1 Neural Networks

1.1 Introduction

Neural networks are functions loosely modeled on the brain. In the brain, we have billions of neurons that connect to one another. Each neuron can be thought of as a node in a graph, and the edges are the connections from one neuron to the next (figure 1.1). The edges are directed; electrical signals propagate in just one direction along the wires in the brain.

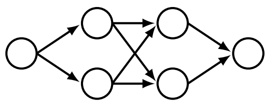
Figure 1.1: A neural network can be drawn as a directed graph.

Outgoing edges are called axons and incoming edges are called dendrites. A neuron fires, sending a pulse down its axon, when the incoming pulses, from the dendrites, exceed a threshold.

1.2 The Perceptron: A Simple Model of A Single Neuron

Let's consider a neuron, shaded in gray, that has four inputs and one output (figure 1.2).

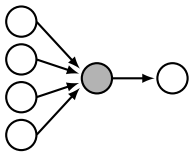
Figure 1.2: Perceptron.

A simple model for this neuron is the **perceptron**. A perceptron is a neuron with N inputs $\{x_i\}_{i=1}^N$ and one output y, that maps inputs to outputs according to the following equations:

$$z = f(\mathbf{x}) = \sum_{i=1}^{N} w_i x_i + b = \mathbf{w}^\mathsf{T}\mathbf{x} + b \qquad \triangleleft \text{ linear layer} \tag{1.1}$$

$$g(z) = \begin{cases} 1, & \text{if} \quad z > 0 \\ 0, & \text{otherwise} \end{cases} \qquad \triangleleft \text{ activation function} \tag{1.2}$$

$$y = g(f(\mathbf{x})) \qquad \triangleleft \text{ perceptron} \tag{1.3}$$

In words, we take a weighted sum of the inputs and, if that sum exceeds a threshold (here 0), the neuron fires (outputs a 1). The function f is called a **linear layer** because it computes a linear function of the inputs, $\mathbf{w}^\mathsf{T}\mathbf{x}$, plus a **bias**, b. The function g is called the **activation function** because it decides whether the neuron activates (fires).

Mathematically, f is an affine function, but by convention we call it a "linear layer." One way to think of it is f is a linear function of $\begin{bmatrix}\mathbf{x}\\1\end{bmatrix}$.

1.2.1 The Perceptron as a Classifier

People got excited about perceptrons in the late 1950s because it was shown that they can learn to classify data [8]. Let's see how that works. We will consider a perceptron with two inputs, x_1 and x_2, and one output, y. Let the incoming connection weights be $w_1 = 2$, $w_2 = 1$, and $b = 0$. The values of z and y, as a function of x_1 and x_2, are shown in figure 1.3.

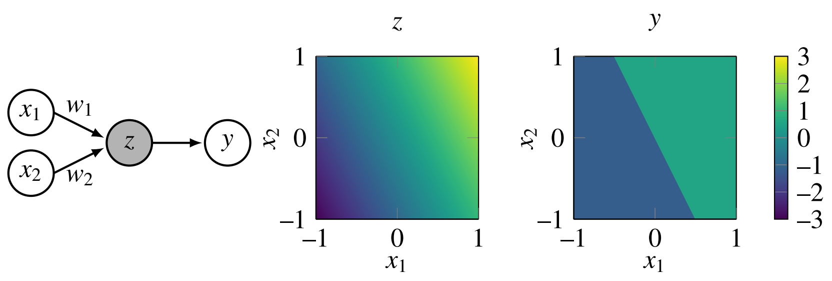

Figure 1.3: Value of hidden unit and output unit in a perceptron, as a function of the input data.

where y takes on values 0 or 1; so you can think of this as a classifier that assigns a class label of 1 to the upper-right half of the plotted region.

1.2.2 Learning with a Perceptron

So a perceptron acts like a classifier, but how can we use it to learn? The idea is that given data, $\{\mathbf{x}^{(i)}, y^{(i)}\}_{i=1}^N$, we will adjust the weights $\mathbf{w}$ and the bias b, in order to minimize a classification loss, $\mathcal{L}$:

$$\mathbf{w}^*, b^* = \arg\min_{\mathbf{w},b} \frac{1}{N}\sum_{i=1}^{N} \mathcal{L}(\mathbf{w}^\mathsf{T}\mathbf{x}^{(i)} + b, y^{(i)}) \tag{1.4}$$

In figure 1.4, this optimization process corresponds to shifting and rotating the **decision boundary**, until you find a line that separates data labeled as $y = 0$ from data labeled as $y = 1$.

You might be wondering, what's the exact optimization algorithm that will find the best line that separates the classes? The original perceptron paper proposed one particular algorithm, the "perceptron learning algorithm." This was an optimizer tailored to the specific structure of the perceptron. Older papers on neural nets are full of specific learning rules for specific architectures: the delta rule, the Rescorla-Wagner model, and so forth [7]. Nowadays we rarely use these special-purpose algorithms. Instead, we use *general-purpose* optimizers like gradient descent (for differentiable objectives) or zeroth-order methods (for nondifferentiable objectives). The next chapter will cover the backpropagation algorithm, which is a general-purpose gradient-based optimizer that applies to essentially all neural

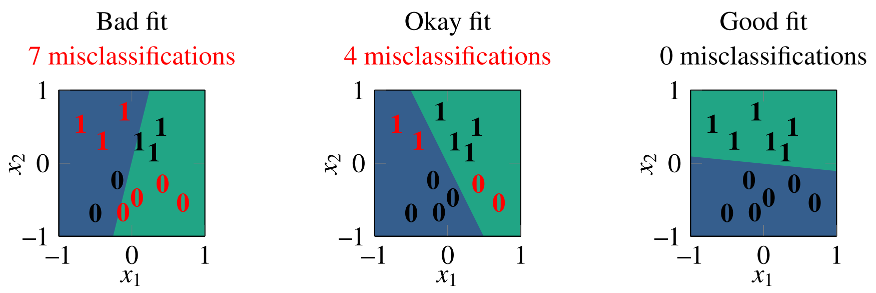

Figure 1.4: Different possible decision surfaces of a perceptron.

networks we will see in this book (but, note that for the perceptron objective, because it has a non-differentiable threshold function, we would instead opt for a zeroth order optimizer).

