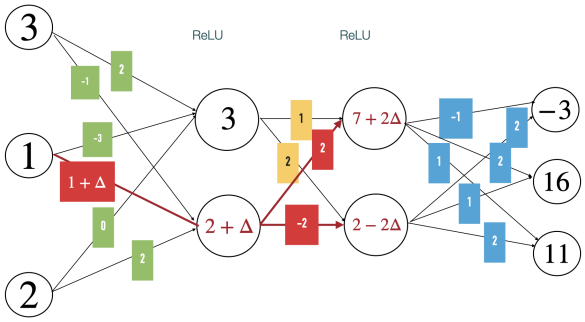


Figure 11.5: The model from Figure 11.2 with inputs $\vec{x} = (3, 1, 2)$. When $w^{(1)}_{2,2}$ is changed by a small number Δ, the second hidden layer is also affected.

Using the updated values of the first hidden layer, the second hidden layer will be calculated as

$$h^{(2)}_1 = ReLU(1 \cdot 3 + 2 \cdot (2 + \Delta)) = 7 + 2\Delta$$
$$h^{(2)}_2 = ReLU(2 \cdot 3 + (-2) \cdot (2 + \Delta)) = 2 - 2\Delta$$

This shows that the rate of change of $h^{(2)}_1$ and $h^{(2)}_2$ with respect to change of $w^{(1)}_{2,2}$ is 2 and -2 respectively.

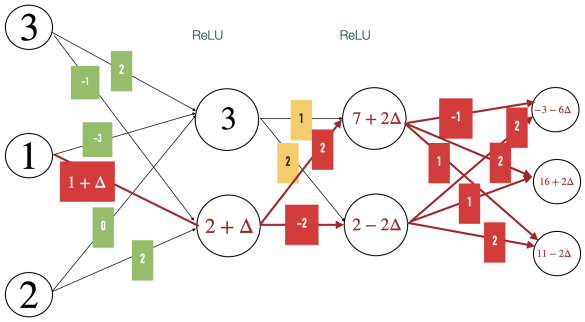


Figure 11.6: The model from Figure 11.2 with inputs $\vec{x} = (3, 1, 2)$. When $w^{(1)}_{2,2}$ is changed by a small number Δ, the output layer is also affected.

Finally the output layer can now be calculated as

$$o_1 = (-1)\cdot(7+2\Delta)+2\cdot(2-2\Delta) = -3-6\Delta$$
$$o_2 = 2\cdot(7+2\Delta)+1\cdot(2-2\Delta) = 16+2\Delta$$
$$o_3 = 1\cdot(7+2\Delta)+2\cdot(2-2\Delta) = 11-2\Delta$$

This shows that the rate of change of o_1 with respect to change of $w^{(1)}_{2,2}$ is -6.

Example 11.3.2. *Now we consider the meaning of $\partial o_1/\partial h^{(2)}_1$: how changing the value of $h^{(2)}_1$ by an infinitesimal Δ affects o_1. Note that this is a thought experiment that does not correspond to a change that is possible if the net were a physical object constructed of nodes and wires — the value of $h^{(2)}_1$ is completely decided by the previous layers and cannot be changed in isolation without touching the previous layers. However, treating these values as variables, it is possible to put on a calculus hat and and think about the rate of change of one with respect to the other.*

If only the value of $h^{(2)}_1$ is changed from 7 to $7+\Delta$ in Figure 11.3, then o_1 can be calculated as

$$o_1 = (-1)\cdot(7+\Delta)+2\cdot 2 = -3-\Delta$$

which shows that $\partial o_1/\partial h^{(2)}_1 = -1$.

Problem 11.3.3. *Following the calculations in Example 11.3.1 and Example 11.3.2, calculate $\partial o_1/\partial h^{(2)}_2$, $\partial h^{(2)}_1/\partial w^{(1)}_{2,2}$, and $\partial h^{(2)}_2/\partial w^{(1)}_{2,2}$. Verify that*

$$\frac{\partial o_1}{\partial w^{(1)}_{2,2}} = \frac{\partial o_1}{\partial h^{(2)}_1}\cdot\frac{\partial h^{(2)}_1}{\partial w^{(1)}_{2,2}} + \frac{\partial o_1}{\partial h^{(2)}_2}\cdot\frac{\partial h^{(2)}_2}{\partial w^{(1)}_{2,2}}$$

Problem 11.3.4. *Following the calculations in Example 11.3.1 and Example 11.3.2, calculate $\partial h^{(2)}_1/\partial h^{(1)}_2$, $\partial h^{(2)}_2/\partial h^{(1)}_2$, and $\partial h^{(1)}_2/\partial w^{(1)}_{2,2}$. Verify that*

$$\frac{\partial o_1}{\partial w^{(1)}_{2,2}} = \frac{\partial o_1}{\partial h^{(2)}_1}\cdot\frac{\partial h^{(2)}_1}{\partial h^{(1)}_2}\cdot\frac{\partial h^{(1)}_2}{\partial w^{(1)}_{2,2}} + \frac{\partial o_1}{\partial h^{(2)}_2}\cdot\frac{\partial h^{(2)}_2}{\partial h^{(1)}_2}\cdot\frac{\partial h^{(1)}_2}{\partial w^{(1)}_{2,2}} \tag{11.10}$$

Some readers may notice that (11.10) is just the result of chain rule in multivariable calculus. It shows that the effect of $w^{(1)}_{2,2}$ on o_1 is the sum of its effect through all possible paths that the values propagate through the network, and the amount of effect for each path can be calculated by multiplying the appropriate partial derivative between each layer.

In this section, we calculated by hand one partial derivative $\partial o_1/\partial w^{(1)}_{2,2}$. But in general, to compute the loss gradient, we see below that we want to calculate the partial derivative of each output node o_i with respect to each weight in the network. Manually applying chain rule for each partial derivative as in (11.10) is too inefficient.[3] Instead,

[3] Putting on your COS 226 hat, you can check that the computational cost of this naive method scales quadratically in the size of the network.

in the next section, we will learn how to utilize matrix operations to combine the computation for multiple partial derivatives into one process.[4]

[4] This efficiency holds even without taking into account the fact that today's GPUs are optimized for fast matrix operations.

11.4 Backpropagation: The General Case

11.4.1 Jacobian Matrix

Suppose some vector $\vec{\mathbf{y}} = (y_1, y_2, \ldots, y_m) \in \mathbb{R}^m$ is a function of $\vec{\mathbf{x}} = (x_1, x_2, \ldots, x_n) \in \mathbb{R}^n$ — that is, there is a mapping $f : \mathbb{R}^n \to \mathbb{R}^m$ such that $\vec{\mathbf{y}} = f(\vec{\mathbf{x}})$, or equivalently, if there are m functions $f_i : \mathbb{R}^n \to \mathbb{R}$ for each $i = 1, 2, \ldots, m$ such that $y_i = f_i(\vec{\mathbf{x}})$.

Then the *Jacobian matrix* of $\vec{\mathbf{y}}$ with respect to $\vec{\mathbf{x}}$, denoted as $J(\vec{\mathbf{y}}, \vec{\mathbf{x}})$, is an $m \times n$ matrix whose (i, j) entry is the partial derivative $\partial y_i / \partial x_j$. Note that each entry of this matrix is itself a function of $\vec{\mathbf{x}}$. A bit confusingly, a Jacobian matrix is also often denoted as $\partial \vec{\mathbf{y}} / \partial \vec{\mathbf{x}}$ when it is clear from the context that $\vec{\mathbf{x}}, \vec{\mathbf{y}}$ are vectors and hence this object is not a partial derivative or gradient. [5]

[5] Note that the i-th row of the Jacobian matrix contains the gradient of y_i, *i.e.* the i-th coordinate of $\vec{\mathbf{y}}$.

The mathematical interpretation of the Jacobian matrix is that if we change $\vec{\mathbf{x}}$ such that each coordinate x_i is updated to $x_i + \delta_i$ for an infinitesimal value δ_i, then the output $\vec{\mathbf{y}}$ changes to $\vec{\mathbf{y}} + J(\vec{\mathbf{y}}, \vec{\mathbf{x}})\vec{\boldsymbol{\delta}}$.

Example 11.4.1. *Suppose $\vec{\mathbf{y}}$ is a linear function of $\vec{\mathbf{x}}$ — that is, there exists a matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ such that $\vec{\mathbf{y}} = \mathbf{A}\vec{\mathbf{x}}$. Then notice that y_i, the i-th coordinate of $\vec{\mathbf{y}}$, can be expressed as*

$$y_i = \mathbf{A}_{i,*}\vec{\mathbf{x}} = A_{i,1}x_1 + A_{i,2}x_2 + \ldots + A_{i,n}x_n$$

Notice that the partial derivative $\partial y_i / \partial x_j$ is equal to A_{ij}. This means that the (i, j) entry of the Jacobian matrix is the (i, j) entry of the matrix $\mathbf{A}$, and hence $J(\vec{\mathbf{y}}, \vec{\mathbf{x}}) = \mathbf{A}$.

Problem 11.4.2. *If $\vec{\mathbf{y}} \in \mathbb{R}^2$ is a function of $\vec{\mathbf{x}} \in \mathbb{R}^3$ such that*

$$y_1 = 2x_1 - x_2 + 3x_3$$
$$y_2 = -x_1 + 2x_3$$

then what is the Jacobian matrix $J(\vec{\mathbf{y}}, \vec{\mathbf{x}})$?

Example 11.4.3. *If $\vec{\mathbf{x}} \in \mathbb{R}^n$ and $\vec{\mathbf{y}} = ReLU(\vec{\mathbf{x}}) \in \mathbb{R}^n$, then notice that*

$$\frac{\partial y_i}{\partial x_i} = \begin{cases} 1 & x_i > 0 \\ 0 & otherwise \end{cases}$$

We can also denote this with an indicator function $\mathbb{1}(x_i > 0)$. Also for any $j \neq i$, we see that $\partial y_i / \partial x_j = 0$. Therefore, the Jacobian matrix $J(\vec{\mathbf{y}}, \vec{\mathbf{x}})$ is a diagonal matrix whose entry down the diagonal is $\mathbb{1}(x_i > 0)$; that is

$$J(\vec{\mathbf{y}}, \vec{\mathbf{x}}) = diag(\mathbb{1}(\vec{\mathbf{x}} > 0))$$

where we take the indicator function element-wise to the vector $\vec{\mathbf{x}}$.

