Contents

Part II: Training

3 Learning a neural network - Part I

3.1 The problem of learning

3.1.1 The network as a parametric function to be learned

3.1.2 Learning the parameters of a network
List of Figures

3.1 Some examples of “desired functions” that we may require a neural network to model ........................................ 4
3.2 A perceptron as a simple threshold unit - Part 2 .................. 6
3.3 Including bias with the input in a perceptron ....................... 6
3.4 Structure of a feed-forward network ................................ 7
3.5 The network as a parametric function ............................... 8
3.6 An arbitrary function and an MLP ................................. 8
3.7 Handcrafting a network: an example ................................ 9
3.8 Constructing a perceptron for one edge of a diamond shaped decision boundary: 1 ........................................... 10
3.9 Constructing a perceptron for one edge of a diamond shaped decision boundary: 2 ........................................... 10
3.10 Constructing a perceptron for one edge of a diamond shaped decision boundary: 3 .......................................... 11
3.11 Constructing a perceptron for one edge of a diamond shaped decision boundary: 4 .......................................... 12
3.12 Constructing a perceptron for one edge of a diamond shaped decision boundary: 5 .......................................... 13
3.13 A desired function and an MLP that must be learned to model it 13
Part II: Training
Chapter 3

Learning a neural network - Part I

In Chapter 3, we learned that MLPs can emulate any function. However, we must still find out how we can use an MLP to emulate a specific type of desired function, e.g. a function that takes an image as input and outputs the labels of all objects in it, or a function that takes speech input and outputs the words in it etc. This – the training of an MLP to emulate any type of desired function – will be the topic of this chapter, and the next.

3.1 The problem of learning

In the previous chapters, we have seen that neural networks have several very desirable properties (and also many limitations): they are universal approximators, they can model any Boolean function, they can model any discrete-valued (classification) function, and they can even model any real-valued function. The constraint with all of these is that they must have an architecture that must be sufficient to represent the function we wish to model, with (at least) the minimal number of neurons required in each layer. If they have fewer than the required number of neurons, they cannot model the function in question.

Let us now understand what is needed for an MLP to model any specific type of “desired function.” Some examples of these are shown in Fig. 3.1.

In Chapter 2, we saw that neural networks are universal approximators: given the
In each case, the function takes in an input, like the voice signal, or the image, or the current game state, and computes an output, like the transcription for the voice recording, the caption of the image, or the next game state.

Figure 3.1: Some examples of “desired functions” that we may require a neural network to model

right architecture, they can model very complicated functions, including functions that perform tasks like speech recognition, image captioning or playing games.

This is however not enough to practically implement an MLP to model any desired function. We must also now consider the mechanics of representing each of the boxes shown in Fig. 3.1, for the network to operate in a desired manner, given the specific types of inputs and outputs that are expected.

For this, we begin by noting that the network is essentially a function that operates on numerical values as inputs, and outputs numerical values. Therefore in all the tasks shown in Fig. 3.1 and in all tasks in general, the input must be represented as numbers. Several questions arise in this context:

a) How do we represent real-world inputs (voices, images, videos etc.) as numbers?
b) What might be “relevant” numerical values for the outputs of such tasks?
c) How can we compose the actual function (i.e. the actual network) that can perform the desired task?

Let us first focus on the last question in this list: how do we compose a network that computes a given function? To understand how, let us return to the basic unit
of our network: the perceptron, yet again shown for reference in Fig. 3.2.

As we have discussed earlier, the perceptron operates on real-valued inputs and first computes an affine combination of the inputs. We recall that linear and affine units in a perceptron operate as follows:

\[
\begin{align*}
\text{Linear} & : \quad f(ax_1 + bx_2) = af(x_1) + bf(x_2) \\
\text{Affine} & : \quad \exists C \text{ such that } g = f(x)C \\
\end{align*}
\]

The affine combination of the inputs computed by the perceptron is a weighted sum of all of the inputs plus a bias. This affine combination, which we will generally represent as \( Z \) in this book, is then put through an activation function. The activation function \( \sigma(Z) \) could be a threshold, or any other function. The output of the perceptron is the output of the activation function. The parameters of the perceptron, which determine how it behaves, are its weights and bias.

Instead of having an explicit bias, we can also think of the neuron (or perceptron) as having one more input component whose value is permanently fixed to 1. The bias term simply becomes the weight associated with the extra fixed input, as shown in Fig. 3.3. Going forward, where we do not explicitly mention the bias, we will assume that every perceptron has an additional input that is always fixed at 1, that implicitly accounts for the bias.

Let us first make some simplifying assumptions about the structure of the perceptron network. We assume a feed-forward network, as shown in Fig. 3.4. In such a network, computations are performed in a strictly directed manner – they are done unidirectionally from input to output. In computing the network output for any input, each neuron is computed exactly once. This means that the output of a neuron does not directly or indirectly feed back to its input. We will consider loopy networks, where the output of a neuron does feed back into its input, in later chapters of this book.
Perceptron: In a general setting, inputs are real valued. A bias $b$ represents a threshold to trigger the perceptron. $\sigma(Z)$ is an activation function, which can be one of many types. $Z$ is the affine combination of inputs. The activation function operates on the affine combination $Z$. Activation functions are not necessarily threshold functions.

