2

Neural Networks

On being asked, "How is Perceptron performing today?" I am often tempted to respond, "Very well, thank you, and how are Neutron and Electron behaving?"

Frank Rosenblatt, inventor of the perceptron and also the Perceptron [6].

With our mathematical lessons concluded, we turn to an introductory overview of deep learning.

In §2.1, we introduce the basic components of neural network architectures – neurons, activations, biases, weights, and layers – in order to define the *multilayer perceptron* (MLP), a simple model that is iteratively composed of these basic components. Given that all deep networks are by definition iteratively composed of many structurally identical layers, MLPs will play the role of archetype network architecture for illustrating the principles of deep learning throughout the book. This class of neural-network models is rich enough to capture all the essential aspects of deep learning theory while simple enough to maintain the pedagogical focus of the book. Nevertheless, we'll also briefly comment on how one could work out an effective theory for other network architectures.

In §2.2, we list some common activation functions that are often used in practice.

Finally, we discuss in §2.3 how MLPs are initialized. Here, we make a key conceptual shift from thinking about the weights and biases as the random variables to thinking about the induced distribution over the neural activities and network outputs. The expressions we derive here will provide a natural starting point for our analysis in §4 when we start developing our effective theory of MLPs with general activation functions.

2.1 Function Approximation

The subject of *artificial neural networks* has a rich history as cognitive science- and neuroscience-inspired artificial intelligence.[1] Here, our starting point will be a discussion of the function, $f(x)$.

[1] The *artificial neuron* was invented by McCulloch and Pitts in 1943 [7] as a model of the *biological neuron*. Their neuron was essentially a perceptron with a bias, but it did not have learnable weights.

https://doi.org/10.1017/9781009023405.004 Published online by Cambridge University Press

Some functions are really simple, easily described in terms of the elementary operations: addition, subtraction, multiplication, and division. For instance, consider either the identity function $f(x) = x$ or the exponential function $f(x) = e^x$. The former is the definition of trivial, involving no operations. The latter is a special function and can be defined in many ways, e.g., through its Taylor series

$$e^x \equiv \sum_{k=0}^{\infty} \frac{x^k}{k!}. \tag{2.1}$$

This definition constructs the exponential function in terms of elementary operations of addition, multiplication, and division: the numerator x^k represents the repeated multiplication of the variable x for k times, and the factorial $k!$ in the denominator represents the repeated multiplication of integers $k! = 1 \times 2 \times \cdots \times (k-1) \times k$. Although this description of the exponential function involves a sum of an infinite number of terms, the actual instructions (2.1) for computing this function in terms of these simple operations are so compact that they take up only about one seventh of a line, and, for many purposes, it only takes the first few terms in the sum to get a useful approximation of e^x.

Some functions are really complicated, and their description in terms of elementary operations is unlikely to fit in the confines of any printed book. For instance, imagine a function $f(x)$ that takes as input an image x_i – represented as a vector of numbers corresponding to a black-and-white pixelated image – and outputs 1 if the image x_i depicts a cat and 0 otherwise. While such a classification function should exist since humans can recognize images of cats, it's not at all clear how to describe such a function in terms of simple operations like addition and multiplication. The subject of **artificial intelligence** (AI) is mostly concerned with functions of this sort: easy for humans to compute but difficult for humans to describe in terms of elementary operations.

The conceptual leap needed to represent such hard-to-describe functions is to start with a flexible *set* of functions $\{f(x;\theta)\}$, constructed from simple components parametrized by a vector of adjustable **model parameters** θ_μ. One then tries to tune these model parameters θ_μ judiciously in order to approximate the original complicated function such that $f(x;\theta^\star) \approx f(x)$. The description of the set of functions $\{f(x;\theta)\}$ along with the settings of the model parameters $\theta^\star_\mu$ then serve as a useful approximate description of the desired function $f(x)$. This is called **function approximation**, and the procedure for adjusting the model parameters θ_μ is called a **learning algorithm**.

To be more concrete, let us represent the collection of inputs to our function $f(x)$ as a set $\mathcal{D}$ of n_0-dimensional vectors,

$$\mathcal{D} = \{x_{i;\alpha}\}_{\alpha=1,\ldots,N_\mathcal{D}}, \tag{2.2}$$

called **input data**. Here, the **sample index** α labels each sample in the dataset of $N_\mathcal{D}$ elements, and the vector index $i = 1, \ldots, n_0$ labels the component of the input vector.

The perceptron model, with learnable weights, was invented by Rosenblatt in 1958 [8]. Deep learning really came into its own in 2012 [9] after the realization that the graphical processing unit (GPU) is well-suited for the parallel computations required to train and run neural networks.

https://doi.org/10.1017/9781009023405.004 Published online by Cambridge University Press

In our motivating example above, each number $x_{i;\alpha}$ refers to the i-th pixel of the α-th image in the dataset $\mathcal{D}$ of $N_{\mathcal{D}}$ images, each of which might or might not depict a cat. By adjusting the model parameters θ_μ so that the function $f(x;\theta^\star)$ outputs the correct answer for as much input data as possible, we can try to approximate the elusive cat-or-not function in a way that no longer defies description. The overall idea of *training* such functions using a dataset $\mathcal{D}$ – rather than *programming* them – goes by the name **machine learning** and stands in contrast to the conventional von Neumann model of the digital computer.

While any set of parameterized functions can be used for function approximation,[2] our focus will be on a particular set of composable functions originally derived from a simplified model of the brain. Such functions were originally termed **artificial neural networks** and are now just referred to as **neural networks**. **Deep learning** is a branch of machine learning that uses neural networks as function approximators, with a particular emphasis on stacking *many layers* of structurally similar components. Let's see how this works in more detail.