1.3 Multilayer Perceptrons

Perceptrons can solve linearly separable binary classification problems, but they are otherwise rather limited. For one, they only produce a single output. What if we want multiple outputs? We can achieve this by adding edges that fan out after the perceptron (figure 1.5).

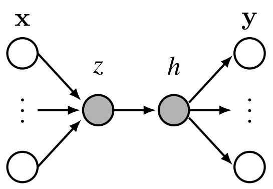

Figure 1.5: Multiple outputs fan out from a neuron.

This network maps an input **layer** of data $\mathbf{x}$ to a layer of outputs $\mathbf{y}$. The neurons in between inputs and outputs are called **hidden units**, shaded in gray. Here, z is a **preactivation** hidden unit and h is a **postactivation** hidden unit, that is, $h = g(z)$ where $g(\cdot)$ is an activation function like in equation (1.2).

More commonly we might have many hidden units in stack, which we call a **hidden layer** (figure 1.6).

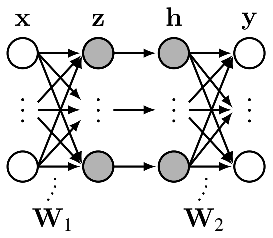

Figure 1.6: Mutilayer perceptron.

How many layers does this net have? Some texts will say two [$\mathbf{W}_1$, $\mathbf{W}_2$], others three [$\mathbf{x}$, $\{\mathbf{z},\mathbf{h}\}$, $\mathbf{y}$], others four [$\mathbf{x}$, $\mathbf{z}$, $\mathbf{h}$, $\mathbf{y}$]. We must get comfortable with the ambiguity.

Because this network has multiple layers of neurons, and because each neuron in this net acts as a perceptron, we call it a **multilayer perceptron** (**MLP**). The equation for this MLP is:

$$\mathbf{z} = \mathbf{W}_1\mathbf{x} + \mathbf{b}_1 \qquad \triangleleft \quad \text{linear layer} \tag{1.5}$$

$$\mathbf{h} = g(\mathbf{z}) \qquad \triangleleft \quad \text{activation function} \tag{1.6}$$

$$\mathbf{y} = \mathbf{W}_2\mathbf{h} + \mathbf{b}_2 \qquad \triangleleft \quad \text{linear layer} \tag{1.7}$$

In general, MLPs can be constructed with any number of layers following this pattern: linear layer, activation function, linear layer, activation function, and so on.

The activation function g could be the threshold function like in equation (1.2), but more generally it can be any pointwise nonlinearity, that is, $g(\mathbf{h}) = [\tilde{g}(h_1), \dots, \tilde{g}(h_N)]$ and $\tilde{g}$ is any nonlinear function that maps $\mathbb{R} \rightarrow \mathbb{R}$.

Beyond MLPs, this kind of sequence (linear layer, pointwise nonlinearity, linear layer, pointwise nonlinearity, and so on) is the prototpyical motif in almost all neural networks, including most we will see later in this book.

1.4 Activations Versus Parameters

When working with deep nets it's useful to distinguish *activations* and *parameters*. The activations are the values that the neurons take on, $[\mathbf{x}, \mathbf{z}_1, \mathbf{h}_1, \dots, \mathbf{z}_{L-1}, \mathbf{h}_{L-1}, \mathbf{y}]$; slightly abusing notation, we use this term for both preactivation function neurons and postactivation function neurons. The activations are the neural representations of the data being processed. Often, we will not worry about distinguishing between inputs, hidden units, and outputs to the net, and simply refer to all data and neural activations in a network, layer by layer, as a sequence $[\mathbf{x}_0, \dots, \mathbf{x}_L]$, in which case $\mathbf{x}_0$ is the raw input data.

A multilayer network is a sequence of transformations $f_1, \dots, f_L$ that produce a series of activations $\mathbf{x}_1, \dots, \mathbf{x}_L$:

$$\begin{tikzcd}
\mathbf{y} \\
\boxed{f_4(\mathbf{x}_3, \theta_4)} \arrow[u] \\
\mathbf{x}_3 \arrow[u] \\
\boxed{f_3(\mathbf{x}_2, \theta_3)} \arrow[u] \\
\mathbf{x}_2 \arrow[u] \\
\boxed{f_2(\mathbf{x}_1, \theta_2)} \arrow[u] \\
\mathbf{x}_1 \arrow[u] \\
\boxed{f_1(\mathbf{x}_0, \theta_1)} \arrow[u] \\
\mathbf{x}_0 \arrow[u]
\end{tikzcd}$$

Conversely, parameters are the weights and biases of the network. These are the variables being learned. Both activations and parameters are tensors of variables.

