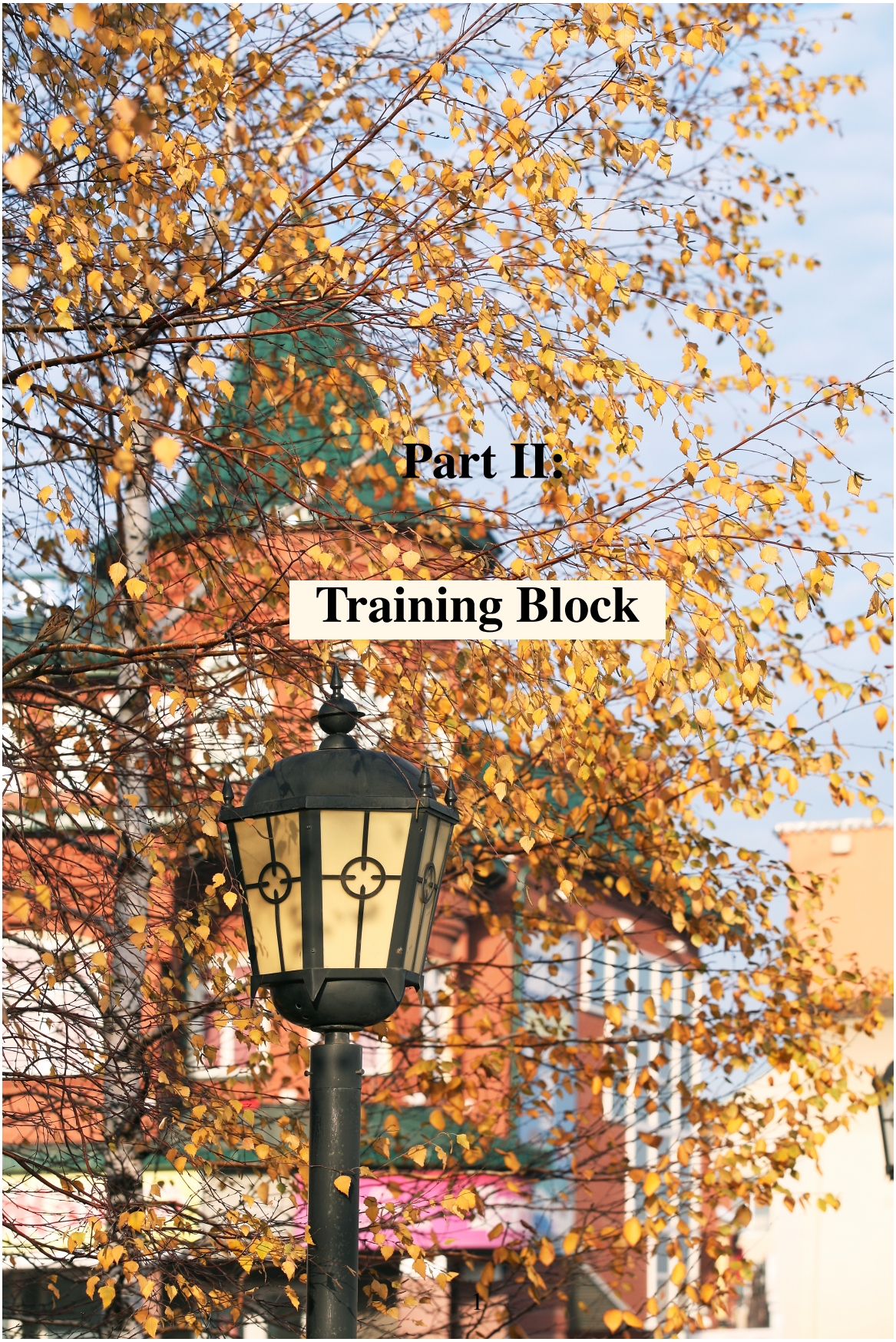
Contents

List of Figures



Part II:

Training Block

Chapter 3

Learning a neural network - Part I

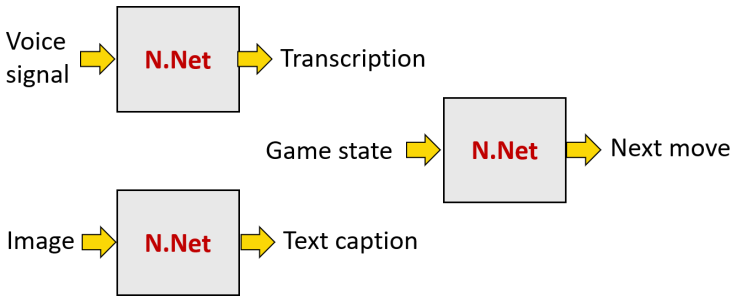
In Chapter 3, we learned that MLPs can emulate any function. However, we must still find out how we can use an MLP to emulate a specific *type* of desired function, e.g. a function that takes an image as input and outputs the labels of all objects in it, or a function that takes speech input and outputs 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?