Definition 11.4.4 (Jacobian Chain Rule)**.** *Suppose vector* $\vec{\mathbf{z}} \in \mathbb{R}^\ell$ *is a function of* $\vec{\mathbf{y}} \in \mathbb{R}^m$ *and* $\vec{\mathbf{y}}$ *is a function of* $\vec{\mathbf{x}} \in \mathbb{R}^n$*, then by the chain rule, the Jacobian matrix* $J(\vec{\mathbf{z}}, \vec{\mathbf{x}}) \in \mathbb{R}^{\ell \times n}$ *is represented as the matrix product:*

$$J(\vec{\mathbf{z}}, \vec{\mathbf{x}}) = J(\vec{\mathbf{z}}, \vec{\mathbf{y}}) J(\vec{\mathbf{y}}, \vec{\mathbf{x}}) \tag{11.11}$$

In context of the feedforward neural network, each hidden layer is a function of the previous layer. Specifically, vector of activations of a hidden layer is a function of the vector of activations of the previous layer as well as of the trainable weights within the layer.

Example 11.4.5 (Gradient calculation for a single layer with ReLU's)**.** *If* $\vec{\mathbf{x}} \in \mathbb{R}^n, \mathbf{A} \in \mathbb{R}^{m \times n}, \vec{\mathbf{y}} = \mathbf{A}\vec{\mathbf{x}} \in \mathbb{R}^m$ *and* $\vec{\mathbf{z}} = ReLU(\vec{\mathbf{y}}) \in \mathbb{R}^m$*, then the Jacobian matrix* $J(\vec{\mathbf{z}}, \vec{\mathbf{x}})$ *can be calculated as*

$$J(\vec{\mathbf{z}}, \vec{\mathbf{x}}) = J(\vec{\mathbf{z}}, \vec{\mathbf{y}}) J(\vec{\mathbf{y}}, \vec{\mathbf{x}}) = diag(\mathbb{1}(\mathbf{A}\vec{\mathbf{x}} > 0))\mathbf{A}$$

11.4.2 *Backpropagation — Efficiency Using Jacobian Viewpoint*

Now we return to backpropagation, and show how the Jacobian viewpoint allows computing the gradient of the loss (with respect to network parameters) with a number of mathematical operations (*i. e.*, additions and multiplications) proportional to the size of the fully connected net.

Recall that we want to find the weights $\mathbf{W}^{(1)}, \mathbf{W}^{(2)}, \ldots, \mathbf{W}^{(o)}$ that minimize the cross-entropy loss ℓ. To apply the standard/stochastic gradient descent algorithm, we need to find the partial derivative $\frac{\partial \ell}{\partial \mathbf{W}^{(k)}_{i,j}}$ of the loss function with respect to each weight $\mathbf{W}^{(k)}_{i,j}$ of each layer k.

To simplify notations, we introduce a new matrix $\frac{\partial \ell}{\partial \mathbf{W}^{(k)}}$ which has the same dimensions as $\mathbf{W}^{(k)}$ (*e.g.*, $\frac{\partial \ell}{\partial \mathbf{W}^{(1)}} \in \mathbb{R}^{2 \times 3}$ in Figure 11.2) and the (i, j) entry of the matrix is:

$$\left(\frac{\partial \ell}{\partial \mathbf{W}^{(k)}}\right)_{i,j} = \frac{\partial \ell}{\partial \mathbf{W}^{(k)}_{i,j}} \tag{11.12}$$

for any layer k. The matrix $\frac{\partial \ell}{\partial \mathbf{W}^{(k)}}$ will be called the gradient with respect to the weights of the k-th layer. [6] Now the update rule for the gradient descent algorithm can be written as the following:

$$\mathbf{W}^{(k)} \to \mathbf{W}^{(k)} - \eta \cdot \frac{\partial \ell}{\partial \mathbf{W}^{(k)}} \tag{11.13}$$

where η is the learning rate. Now the question remains as how to calculate these gradients. As the name "backpropagation" suggests, we will first compute the gradient of the loss ℓ with respect to the output nodes; we then inductively compute the gradient for the previous layers, until we reach the input layer.

[6] Alternatively, you can think of flattening $\mathbf{W}^{(k)}$ into a single vector, then finding the Jacobian matrix $\partial \ell / \partial \mathbf{W}^{(k)}$, and later converting it back to a matrix form.

1. Output Layer: First recall that the cross-entropy loss is

$$\begin{aligned}\ell &= -\log\left(\frac{e^{o_y}}{\sum_{i=1}^{d_{out}} e^{o_i}}\right)\\ &= -\log(e^{o_y}) + \log\left(\sum_{i=1}^{d_{out}} e^{o_i}\right)\\ &= -o_y + \log\left(\sum_{i=1}^{d_{out}} e^{o_i}\right)\end{aligned}$$

where $y \in \{1, 2, \ldots, d_{out}\}$ is the ground truth value. Therefore, the gradient with respect to the output layer is

$$\frac{\partial \ell}{\partial o_i}_{1\le i\le d_{out}} = \begin{cases} -1 + \widehat{o}_i & y = i \\ \widehat{o}_i & y \neq i \end{cases}$$

To simplify notations, we introduce a *one-hot encoding vector* $\vec{\mathbf{e}}_y$, which has 1 only at the y-th coordinate and 0 everywhere else. Then, we can rewrite the equation above as: [7]

$$\frac{\partial \ell}{\partial \vec{\mathbf{o}}} = \left(\vec{\widehat{\mathbf{o}}} - \vec{\mathbf{e}}_y\right)^\intercal \in \mathbb{R}^{1\times d_{out}} \tag{11.14}$$

This is the Jacobian matrix of the loss ℓ with respect to the output layer $\vec{\mathbf{o}}$.

[7] Note that $\partial\ell/\partial\vec{\mathbf{o}}$, the term on the left hand side, is a Jacobian matrix in $\mathbb{R}^{1\times d_{out}}$. But $\vec{\widehat{\mathbf{o}}}$ and $\vec{\mathbf{e}}_y$, the terms on the right hand side, are both column vectors, or equivalently a $d_{out} \times 1$ matrix. We resolve the problem by taking the transpose.

2. Jacobian With Respect To Hidden Layer: We first compute $\partial\ell/\partial\vec{\mathbf{h}}^{(L-1)}$, the Jacobian matrix with respect to the last hidden layer before the output layer. Since $\vec{\mathbf{o}} = \mathbf{W}^{(o)}\vec{\mathbf{h}}^{(L-1)}$, we can apply the result from Example 11.4.1 and get

$$\begin{aligned}\frac{\partial \ell}{\partial \vec{\mathbf{h}}^{(L-1)}} &= \frac{\partial \ell}{\partial \vec{\mathbf{o}}} J(\vec{\mathbf{o}}, \vec{\mathbf{h}}^{(L-1)})\\ &= \frac{\partial \ell}{\partial \vec{\mathbf{o}}}\mathbf{W}^{(o)}\end{aligned} \tag{11.15}$$

Now as an inductive hypothesis, assume that we have already computed the gradient (or Jacobian matrix) $\partial\ell/\partial\vec{\mathbf{h}}^{(k+1)}$. We now compute $\partial\ell/\partial\vec{\mathbf{h}}^{(k)}$ using the result from Example 11.4.5.

$$\begin{aligned}\frac{\partial \ell}{\partial \vec{\mathbf{h}}^{(k)}} &= \frac{\partial \ell}{\partial \vec{\mathbf{h}}^{(k+1)}} J(\vec{\mathbf{h}}^{(k+1)}, \vec{\mathbf{h}}^{(k)})\\ &= \frac{\partial \ell}{\partial \vec{\mathbf{h}}^{(k+1)}} diag(\mathbb{1}(\mathbf{W}^{(k+1)}\vec{\mathbf{h}}^{(k)} > 0))\mathbf{W}^{(k+1)}\end{aligned} \tag{11.16}$$

3. Gradient With Respect to Weights: We first compute $\partial\ell/\partial\mathbf{W}^{(o)}$, the gradients with respect to the weights of the output layer. Notice that

a particular weight $w_{i,j}^{(o)}$ is only used in computing o_i out of all output nodes:

$$o_i = w_{i,1}^{(o)} h_1^{(L-1)} + \ldots + w_{i,j}^{(o)} h_j^{(L-1)} + \ldots + w_{i,d_{L-1}}^{(o)} h_{d_{L-1}}^{(L-1)}$$

Therefore, the gradient with respect to $w_{i,j}^{(o)}$ can be calculated as

$$\frac{\partial \ell}{\partial w_{i,j}^{(o)}} = \frac{\partial \ell}{\partial o_i} \cdot \frac{\partial o_i}{\partial w_{i,j}^{(o)}} = \frac{\partial \ell}{\partial o_i} \cdot h_j^{(L-1)}$$

We can combine these results into the following matrix form

$$\frac{\partial \ell}{\partial \mathbf{W}^{(o)}} = \left(\frac{\partial \ell}{\partial \vec{\mathbf{o}}}\right)^\intercal \left(\vec{\mathbf{h}}^{(L-1)}\right)^\intercal \tag{11.17}$$

Now as an inductive hypothesis, assume that we have already computed the gradient (or Jacobian) $\partial \ell / \partial \vec{\mathbf{h}}^{(k)}$. We now compute $\partial \ell / \partial \mathbf{W}^{(k)}$.