Often we think of a layer as a function $\mathbf{x}_{l+1} = f_{l+1}(\mathbf{x}_l)$, but we can also make the parameters explicit and think of each layer as a function:

$$\mathbf{x}_{l+1} = f_{l+1}(\mathbf{x}_l, \theta_{l+1}) \tag{1.8}$$

That is, each layer takes the activations from the previous layer, as well as parameters of the current layer as input, and produces activations of the next layer. Varying either the input activations or the input parameters will affect the output of the layer. From this perspective, anything we can do with parameters, we can do with activations instead, and vice versa, and that is the basis for a lot of applications and tricks. For example, while normally we learn the values of the parameters, we could instead hold the parameters fixed and learn the values of the activations that achieve some objective. In fact this is what is done in applications such as style transfer, adversarial attacks, and network visualization, which we will see in more detail in later chapters.

1.4.1 Fast Activations and Slow Parameters

So what's different about activations versus parameters? One way to think about it is that activations are *fast* functions of a data*point*: they are the result of a few layers of processing this datapoint. Parameters are *also* functions of the data (they are learned from data) but they are *slow* functions of data*sets*: the parameters are arrived at via an optimization procedure over a whole dataset. So, both activations and parameters are statistics of the data, that is, information extracted about about the data that organizes or summarizes it. The parameters are a kind of metasummary since they specify a functional transformation that produces activations from data, and activations themselves are a summary of the data. Figure 1.4.1 shows how this looks.

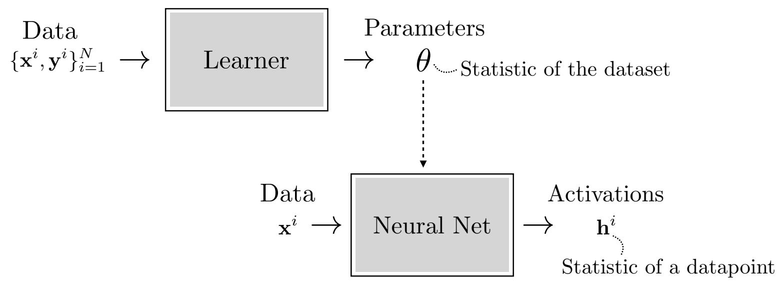

Figure 1.7: Learning is a function that maps a dataset to parameters. Inference, through a neural net, is a function that maps a datapoint to activations.

1.5 Deep Nets

Deep nets are neural nets that stack the above motif many times (figure 1.8).

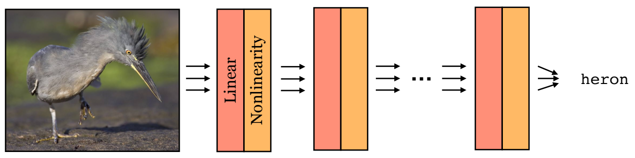

Figure 1.8: Deep nets consist of linear layers interleaved with nonlinearities.

Each layer is a function. Therefore, a deep net is a composition of many functions:

$$f(\mathbf{x}) = f_L(f_{L-1}(\ldots f_2(f_1(\mathbf{x})))) \tag{1.9}$$

The L is the number of layers in the net.

These functions are parameterized by weights $[\mathbf{W}_1, \ldots, \mathbf{W}_L]$ and biases $[\mathbf{b}_1, \ldots, \mathbf{b}_L]$. Some layers we will see later have other parameters. Collectively, we will refer to the concatenation of all the parameters in a deep net as θ.

Deep nets are powerful because they can perform nonlinear mappings. In fact, a deep net with sufficiently many neurons can fit almost any desired function arbitrarily closely, a property we will investigate further in section 1.5.2.

1.5.1 Deep Nets Can Perform Nonlinear Classification

Let's return to our binary classification problem shown previously, but now let's make the two classes not linearly separable. Our new dataset is shown in figure 1.9.

Here there is no line that can separate the zeros from the ones. Nonetheless, we will demonstrate a multilayer network that can solve this problem. The trick is to just add more layers! We can use the two layer MLP shown in figure 1.10.

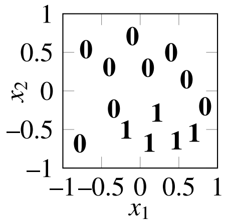

Figure 1.9: Dataset that is not linearly separable.

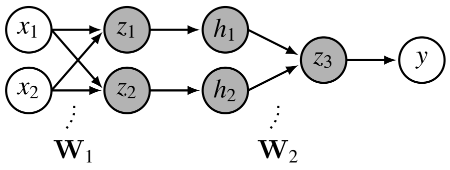

Figure 1.10: A simple MLP network.

Consider using the following settings for $\mathbf{W}_1$ and $\mathbf{W}_2$:

$$\mathbf{W}_1 = \begin{bmatrix} -1 & 1 \\ 1 & 2 \end{bmatrix}, \qquad \mathbf{W}_2 = \begin{bmatrix} 1 & -1 \end{bmatrix} \tag{1.10}$$

The full net then performs the following operation:

$$z_1 = x_1 - x_2, \quad z_2 = 2x_1 + x_2 \qquad \triangleleft \ \texttt{linear} \tag{1.11}$$

$$h_1 = \max(z_1, 0), \quad h_2 = \max(z_2, 0) \qquad \triangleleft \ \texttt{relu} \tag{1.12}$$

$$z_3 = h_1 - h_2 \qquad \triangleleft \ \texttt{linear} \tag{1.13}$$

$$y = \mathbb{1}(z_3 > 0) \qquad \triangleleft \ \texttt{threshold} \tag{1.14}$$

The $\mathbb{1}$ is the indicator function, which we define as:

$$\mathbb{1}(x) = \begin{cases} 1 & \text{if } x \text{ is true} \\ 0 & \text{otherwise} \end{cases}$$

Here we have introduced a new pointwise nonlinearity, the **Rectified linear unit** (**relu**), which is like a graded version of the `threshold` function we saw above, and performs better in practice.