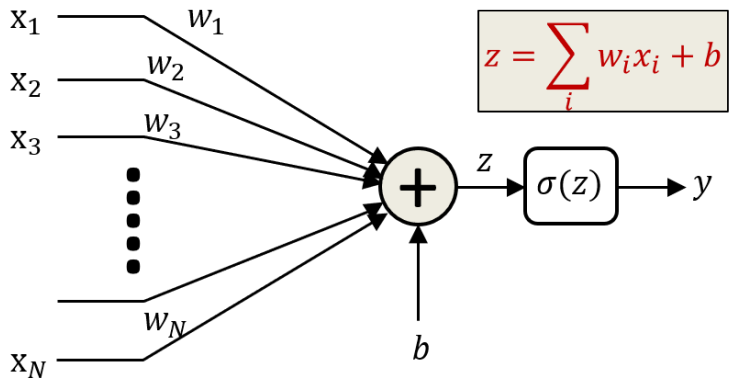
Let us first focus on the last question in this list: how do we compose a network that computes a given function? To understand how, let us return to the basic unit

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:

$$Linear: \quad f(ax_1 + bx_2) = af(x_1) + bf(x_2) \tag{3.1}$$
$$Affine: \quad \exists\, C \; such\; that\; g = f(x)C \tag{3.2}$$
$$g(ax_1 + bx_2) = ag(x_1) + bg(x_2) \tag{3.3}$$

The affine function of the inputs computed by the perceptron is a weighted sum of all of the inputs plus a bias. This affine function, 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.

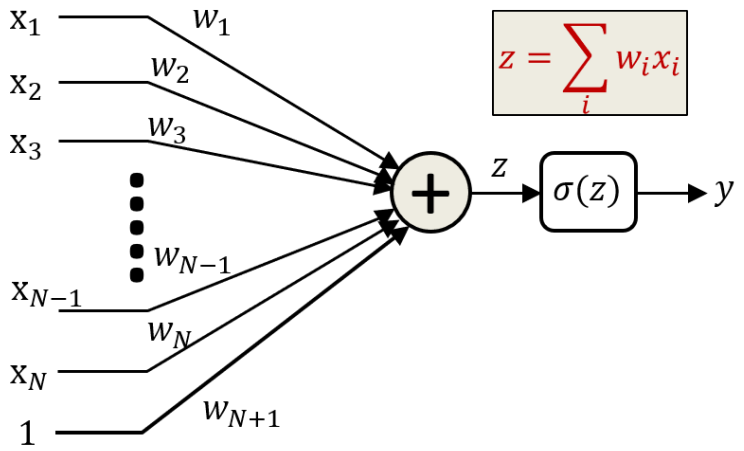


Perceptron: In a general setting, inputs are real valued. A bias b represents a threshold to trigger the perceptron. $\sigma(Z)$ is an activation function, which can be one of many types. Z is the affine function of inputs. The activation function operates on the affine value Z. Activation functions are not necessarily threshold functions.

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

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.

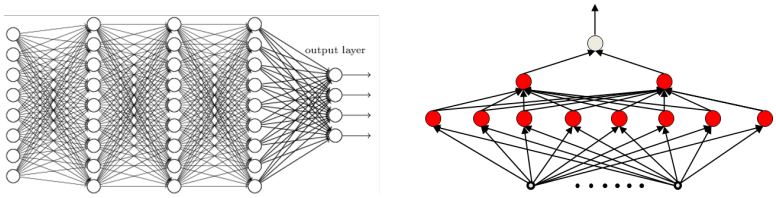


The bias can also be viewed as the weight of another input component that is always set to 1.

Figure 3.3: Including bias with the input in a perceptron

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.

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

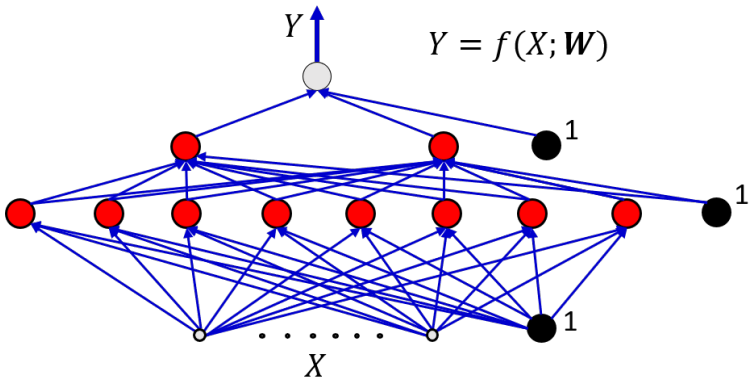
A perceptron (or neural) network is a function. It takes an input and produces an output. Fig. 3.5 shows a typical neural network. The parameters of the function are the weights and biases of the neurons in the network (represented by the blue arrows in the figure).