To do this, we introduce an intermediate variable $\vec{\mathbf{z}}^{(k)} = \mathbf{W}^{(k)} \vec{\mathbf{h}}^{(k-1)}$ such that $\vec{\mathbf{h}}^{(k)} = ReLU(\vec{\mathbf{z}}^{(k)})$. Then the gradient with respect to a particular weight $w_{i,j}^{(k)}$ can be calculated as

$$\frac{\partial \ell}{\partial w_{i,j}^{(k)}} = \frac{\partial \ell}{\partial z_i^{(k)}} \cdot \frac{\partial z_i^{(k)}}{\partial w_{i,j}^{(k)}} = \frac{\partial \ell}{\partial z_i^{(k)}} \cdot h_j^{(k-1)}$$

We can combine these results into the following matrix form

$$\begin{aligned}
\frac{\partial \ell}{\partial \mathbf{W}^{(k)}} &= \left(\frac{\partial \ell}{\partial \vec{\mathbf{z}}^{(k)}}\right)^\intercal \left(\vec{\mathbf{h}}^{(k-1)}\right)^\intercal \\
&= \left(\frac{\partial \ell}{\partial \vec{\mathbf{h}}^{(k)}} J(\vec{\mathbf{h}}^{(k)}, \vec{\mathbf{z}}^{(k)})\right)^\intercal \left(\vec{\mathbf{h}}^{(k-1)}\right)^\intercal \\
&= \left(J(\vec{\mathbf{h}}^{(k)}, \vec{\mathbf{z}}^{(k)})\right)^\intercal \left(\frac{\partial \ell}{\partial \vec{\mathbf{h}}^{(k)}}\right)^\intercal \left(\vec{\mathbf{h}}^{(k-1)}\right)^\intercal \\
&= diag(\mathbb{1}(\mathbf{W}^{(k)} \vec{\mathbf{h}}^{(k-1)} > 0)) \left(\frac{\partial \ell}{\partial \vec{\mathbf{h}}^{(k)}}\right)^\intercal \left(\vec{\mathbf{h}}^{(k-1)}\right)^\intercal
\end{aligned} \tag{11.18}$$

4. Full Backpropagation Process We summarize the results above into the following four steps:

1. Compute the Jacobian matrix with respect to the output layer, $\frac{\partial \ell}{\partial \vec{\mathbf{o}}} \in \mathbb{R}^{1 \times d_{out}}$:

$$\frac{\partial \ell}{\partial \vec{\mathbf{o}}} = \left(\vec{\hat{\mathbf{o}}} - \vec{\mathbf{e}}_y\right)^\intercal \qquad \text{((11.14) revisited)}$$

2. Compute the Jacobian matrix with respect to the last hidden layer, $\frac{\partial \ell}{\partial \vec{\mathbf{h}}^{(L-1)}} \in \mathbb{R}^{1 \times d_{L-1}}$:

$$\frac{\partial \ell}{\partial \vec{\mathbf{h}}^{(L-1)}} = \frac{\partial \ell}{\partial \vec{\mathbf{o}}} \mathbf{W}^{(o)} \qquad \text{((11.15) revisited)}$$

Then, compute the gradient with respect to the output weights, $\frac{\partial \ell}{\partial \mathbf{W}^{(o)}} \in \mathbb{R}^{d_{out} \times d_{L-1}}$:

$$\frac{\partial \ell}{\partial \mathbf{W}^{(o)}} = \left(\frac{\partial \ell}{\partial \vec{\mathbf{o}}}\right)^\intercal \left(\vec{\mathbf{h}}^{(L-1)}\right)^\intercal \qquad \text{((11.17) revisited)}$$

3. For each successive layer k, calculate the Jacobian matrix with respect to the k-th hidden layer $\frac{\partial \ell}{\partial \vec{\mathbf{h}}^{(k)}} \in \mathbb{R}^{1 \times d_k}$:

$$\frac{\partial \ell}{\partial \vec{\mathbf{h}}^{(k)}} = \frac{\partial \ell}{\partial \vec{\mathbf{h}}^{(k+1)}} diag(\mathbb{1}(\mathbf{W}^{(k+1)}\vec{\mathbf{h}}^{(k)} > 0))\mathbf{W}^{(k+1)} \quad \text{((11.16) revisited)}$$

Next, compute the gradient with respect to the $(k+1)$-th hidden layer weights $\frac{\partial \ell}{\partial \mathbf{W}^{(k+1)}} \in \mathbb{R}^{d_{k+1} \times d_k}$:

$$\frac{\partial \ell}{\partial \mathbf{W}^{(k+1)}} = diag(\mathbb{1}(\mathbf{W}^{(k+1)}\vec{\mathbf{h}}^{(k)} > 0)) \left(\frac{\partial \ell}{\partial \vec{\mathbf{h}}^{(k+1)}}\right)^\intercal \left(\vec{\mathbf{h}}^{(k)}\right)^\intercal$$
((11.18) revisited)

4. Repeat Step 3 until we reach the input layer.

Note that these instructions are based on a model that adopts the cross-entropy loss and the *ReLU* activation function. Using alternative losses and/or activation functions would result in a similar form, although the actual derivatives may be slightly different.

Problem 11.4.6. *(i) Show that if $\boldsymbol{A}$ is an $m \times n$ matrix and $\vec{\mathbf{h}} \in \mathbb{R}^n$ then computing $\boldsymbol{A}\vec{\mathbf{h}}$ requires mn multiplications and m additions. (ii) Using the previous part, argue that the number of arithmetic operations (additions or multiplications) in backpropagation algorithm on an FC net with ReLU activations is proportional to the number of parameters in the net.*

While the above calculation is in line with your basic algorithmic training, it doesn't exactly describe running time in modern ML environments with GPUs, since certain operations are parallelized, and compilers are optimized to run backpropagation as fast as possible.

11.4.3 Using a Different Activation Function

We briefly consider what happens if we choose a different activation function for the hidden layers. Consider the sigmoid activation function $\sigma(z) = \frac{1}{1+e^{-z}}$. Its derivative is given by:

$$\begin{aligned} \sigma'(z) &= \frac{1}{1+e^{-z}} \\ &= \frac{e^{-z}}{(1+e^{-z})^2} \\ &= \sigma(z) \cdot \left(\frac{e^{-z}}{1+e^{-z}}\right) \\ &= \sigma(z) \cdot (1-\sigma(z)) \end{aligned} \qquad (11.19)$$

There is also the hyperbolic tangent function $\tanh(z) = \frac{e^{2z}-1}{e^{2z}+1}$.

Problem 11.4.7. *Compute $f'(z)$ for $f(z) = \tanh(z)$; show how $f'(z)$ can be written in terms of $f(z)$.*

Problem 11.4.8. *Say a neural network uses an activation function $f(z)$ at layer $k+1$ such that $f'(z)$ is a function of $f(z)$. That is, $f'(z) = g(f(z))$ for some function g. Then verify that (11.16, 11.18) can be rewritten as:*

$$\frac{\partial \ell}{\partial \vec{\mathbf{h}}^{(k)}} = \frac{\partial \ell}{\partial \vec{\mathbf{h}}^{(k+1)}} diag(g(\mathbf{W}^{(k+1)}\vec{\mathbf{h}}^{(k)}))\mathbf{W}^{(k+1)}$$
$$\frac{\partial \ell}{\partial \mathbf{W}^{(k+1)}} = diag(g(\mathbf{W}^{(k+1)}\vec{\mathbf{h}}^{(k)})) \left(\frac{\partial \ell}{\partial \vec{\mathbf{h}}^{(k+1)}}\right)^{\mathsf{T}} \left(\vec{\mathbf{h}}^{(k)}\right)^{\mathsf{T}}$$

11.4.4 Final Remark

Directly following the steps of backpropagation is complicated and involves a lot of calculations. But remember that backpropagation is simply *computing gradients by the chain rule*. At a high level, we can think of the loss as a function of inputs and all the weights and note that backpropagation simply entails calculating derivatives with respect to each variable. The good news is that modern deep learning software does all the gradient calculations for users. All the model designer needs to do is to determine the neural network architecture (*e.g.*, choose number of layers, number of hidden units, and the activation functions).

One note of caution is that the loss function for deep neural nets is highly non-convex with respect to the parameters of the network. Just as we discussed in Chapter 3, the gradient descent algorithm is not guaranteed to find the actual minimizer in such situation, and the choice of the initial values of the parameters matter a lot.

11.5 Feedforward Neural Network in Programming

In this section, we discuss how to write Python code to build neural network models and perform forward propagation and backpropagation. As usual, we use the *numpy* package to speed up computation. Additionally, we use the *torch* package to easily design and train the neural network.

```
# import necessary packages
import numpy as np
import torch
import torch.nn as nn

# define the neural network
class Net(nn.Module):
    def __init__(self, input_size=2, hidden_dim1=2, hidden_dim2=3,
                                        hidden_dim3=2):
```

```
        super(Net, self).__init__()

        self.hidden1 = nn.Linear(input_size, hidden_dim1, bias=False)
        self.hidden1.weight = nn.Parameter(torch.tensor([[-2., 1.], [3., -1.
                                            ]]))

        self.hidden2 = nn.Linear(hidden_dim1, hidden_dim2, bias=False)
        self.hidden2.weight = nn.Parameter(torch.tensor([[0., 1.], [2., -1.]
                                            , [1., 2.]]))

        self.hidden3 = nn.Linear(hidden_dim2, hidden_dim3, bias=False)
        self.hidden3.weight = nn.Parameter(torch.tensor([[-1., 2., 1.], [3.,
                                             0., 0.]]))

        self.activation = nn.ReLU()

    # single step of forward propagation
    def forward(self, x):
        h1 = self.hidden1(x)
        h1 = self.activation(h1)
        h2 = self.hidden2(h1)
        h2 = self.activation(h2)
        h3 = self.hidden3(h2)
        return h3

net = Net()

# forward propagation with sample input
x = torch.tensor([3., 1.])
y_pred = net.forward(x)
print('Predicted value:', y_pred)

# backpropagation with sample input
loss = nn.functional.cross_entropy(y_pred.unsqueeze(0), torch.LongTensor([1]
                                    ))
loss.backward()
print(loss)
print(net.hidden1.weight.grad)
```

We start the code by importing all necessary packages.

```
import numpy as np
impoort torch
import torch.nn as nn
```

With *PyTorch,* we can design the architecture of any neural network by defining the corresponding *class*.

```
class Net(nn.Module):
    def __init__(self, input_size=2, hidden_dim1=2, hidden_dim2=3,
                                        hidden_dim3=2):
        super(Net, self).__init__()

        self.hidden1 = nn.Linear(input_size, hidden_dim1, bias=False)
        self.hidden1.weight = nn.Parameter(torch.tensor([[-2., 1.], [3., -1.
                                            ]]))

        self.hidden2 = nn.Linear(hidden_dim1, hidden_dim2, bias=False)
        self.hidden2.weight = nn.Parameter(torch.tensor([[0., 1.], [2., -1.]
                                            , [1., 2.]]))
```

```
        self.hidden3 = nn.Linear(hidden_dim2, hidden_dim3, bias=False)
        self.hidden3.weight = nn.Parameter(torch.tensor([[-1., 2., 1.], [3.,
                                            0., 0.]]))

        self.activation = nn.ReLU()

    def forward(self, x):
        h1 = self.hidden1(x)
        h1 = self.activation(h1)
        h2 = self.hidden2(h1)
        h2 = self.activation(h2)
        h3 = self.hidden3(h2)
        return h3
```

In the constructor, we define all the layers and activation functions we are going to use in the network. In particular, we specify that we need fully-connected layers by making instances of the *nn.Linear* class and that we need *ReLU* activation function by making an instance of the *nn.Relu* class. Then in the *forward()* function, we specify the order in which to apply the layers and activations. See Figure 11.7 for a visualization of this neural network architecture.