Let's visualize the values that the neurons take on as a function of x_1 and x_2 in figure 1.11.

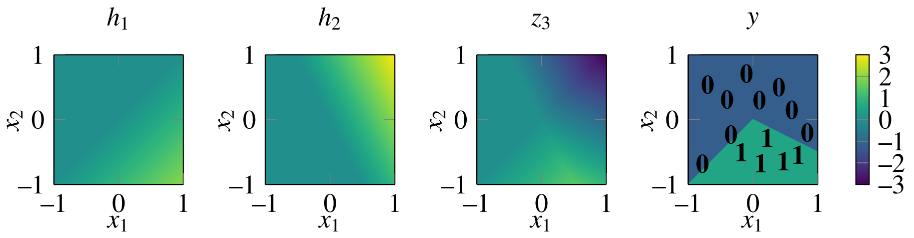

Figure 1.11: Values of hidden units and output unit for the MLP in figure 1.10.

As can be seen in the rightmost plot in figure 1.11, at the output y, this neural net successfully assigns a value of 1 to the region of the dataspace where the datapoints labeled as 1 live. This example demonstrates that is possible to solve nonlinear classification problems with a deep net. In practice, we would want to *learn* the parameter settings that achieve this classification. One way to do so would be to enumerate all possible parameter settings and pick one that successfully separates the zeros from the ones. This kind of exhaustive enumeration is a slow process, but don't worry, in later chapters we will see how to speed things up using methods from optimization (in particular, gradient descent). But it's worth remarking that enumeration is always a sufficient solution, at least when possible parameter values form a finite set.

1.5.2 Deep Nets Are Universal Approximators

Not only can deep nets perform nonlinear classification, they can in principle perform *any* continuous input-output mapping. The **universal approximation theorem** [2] states that this is true even for a network with just a single hidden layer. The caveat is that the number of neurons in the hidden layers will have to be very large in order to fit complicated functions.

Technically, this theorem only holds for continuous functions on compact subsets of $\mathbb{R}^N$ – for example a neural net cannot fit noncomputable functions. We will not be rigorous in this section. We direct the reader to [12] for a formal treatment of universal approximation.

To get an intuition for why this is true, we will consider the case of approximating an arbitrary function from $\mathbb{R} \to \mathbb{R}$ with a `relu`-network. First observe that any function can be approximated arbitrarily well by a sum of indicator functions, that is, bumps placed at different positions:

$$f(x) \approx \sum_i w_i \mathbb{1}(\alpha_i < x < \beta_i) \tag{1.15}$$

As an example, in figure 1.12 we show a curve (blue line) approximated in this way. As the width, $\beta - \alpha$, of the bumps (black lines) goes to zero, the error in the fit goes to zero.

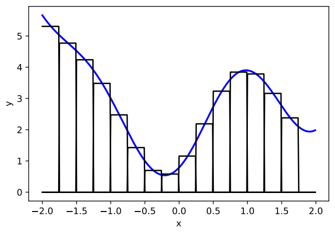

Figure 1.12: Any function from $\mathbb{R} \to \mathbb{R}$ can be approximated arbitrarily well by a sum of elementary bumps.

While we only consider scalar functions $\mathbb{R} \to \mathbb{R}$ here, a similar construction can be used to approximate general functions of the form $\mathbb{R}^n \to \mathbb{R}^m$.

Next we will show that a `relu`-net can represent equation (1.15). The weighted sum $\sum_i w_i \ldots$ is the easy part: that's just a `linear` layer. So we just have to show that we can also write $\mathbb{1}(\alpha < x < \beta)$ using `linear` and `relu` layers. It turns out the construction is rather simple:

$$\mathbb{1}(\alpha < x < \beta) \approx \texttt{relu}\left(\frac{x-(\alpha-\gamma)}{\gamma}\right) - \texttt{relu}\left(\frac{x-\alpha}{\gamma}\right) - \texttt{relu}\left(\frac{x-(\beta-\gamma)}{\gamma}\right) + \texttt{relu}\left(\frac{x-\beta)}{\gamma}\right) \tag{1.16}$$

Here we show how a neural net can represent a function as a sum of basis functions. This idea is also foundational in signal processing, where signals are often represented as a sum of sine waves (chapter **??**), boxes (figure **??**), or trapezoids (figure **??**).

As $\gamma \to 0$, this approximation becomes exact. The input to each of the four `relu`s in equation (1.16) is an affine function of the input x, hence these four values can be represented by a `linear` layer with four outputs. Then we apply a `relu` layer to these four values, and finally we apply a `linear` layer to compute the sum over these `relu`s (a weighted sum with weights [1,-1,-1,1]). Therefore equation (1.16) can be implemented as a `linear-relu-linear` network. In figure 1.13 below, we show an example of constructing a bump in this way.

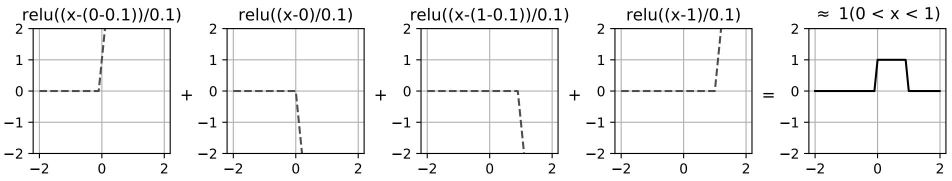

Figure 1.13: A bump can be represented as a weighted sum of shifted and scaled `relu`s.