The most basic component of the neural network is the **neuron**. Loosely inspired by the behavior of biological neurons, the artificial neuron essentially consists of two simple operations:

- The **preactivation** z_i of a neuron is a linear aggregation of incoming signals s_j where each signal is weighted by W_{ij} and biased by b_i:

$$z_i(s) = b_i + \sum_{j=1}^{n_{\text{in}}} W_{ij}\, s_j \quad \text{for} \quad i = 1, \ldots, n_{\text{out}}. \tag{2.3}$$

- Each neuron then *fires* or not according to the weighted and biased evidence, i.e., according to the value of the preactivation z_i, and produces an **activation**

$$\sigma_i \equiv \sigma(z_i)\,. \tag{2.4}$$

  The scalar-valued function $\sigma(z)$ is called the **activation function** and acts independently on each component of the preactivation vector.

Taken together, these n_{out} neurons form a **layer**, which takes in the n_{in}-dimensional vector of signals s_j and outputs the n_{out}-dimensional vector of activations σ_i. With this collective perspective, a layer is parameterized by a vector of **biases** b_i and a matrix of **weights** W_{ij}, where $i = 1, \ldots, n_{\text{out}}$ and $j = 1, \ldots, n_{\text{in}}$, together with a fixed activation function $\sigma(z)$.

With these components, we can make an increasingly flexible set of functions by organizing many neurons into a layer and then iteratively stacking many such layers, so that the outgoing activations of the neurons in one layer become the input signals to the neurons in some other layer. The organization of the neurons and their pattern

[2] E.g., consider a sum of Gaussian functions, where the mean and variance of each Gaussian play the role of the adjustable parameters.

https://doi.org/10.1017/9781009023405.004 Published online by Cambridge University Press

of connections is known as the neural network **architecture**. The archetypal neural network architecture based on this principle of stacking layers of many neurons is called the **multilayer perceptron** (MLP).[3]

The activation function is usually chosen to be a nonlinear function in order to increase the expressivity of the neural-network function $f(x;\theta)$. The simplest – and historically first – activation function either fires or does not fire: $\sigma(z) = 1$ for $z \geq 0$ and $\sigma(z) = 0$ for $z < 0$. In other words, each neuron fires if and only if the weighted accumulated evidence $\sum_j W_{ij}\, x_j$ exceeds the firing threshold $-b_i$. More generally, activation functions are not binary and can incorporate the strength of the evidence into their output. In §2.2 we'll describe many of the commonly-used activation functions in deep learning.

The MLP is recursively defined through the following iteration equations:

$$z_i^{(1)}(x_\alpha) \equiv b_i^{(1)} + \sum_{j=1}^{n_0} W_{ij}^{(1)} x_{j;\alpha}\,, \quad \text{for} \quad i = 1, \ldots, n_1\,, \tag{2.5}$$

$$z_i^{(\ell+1)}(x_\alpha) \equiv b_i^{(\ell+1)} + \sum_{j=1}^{n_\ell} W_{ij}^{(\ell+1)} \sigma\Big(z_j^{(\ell)}(x_\alpha)\Big)\,, \quad \text{for} \quad i = 1, \ldots, n_{\ell+1}\,;\ \ell = 1, \ldots, L-1\,,$$

which describe a network with L layers of neurons, with each layer ℓ composed of n_ℓ neurons.[4] We depict an example MLP architecture in Figure 2.1. The number of layers L defines the **depth** of the network, and the different numbers of neurons in each layer $n_{\ell=1,\ldots,L-1}$ define the **widths** of the layers. The depth and hidden-layer widths are variable **architecture hyperparameters** that define the shape of the network, while the values of n_0 and n_L are set by input and output dimensions, respectively, of the function-approximation task. In particular, the final-layer preactivations computed by the network,

$$f(x;\theta) = z^{(L)}(x), \tag{2.6}$$

serve as the function approximator, with its model parameters θ_μ being the union of the biases and weights from all the layers. Sometimes it will be convenient to think of this collection of model parameters as an explicit vector θ_μ whose components cover all

[3] Here, the name "perceptron" was inherited from Rosenblatt's Perceptron architecture [8], which was originally envisioned for emulating *human perception*. The name perceptron is also used to refer to the original step-function activation function, cf. the first entry of §2.2.

[4] A more modern name for the MLP is the *fully-connected network* (FCN), highlighting the fact that each neuron in a given layer ℓ has a connection to every neuron in layer $\ell+1$, as Figure 2.1 makes clear. Such a dense pattern of connections is computationally expensive in terms of the number of parameters required for the architecture and should be contrasted with the sparser architectures described at the end of this section. To place an emphasis on the *deepness* of networks rather than on the *density* of the connections, we'll mainly stick with the name *multilayer* perceptron over the name *fully-connected* network in this book.

https://doi.org/10.1017/9781009023405.004 Published online by Cambridge University Press