We represent this function as $f(X; \mathbf{W})$, where X is the input to the network, and $\mathbf{W}$ are its parameters – its weights and biases. For the network to compute a specific (desired) function, we must set the parameters $\mathbf{W}$ 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.

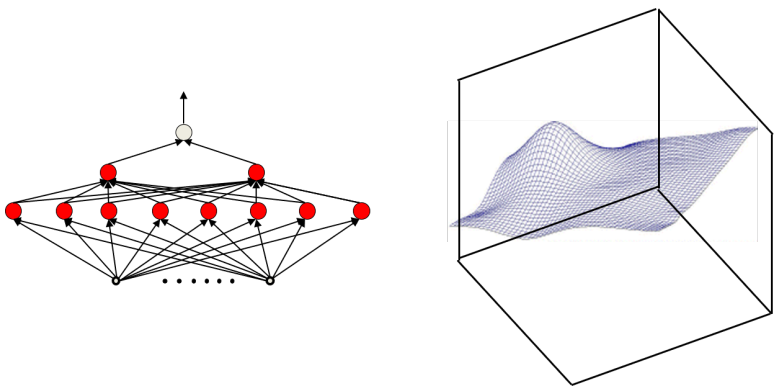
One approach is to try to handcraft a network (i.e., manually construct it). We can understand this approach with the help of the example in Fig. 3.7. To construct an MLP to compute the decision boundary shown in this figure over 2-dimensional inputs, a network must output 1 inside the diamond shape, and 0 outside. For this, let us use a network where the individual perceptrons have threshold activations.

We first construct a perceptron for one boundary, e.g. the boundary pointed to by



The network is a function $f()$ with parameters **W** which must be set to the appropriate values to get the desired behavior from the net.

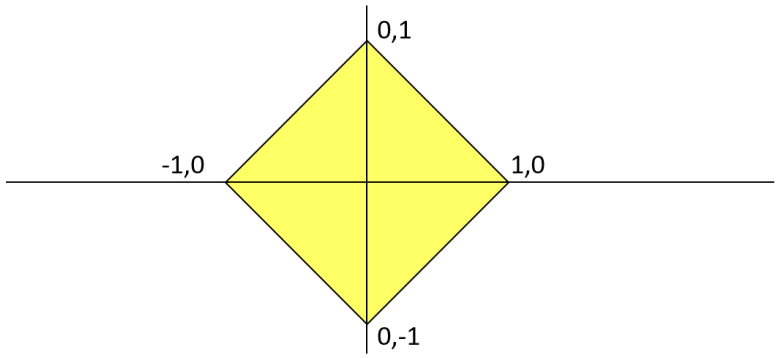
Figure 3.5: The network as a parametric function



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

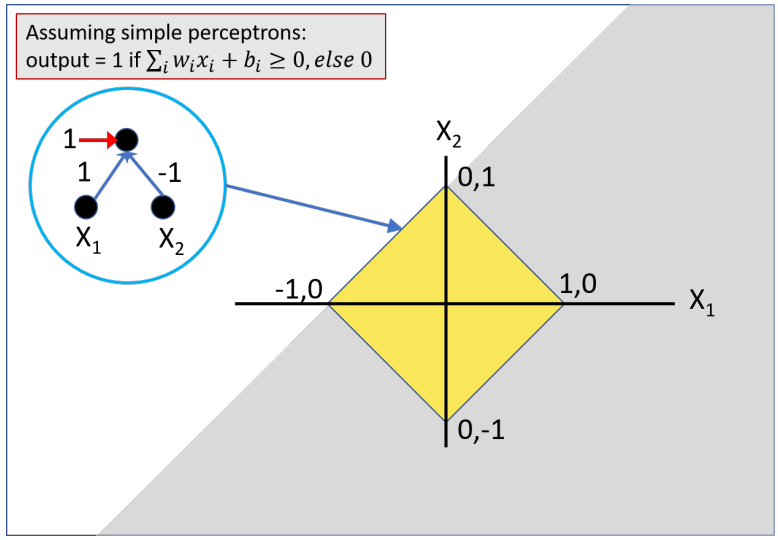
Figure 3.6: An arbitrary function and an MLP

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.