Putting everything together, we have a `linear-relu-linear` for each bump, followed by a `linear` layer for summing up all the bumps. The two linear layers in sequence can be collapsed to a single `linear` layer, and hence the full function can therefore be approximated, to arbitrary precision, by a `linear-relu-linear` net.

Most literature refers to such a net as having a single hidden layer, using the convention that we don't count pre- and postactivation neurons as separate layers.

Notice that in this approximation, we need four `relu` neurons for each bump we are modeling. Therefore if we want to approximate a very bumpy function, say with N bumps, we will need $4N$ `relu` neurons. In general, to achieve arbitrarily good approximation to a curve we may need an unbounded number of neurons in our network.

1.5.3 Depth versus Width

If different layers have different numbers of neurons, then we may specify the width per layer. Here we will assume all layers have the same width and simply speak of the width of the network.

Above we saw that if you have a hidden layer with N neurons, you can fit a function with $\mathcal{O}(N)$ bumps. The number of neurons on a single hidden layer is called its **width**. So, as we increase the width of a network, we can fit ever more complicated functions. What if we instead increase the **depth** of a network, that is, its number of layers? It turns out that this can also be an effective way to increase the capacity of the net, but its effect is a bit different than increasing width.

Interestingly, it is sometimes the case that *deep* nets require far fewer parameters to fit data than *wide* nets. Evidence for this statement comes mostly from empiricism, where researchers have found that deeper nets just work better in practice on many popular problems. However, there is also the beginning of a mathematical theory of when and why this can happen. The basic idea of this theory is to establish that there are certain classes of function that can be represented with a polynomial number of neurons in a depth d network but require an exponential number of neurons in a depth d' network, for certain $d' < d$. Arguments along these lines are called **depth separations**, and the interested reader can refer to [11] to learn more about this ongoing line of research.

1.6 Deep Learning: Learning with Neural Nets

Using the formalism we defined in chapter **??**, learning consists of using an *optimizer* to find a function in a *hypothesis space*, that maximizes an *objective*. From this perspective, neural nets are simply a special kind of hypothesis space (and a particular parameterization of that hypothesis space). **Deep learning** refers to learning algorithms that use this parameterized hypothesis space.

Deep learning also typically involves using gradient-based optimization to search the hypothesis space for the best fit to the data. We will investigate this approach in detail in chapter **??**, where we will learn about the **backpropagation** algorithm for gradient-based learning with neural nets. However, it is certainly possible to optimize neural nets with other methods, including zeroth-order optimizers like evolution strategies (section **??**; [9]).

One intriguing alternative to backpropagation is called **Hebbian learning** [3]. Backpropagation is a *top-down* learning algorithm, where errors incurred at the output (top) of the net are propagated backward to inform earlier layers how to update their weights and biases to minimize the loss; a form of learning from *feedback*. Hebbian learning, in contrast, is a *bottom-up* approach, where neurons wire up just based on the *feedforward* pattern of activity in the net. The canonical learning rule in Hebbian methods is **Hebb's rule**: "fire together, wire together." That is, we increase the weight of the connection between two neurons whenever the two neurons are active at the same time. Although this learning rule is not explicitly minimizing a loss function, it has been shown to lead to effective neural representations. For example, Hebb-like rules can learn **infomax** representations, which capture, in the neural activations, as much information as possible about the input signal [6]. Similar rules lead to networks that act like memory banks [4]. Hebbian learning is also of interest because it is considered to be more biologically plausible than backpropagation. This is because Hebb's rule can be computed *locally*—each neuron strengthens and weakens its weights based just on the activity of adjacent neurons—whereas backpropagation requires global coordination throughout the neural network. It is currently unknown how this global coordination can be achieved in biological brains.

1.6.1 Data Structures for Deep Learning: Tensors and Batches

The main data structure that we will encounter in deep learning is the **tensor**, which is just a multidimensional array. This may seem simple, but it's important to get comfortable with the conventions of tensor processing.

In general, everything in deep learning is represented as tensors—the input is one tensor, the activations are tensors, the weights are tensors, the outputs are tensors. If you have data that is not natively represented as a tensor, the first step, before feeding it to a deep net, is usually to convert it into a tensor format. Most often we use tensor of real numbers, that is, the elements of the tensor are in $\mathbb{R}$.

Suppose we have a dataset $\{\mathbf{x}^{(i)}, \mathbf{y}^{(i)}\}_{i=1}^{N}$ of images $\mathbf{x}$ and labels $\mathbf{y}$. The tensor way of thinking about such a dataset is as two tensors, $\mathbf{X} \in \mathbb{R}^{N \times C_0 \times H \times W}$ and $\mathbf{Y} \in \mathbb{R}^{N \times K}$. The first dimension of the tensor is the number of elements in our dataset. The remaining dimensions are the dimensionality of the images (C_0 color channels by height H by width W) and labels (K-way classification).

The activations in the network are also tensors. For the MLP networks we have seen so far, the activation tensors have shape $N \times C_\ell$, where C_ℓ is the number of neurons on layer ℓ, sometimes also called **channels** in analogy to the color channels of the input image. In later chapters we will encounter other architectures where the activation layers have additional dimensions, for example, in convolutional networks we will see activation layers that are of shape $N \times C_\ell \times H_\ell \times W_\ell$.