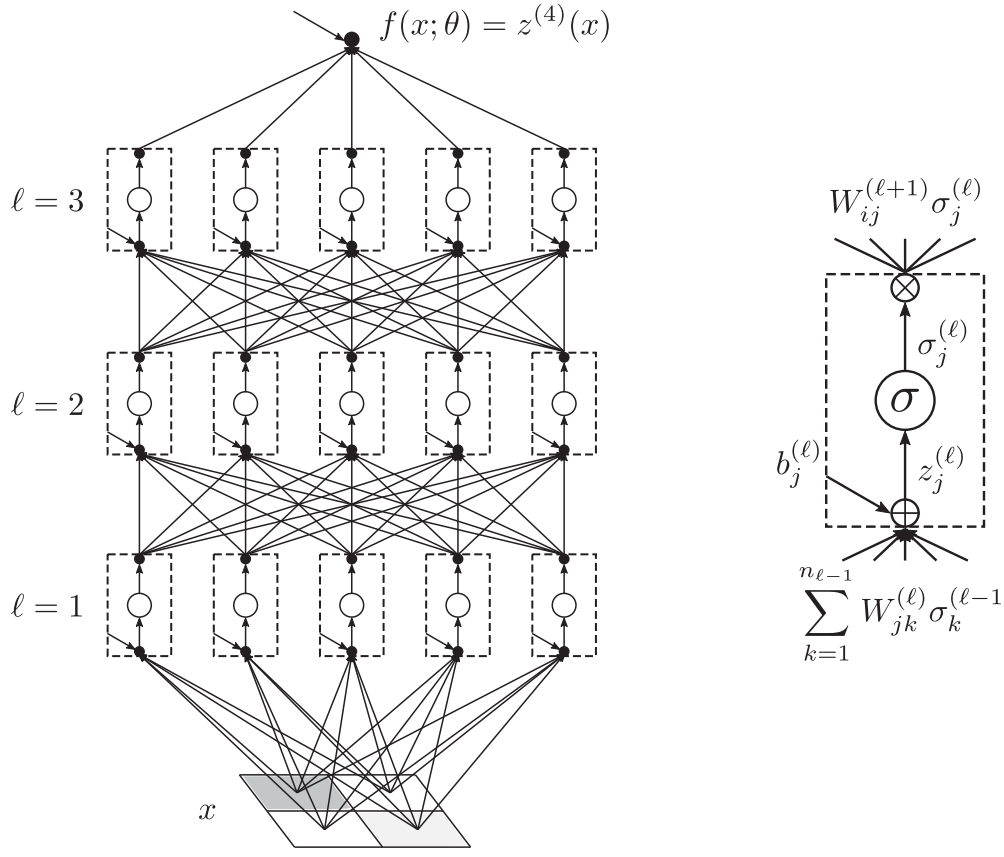

Figure 2.1 **Left:** depiction of the neurons and connections for an example multilayer perceptron (MLP) architecture. This particular MLP has $L = 4$ layers, defining a set of functions $f(x;\theta)$ with input dimension $n_0 = 4$ and output dimension $n_4 = 1$. The three hidden layers have five neurons each, $n_1, n_2, n_3 = 5$, implying $P = 91$ total model parameters. The graph describing the connections between neurons is a *directed acyclic graph*, meaning that signals only propagate in one direction and do not loop inside the network. For this reason, MLPs are also sometimes called *feedforward* networks. **Right:** the detailed structure of each neuron that *(i)* adds the bias and the weighted signals to produce the preactivation, *(ii)* generates the activation from the preactivation, and *(iii)* multiplies the activation by the next-layer weight.

the model parameters. In that case, the dimension of θ_μ and therefore the total number of model parameters is given by

$$P = \sum_{\ell=1}^{L} (n_\ell + n_\ell n_{\ell-1}) \,, \tag{2.7}$$

which scales quadratically with the widths of the network and linearly with the depth.

The intermediate layers $\ell = 1, \ldots, L-1$ are referred to as **hidden layers**, since preactivations and activations of the neurons from those layers are not part of the

https://doi.org/10.1017/9781009023405.004 Published online by Cambridge University Press

network's output. On the one hand, the variables $z^{(\ell)}(x)$ for $\ell < L$ are simply temporary variables introduced to construct an increasingly flexible set of functions, expressive enough to have a chance of approximating hard-to-describe functions. On the other hand, by analogy to the physical brain, these variables are thought to encode useful information about *how* the neural network is approximating; for example, a particular neuron might fire if it recognizes a tail, a whisker, or a pattern representing fur – all potentially useful *features* for determining whether an image contains a cat or not.

Moving beyond MLPs, the choice of neural network architecture is often motivated by the nature of the function we are trying to approximate. For instance, the properties of the dataset $\mathcal{D}$, when known and articulated, can be used to build **inductive biases** into the architecture so that the resulting set of functions may better represent the underlying function.[5] Let's look at a few examples.

- For **computer vision** (CV) applications, **convolutional neural networks** (CNN) or conv-nets [9–13] are used to take advantage of the fact that information in images is organized in a spatially local manner, often respecting *translational invariance*.[6]

- For **natural language processing** (NLP) applications, the **transformer** architecture (no acronym yet) is used to process sequential input – such as a paragraph of text or an amino acid sequence coding a protein – in a way that encourages correlations to develop between any of the elements in the sequence [14]. This property of the model is aptly called *attention*.

An important property of these inductive biases is that they induce constraints or relationships between the weights. For instance, we can think of the convolutional layer as a particular type of MLP layer, where many weights are set to zero, and the values of remaining weights are further shared among several different neurons. This property is known as *weight tying*. That means that convolutional layers are actually within the class of functions describable by using MLP layers, but they are very unlikely to be found via training unless the constraints are explicitly enforced. As long as the inductive

[5] We'll discuss the inductive bias of MLPs from various different perspectives in §6, §11, and Epilogue ε.

[6] For a two-dimensional convolutional layer, the iteration equation (2.5) for MLPs is replaced by

$$z^{(\ell+1)}_{i,(c,d)}(x_\alpha) \equiv b^{(\ell+1)}_i + \sum_{j=1}^{n_\ell} \sum_{c'=-k}^{k} \sum_{d'=-k}^{k} W^{(\ell+1)}_{ij} \sigma\left(z^{(\ell)}_{j,(c+c',d+d')}(x_\alpha)\right), \tag{2.8}$$

where in $z^{(\ell)}_{i,(c,d)}$, the first index i is an auxiliary *channel* index and the paired index (c, d) is a two-dimensional spatial index, and the number k is a fixed constant for each layer, determining the size of the convolutional window. In particular, the same weights are used on different spatial locations of the input, which promotes the inductive bias that image data are often translationally invariant. In other words, a cat is still a cat regardless of its location in an image. At the time of writing, the convolutional layer is an essential part of many modern deep learning architectures, but this situation may change in the future. Please pay *attention*.

https://doi.org/10.1017/9781009023405.004 Published online by Cambridge University Press

bias of spatial locality and translational invariance is well founded, the convolution layer has obvious computational advantages by heavily curtailing the number of weights to be trained and stored.