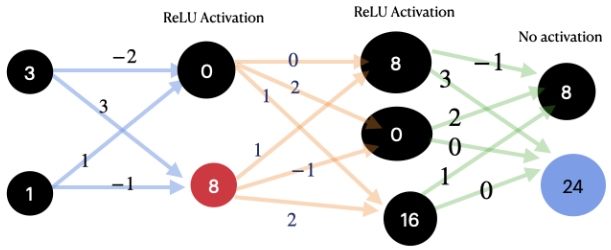


Figure 11.7: A sample feedforward neural network with two hidden layers and two output nodes.

We can simulate one step of forward propagation by calling the *forward()* function of the class *Net* we defined.

```
x = torch.tensor([3., 1.])
y_pred = net.forward(x)
print('Predicted value:', y_pred)
```

Similarly, we can implement backpropagation by specifying which loss function we want to use, and calling its *backward()* function.

```
loss = nn.functional.cross_entropy(y_pred.unsqueeze(0), torch.LongTensor([1]
                                    ))
loss.backward()
print(loss)
print(net.hidden1.weight.grad)
```

Each function call of *backward()* will evaluate the gradients of *loss* with respect to every parameter in the network. Gradients can be manually accessed through the following code.

```
print(net.hidden1.weight.grad)
```

Note that calling *backward()* multiple times will cause gradients to accumulate. While we do not update model weights in this code sample, it is important to periodically clear the gradient buffer when doing so to prevent unintended training. [8] We will discuss how to do this in the next chapter.

[8] For more information, see https://pytorch.org/docs/stable/generated/torch.Tensor.backward.html

12
Convolutional Neural Network

In Chapter 11, we focused on a type of a neural network called feed-forward neural networks. But different data has different structure (*e.g.*, image, text, audio, etc.) and we need better ways of exploiting them. This can help reduce the number of parameters needed in the network, which may allow easier or more data-efficient training. In this chapter, we present a type of a neural network common in image processing called *Convolutional Neural Network* (CNN); these models use a mathematical technique called convolution in order to extract important visual features from input images.

12.1 Introduction to Convolution

Roughly speaking, *convolution* refers to a mathematical operation where two functions are "mixed" to output a new function. In machine learning, the main idea of convolution is to reuse the same set of parameters on different portions of input. This is particularly effective at exploiting the structure of images. It was originally motivated by studies of the structure of cortical cells in the $V1$ visual cortex of the human brain (Hubel and Wiesel won the Nobel Prize in 1981 for this breakthrough discovery). [1] Let's first consider an example of a $1D$ convolution.

[1] Paper: https://www.jstor.org/stable/24965293.

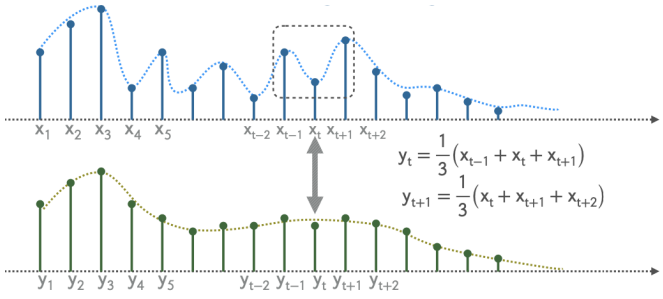


Figure 12.1: The effects of $1D$ convolution on graph of COVID-19 positive cases.

Example 12.1.1. *Consider the 3-day moving average of daily COVID-19 cases as shown in Figure 12.1. Let x_t denote the number of daily cases on day t. We can then take three consecutive values, compute their average and create a new output sequence from averages: $y_t = \frac{1}{3}(x_{t-1} + x_t + x_{t+1})$. Then if we set $w_1 = w_2 = w_3 = \frac{1}{3}$ and denote $\vec{\mathbf{w}} = (w_1, w_2, w_3)$, we can write:* [2]

$$y_t = w_1 x_{t-1} + w_2 x_t + w_3 x_{t+1} = \vec{\mathbf{w}} \cdot (x_{t-1}, x_t, x_{t+1})$$
$$y_{t+1} = w_1 x_t + w_2 x_{t+1} + w_3 x_{t+2} = \vec{\mathbf{w}} \cdot (x_t, x_{t+1}, x_{t+2})$$
$$y_{t+2} = w_1 x_{t+1} + w_2 x_{t+2} + w_3 x_{t+3} = \vec{\mathbf{w}} \cdot (x_{t+1}, x_{t+2}, x_{t+3})$$

Notice that we are reusing the same weights and applying them to multiple different values of x_t to calculate y_t. It is almost like sliding a filter down the array of x_t's and applying it to every set of 3 consecutive inputs. For this reason, we call $\vec{\mathbf{w}}$ the ***convolution filter weight*** *of length 3.*

[2] If we set the weights to a different value, we can find a weighted moving average.

Example 12.1.2. *Consider an input sequence $\vec{\mathbf{x}} = (2, 1, 1, 7, -1, 2, 3, 1)$ and a convolution filter $\vec{\mathbf{w}} = (3, 2, -1)$. The first two output values will be:*

$$y_1 = 2 \times 3 + 1 \times 2 + 1 \times (-1) = 7$$
$$y_2 = 1 \times 3 + 1 \times 2 + 7 \times (-1) = -2$$

Following a similar calculation for the other values, we see that the full output sequence is $\vec{\mathbf{y}} = (7, -2, 18, 17, -2, 11)$. Note that the length of $\vec{\mathbf{y}}$ should be $|\vec{\mathbf{x}}| - |\vec{\mathbf{w}}| + 1 = 8 - 3 + 1 = 6$.

Problem 12.1.3. *If $y_t = 2x_{t-1} - x_{t+1}$, $y_{t+1} = 2x_t - x_{t+2}$, and $y_{t+2} = 2x_{t+1} - x_{t+3}$, what is the convolution filter weight?*

12.2 Convolution in Computer Vision

In this section, we now focus on the application of convolution in computer vision. By the nature of image data, we will be primarily dealing with 2D convolution. Generally, 2D convolution filters are called *kernels*.

Figure 12.2: The effect of local smoothing on a sample image. (The person depicted here is Admiral Grace Murray Hopper, a computing pioneer.)

Example 12.2.1 (Local Smoothing (Blurring))**.** *An image can be blurred by constructing a filter that replaces each pixel by the average of neighboring pixels:*

$$y_{i,j} = \frac{1}{9} \sum_{b_1, b_2 \in \{-1,0,1\}} x_{i+b_1, j+b_2}$$

An example is shown in Figure 12.2.

Figure 12.3: The effect of local sharpening on a sample image

Example 12.2.2 (Local Sharpening (Edge Detection))**.** *The edge of objects in an image can be detected by constructing a filter that replaces each pixel by its difference with the average of neighboring pixels:*

$$y_{i,j} = x_{i,j} - \frac{1}{9} \sum_{b_1,b_2 \in \{-1,0,1\}} x_{i+b_1,j+b_2}$$

An example is shown in Figure 12.3.

12.2.1 *Convolution Filters for Images*

Computationally, we perform 2D convolution on an image by "sliding" the filter around every possible location in the image and taking the inner product:

$$y_{i,j} = \sum_{-k \le r,s \le k} w_{r,s} x_{i+r,j+s} \tag{12.1}$$

The result is a new image and we can view each filter as a transformation which takes an image and returns an image. In the above equation, the filter size is $(2k+1) \times (2k+1)$. For example, if $k = 1$, we can consider the convolution weight filter to be

$$\begin{bmatrix} w_{-1,-1} & w_{-1,0} & w_{-1,+1} \\ w_{0,-1} & w_{0,0} & w_{0,+1} \\ w_{+1,-1} & w_{+1,0} & w_{+1,+1} \end{bmatrix}$$

An image of size $m \times n$ can only be applied the filter at a location where the filter completely fits inside the image. Therefore, the locations in the input image where the center of the filter can be placed are $k < i \le m - k$, $k < j \le n - k$ and the size of the output image is $(m - 2k) \times (n - 2k)$.