One other important concept is **batch processing**. Normally, we don't process one image at a time through a neural net. Instead we run a *batch* of images all at once, and they are processed in parallel. A batch sampled from the training data can be denoted as $\{\mathbf{x}^{(i)}_{\texttt{batch}}, \mathbf{y}^{(i)}_{\texttt{batch}}\}_{i=1}^{N_{\texttt{batch}}}$, and the batch represented as a tensor has shape $\mathbf{X} \in \mathbb{R}^{N_{\texttt{batch}} \times C_0 \times H \times W}$ and $\mathbf{Y} \in \mathbb{R}^{N_{\texttt{batch}} \times K}$.

The weights and biases of the net are also usually represented as tensors. The weights and biases of a linear layer will be tensors of shape $\mathbf{W}_\ell \in \mathbb{R}^{C_{\ell+1} \times C_\ell}$ and $\mathbf{b}_\ell \in \mathbb{R}^{C_{\ell+1}}$.

As an example, in figure 1.14 below, we visualize all the tensors associated with a batch of three datapoints being processed by the MLP from figure 1.10. For this network, the input is not a set of images but instead a set of vectors $\mathbf{X} \in \mathbb{R}^{N_{\texttt{batch}} \times C_0}$. The output is one value for each input vector, so we have $\mathbf{Y} \in \mathbb{R}^{N_{\texttt{batch}} \times 1}$.

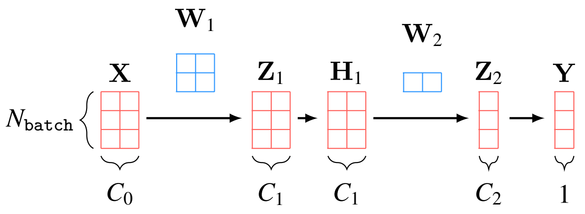

Figure 1.14: The tensors that represent one pass through the MLP in figure 1.10.

where the capital letters are the batches of datapoints and activations corresponding to the lowercase names of datapoints and hidden units in figure 1.10.

This example shows the basic concept of working with tensors and batches for one-dimensional data, but, in vision, most of the time we will be working with higher-dimensional tensors. For image data we typically use four-dimensional tensors: batch $\times$ channels $\times$ height $\times$ width; for videos we may use five-dimensional tensors: batch $\times$ channels $\times$ height $\times$ width $\times$ time. Three-dimensional (3D) scans have an additional *depth* spatial dimension; videos of 3D data could therefore be represented by six-dimensional tensors. As you can see, thinking in terms of two-dimensional matrices is not quite sufficient. Instead, you should be imagining data processing as operating on N-dimensional tensors, sliced and diced in different ways. As a step in this direction, you may find it useful to visualize tensors in 3D, as shown in figure 1.15.

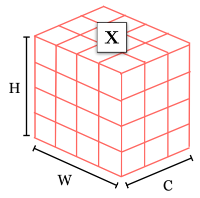

Figure 1.15: A 3D tensor that could represent an $C \times H \times W$ color image.

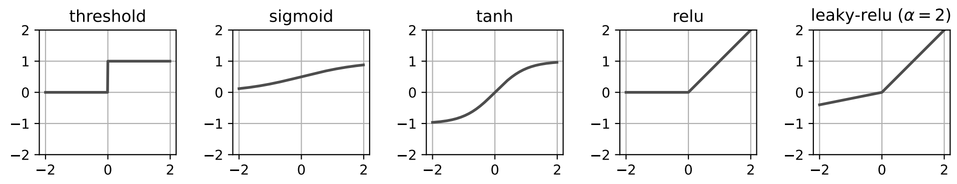

Figure 1.16: Common pointwise nonlinearities.

This is closer to the actual ND tensors vision systems work with, and many concepts can be adequately captured just by thinking in 3D. We will see some examples in later chapters.

1.7 Catalog of Layers

Below, we use the color blue to denote **parameters** and the color red to denote **data/activations** (inputs and outputs to each layer).

1.7.1 Linear layers

Linear layers are the workhorses of deep nets. Almost all parameters of the network are contained in these layers; we call these parameters the weights and biases. We have already introduced linear layers previously. They look like this:

$$\mathbf{x}_{\texttt{out}} = \mathbf{W}\mathbf{x}_{\texttt{in}} + \mathbf{b} \qquad \triangleleft \quad \texttt{linear} \tag{1.17}$$

1.7.2 Activation layers

If a net only contained linear layers then it could only compute linear functions. This is because the composition of N linear functions is a linear function. **Activation layers** add nonlinearity. Activation layers are typically pointwise functions, applying a scalar to scalar mapping on each dimension of the input vector. Typically parameters of these layers, if any, are not learned (but they can be). Some common activation layers are defined below and are plotted in figure 1.16:

$$x_{\texttt{out}}[i] = \begin{cases} 1, & \text{if} \quad x_{\texttt{in}}[i] > 0 \\ 0, & \text{otherwise} \end{cases} \qquad \triangleleft \quad \texttt{threshold} \tag{1.18}$$

$$x_{\texttt{out}}[i] = \frac{1}{1 + e^{-x_{\texttt{in}}[i]}} \qquad \triangleleft \quad \texttt{sigmoid} \tag{1.19}$$

$$x_{\texttt{out}}[i] = 2 * \texttt{sigmoid}(2 * x_{\texttt{in}}[i]) - 1 \qquad \triangleleft \quad \texttt{tanh} \tag{1.20}$$

$$x_{\texttt{out}}[i] = \max(x_{\texttt{in}}[i], 0) \qquad \triangleleft \quad \texttt{relu} \tag{1.21}$$

$$x_{\texttt{out}}[i] = \begin{cases} \max(x_{\texttt{in}}[i], 0), & \text{if} \quad x_{\texttt{in}}[i] \geq 0 \\ a \min(x_{\texttt{in}}[i], 0), & \text{otherwise} \end{cases} \qquad \triangleleft \quad \texttt{leaky-relu} \tag{1.22}$$