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

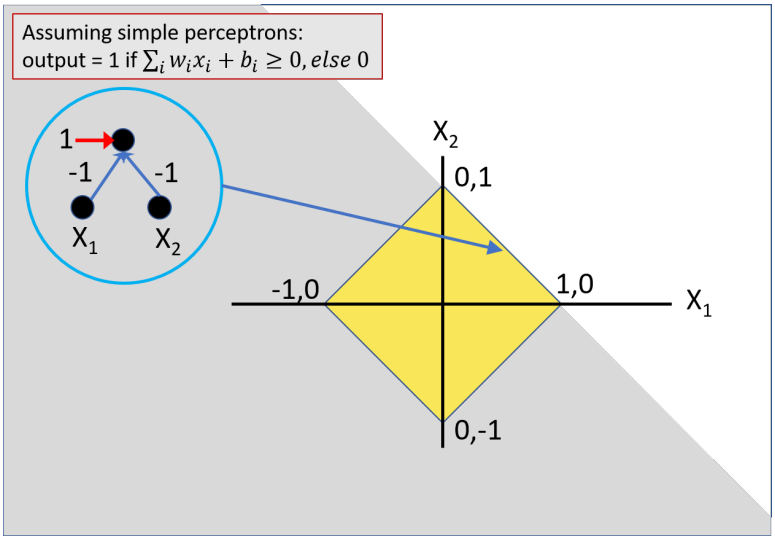


At both (0,1) and (-1,0) the output must become 0 along the straight line that represents the boundary to the left.

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

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.

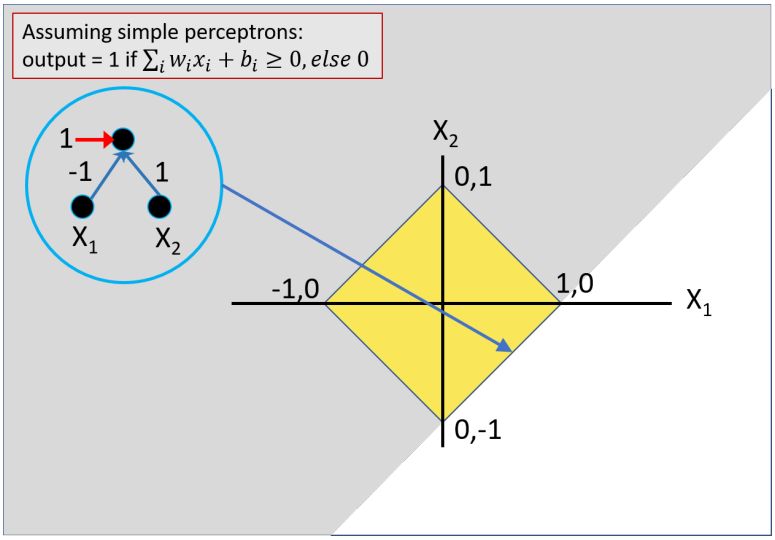
The third boundary, shown in Fig. 3.10 is captured by a perceptron with weights



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

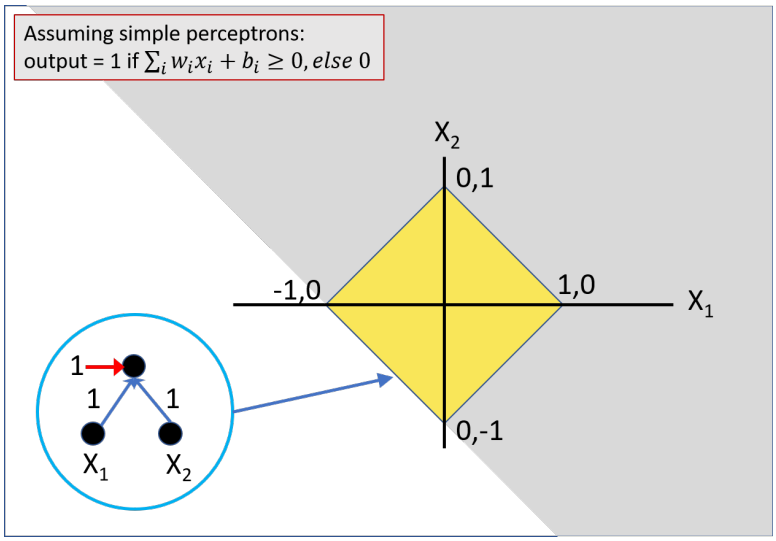
$-1, 1$ and bias 1.



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

The fourth boundary, shown in Fig. 3.11 is captured by a perceptron with weights $1, 1$ and bias 1.



The boundary in this case corresponds to weights $1, 1$ and bias 1. The corresponding perceptron is shown on the left.