Example 12.2.3. *If the input and convolution filter are given as following:*

$$\mathbf{X} = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 \end{bmatrix} \quad \textit{and} \quad \mathbf{W} = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$$

then the pixel at $(1,1)$ *of the resulting image can be calculated by applying the filter at the top left corner of the input image. That is, we take the inner product of the following parts (in red) of the two matrices*

$$\begin{bmatrix} 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 \end{bmatrix} \quad \textit{and} \quad \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$$

which is

$$1 \times 1 + 1 \times 0 + 1 \times 1 + 0 \times 0 + 1 \times 1 + 1 \times 0 + 0 \times 1 + 0 \times 0 + 1 \times 1 = 4$$

Therefore, the $(1,1)$ *entry of the resulting image is* 4*. Similarly, the remaining pixels of the resulting image can be calculated by moving around the filter as in Figure 12.4. The output image is given as:*

$$\mathbf{Y} = \begin{bmatrix} 4 & 3 & 4 \\ 2 & 4 & 3 \\ 2 & 3 & 4 \end{bmatrix}$$

In this example, $\mathbf{X} \in \mathbb{R}^{5\times5}, \mathbf{W} \in \mathbb{R}^{3\times3}, k = 1$ *and* $\mathbf{Y} \in \mathbb{R}^{3\times3}$*.*

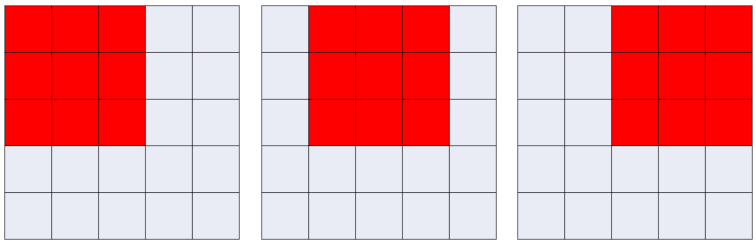


Figure 12.4: Visual representation of applying a 3×3 convolutional filter to a 5×5 image.

Problem 12.2.4. *Suppose we have a* 10×10 *image and a* 5×5 *filter. What is the size of the output image?*

Figure 12.5 shows some common filters used in image processing. Note that all these filters are hand-crafted and require domain-specific knowledge. However, in convolutional neural networks, we don't set these weights by hand and we learn all the filter weights from the data!

12.2.2 *Padding*

In standard 2*D* convolution, the size of the output image is not equal to the size of the input image because we only consider locations where the filter fits completely in the image. However, sometimes we may want their sizes to be the same. In such a case, we apply a

Box blur (normalized)	$\frac{1}{9}\begin{bmatrix}1&1&1\\1&1&1\\1&1&1\end{bmatrix}$	
Gaussian blur 3 × 3 (approximation)	$\frac{1}{16}\begin{bmatrix}1&2&1\\2&4&2\\1&2&1\end{bmatrix}$	
Gaussian blur 5 × 5 (approximation)	$\frac{1}{256}\begin{bmatrix}1&4&6&4&1\\4&16&24&16&4\\6&24&36&24&6\\4&16&24&16&4\\1&4&6&4&1\end{bmatrix}$	
Unsharp masking 5 × 5 Based on Gaussian blur with amount as 1 and threshold as 0 (with no image mask)	$\frac{-1}{256}\begin{bmatrix}1&4&6&4&1\\4&16&24&16&4\\6&24&-476&24&6\\4&16&24&16&4\\1&4&6&4&1\end{bmatrix}$	

Figure 12.5: Some common filters and corresponding weights used in image processing. Source: https://en.wikipedia.org/wiki/Kernel_(image_processing)

technique called *padding*. The idea is to pad pixels to all four edges of the input image (left, right, up, and down) so that the number of valid locations for the filter is the same as the number of pixels in the original image. In particular, if the filter size is $(2k+1) \times (2k+1)$, we need to pad k pixels on each side.

There are multiple ways to implement padding. *Zero padding* is when the values at all padded pixels are set to 0. *"Same" padding* is when the values at padded pixels are set to the value of the nearest pixel at the edge of the input image. In practice, zero padding is a more common form of padding (it is equivalent to adding a black frame around the image).

12.2.3 Downsampling Input with Stride

Another common operation in convolutional neural networks is called *stride*. Stride controls how the filter convolves around the input image. Instead of moving the filter by 1 pixel every time, we can also move the filter every 2 (or in general, s) pixels. Essentially, we are applying each of the filter weights at fewer locations of the image than before. This can be viewed as a downsampling strategy, which gives a smaller output image while greatly preserving the information from the original input.

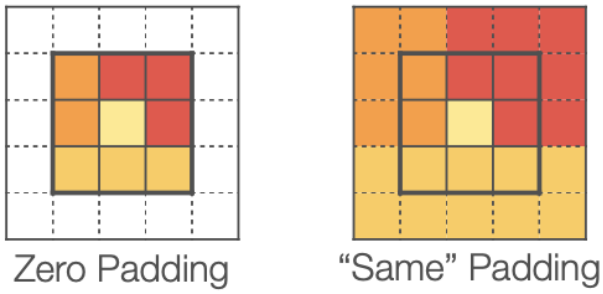


Figure 12.6: A visual comparison between two common types of padding: zero padding and "same" padding

Suppose we have an input image of size $m \times n$ and a filter of size $(2k+1) \times (2k+1)$. If padding is applied to the image, the output image size is $\lfloor (m+s-1)/s \rfloor \times \lfloor (n+s-1)/s \rfloor$. [3] If padding is not applied to the image, the output image size is $\lfloor (m+s-2k-1)/s \rfloor \times \lfloor (n+s-2k-1)/s \rfloor$; this is because convolution with stride is performed directly on the input image itself, making the effective input image size $(m-2k) \times (n-2k)$.

[3] As a sanity check, you can verify that in the special case of $s = 1$ the output image size will be the same as the input image size

Example 12.2.5. *Suppose we have an input image of size* 5×5*. If we apply padding and take stride size* $s = 2$*, then output size is* 3×3*.*

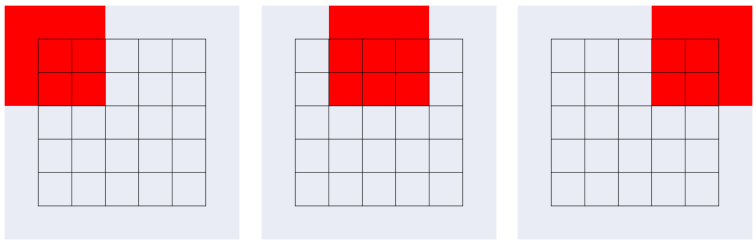


Figure 12.7: Visual representation of applying a 3×3 convolutional filter to a 5×5 image with padding and stride size 2.

12.2.4 *Nonlinear Convolution*

For each location in the image (original or padded) and a single convolution filter, we can apply a nonlinear activation function after the convolution

$$y_{i,j} = g\left(\sum_{-k \le r,s \le k} w_{r,s} x_{i+r,j+s}\right) \tag{12.2}$$

where g is some function like *ReLU*, sigmoid, tanh. The intuition is similar to what we had earlier in feedforward neural networks — if we don't add non-linear activation functions, a multi-layered convolutional neural network can be easily reduced to a linear model!

12.2.5 *Channels*

In general, we do not only use one convolution filter. We construct a network of multiple layers, and for each of the layers, we apply multiple convolutional filters. Different filters will be able to detect different features of the image (*e.g.*, one filter detects edges and one filter detects dots), and we want to apply different filters independently on the input image. The result of applying a given filter on a single input image is called a *channel*. We can stack the channels to create a $3D$ structure, as shown in Figure 12.8.

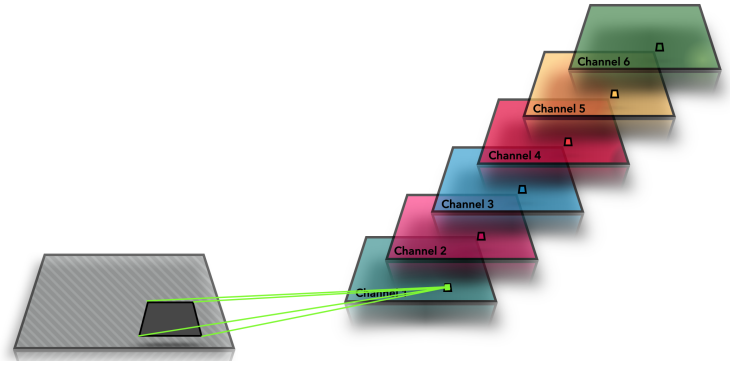


Figure 12.8: Each filter creates one channel. The output of a convolutional layer has multiple output channels.

Next, let's imagine that we want to build a deep neural network with multiple convolutional layers (state-of-the-art CNNs have 100 or even 1000 layers!). A typical convolutional layer in the middle of the network will have several input channels (equivalent to the number of output channels from the previous layer) and multiple output channels. How can we determine the number of filters needed?

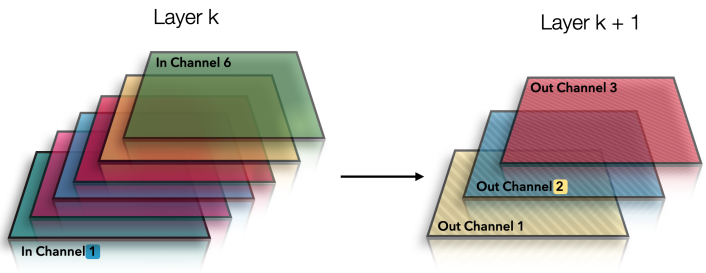


Figure 12.9: A convolutional layer which has multiple input and multiple output channels.

In this case, we want to define a filter for every possible pair of input and output channels. The output image of a particular output channel will be the summation of the output images from each of the input channels, after applying the corresponding filter. We can also add a nonlinear activation function g after taking the summation of the output images. That is, (12.2) can be rewritten for the output

image in the v-th output channel as:

$$y_{i,j}^{(v)} = g\left(\sum_{u=1}^{n_{in}} \sum_{-k\le r,s\le k} w_{r,s}^{(u,v)} x_{i+r,j+s}^{(u)}\right) \tag{12.3}$$

where n_{in} is the number of input channels, $\mathbf{X}^{(u)}$ is the image at the u-th input channel, $\mathbf{Y}^{(v)}$ is the image at the v-th output channel, and $\mathbf{W}^{(u,v)}$ is the filter between the u-th input channel and the v-th output channel.