1.7.3 Normalization layers

Normalization layers add another kind of nonlinearity. Instead of being a pointwise nonlinearity, like in activation layers, they are nonlinearities that perturbs each neuron based on the collective behavior of a set of neurons. Let's start with the example of **batch normalization** (**batchnorm**) [5].

Recall from statistics that the standard score of a draw of a random variable is how many standard deviations it differs from the mean: $z = \frac{x-\mu}{\sigma}$.

Batchnorm **standardizes** each neural activation with respect to its mean and variance over a batch of datapoints. Mathematically,

$$x_{\texttt{out}}[i] = \gamma \frac{x_{\texttt{in}}[i] - \mathbb{E}[x_{\texttt{in}}[i]]}{\sqrt{\texttt{Var}[x_{\texttt{in}}[i]]}} + \beta \qquad \triangleleft \quad \texttt{batchnorm} \tag{1.23}$$

where γ and β are learned parameters of this layer that maintain expressivity so that the layer can output values with non-zero mean and non-unit variance. Most commonly batchnorm is applied using training batch statistics to compute the mean and variance, which change batch to batch. At test time, aggregate statistics from the training data are used. However, using test batch statistics can be useful for achieving invariance to changes in the statistics from training data to test data [13].

There are numerous other normalization layers that have been defined over the years. Two more that we will highlight are L_2 **normalization** and **layer normalization** (**layernorm**) [1]. L_2 normalization projects the inputs onto the unit hypersphere, which useful for bounding the activations to unit vectors:

$$x_{\texttt{out}}[i] = \frac{x_{\texttt{in}}[i]}{\|\mathbf{x}_{\texttt{in}}\|_2} \qquad \triangleleft \quad \texttt{L2-norm} \tag{1.24}$$

Layernorm is similar except that it standardizes the vector of input activations:

Notice that layernorm, like L_2-normalization, maps the activation vector to the surface of a hypersphere, but it also centers the activations to have zero mean, and then scales and shifts the activations via γ and β. As an exercise, see if you can write layernorm using L_2-normalization as one step.

$$\mu = \frac{1}{N}\sum_{i=1}^{N} x_{\texttt{in}}[i] \tag{1.25}$$

$$\sigma^2 = \frac{1}{N}\sum_{i=1}^{N} (x_{\texttt{in}}[i] - \mu)^2 \tag{1.26}$$

$$x_{\texttt{out}}[i] = \gamma \frac{x_{\texttt{in}}[i] - \mu}{\sigma} + \beta \qquad \triangleleft \quad \texttt{layernorm} \tag{1.27}$$

Notice that layernorm also looks quite similar to batchnorm. Both standardize activations but do so with respect to different statistics. Layernorm computes a mean and variance over elements of a datapoint $\mathbf{x}_{\texttt{in}}$, and will do so separately for each such datapoint in a batch. Batchnorm computes the mean and variance per channel over datapoints in a batch. If we have a batch stored in the tensor $\mathbf{X} \in \mathbb{R}^{N_{\texttt{batch}} \times C}$, then what layernorm does looks just like a "transpose" of what batchnorm does. Batchnorm standardizes each element of the tensor by the mean and variance of its column. Layernorm standardizes each element by the mean and variance of its row:

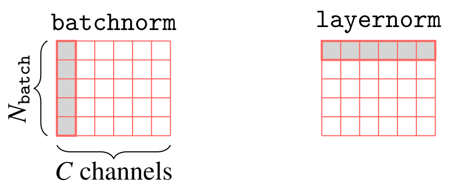

Figure 1.17: Batchnorm vs layernorm. Gray indicates the region over which mean and variance are computed.

See Figure 2 of [14] for further normalization schemes visualized this way.

One issue with batchnorm is that it requires processing a batch of datapoints all at once, and introduces a dependency between each datapoint in the batch. This violates the principle that datapoints should be processed independently and identically (iid), and this can lead to bugs if your method relies on the iid assumption. Layernorm does not have this problem and does indeed process each datapoint in an iid fashion.

1.7.4 Output layers

The last piece we need is an **output layer** that maps a neural representation—a high-dimensional array of floating point numbers—to a desired output representation. In classification problems, the desired output is a class label, and the most common output operation is the softmax function, which we have already encountered in previous chapters. In image synthesis problems, the desired output is typically a 3D array with dimensions $N \times M \times 3$, and values in the range $[0, 255]$. A sigmoid multiplied by 255 is a typical output transformation for this setting. The equations for these two layers are:

$$x_{\texttt{out}}[i] = \frac{e^{-\tau x_{\texttt{in}}[i]}}{\sum_{k=1}^{K} e^{-\tau x_{\texttt{in}}[k]}} \qquad \triangleleft \quad \texttt{softmax} \tag{1.28}$$

$$x_{\texttt{out}}[i] = 255 * \texttt{sigmoid}(x_{\texttt{in}}[i]) \qquad \triangleleft \quad \text{common layer for image output problems} \tag{1.29}$$

In the softmax definition we have added a **temperature** parameter τ, which is commonly used to scale how peaky, or confident, the predictions are.