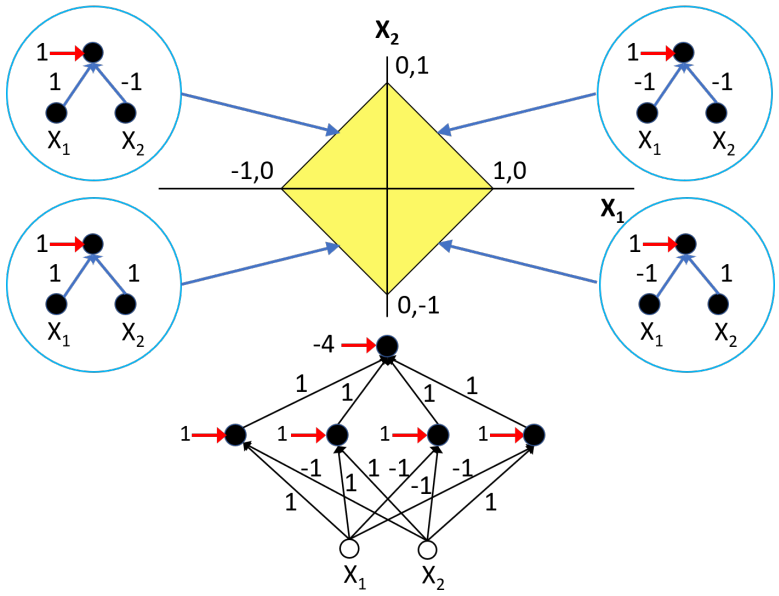
Figure 3.11: Constructing a perceptron for one edge of a diamond shaped decision boundary: 4

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 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 bounday.

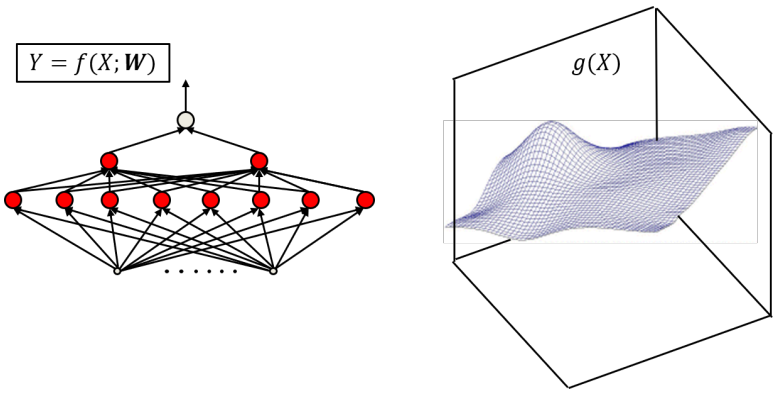
Figure 3.12: Constructing a perceptron for one edge of a diamond shaped decision boundary: 5

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.

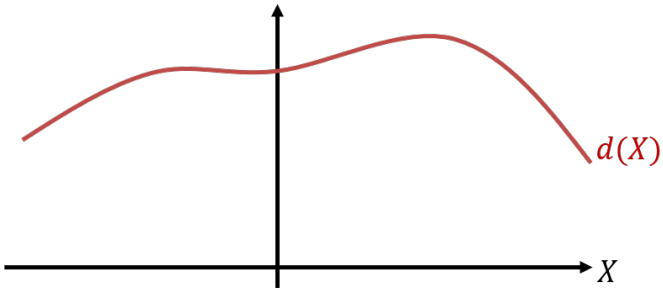
For *any* given setting of the weights $\mathbf{W}$, the network will actually model some function $f(X; \mathbf{W})$, as shown in Fig. 3.14().

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 $\mathbf{W}$ to minimize the area (shown highlighted in Fig. 3.16) that represents



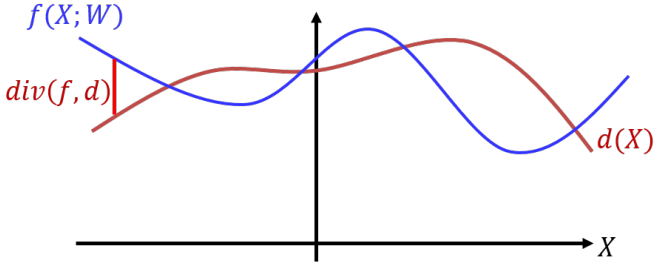
Right: A function $g(X)$ that we want to model. **Left:** An MLP that is sufficient to model it (schematically shown). We must derive the parameters of this network computationally, so that it can model $g(X)$ accurately.

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**

the error.