Example 12.2.6. *Assume there are 6 input channels and 3 output channels, and the filter size is* 5×5*. Then for every* 6×3 *pair of input and output channel, we have a kernel of weights of size* 5×5*, so there are a total of* $6 \times 3 \times 5 \times 5 = 450$ *weights.*

12.2.6 Pooling

Pooling is another popular way to reduce the size of the output of a convolutional layer. In contrast to stride, which applies convolution operation every s pixels, pooling partitions each image (channel) to patches of size $\Delta \times \Delta$ and performs a reduction operation on each patch. You can think of this as similar to what happens when you lower the resolution of an image. The reduction operation can either involve taking the max of all the values in the patch ("max-pooling"):

$$y_{i,j} = \max_{1\le r,s\le \Delta} X_{(i-1)\cdot\Delta+r,(j-1)\cdot\Delta+s}$$

or taking the average of all the values in the patch ("mean-pooling"):

$$y_{i,j} = \frac{1}{\Delta^2}\sum_{r,s=1}^{\Delta} X_{(i-1)\cdot\Delta+r,(j-1)\cdot\Delta+s}$$

The pooling operation can reduce the image size by a factor of Δ^2.

If the input image is of size $m \times n$, the size of the image after pooling will be $\lfloor m/\Delta \rfloor \times \lfloor n/\Delta \rfloor$.

Example 12.2.7. *If the size of an input image to a pooling layer is* 6×6 *and* $\Delta = 2$*, then the output is of size* 3×3*.*

12.2.7 A Full Convolutional Neural Network

Let's put everything together and consider a full convolutional neural network. Figure 12.11 shows a typical example of a convolutional neural network. A convolutional neural network typically begins by stacking multiple convolutional layers and pooling layers. Each convolutional layer has its own kernel size and number of output channels; similarly, each pooling layer has its own kernel size. This is

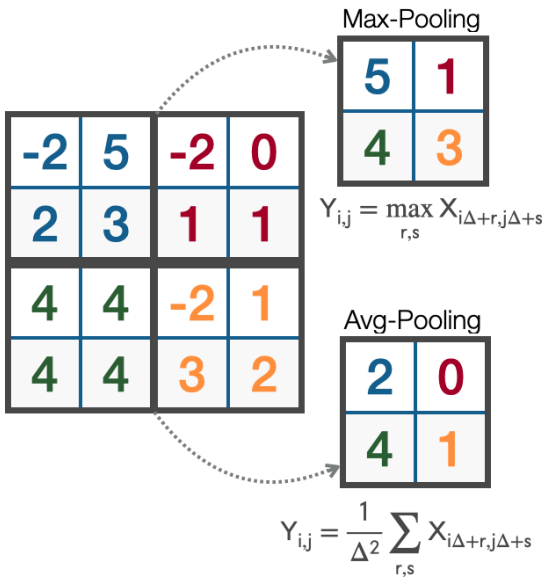


Figure 12.10: Max-pooling vs mean-pooling.

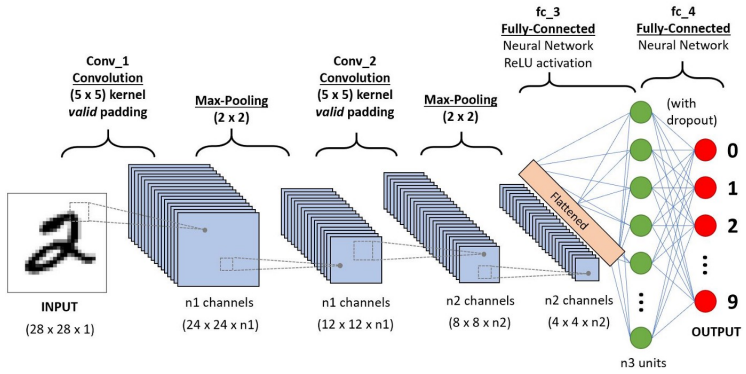


Figure 12.11: A illustration of a full convolutional neural network.

followed by several fully-connected layers at the end. Since the output images of convolutional layers are 2-dimensional, it is customary to "flatten" the images into a $1D$ vector (*i.e.*, append one row after another) before applying the fully connected layers. Intuitively, we can think of the convolutional and pooling layers as learning interesting image features (*e.g.*, stripes or dots) while the fully-connected layers map these features to output classes (*e.g.*, zebras have a lot of stripes).

All the weights in a convolutional neural network (including weights in kernels, fully-connected layers) can be learned via the backpropagation algorithm in Chapter 11. Again, modern deep learning libraries (*e.g.*, PyTorch, Tensorflow) have all the convolutional and pooling layers implemented and can compute gradients

automatically!

Finally, the above convolutional neural network is still a simple network, compared to modern convolutional neural networks. Interested students can look up architectures such as AlexNet, Inception, VGG, ResNet and DenseNet.

12.2.8 Designing a Convolutional Network

While we described convolutional nets above, we did not give a good explanation of why they are well-suited to solve vision tasks. Working through the next few examples will help you understand their power. The idea is that convolution is a parameter-efficient [4] architecture that can "search for patterns" anywhere in the image. For example, suppose the net has to decide if the input image has a triangle anywhere in it. If the net were fully connected, it would have to learn how to detect triangles centered at every possible pixel location (i, j). By contrast, if a simple convolutional filter can detect a triangle, we can just replicate this filter in all patches to detect a triangle anywhere in the image.

[4] Which usually goes with sample-efficiency!

Now consider the CNN architecture in Figure 12.12. The architecture has two convolutional layers, the first with a *ReLU* activation function, and the second with a sigmoid activation function. [5] We will choose an appropriate convolutional weight and bias such that the architecture can detect a particular simple visual pattern.

[5] In both Example 12.2.8 and Example 12.2.9, the second convolutional layer can be considered a fully connected layer if we flatten image Y

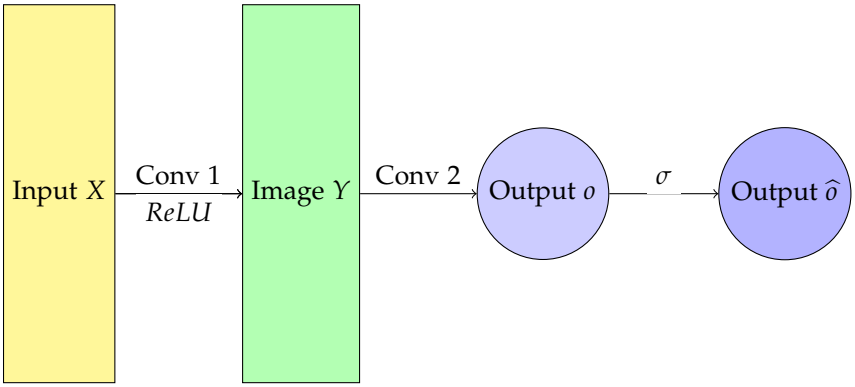


Figure 12.12: A sample CNN architecture that can be used to detect the patterns as aligned in Example 12.2.8 and Example 12.2.9.

Example 12.2.8. *The input to the network is a gray-scale image of size* 8×8 *(*1 *channel), and each pixel takes an integer value between* 0 *and* 255*, inclusive. If at least one pixel of the image has value exactly* 255*, the output of the CNN should have a value close to* 1 *and otherwise the output should have a value close to* 0*.*

We will now solve Example 12.2.8 by individually configuring the parameters for each convolutional layer in Figure 12.12. The first

convolutional layer will have a 1×1 filter of weight 1, a bias of -254, and a ReLU activation function. The convolution will be applied with no padding, and with stride 1. That is, the (i, j)-th entry of the output image of the first convolutional layer will be

$$y_{i,j} = ReLU(x_{i,j} - 254) \qquad 1 \leq i, j \leq 8$$

where $x_{i,j}$ is the (i, j)-th entry of the input image. Notice that this value is zero everywhere, except if $x_{i,j} = 255$, in which case $y_{i,j}$ takes the value 1. That is,

$$y_{i,j} = \begin{cases} 1 & x_{i,j} = 255 \\ 0 & otherwise \end{cases}$$

See Figure 12.13 to see the effect of this choice of convolutional layer on a sample image. We see that we have now successfully identified the pixels in the input image that take the value 255.

$$\begin{bmatrix} 0 & 100 & 200 \\ 50 & 150 & 250 \\ 55 & 155 & 255 \end{bmatrix} \xrightarrow{\text{Conv } 1} \begin{bmatrix} -254 & -154 & -54 \\ -204 & -104 & -4 \\ -199 & -99 & 1 \end{bmatrix} \xrightarrow{ReLU} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Figure 12.13: The effect of the choice of first convoluational layer for Example 12.2.8 on a sample image. Only a portion of the image is shown.

Next, consider the second convolutional layer with a 8×8 filter of all weights equal to 10, a bias of -5, and a sigmoid activation function. The convolution will be applied with no padding, and with stride 1. [6] The output, before the sigmoid, will be

$$o = \left(\sum_{i,j=1}^{8} 10 y_{i,j} \right) - 5$$

Notice that this value is -5 if and only if there is no pixel such that $y_{i,j} = 1$ (*i.e.*, $x_{i,j} = 255$). If there is one such pixel, the output is 5; if there are two, the output is 15. The important thing is, the output is at least 5 if there is at least one pixel in the input image whose value is 255. That is

$$o = \begin{cases} \geq 5 & \exists (i, j) : x_{i,j} = 255 \\ -5 & otherwise \end{cases}$$

Finally, when we apply the sigmoid function, the final output of the model will be

$$\widehat{o} = \sigma(o) = \begin{cases} \geq 0.99 & \exists (i, j) : x_{i,j} = 255 \\ 0.01 & otherwise \end{cases}$$

This is exactly what we wanted in Example 12.2.8.

[6] Once the output image of the first convolutional layer is flattened to a vector of length 64, this can also be thought of as a fully-connected layer with input size 64 and output size 1.