Figure 3.2: A perceptron as a simple threshold unit - Part 2

The bias can also be viewed as the weight of another input component that is always set to 1.

Figure 3.3: Including bias with the input in a perceptron
To compute any function, a major part of designing a feed-forward (or even a loopy) network is the architecture of the net. The architecture comprises the number of layers that it must have, the number of neurons in each layer and their exact interconnections. For now we assume that the architecture of the network is given to us, and the network is capable of representing the desired function.

A feed-forward network has no loops. In such networks, neuron outputs do not feed back to their inputs directly or indirectly. The computation is unidirectional, and in this example it is done from left to right.

Figure 3.4: Structure of a feed-forward network

### 3.1.1 The network as a parametric function to be learned

A perceptron (or neural) network is a function. It takes an input and produces an output. Fig. 3.5 shows a typical neural network. The parameters of the function are the weights and biases of the neurons in the network (represented by the blue arrows in the figure).

We represent this function as \( f(x; W) \), where \( x \) is the input to the network, and \( W \) are its parameters – its weights and biases. For the network to compute a specific (desired) function, we must set the parameters \( W \) appropriately. The process of learning a network involves estimating the set of parameters – the weights and biases – such that the network computes the desired function.

From the chapters in Part I of this book, we know that given any function, it is possible to construct an MLP that computes it. Fig. 3.6 shows an arbitrary function of 3 variables as an example. Let us understand how we can do this.

One approach is to try to handcraft a network (i.e., manually construct it). We can
The network is a function $f(\cdot)$ with parameters $W$ which must be set to the appropriate values to get the desired behavior from the net.

Figure 3.5: The network as a parametric function

An MLP can be constructed to represent any function. The panel on the right shows an arbitrary function in the space of three variables.

Figure 3.6: An arbitrary function and an MLP
understand this approach with the help of the example in Fig. 3.7. To construct an MLP to compute the decision boundary shown in this figure over 2-dimensional inputs, a network must output 1 inside the diamond shape, and 0 outside. For this, let us use a network where the individual perceptrons have threshold activations. 

![Diagram of a diamond-shaped decision boundary]

A network with a diamond shaped decision boundary can be handcrafted to output 1 inside the diamond, and 0 outside it. Our goal is to build an MLP to classify the input using this diamond-shaped decision boundary.

Figure 3.7: Handcrafting a network: an example

We first construct a perceptron for one boundary, e.g. the boundary pointed to by the blue arrow in Fig. 3.8. This boundary represents a line, with slope 1 that goes through (−1, 0) and (0, 1). The equation for such a line is either \( x_1 - x_2 + 1 = 0 \) or its negation \( -x_1 + x_2 - 1 = 0 \). The weights and constant of the equation constitute the weights and bias of the neuron. We also want (0, 0), which is in the yellow region, to fall on the “positive” side of the boundary. This means that inserting (0, 0) into the left hand side of the appropriate equation should give us a positive value. That leaves us with \( x_1 - x_2 + 1 \) as the equation for the perceptron, giving us the perceptron shown in Fig. 3.8.

We can similarly reason that the boundary shown in Fig. 3.9 is captured by the perceptron shown in the figure, which has weights \(-1, -1\) and bias 1.

The third boundary, shown in Fig. 3.10 is captured by a perceptron with weights \(-1, 1\) and bias 1.

The fourth boundary, shown in Fig. 3.11 is captured by a perceptron with weights
At both (0,1) and (-1,0) the output must become 0 along the straight line that represents the boundary to the left.

Figure 3.8: Constructing a perceptron for one edge of a diamond shaped decision boundary: 1

The boundary corresponds to weights \(-1, -1\) and bias 1. The corresponding perceptron is shown on the left.

Figure 3.9: Constructing a perceptron for one edge of a diamond shaped decision boundary: 2
The boundary in this case corresponds to weights $-1, 1$ and bias 1. The corresponding perceptron is shown on the left.

Figure 3.10: Constructing a perceptron for one edge of a diamond shaped decision boundary: 3

1, 1 and bias 1.

Finally we add an AND perceptron as shown in Fig. 3.12, which compares the sum of the four outputs to a threshold of 4, which is the same as having a bias of -4. Fig. 3.12 shows the network we built so far. This perceptron now models the desired function (or decision boundaries), and gives us the desired outputs.

Such a process wherein a network is hand-crafted can only be done for simple cases. It becomes infeasible for more complex functions. We must find ways to do this computationally, in an automated fashion.

### 3.1.2 Learning the parameters of a network

To understand how we can do this in an automated fashion, let us use the example in Fig. 3.13: given a function $g(X)$ that we want to model, we must derive the parameters (weights and biases) of the network shown to model it accurately.
The boundary in this case corresponds to weights 1, 1 and bias 1. The corresponding perceptron is shown on the left.