The output layer is the input to the loss function, thus completing our specification of the deep learning problem. However, to use the outputs in practice requires translating them into actual pictures, or actions, or decisions. For a classification problem, this might mean taking the argmax of the softmax distribution, so that we can report a single class. For image prediction problems, it might mean rounding each output to an integral value since common image formats represent red-green-blue values as integers.

There are of course many other output transformations you can try. Often, they will be very problem specific since they depend on the structure of the output space you are targeting.

1.8 Why Are Neural Networks a Good Architecture?

As you will soon learn, almost all modern computer vision algorithms involve neural nets in one way or another. So you may be wondering: why are deep nets such a good architecture? We will highlight here five reasons:

1. They are high capacity (big enough nets are universal approximators).
2. They are differentiable (the parameters can be optimized via gradient descent).
3. They have good inductive biases (neural architectures reflect real structure in the world).
4. They run efficiently on parallel hardware.

5. They build abstractions.

Let's look at reasons 1-3 in light of the discussion of searching for truth from chapter **??** (see figure **??**). Reason 1 relates to the size of the hypothesis space. The hypothesis space can be made very big if we use a large neural network with many parameters. So we can usually be sure that our true solution (or a close approximation to it) does indeed lie in the space spanned by the neural net architecture. Reason 2 says that searching for the solution within this space is relatively easy, since we can use gradients to direct us toward ever better fits to the data. Reason 3 is one we will only come to appreciate later in the book as we investigate more advanced neural net architectures. It turns out that these architectures impose constraints and regularizers that bias our search toward solutions that capture true structure about the visual world, and this leads to learned solutions that generalize.

Reason 4 is equally important to the first three: it says we can do all this *efficiently* because most computations can be parallelized on modern hardware; in particular both matrix multiplies (`linear` layers) and pointwise operations (e.g., `relu` layers) are easy to parallelize on graphical processing units. Further, most operations are applied to image batches, where each item in the batch can be sent to a different parallel compute node.

Reason 5 is the perhaps the most subtle. It is related to the layered structure of neural nets. Layer by layer, neural nets build up increasingly abstracted representations of the input data, and these abstractions tend to be increasingly useful. This argument is not easy to appreciate at first glance, but it will be a major theme of the subsequent chapters in this book, especially those on representation learning. For now, just keep in mind that the *internal representations* that are built up layer by layer in deep nets are useful and important beyond just the net's overall input-output behavior.

1.9 Concluding Remarks

Neural nets are a very simple and useful parameterized hypothesis space. They are universal approximators that can be trained via gradient descent and run on parallel hardware. *Deep* nets are especially effective in computer vision; as we will soon see, deep architectures can be constructed that specifically reflect structure in the visual world, and make visual processing highly efficient and performant. Artificial neural nets also have connections to the real neural nets in our brains. This connection runs deeper than merely sharing a name: the deep net architectures we will see later in this book (e.g., convolutional networks, transformers) are our best current models of computation in animal brains, in the sense that they explain brain data better than any competing models [10]. This is a class of models truly worth paying attention to.

References

[1] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. *Layer Normalization*. 2016. arXiv: 1607.06450 [stat.ML].

[2] G. Cybenko. “Approximation by Superpositions of a Sigmoidal Function”. In: *Mathematics of Control, Signals and Systems* 2.4 (Dec. 1989), pp. 303–314. DOI: 10.1007/BF02551274.

[3] Donald Olding Hebb. *The Organization of Behavior: A Neuropsychological Theory*. Psychology Press, 2005.

[4] John J Hopfield. “Neural Networks and Physical Systems with Emergent Collective Computational Abilities.” In: *Proceedings of the National Academy of Sciences* 79.8 (1982), pp. 2554–2558.

[5] Sergey Ioffe and Christian Szegedy. “Batch normalization: Accelerating deep network training by reducing internal covariate shift”. In: *International Conference on Machine Learning* (2015), pp. 448–456.

[6] Ralph Linsker. “Self-Organization in a Perceptual Network”. In: *Computer* 21.3 (1988), pp. 105–117.

[7] Robert A Rescorla. “A Theory of Pavlovian Conditioning: Variations in the Effectiveness of Reinforcement and Non-Reinforcement”. In: *Classical Conditioning, Current Research and Theory* 2 (1972), pp. 64–69.

[8] Frank Rosenblatt. “The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain.” In: *Psychological review* 65.6 (1958), p. 386.

[9] Tim Salimans et al. “Evolution Strategies as a Scalable Alternative to Reinforcement Learning”. In: (2017). https://arxiv.org/abs/1703.03864.

[10] Martin Schrimpf et al. “Integrative Benchmarking to Advance Neurally Mechanistic Models of Human Intelligence”. In: *Neuron* (2020).

[11] Matus Telgarsky. “Benefits of Depth in Neural Networks”. In: *Conference on Learning Theory*. PMLR. 2016, pp. 1517–1539.

[12] Matus Telgarsky. *Deep learning theory*. Lecture notes. 2021.

[13] Dequan Wang et al. “Fully Test-Time Adaptation by Entropy Minimization”. In: (2020). https://arxiv.org/abs/2006.10726.

[14] Yuxin Wu and Kaiming He. “Group normalization”. In: *Proceedings of the European conference on computer vision (ECCV)*. 2018, pp. 3–19.