Example 12.2.9. *The input to the network is a gray-scale image of size* 8×8 *(*1 *channel), and each pixel takes an integer value between* 0 *and* 255*, inclusive. If any part of the input image contains the following pattern:*

$$\begin{bmatrix} * & 255 & * \\ 255 & 255 & 255 \\ * & 255 & * \end{bmatrix} \tag{12.4}$$

the output of the CNN should have a value close to 1 *and otherwise the output should have a value close to* 0*.*

We use the same architecture as in Figure 12.12, but now with a different choice of parameters for the convolutional layers. The first convolutional layer will have a 3×3 filter with the following weights:

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

a bias of -1274, and a ReLU activation function. The convolution will be applied with no padding, and with stride 1. That is, the (i, j)-th entry of the output image of the first convolutional layer will be

$$y_{i,j} = ReLU\left(x_{i-1,j} + x_{i,j-1} + x_{i,j} + x_{i,j+1} + x_{i+1,j} - 1274\right) \quad 2 \leq i, j \leq 7$$

where $x_{i,j}$ is the (i, j)-th entry of the input image. [7] Notice that this value is zero everywhere, except if $x_{i,j} + x_{i-1,j} + x_{i,j-1} + x_{i,j+1} + x_{i+1,j} = 1275$, in which case $y_{i,j}$ takes the value 1. This can only happen if $x_{i-1,j} = x_{i,j-1} = x_{i,j} = x_{i,j+1} = x_{i+1,j} = 255$. That is, if the input image has the pattern in (12.4) centered around (i, j).

[7] Since there is no padding, the values $y_{1,1}, y_{1,8}, y_{8,1}, y_{8,8}$ are not defined.

$$y_{i,j} = \begin{cases} 1 & \textit{Pattern in (12.4) exists at } (i, j) \\ 0 & \textit{otherwise} \end{cases}$$

See Figure 12.14 to see the effect of this choice of convolutional layer on two sample images.

$$\begin{bmatrix} 0 & 255 & 0 \\ 255 & 250 & 255 \\ 0 & 255 & 0 \end{bmatrix} \xrightarrow{\text{Conv 1}} \begin{bmatrix} * & * & * \\ * & -4 & * \\ * & * & * \end{bmatrix} \xrightarrow{ReLU} \begin{bmatrix} * & * & * \\ * & 0 & * \\ * & * & * \end{bmatrix}$$

$$\begin{bmatrix} 0 & 255 & 0 \\ 255 & 255 & 255 \\ 0 & 255 & 0 \end{bmatrix} \xrightarrow{\text{Conv 1}} \begin{bmatrix} * & * & * \\ * & +1 & * \\ * & * & * \end{bmatrix} \xrightarrow{ReLU} \begin{bmatrix} * & * & * \\ * & 1 & * \\ * & * & * \end{bmatrix}$$

Figure 12.14: The effect of the choice of first convoluational layer for Example 12.2.9 on two sample images. Only a portion of the images is shown.

Next, consider the second convolutional layer with a 6×6 filter of all weights equal to 10, a bias of -5, and a sigmoid activation function. The convolution will be applied with no padding, and with stride 1. The output, before the sigmoid, will be

$$o = \left(\sum_{i,j=2}^{7} 10 y_{i,j} \right) - 5$$

Notice that this value is -5 if and only if there is no pixel such that $y_{i,j} = 1$ (*i.e.*, the pattern exists at (i, j)). If there is one such pixel, the output is 5; if there are two, the output is 15. The important thing is, the output is at least 5 if there is at least one copy of the given pattern. That is

$$o = \begin{cases} \geq 5 & \exists (i,j) : \textit{Pattern in } (12.4) \textit{ exists at } (i,j) \\ -5 & \textit{otherwise} \end{cases}$$

Finally, when we apply the sigmoid function, the final output of the model will be

$$\widehat{o} = \sigma(o) = \begin{cases} \geq 0.99 & \exists (i,j) : \textit{Pattern in } (12.4) \textit{ exists at } (i,j) \\ 0.01 & \textit{otherwise} \end{cases}$$

This is exactly what we wanted in Example 12.2.9.

12.3 *Backpropagation for Convolutional Nets*

A convolutional neural network is a special case of a feedforward neural network where we use convolutional layers, instead of fully-connected layers as in Chapter 11. Therefore, we can apply the same basic idea of backpropagation so that we can run the gradient descent algorithm, although the details of the calculation are slightly different.

The biggest difference is that in a fully-connected layer, each weight is used exactly once, while in a convolutional layer, each weight is applied multiple times throughout the input image. [8] This makes the computation for the gradient slightly more convoluted. But the basic idea is the same — identify all paths through which the corresponding weight affects the output of the model and add up the amount of effect for each path.

[8] This phenomenon is also known as *weight sharing*.

Figure 12.15 shows a portion of a sample neural network where weight sharing occurs. That is, the same weight w is used between the following four pairs of nodes: $(x_1, y_1), (x_2, y_2), (x_2, y_3), (x_3, y_4)$. If we wanted to find the gradient $\partial o / \partial w$, we need to consider the four paths that the weight w affects the output: $w \to y_i \to o$ where $1 \leq i \leq 4$.

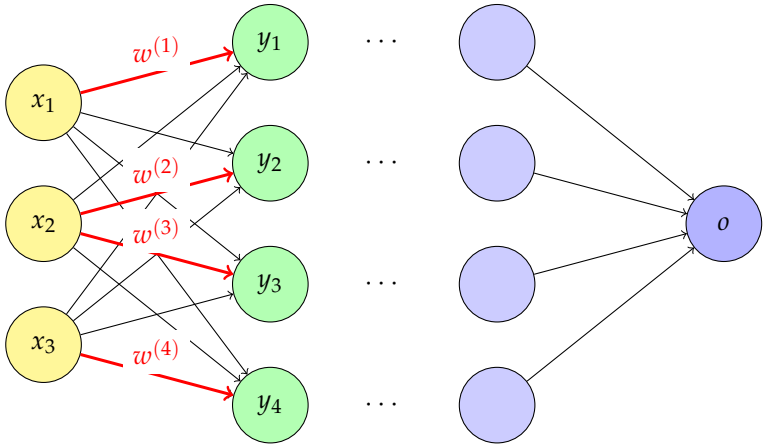


Figure 12.15: A sample neural network where weight sharing occurs. $w^{(1)}, w^{(2)}, w^{(3)}, w^{(4)}$ are the copies of the same weight w.

What we will do is consider the four copies of the weight w as separate weights that will be denoted as $w^{(i)}$ where $1 \leq i \leq 4$. Since these weights are only used in one place in the layer, we are already familiar with computing the gradients $\partial o / \partial w^{(i)}$. Then we will add (or *pool*) these values to get the gradient $\partial o / \partial w$. This works because we can think of each $w^{(i)}$ as a function of w where $w^{(i)} = w$. Then by Chain Rule, we have

$$\frac{\partial o}{\partial w} = \sum_{i=1}^{4} \frac{\partial o}{\partial w^{(i)}} \cdot \frac{\partial w^{(i)}}{\partial w} = \sum_{i=1}^{4} \frac{\partial o}{\partial w^{(i)}}$$

12.3.1 *Deriving Backpropagation Formula for Convolutional Layers*

In this subsection, we derive the backpropagation formula for a convolutional layer. (As in many other places, if your instructor did not teach it in COS 324, consider this to be advanced reading.)

Recall that in a fully-connected layer, which computes $\vec{\mathbf{h}}^k = \mathbf{W}^{(k)} \vec{\mathbf{h}}^{(k-1)}$, the gradient with respect to a particular weight $w_{i,j}^{(k)}$ can be simply computed as

$$\frac{\partial \ell}{\partial w_{i,j}^{(k)}} = \frac{\partial \ell}{\partial h_i^{(k)}} \cdot \frac{\partial h_i^{(k)}}{\partial w_{i,j}^{(k)}} = \frac{\partial \ell}{\partial h_i^{(k)}} \cdot h_j^{(k-1)}$$

This is because the weight $w_{i,j}^{(k)}$ is only used to compute $h_i^{(k)}$ out of all nodes in the next hidden layer. In comparison, consider a convolutional layer, which computes an output image $\mathbf{Y} \in \mathbb{R}^{n \times n}$ from an input image $\mathbf{X} \in \mathbb{R}^{m \times m}$ and filter $\mathbf{W} \in \mathbb{R}^{(2k+1) \times (2k+1)}$. Notice that the weight $w_{i,j}$ is used to compute all of the pixels in the output image. Therefore, we just need to add (or *pool*) the gradient flow from each of these paths. The gradient with respect to a particular weight

$w_{i,j}$ will be

$$\begin{aligned}\frac{\partial \ell}{\partial w_{i,j}} &= \left(\frac{\partial \ell}{\partial y_{1,1}} \cdot \frac{\partial y_{1,1}}{\partial w_{i,j}} + \frac{\partial \ell}{\partial y_{1,2}} \cdot \frac{\partial y_{1,2}}{\partial w_{i,j}} + \ldots + \frac{\partial \ell}{\partial y_{1,n}} \cdot \frac{\partial y_{1,n}}{\partial w_{i,j}}\right) \\ &+ \left(\frac{\partial \ell}{\partial y_{2,1}} \cdot \frac{\partial y_{2,1}}{\partial w_{i,j}} + \frac{\partial \ell}{\partial y_{2,2}} \cdot \frac{\partial y_{2,2}}{\partial w_{i,j}} + \ldots + \frac{\partial \ell}{\partial y_{2,n}} \cdot \frac{\partial y_{2,n}}{\partial w_{i,j}}\right) \\ &+ \ldots \\ &+ \left(\frac{\partial \ell}{\partial y_{n,1}} \cdot \frac{\partial y_{n,1}}{\partial w_{i,j}} + \frac{\partial \ell}{\partial y_{n,2}} \cdot \frac{\partial y_{n,2}}{\partial w_{i,j}} + \ldots + \frac{\partial \ell}{\partial y_{n,n}} \cdot \frac{\partial y_{n,n}}{\partial w_{i,j}}\right)\end{aligned}$$

Assuming there is zero padding, this can be calculated as

$$\begin{aligned}\frac{\partial \ell}{\partial w_{i,j}} &= \left(\frac{\partial \ell}{\partial y_{1,1}} \cdot x_{i+1,j+1} + \frac{\partial \ell}{\partial y_{1,2}} \cdot x_{i+1,j+2} + \ldots + \frac{\partial \ell}{\partial y_{1,n}} \cdot x_{i+1,j+n}\right) \\ &+ \left(\frac{\partial \ell}{\partial y_{2,1}} \cdot x_{i+2,j+1} + \frac{\partial \ell}{\partial y_{2,2}} \cdot x_{i+2,j+2} + \ldots + \frac{\partial \ell}{\partial y_{2,n}} \cdot x_{i+2,j+n}\right) \\ &+ \ldots \\ &+ \left(\frac{\partial \ell}{\partial y_{n,1}} \cdot x_{i+n,j+1} + \frac{\partial \ell}{\partial y_{n,2}} \cdot x_{i+n,j+2} + \ldots + \frac{\partial \ell}{\partial y_{n,n}} \cdot x_{i+n,j+n}\right)\end{aligned}$$

Notice that the equation above can be rewritten as

$$\frac{\partial \ell}{\partial w_{i,j}} = \sum_{1 \leq r,s \leq n} \frac{\partial \ell}{\partial y_{r,s}} \cdot x_{i+r,j+s} \tag{12.5}$$

That is, the Jacobian matrix $\partial \ell / \partial \mathbf{W}$ is the output when applying a convolution filter $\partial \ell / \partial \mathbf{Y}$ to the input matrix $\mathbf{X}$.