Regardless of these specific inductive biases ingrained into modern neural network architectures used in deep learning, the common thread to all is the idea of constructing a flexible set of functions by organizing neural components into many iterated layers. MLPs are the simplest of these neural network architectures that hinge on this stacking idea, and thus provide a *minimal model* for an **effective theory of deep learning**. Specifically, we expect (*a*) the *principles of deep learning theory* that we uncover to be general and valid across the large variety of architectures that are based on the idea of stacking many layers of neural components and (*b*) that the resulting effective theory formalism can be specialized to specific architectures of interest as needed, using this book as a guide for how to work out such a theory. In particular, one can study other architectures in our formalism simply by swapping out the MLP iteration equation (2.5) – e.g., for the convolution layer iteration equation (2.8) – in the appropriate place. We'll provide pointers on where to make such substitutions when we begin working out our effective theory in §4.

Finally, in Appendix B we'll study neural networks with *residual connections*, known as **residual networks**. These architectures are specially modified to enable the training of deeper and deeper networks. In the final section of that appendix, we'll also explain how our effective theory approach can be extended to *general* residual networks, including the *residual convolutional network* – or **ResNet** – and the *transformer* architecture.

2.2 Activation Functions

In this section, we discuss some of the most common activation functions. This list is not exhaustive, so hopefully you won't find this section exhausting. To make it easier for you, we've plotted all these activation functions together in Figure 2.2. In §5, we'll use our effective theory to evaluate the relative usefulness of these activation functions in allowing input signals to effectively pass through a deep network.

Perceptron

The `perceptron` was the original activation function [7]. It is just a step function

$$\sigma(z) = \begin{cases} 1\,, & z \geq 0, \\ 0\,, & z < 0, \end{cases} \tag{2.9}$$

corresponding to a computer scientist's notion of simplicity: the neuron either *fires* and outputs 1 or *doesn't fire* and outputs 0.[7]

Despite the logical simplicity, this turns out to be a poor choice. As we will see, in order to both effectively pass signals through networks (§5 and §9) and train them (§10),

[7] Alternatively, the `perceptron` may be shifted and scaled such that $\sigma(z) = \text{sign}(z)$.

https://doi.org/10.1017/9781009023405.004 Published online by Cambridge University Press

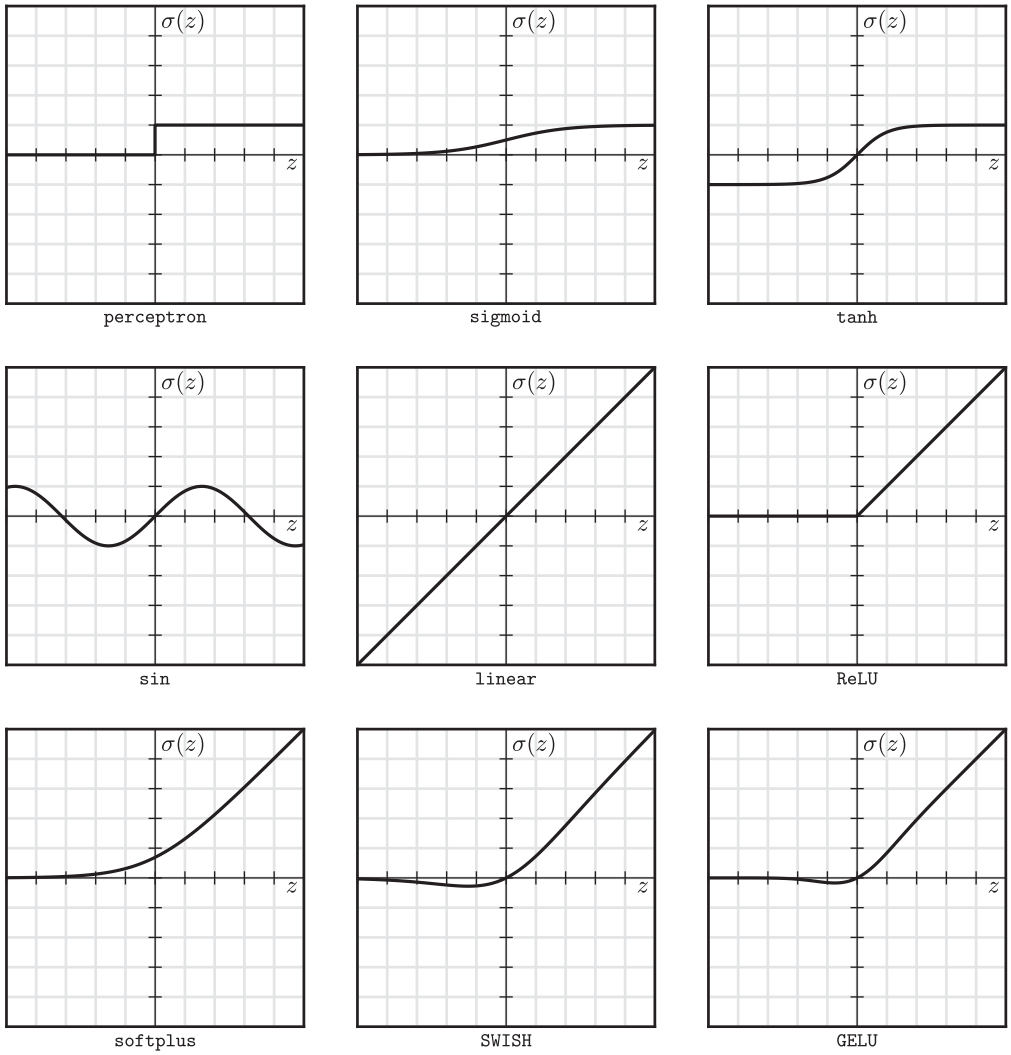

Figure 2.2 Commonly-used activation functions $\sigma(z)$. Grids are in units of one for both the preactivation z and activation σ. (The `leaky ReLU` is not shown.)

it's helpful to propagate more than one bit of information about the preactivation z. The `perceptron` has historical significance but is never used in deep neural networks.