To quantify this area, we will first define a divergence function $div()$, which

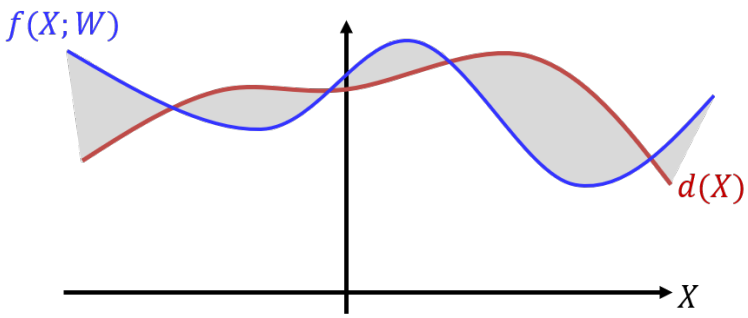


Figure 3.16: The divergence as an area

quantifies the difference between the output of the network $f(X;\mathbf{W})$ 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 $f(X;\mathbf{W})$ is exactly equal to $d(X)$, and is positive when the two are not the same.

When $f(X;\mathbf{W})$ has the capacity to exactly represent $d(X)$, the setting of $\mathbf{W}$ that achieves it is given by

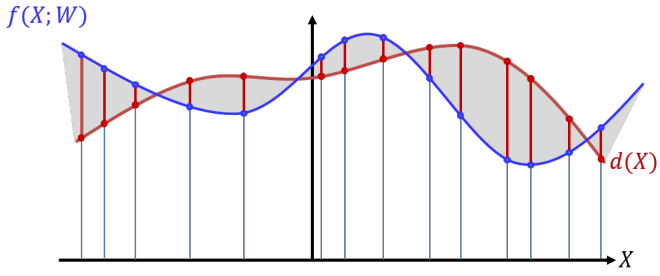
$$\hat{\mathbf{W}} = \underset{\mathbf{W}}{\operatorname{argmin}} \int_X div(f(X;\mathbf{W}), d(X))dX \tag{3.4}$$

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

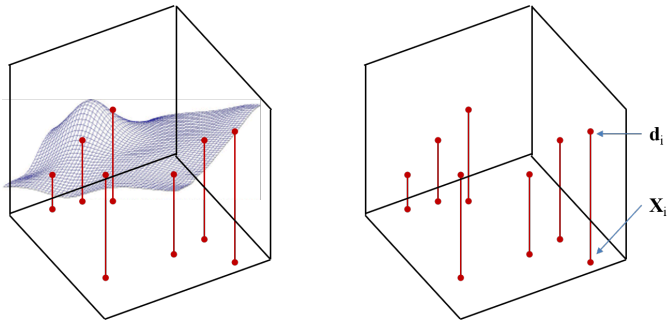
The entire area shown highlighted in the equation above is the integral of the divergence function, computed over all valid values of the input. This quantifies the error between the actual output of the network, and the function we want the net to compute, and our goal is to compute the $\mathbf{W}$ 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 $div(f(X;\mathbf{W}), g(X))$.

The problem here is that in order to compute the integral in Eq. 3.18, 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.



Sampling of $d(X)$.



Samples of X are drawn from $P(X)$. The input-output pairs from the samples are used to learn the network. We estimate the network parameters to "fit" the training data exactly.

Figure 3.17: Sampling the function to be modeled for learning

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 applications, 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. The X_i, d_i pairs that we collect comprise our "training" samples. The entire function must now be learned from these samples. This is turn specifically requires that we learn **W** to correctly compute the output at the sampled input values.

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. 3.17). 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.

Before we proceed, it is important to be clear about the notations used in this context. They are listed below.

Info box 1.1: Notations

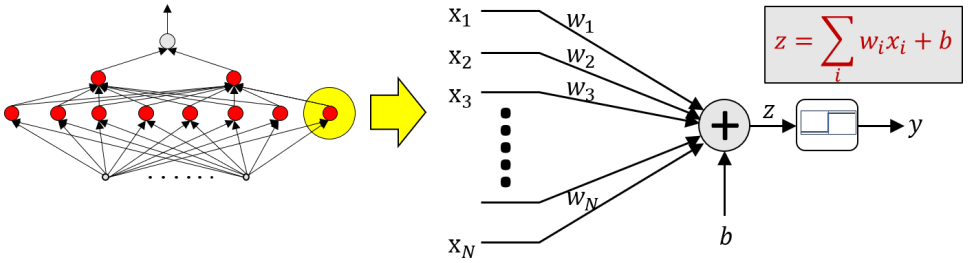
X: An input vector
x: A scalar input
x_i: The i^{th} dimension of a vector X
X_i: The i^{th} input vector
Y: An output vector (may be used for a 1-D vector when it can generalize to multiple dimensions)
y: A scalar output
$\mathbf{W}$: Set of weights (also, the weight vector)
w_i: The i^{th} weight (i^{th} element of $\mathbf{W}$, or the i^{th} dimension of $\mathbf{W}$)
b: Bias of a unit

To learn how to find a function that best fits a set of sample points, let us begin with a simple task – that of learning a function that is a classifier. This is simpler than learning a regression. Here we will specifically consider binary classification, though the discussion that follows also generalizes to multi-class classification.