Figure 3.11: Constructing a perceptron for one edge of a diamond shaped decision boundary: 4

For now, we continue to assume that the architecture of the network – the complete arrangement of neurons and their connections – is given to us, and this architecture is sufficient for modeling the desired function.

We illustrate the problem through an even simpler example – a 1-dimensional function. Fig. 3.14 shows a desired function \( d(X) \) in one dimension. The \( X \) axis represents the input \( X \), and the curve shown represents a function \( d(X) \) that we want a network to compute.

For any given setting of the weights \( W \), the network will actually model some function \( f(X; W) \), as shown in Fig. 3.14).

The area between the output of the function, and the one we want to model, represents the error between the two. To learn the function, then, we must learn the weights \( W \) to minimize the area (shown highlighted in Fig. 3.16) that represents the error.

To quantify this area, we will first define a divergence function \( \text{div()} \), which
The decision boundary is fully represented by the four perceptrons shown. A final perceptron is needed to AND these. **Bottom panel:**
The final network that models the desired decision boundary.

Figure 3.12: Constructing a perceptron for one edge of a diamond shaped decision boundary: 5

**Right:** A function $g(X)$ that we want to model. **Left:** An MLP that is sufficient to model it (schematically shown). We must derive the parameters of this network computationally, so that it can model $g(X)$ accurately.

Figure 3.13: A desired function and an MLP that must be learned to model it
Right: A one-dimensional function $d(X)$ that we want to model.

Figure 3.14: A desired function in one dimension

The function models a curve for any setting of $W$

Figure 3.16: The divergence as an area
quantifies the difference between the output of the network $f(X; W)$ and the actual desired output at any $X$. This, to reiterate, is the difference that is shown as an area in Fig. 3.16. The divergence function has the property that it goes to 0 when $f()$ is exactly equal to $d()$, and is positive when the two are not the same.

When $f(X; W)$ has the capacity to exactly represent $d(x)$

$$\hat{W} = \arg\min_w \int_X \text{div}(f(X; W), d(X)) dX$$

(3.4)

where $\text{div}()$ is a divergence function that goes to zero when $f(X; W) = d(X)$

The entire area shown highlighted in the equation above is the integral of the divergence function, computed over all valid values of the input. This quantifies the error between the actual output of the network, and the function we want the net to compute, and our goal is to compute the $W$ to minimize this value. This is exactly what we would do in the case of the function $g(X)$ in Fig. 3.13 (or indeed any other function that we want to model) as well. The divergence function in that case would be $\text{div}(f(X; W), g(X))$.

The problem here is that in order to compute the integral in Eq. 3.4, the function $d(X)$ (or $g(X)$ as the case may be) must be known for every input $X$, i.e. it must be fully specified everywhere in $X$. In practice though, when we attempt to build a network to compute some functions, such as one for classifying patterns, we will generally not have a full specification of the function to begin with.

This problem can be resolved through a reasonable approximation: we can sample the function, as shown (schematically) for both $d(X)$ and $g(X)$ in Fig. 3.17. Doing this is simple. For a number of inputs, we simply compute the target output $d(X)$ (or $g(X)$) to obtain a set of input-output pairs $(X, d(X))$ (or $(X, g(X))$). In other words, for a number of samples of input $X_i$, we obtain the pairs $(X_i, d_i)$, where $d_i$ is the value of the target function ($d(X)$ or $g(X)$ in our illustrations) at $X_i$. In good sampling, the samples of $X$ will be drawn from the true probability distribution of the input, $P(X)$.

These input-output pairs are then used to learn the network. For real-world ap-
It is easy to collect such input-output pairs: we simply collect some task-specific real-world input-output pairs of data, such as images and their class labels, or speech recordings and their transcriptions.

Computationally, to learn the function from data samples, we must estimate $W$ such that the function $f()$ that the network computes exactly (or near exactly) takes the value $d_i$ at each $X_i$, i.e. it goes through the sampled points (the blue dots shown on the function traces in Fig. fig:ch3slide24). In doing so, we expect that the resulting function would also be correct where we have not sampled it – i.e., for regions of the input space where we don’t have training samples. Thus, in general, learning the function in this manner essentially involves fitting the function to the sampled points. It is important to keep in mind that we make two
assumptions here: a) that the network architecture is sufficient for such a fit and b) that for any specific training sample $X_i$, there is a unique target value $d_i$.

Recap 1.1

1. “Learning a neural network is the same as determining the parameters of the network (weights and biases) required for it to model a desired function

2. For this, the network must have sufficient capacity to model the function

3. Ideally, we would like to optimize the network to represent the desired function everywhere

4. However this requires knowledge of the function everywhere

5. Instead, we draw “input-output” sample pairs as training instances from the function, and estimate network parameters to “fit” the input-output relation at these instances

6. We hope that this process fits the function elsewhere (regions that we did not sample) as well

More coming up....