Sigmoid

The `sigmoid` activation function is a logistic function

$$\sigma(z) = \frac{1}{1+e^{-z}} = \frac{1}{2} + \frac{1}{2}\tanh\left(\frac{z}{2}\right), \tag{2.10}$$

which is a smoothed version of the `perceptron`. Not only is it continuous, but it also preserves information about the magnitude of the preactivation, albeit mostly in the

https://doi.org/10.1017/9781009023405.004 Published online by Cambridge University Press

range near $z = 0$ where the function is nearly linear. Outside of this range, the `sigmoid` heavily compresses such information as it becomes more and more `perceptron`-like, *saturating* as $\sigma(z) = 1$ when $z \to \infty$ and as $\sigma(z) = 0$ when $z \to -\infty$.

As a mapping from the domain of $(-\infty, \infty)$ to the range $[0, 1]$, the `sigmoid` also has a natural interpretation of converting log-odds to a probability, which is its main application in machine learning. For deep learning, the differentiability of the `sigmoid` was essential in the development of a learning algorithm – backpropagation – for training neural networks with hidden layers [15]. Nevertheless, the `sigmoid` activation function is still a poor choice in *deep* neural networks: as we'll see in §5, a problem arises from the fact that it doesn't pass through the origin.

Tanh

The hyperbolic tangent or `tanh` activation function

$$\sigma(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} = \frac{e^{2z} - 1}{e^{2z} + 1}, \tag{2.11}$$

is a scaled (both before and after the activation) and shifted `sigmoid`, as is clear from (2.10). Of particular importance is the fact that it's shifted such that $\sigma(0) = 0$ [16].

The `tanh` is probably the most popular choice of activation function aside from the `ReLU` or `ReLU`-like activation functions to be discussed shortly, and arguably `tanh` is the most popular smooth activation function. As an exemplary smooth activation function, the `tanh` will be of significant interest for us in this book.

Sin

The `sin` activation function is just what it sounds like:

$$\sigma(z) = \sin(z), \tag{2.12}$$

i.e., one of the three standard trigonometric functions. Periodic nonlinearities have been cycling in and out of popularity for a long while now, see, e.g., [17], though they have never really achieved true popularity.

Scale-Invariant: Linear, ReLU, and Leaky ReLU

A scale-invariant activation function is any activation function that satisfies

$$\sigma(\lambda z) = \lambda \sigma(z)\,, \tag{2.13}$$

for any positive rescaling λ. We call these activation functions scale-invariant because any scaling of the preactivation $z \to \lambda z$ can be undone by an inverse scaling of the activation $\sigma(z) \to \lambda^{-1}\sigma(z)$. This condition is met by – and only by[8] – activation functions of the form

[8] In order to prove this necessity statement, first take the derivative of the scale-invariance equation (2.13) with respect to z, which gives $\sigma'(\lambda z) = \sigma'(z)$ for any $\lambda > 0$. Then note that this enforces a constant derivative, a_+, for $z > 0$ and another constant derivative, a_-, for $z < 0$. Finally, to satisfy (2.13) we also must have $\lim_{z \to \pm 0} \sigma(z) = 0$. Quantum Electrodynamics.

https://doi.org/10.1017/9781009023405.004 Published online by Cambridge University Press

$$\sigma(z) = \begin{cases} a_+ z\,, & z \geq 0, \\ a_- z\,, & z < 0. \end{cases} \tag{2.14}$$

The class of scale-invariant activation functions includes `linear` ($a_+ = a_- = a$), Rectified Linear Unit or `ReLU` ($a_+ = 1$, $a_- = 0$) [18, 19], and `leaky ReLU` ($a_+ = 1$, $a_- = a$) [20] activation functions. The `ReLU` is the most popular of the activation functions used in deep neural networks and therefore will be of substantial interest for us in this book.

In order to deepen our understanding of scale invariance, let's consider how other activation functions can break it. For instance, consider the `tanh` activation function $\sigma(z) = \tanh(z)$. Mathematically, `tanh` violates scale invariance because $\tanh(\lambda z) \neq \lambda \tanh(z)$ unless $\lambda = 1$. In particular, while the activation function is approximately linear for small preactivations, i.e., $\tanh(z) \approx z$ for $|z| \ll 1$, it saturates for large preactivations, i.e., $|\tanh(z)| \approx 1$ for $|z| \gg 1$. Thus, `tanh` comes with an intrinsic crossover scale $|z| \sim 1$ that separates the two regimes. We can see this visually in Figure 2.2: if we zoom out, all the non-scale-invariant activation functions – e.g., `perceptron`, `sigmoid`, and `tanh` – will look squashed, while the scale-invariant activation functions – e.g., `ReLU` and `linear` – will look the same at any scale.

Finally, note that all the scale-invariant activation functions – except the aptly-named `linear` activation – create a nonlinear relationship between the network inputs and outputs due to the kink at the origin $z = 0$. Stacking up many layers of neurons with these nonlinear activation functions accumulates the nonlinearity, allowing such deep neural networks to express highly nonlinear functions.

ReLU-like: Softplus, SWISH, and GELU

Despite the popularity of the `ReLU`, there's an uneasiness about the fact that it's not smooth. In an attempt to rectify the situation, a variety of smoothed-out `ReLU`-like activations have been proposed and have achieved some popularity, of which we will consider the following three:

- The `softplus` activation function [21],

$$\sigma(z) = \log(1 + e^z)\,, \tag{2.15}$$

  behaves linearly $\sigma(z) \approx z$ for a large argument $z \gg 1$ and vanishes exponentially for a negative argument, $\sigma(z) \approx e^{-|z|}$ for $z < 0$. Importantly, the `softplus` does not pass through the origin: $\sigma(0) = \log(2)$.