Similarly, we can try to calculate the Jacobian matrix with respect to the input matrix $\mathbf{X}$. Each input pixel $x_{i,j}$ is used to calculate the output pixels $y_{i+r,j+s}$ where $-k \leq r, s \leq k$. The gradient with respect to a particular input pixel will be

$$\begin{aligned}\frac{\partial \ell}{\partial x_{i,j}} &= \left(\frac{\partial \ell}{\partial y_{i-k,j-k}} \cdot \frac{\partial y_{i-k,j-k}}{\partial x_{i,j}} + \ldots + \frac{\partial \ell}{\partial y_{i-k,j+k}} \cdot \frac{\partial y_{i-k,j+k}}{\partial x_{i,j}}\right) \\ &+ \left(\frac{\partial \ell}{\partial y_{i-k+1,j-k}} \cdot \frac{\partial y_{i-k+1,j-k}}{\partial x_{i,j}} + \ldots + \frac{\partial \ell}{\partial y_{i-k+1,j+k}} \cdot \frac{\partial y_{i-k+1,j+k}}{\partial x_{i,j}}\right) \\ &+ \ldots \\ &+ \left(\frac{\partial \ell}{\partial y_{i+k,j-k}} \cdot \frac{\partial y_{i+k,j-k}}{\partial x_{i,j}} + \ldots + \frac{\partial \ell}{\partial y_{i+k,j+k}} \cdot \frac{\partial y_{i+k,j+k}}{\partial x_{i,j}}\right)\end{aligned}$$

Assuming zero padding, this is calculated as

$$\begin{aligned}\frac{\partial \ell}{\partial x_{i,j}} = &\left(\frac{\partial \ell}{\partial y_{i-k,j-k}} \cdot w_{k,k} + \ldots + \frac{\partial \ell}{\partial y_{i-k,j+k}} \cdot w_{k,-k}\right) \\ &+ \left(\frac{\partial \ell}{\partial y_{i-k+1,j-k}} \cdot w_{k-1,k} + \ldots + \frac{\partial \ell}{\partial y_{i-k+1,j+k}} \cdot w_{k-1,-k}\right) \\ &+ \ldots \\ &+ \left(\frac{\partial \ell}{\partial y_{i+k,j-k}} \cdot w_{-k,k} + \ldots + \frac{\partial \ell}{\partial y_{i+k,j+k}} \cdot w_{-k,-k}\right)\end{aligned}$$

which can be rewritten as

$$\frac{\partial \ell}{\partial x_{i,j}} = \sum_{-k \le r,s \le k} \frac{\partial \ell}{\partial y_{i+r,j+s}} \cdot w_{-r,-s} \tag{12.6}$$

That is, the Jacobian matrix $\partial \ell / \partial \mathbf{X}$ is the output when applying the horizontally and vertically inverted image of $\mathbf{W}$ as the convolutional filter to the input matrix $\partial \ell / \partial \mathbf{Y}$.

12.4 *CNN in Programming*

In this section, we discuss how to write Python code to implement Convolutional Neural Networks (CNN). As usual, we use the *numpy* package to speed up computation and the *torch* package to easily design and train the neural network. We also introduce the *torchvision* package:

- *torchvision*: This package focuses on computer vision applications and is integrated with the broader PyTorch framework. It provides access to pre-built models, popular datasets, and a variety of image transform capabilities. [9]

[9] Documentation is available at https://pytorch.org/vision/stable/index.html

The following code sample implements a CNN and trains it on a single image.

```
# import necessary packages
import random
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision
import torchvision.transforms as transforms
from torch.utils.data import DataLoader

# set random seeds to ensure reproducibility
torch.manual_seed(0)
np.random.seed(0)
random.seed(0)
```

```
# load CIFAR10 data
transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])
train_data = torchvision.datasets.CIFAR10(root='./data', train=True,
                                          download=True, transform=transform)
test_data = torchvision.datasets.CIFAR10(root='./data', train=False,
                                          transform=transform)

# helps iterate through the train/test data in batches
train_loader = DataLoader(dataset=train_data, batch_size=8, shuffle=True,
                                          num_workers=0)
test_loader = DataLoader(dataset=test_data, batch_size=8, shuffle=False,
                                          num_workers=0)

# define the CNN architecture
class ConvNet(nn.Module):
    def __init__(self):
        super(ConvNet, self).__init__()
        # conv2d takes # input channels, # output channels, kernel size
        self.conv1 = nn.Conv2d(3, 3, 5)
        self.pool1 = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(3, 16, 5)
        self.pool2 = nn.AvgPool2d(2, 2)
        self.fc1 = nn.Linear(16*5*5, 120)
        self.fc2 = nn.Linear(120, 10)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = self.pool1(x)
        x = F.relu(self.conv2(x))
        x = self.pool2(x)
        x = x.view(-1, 16*5*5)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)

        return x

# choose the optimization technique to implore
optimizer = torch.optim.SGD(net.parameters(), lr=0.01)

# extract one image from the dataset
images, labels = next(iter(train_loader))
image = images[0].unsqueeze(0)

# forward propagation
net = ConvNet()
output = net(image)

# backpropagation
loss = torch.norm(output - torch.ones(output.shape[1]))**2
loss.backward()
optimizer.step()
optimizer.zero_grad()
```

As usual, we start by importing packages.

```
import random
import numpy as np
import torch
import torch.nn as nn
```

```
import torch.nn.functional as F
import torchvision
import torchvision.transforms as transforms
from torch.utils.data import DataLoader
```

The *DataLoader* class helps iterate through a dataset in batches.

Next, we fix all random seeds to ensure reproducibility.

```
torch.manual_seed(0)
np.random.seed(0)
random.seed(0)
```

Recall that programming languages on a classical computer can only implement pseudorandom methods, which always produce the same result for a given seed.

Then we load the CIFAR-10 dataset.

```
transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])
train_data = torchvision.datasets.CIFAR10(root='./data', train=True,
                                          download=True, transform=transform)
test_data = torchvision.datasets.CIFAR10(root='./data', train=False,
                                         transform=transform)
```

The CIFAR-10 dataset contains simple images of a single object, and the images are labeled with the category of the objects they contain. Note that we normalize the dataset with a mean of 0.5 and standard deviation of 0.5 per color channel. Figure 12.16 shows a sampling of images from the dataset after the normalization.

Figure 12.16: Sample images from the CIFAR10 dataset.

Next we create *DataLoader* objects to help iterate through the dataset in batches. Each batch will consist of 8 images and 8 labels.

```
train_loader = DataLoader(dataset=train_data, batch_size=8, shuffle=True,
                                          num_workers=0)
test_loader = DataLoader(dataset=test_data, batch_size=8, shuffle=False,
                                          num_workers=0)
```

Then we define our CNN architecture in the *ConvNet* class.

```
class ConvNet(nn.Module):
    def __init__(self):
        super(ConvNet, self).__init__()
        # conv2d takes # input channels, # output channels, kernel size
        self.conv1 = nn.Conv2d(3, 3, 5)
        self.pool1 = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(3, 16, 5)
        self.pool2 = nn.AvgPool2d(2, 2)
```

```
        self.fc1 = nn.Linear(16*5*5, 120)
        self.fc2 = nn.Linear(120, 10)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = self.pool1(x)
        x = F.relu(self.conv2(x))
        x = self.pool2(x)
        x = x.view(-1, 16*5*5)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)

        return x
```

Just like the FFNN code from the previous chapter, we define all the layers and activations we are going to use in the constructor. Note that in addition to instances of the *nn.Linear* class and the *nn.Relu* class, we also make use of classes like *nn.Conv2d* and *nn.MaxPool2d* which are specifically designed for CNNs.

We extract one training image with the following code.

```
images, labels = next(iter(train_loader))
image = images[0].unsqueeze(0)
```

The *unsqueeze()* function adds one dimension to the training data. This is called a *batch dimension*. Normally, we would run the training in batches, and the size of the data along the batch dimension will be equal to the number of images in each batch. Here, we only use one image for the sake of exposition.

We can now run forward propagation on a sample image with the code below.

```
net = ConvNet()
output = net(image)
```

We then implement the *squared error* loss. Alternatively, we could have chosen the cross-entropy loss or any other valid loss function.

```
loss = torch.norm(output - torch.ones(output.shape[1]))**2
```

Next, we calculate the gradients of the *loss* with the following line of code

```
loss.backward()
```

and update each of the parameters according to the Gradient Descent algorithm with the following line.

```
optimizer.step()
```

Finally, we reset the values of the gradients to zero with the following code.

```
optimizer.zero_grad()
```

Recall as discussed in the previous chapter that failing to do so will cause unintended training during subsequent iterations of backpropagation. Here, we called the *zero_grad()* function at the end of one iteration of backpropagation, but it may be a good idea to call this function right before calling *backward()*, just in case there are already gradients in the buffer before program execution (*e.g.*, if someone was working with the model beforehand in the interpreter).

In this section, we only showed how to run forward propagation and backpropagation on a single data point. In general, we train the model on the entire dataset multiple times. A single pass over the entire dataset is called an *epoch*.