We start with the original MLP as proposed by Minsky, shown in Fig. 3.18. This is a network of threshold-activation perceptrons. Let us see how we can train this network using only a set of training instances of input-output pairs.

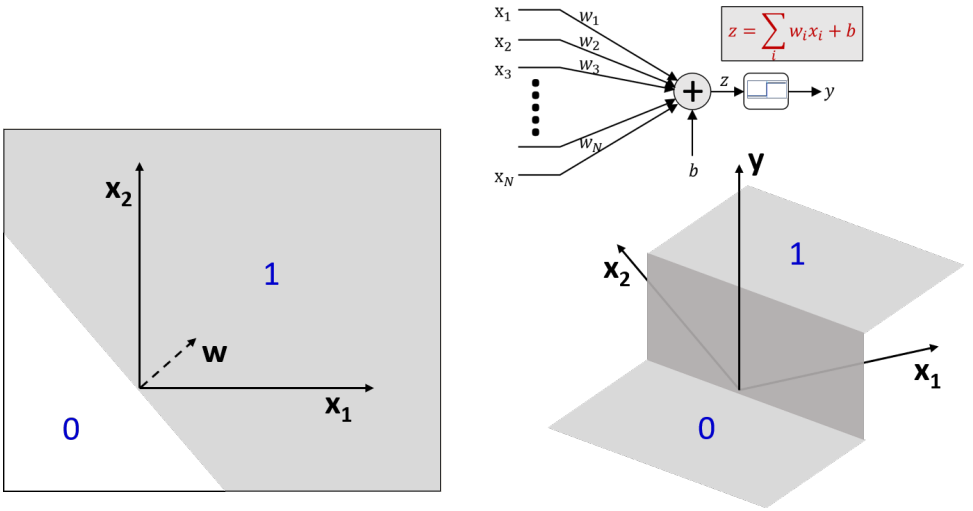
We first consider the simplest MLP, which has only one unit: a single perceptron, also shown in Fig. 3.18. As we have seen in Chapter 1, the perceptron models a step function, where the output is 0 on one side of a hyperplane and 1 on the other side. Let us consider the problem of how we can learn this step function.

In this case too, we may not be given the entire function – we may only have



Left: A network of threshod units. **Right:** A single unit (a perceptron).

Figure 3.18: The original MLP: a network of threshold units



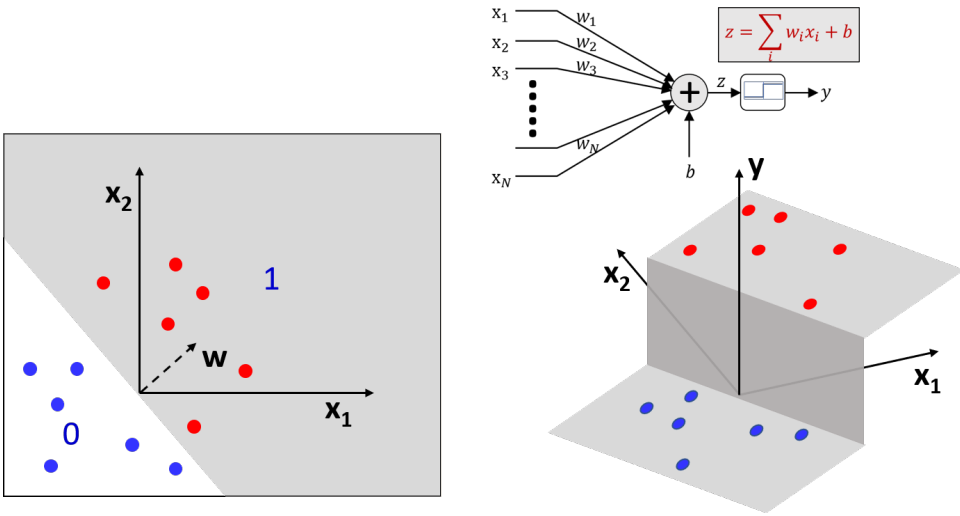
We must learn the step function that this single perceptron models.