- The `SWISH` activation function [22] is defined as

$$\sigma(z) = \frac{z}{1 + e^{-z}}, \tag{2.16}$$

  which is a logistic function (2.10) multiplied by the preactivation z. The logistic function behaves as a continuous on/off switch, and so the `SWISH` approximates the `ReLU`, which we recall was defined as a discrete on/off switch multiplied by the preactivation z. In particular, for $z > 0$ the `SWISH` behaves as $\sigma(z) \approx z$, but for $z < 0$ it behaves as $\sigma(z) \approx 0$. Also, the multiplication by z ensures that the `SWISH` passes through the origin: $\sigma(0) = 0$.

https://doi.org/10.1017/9781009023405.004 Published online by Cambridge University Press

- The Gaussian Error Linear Unit (`GELU`) activation function [23] is a lot like the `SWISH`. It's given by the expression

$$\sigma(z) = \left[\frac{1}{2} + \frac{1}{2}\text{erf}\left(\frac{z}{\sqrt{2}}\right)\right] \times z, \tag{2.17}$$

  where the error function $\text{erf}(z)$ is given by

$$\text{erf}(z) \equiv \frac{2}{\sqrt{\pi}} \int_0^z dt\, e^{-t^2}\,, \tag{2.18}$$

  which is a partial integration of the Gaussian function. In particular, the graph of $\text{erf}(z)$ looks very similar to graph of $\tanh(z)$, and so the graph of the scaled and shifted version used in the definition of the `GELU`, $\frac{1}{2} + \frac{1}{2}\text{erf}\left(\frac{z}{\sqrt{2}}\right)$, looks very similar to the graph of the logistic function (2.10). Like the `SWISH`, it crosses the origin and behaves more like the `ReLU` the further we go away from 0 in either direction.

In smoothing the `ReLU`, all three of these activation functions introduce an intrinsic scale and violate the scale-invariance condition (2.13).

2.3 Ensembles

As we discussed in §2.1, neural networks are *trained* rather than *programmed*. Practically speaking, to begin training a neural network for function approximation, we need to set initial values of the biases $b_i^{(\ell)}$ and weights $W_{ij}^{(\ell)}$. Since the learned values of these model parameters are almost always iteratively built up from their initial values, the initialization strategy can have a major impact on the success or failure of the function approximation.

Perhaps the simplest strategy would be to set all the biases and weights to zero: $b_i^{(\ell)} = W_{ij}^{(\ell)} = 0$. However, this initialization fails to break the permutation symmetry among the n_ℓ different neurons in a hidden layer ℓ. If this symmetry isn't broken, then we cannot distinguish between the different neurons in a layer as all these neurons perform exactly the same computation. In effect, the network would behave as if it only had a single neuron $n_\ell = 1$ in each hidden layer. Thus, in order to leverage all the different components of the biases and weights in a wider network, we need to somehow break the permutation symmetry.

Perhaps the simplest strategy that breaks this permutation symmetry is to sample each bias and weight independently from some *probability distribution*. Theoretically speaking, we should pick this **initialization distribution** so that the resulting **ensemble** of networks is well behaved with respect to the function-approximation task. This section initializes ourselves for analyzing such an ensemble.

https://doi.org/10.1017/9781009023405.004 Published online by Cambridge University Press

Initialization Distribution of Biases and Weights

Among the many potential reasonable choices for the initialization distribution, the obvious choice is the Gaussian distribution.[9] As we discussed, Gaussian distributions are defined solely in terms of their mean and variance, so they're easy to specify and work with in theory. They're also extremely easy to sample from, which is also an essential consideration when picking a sampling distribution in practice.

In particular, to initialize MLPs, we'll independently sample each bias and each weight from zero-mean Gaussian distributions with variances given by

$$\mathbb{E}\left[b_{i_1}^{(\ell)} b_{i_2}^{(\ell)}\right] = \delta_{i_1 i_2} C_b^{(\ell)}, \tag{2.19}$$

$$\mathbb{E}\left[W_{i_1 j_1}^{(\ell)} W_{i_2 j_2}^{(\ell)}\right] = \delta_{i_1 i_2}\delta_{j_1 j_2}\frac{C_W^{(\ell)}}{n_{\ell-1}}, \tag{2.20}$$

respectively. Here the Kronecker deltas indicate that each bias and each weight are all drawn independently from the others. Explicitly, the functional forms of these Gaussian distributions are given by

$$p\left(b_i^{(\ell)}\right) = \frac{1}{\sqrt{2\pi C_b^{(\ell)}}} \exp\left[-\frac{1}{2C_b^{(\ell)}}\left(b_i^{(\ell)}\right)^2\right], \tag{2.21}$$

$$p\left(W_{ij}^{(\ell)}\right) = \sqrt{\frac{n_{\ell-1}}{2\pi C_W^{(\ell)}}} \exp\left[-\frac{n_{\ell-1}}{2C_W^{(\ell)}}\left(W_{ij}^{(\ell)}\right)^2\right]. \tag{2.22}$$

Here the normalization of weight variances by $1/n_{\ell-1}$ is purely conventional, but, as we will show explicitly in §3 and §4, it is necessary for wide neural networks and natural for comparing the behavior of networks with different widths.[10] Also note that we allow the bias variance $C_b^{(\ell)}$ and rescaled weight variance $C_W^{(\ell)}$ to potentially vary from layer to layer. Together, the set of bias variances $\left\{C_b^{(1)}, \ldots, C_b^{(L)}\right\}$ and the set of rescaled weight variances $\left\{C_W^{(1)}, \ldots, C_W^{(L)}\right\}$ are called **initialization hyperparameters**. One practical result of our effective theory approach will be prescriptions for setting these initialization hyperparameters so that the output of the neural network is well behaved.

[9] Two other choices seen in the wild for the initialization distribution are the uniform distribution and the truncated normal distribution. For the weights, the difference between the Gaussian distribution and any other distribution – when the means are set zero and the variances are set equal – turns out to be suppressed by 1/width for wide networks. That is, due to the central limit theorem, ultimately only the first and second moment – i.e., the mean and variance – for the weight initialization distribution is of any real consequence. Thus, we might as well just use a Gaussian distribution. For the biases, the difference between the Gaussian distribution and any other distribution is mostly moot in practice because we shall find that the bias variance $C_b^{(\ell)}$ should be set to zero for good activation functions.

[10] We can trace this convention to the MLP iteration equation (2.5). To compute the preactivation $z_{i;\alpha}^{(\ell)} = b_i^{(\ell)} + \sum_{j=1}^{n_{\ell-1}} W_{ij}^{(\ell)} \sigma_{j;\alpha}^{(\ell-1)}$, we essentially add together $n_{\ell-1}$ random weights. For large $n_{\ell-1}$, the normalization factor of $1/n_{\ell-1}$ in the variance – which is tantamount to normalizing each weight by $1/\sqrt{n_{\ell-1}}$ – essentially counteracts this summation of many zero-mean random numbers. Since there is no such summation for the biases, there is no need for such a normalization factor.

https://doi.org/10.1017/9781009023405.004 Published online by Cambridge University Press

Induced Distributions

Given a dataset $\mathcal{D} = \{x_{i;\alpha}\}$ consisting of $N_{\mathcal{D}}$ input data, an MLP with model parameters $\theta_\mu = \left\{b_i^{(\ell)}, W_{ij}^{(\ell)}\right\}$ evaluated on $\mathcal{D}$ outputs an array of $n_L \times N_{\mathcal{D}}$ numbers

$$f_i(x_\alpha; \theta) = z_i^{(L)}(x_\alpha) \equiv z_{i;\alpha}^{(L)}, \tag{2.23}$$

indexed by both **neural indices** $i = 1, \ldots, n_L$ and **sample indices** $\alpha = 1, \ldots, N_{\mathcal{D}}$. Each time we instantiate MLPs by drawing model parameters θ_μ from the initialization distribution $p(\theta)$, we get a different initial set of outputs $z_{i;\alpha}^{(L)}$. It follows that since the biases $b_i^{(\ell)}$ and weights $W_{ij}^{(\ell)}$ are random variables at initialization, then so must be the network outputs $z_{i;\alpha}^{(L)}$. In this way, the initialization distribution *induces* a distribution on the network outputs.

This **output distribution** $p\left(z^{(L)} \middle| \mathcal{D}\right)$ controls the statistics of network outputs at the point of initialization. In practice, the properties of this distribution are directly related to how hard it is for a network to approximate its target function through iterated adjustments of its model parameters. As such, having control over this distribution is of significant interest from a practitioner's perspective. From a theorist's perspective, even though the initialization distribution for model parameters is simple by design, the induced output distribution is not. In theory, we need to calculate the following gigantic integral over all the model parameters:

$$p\left(z^{(L)} \middle| \mathcal{D}\right) = \int \left[\prod_{\mu=1}^{P} d\theta_\mu\right] p\left(z^{(L)} \middle| \theta, \mathcal{D}\right) p(\theta). \tag{2.24}$$

Before performing this heroic integration, notice that the conditional distribution $p\left(z^{(L)} \middle| \theta, \mathcal{D}\right)$ in the integrand (2.24) is actually *deterministic*. In other words, if we know the set of inputs $\mathcal{D}$ and the settings of all the model parameters θ_μ, then we know how to compute the network outputs: we just use the iteration equation (2.5) that defines the MLP. What we don't yet know is how to express this determinism as a distribution.

Deterministic Distributions and the Dirac Delta Function

What kind of a probability distribution is *deterministic*? Let's abstractly denote such a distribution as $p(z|s) = \delta(z|s)$, which intends to encode the deterministic relationship $z = s$. What properties should this distribution have? First, the mean of z should be s:

$$\mathbb{E}\left[z\right] = \int dz\ \delta(z|s)\, z \equiv s. \tag{2.25}$$

Second, the variance should vanish, since this is a deterministic relationship. In other words,

$$\mathbb{E}[z^2] - \left(\mathbb{E}\left[z\right]\right)^2 = \left[\int dz\ \delta(z|s)\, z^2\right] - s^2 \equiv 0, \tag{2.26}$$

https://doi.org/10.1017/9781009023405.004 Published online by Cambridge University Press

or, equivalently,

$$\int dz\ \delta(z|s)\, z^2 \equiv s^2. \tag{2.27}$$

In fact, this determinism implies an even stronger condition. In particular, the expectation of any function $f(z)$ of z, should evaluate to $f(s)$:

$$\mathbb{E}\left[f(z)\right] = \int dz\ \delta(z|s)\, f(z) \equiv f(s), \tag{2.28}$$

which includes the properties (2.25) and (2.27) as special cases when $f(z) = z$ and $f(z) = z^2$, respectively, as well as the probability normalization condition

$$\int dz\ \delta(z|s) = 1, \tag{2.29}$$

when $f(z) = 1$.[11] In fact, (2.28) is the defining property of the **Dirac delta function**.[12]

As a representation though, (2.28) is a little too abstract, even for us. However, our discussion above paves the way for a much more concrete representation. Since the Dirac delta function is a normalized distribution (2.29) with mean s (2.25) and zero variance (2.26), let's consider a normalized Gaussian distribution with mean s (1.9) and take the limit as the variance K goes to zero:

$$\delta(z|s) \equiv \lim_{K\to+0} \frac{1}{\sqrt{2\pi K}} e^{-\frac{(z-s)^2}{2K}}. \tag{2.30}$$

This distribution is infinitely peaked at $z = s$ while vanishing everywhere else, so any function $f(z)$ integrated against (2.30) will give $f(s)$ after taking the limit. In other words, it satisfies the defining property of the Dirac delta function (2.28).

The limit in (2.30) should always be taken after integrating the distribution against some function. Having said that, perhaps this representation still makes you a little bit uncomfortable as it is still a very singular limit. Let's try to fix this and find an even better representation. Here's a magic trick: starting from (2.30), let's insert "1" on the right-hand side as

$$\begin{aligned}\delta(z|s) &= \lim_{K\to+0} \frac{1}{\sqrt{2\pi K}} e^{-\frac{(z-s)^2}{2K}} \left\{ \frac{1}{\sqrt{2\pi/K}} \int_{-\infty}^{\infty} d\Lambda\ \exp\left[-\frac{K}{2}\left(\Lambda - \frac{i(z-s)}{K}\right)^2 \right] \right\} \\ &= \lim_{K\to+0} \frac{1}{2\pi} \int_{-\infty}^{\infty} d\Lambda\ \exp\left[-\frac{1}{2}K\Lambda^2 + i\Lambda(z-s) \right],\end{aligned} \tag{2.31}$$

[11] A random variable that obeys $\mathbb{E}\left[f(z)\right] = f(\mathbb{E}\left[z\right])$ is said to *self-average*, meaning that we can exchange the order of the expectation with the function evaluation. The condition (2.28) is equivalent to saying that the distribution $\delta(z|s)$ is self-averaging.

[12] The Dirac delta function is really a generalization of the Kronecker delta (1.26) for continuous variables. In this footnote we also include the obligatory disclaimer that – despite its name – the Dirac delta function is a *distribution* and not a *function*, as should have been clear from our discussion. Despite this, we will stick with common convention and continue to refer to it as the *Dirac delta function*.

https://doi.org/10.1017/9781009023405.004 Published online by Cambridge University Press

where in the curly brackets we inserted an integral over a dummy variable Λ of a normalized Gaussian with variance $1/K$ and imaginary mean $i(z-s)/K$, and on the second line we simply combined the exponentials. Now we can easily take the limit $K \to +0$ to find an *integral representation* of the Dirac delta function:

$$\delta(z|s) = \frac{1}{2\pi}\int_{-\infty}^{\infty} d\Lambda\; e^{i\Lambda(z-s)} \equiv \delta(z-s). \tag{2.32}$$

In this final expression, we note that the function depends only on the difference $z-s$. This integral representation will come in handy in §4.

Induced Distributions, Redux

Now that we are familiar with the Dirac delta function, we can use it to express the output distribution (2.24) more concretely. To start, for a one-layer network of depth $L=1$, the distribution of the first layer output (2.5) is given by

$$\begin{aligned} p\Big(z^{(1)}\Big|\mathcal{D}\Big) = \int \left[\prod_{i=1}^{n_1} db_i^{(1)}\; p\Big(b_i^{(1)}\Big)\right]\left[\prod_{i=1}^{n_1}\prod_{j=1}^{n_0} dW_{ij}^{(1)}\; p\Big(W_{ij}^{(1)}\Big)\right] \\ \times \left[\prod_{i=1}^{n_1}\prod_{\alpha\in\mathcal{D}} \delta\left(z_{i;\alpha}^{(1)} - b_i^{(1)} - \sum_{j=1}^{n_0} W_{ij}^{(1)} x_{j;\alpha}\right)\right]. \end{aligned} \tag{2.33}$$

Here, we needed $n_1 \times N_{\mathcal{D}}$ Dirac delta functions, one for each component of $z_{i;\alpha}^{(1)}$. In §4.1 we will explicitly evaluate the above integrals, though you should feel free to do so now on your own, if you're impatient. In passing, let us also introduce a cousin of (2.33),

$$\begin{aligned} p\Big(z^{(\ell+1)}\Big|z^{(\ell)}\Big) = \int \left[\prod_{i=1}^{n_{\ell+1}} db_i^{(\ell+1)}\; p\Big(b_i^{(\ell+1)}\Big)\right]\left[\prod_{i=1}^{n_{\ell+1}}\prod_{j=1}^{n_\ell} dW_{ij}^{(\ell+1)}\; p\Big(W_{ij}^{(\ell+1)}\Big)\right] \\ \times \left[\prod_{i=1}^{n_{\ell+1}}\prod_{\alpha\in\mathcal{D}} \delta\left(z_{i;\alpha}^{(\ell+1)} - b_i^{(\ell+1)} - \sum_{j=1}^{n_\ell} W_{ij}^{(\ell+1)}\sigma\Big(z_{j;\alpha}^{(\ell)}\Big)\right)\right], \end{aligned} \tag{2.34}$$

which determines the distribution of the preactivations in the $(\ell+1)$-th layer, conditioned on the preactivations in the ℓ-th layer, after integrating out the model parameters.

More generally, for any parameterized model with output $z_{i;\alpha}^{\text{out}} \equiv f_i(x_\alpha;\theta)$ for $i = 1,\ldots,n_{\text{out}}$ and with the model parameters θ_μ distributed according to $p(\theta)$, the output distribution (2.24) can be written using the Dirac delta function as

$$p\Big(z^{\text{out}}\Big|\mathcal{D}\Big) = \int \left[\prod_{\mu=1}^{P} d\theta_\mu\right] p(\theta)\left[\prod_{i=1}^{n_{\text{out}}}\prod_{\alpha\in\mathcal{D}} \delta\Big(z_{i;\alpha}^{\text{out}} - f_i(x_\alpha;\theta)\Big)\right]. \tag{2.35}$$

Our pretraining was designed precisely to prepare ourselves for performing this integral for MLPs.

https://doi.org/10.1017/9781009023405.004 Published online by Cambridge University Press