Figure 3.19: The function modeled by a single perceptron with a threshold unit

some input-output pairs, shown as blue and red dots in Fig. 3.20. Each of the dots shown represents an (X, Y) pair, where X are the coordinates of the input, and Y is the desired output, which is either 0 or 1. We must learn the weights of the perceptron that models the step function from only these pairs.

3.1.2.1 Learning the perceptron

The output of the perceptron is given by:



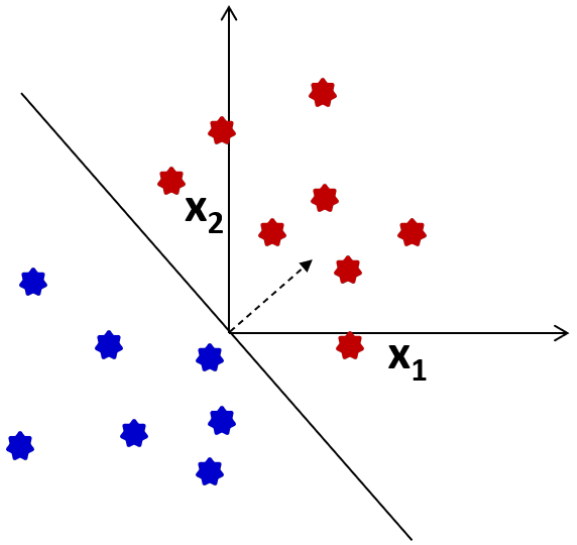
Each of the dots shown represents an (X, Y) pair, where X are the coordinates of the input, and Y is the desired output, which is either 0 or 1 in this case.

Figure 3.20: Sampling the step function modeled by a single perceptron

$$y = \begin{cases} 1 & if \ \sum_i^N w_i x_i + b \geq 0 \\ 0 & else \end{cases} \tag{3.5}$$

The goal is to learn $\mathbf{W} = [w_1, w_2, \ldots, w_N]^T$ and b , given several (X,Y) pairs. The expression in Eq. 3.6 is 1 when the affine combination of inputs is positive (or rather non-negative), and 0 otherwise. The parameters of the perceptron are its weights $\mathbf{W}$ and bias b. The classification boundary is a hyperplane, as shown in Fig. 3.20. Learning the perceptron is the same as learning the equation of a hyperplane such that all training instances with class label 1 (shown by red dots) lie on one side of the plane, and all instances with label 0 (shown by blue dots) lie on the other. We would, for example, like to learn the equation of the line (which is the classification boundary) shown in Fig. 3.21.

For convenience, we reconfigure the perceptron by adding an additional input whose value is fixed to 1. The weight for that input is now the bias, and $x_{N+1} = 1$.



Learning the perceptron is the same as learning the equation of a hyperplane such that all red dots (which indicate an output of 1) lie on one side of the plane, and all blue dots lie on the other. In this case it is the equation of the line shown.

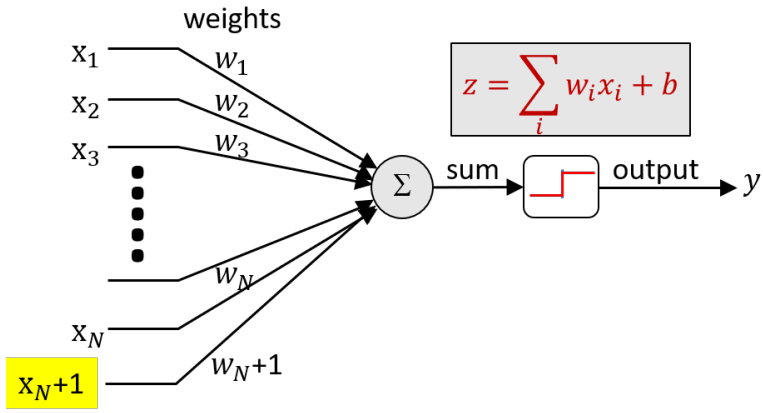
Figure 3.21: Learning the parameters of the perceptron from samples

$$y = \begin{cases} 1 & if \ \sum_i^{N+1} w_i x_i \geq 0 \\ 0 & else \end{cases} \tag{3.6}$$

Note that the boundary $\sum_i^{N+1} w_i x_i \geq 0$ is now a hyperplane through origin. We can now restate the perceptron by saying the output is 1 when the weighted combination of inputs is positive, and is 0 otherwise. The boundary is where the weighted sum of inputs is exactly 0: $\sum_i^{N+1} w_i x_i = 0$. This is the equation of a hyperplane through the origin.

In this format what we really want to do is to find the equation of the hyperplane passing through origin such as the one schematically shown in Fig. 3.23 by the red line, such that all training instances from class 1 (red dots lie on one side) and instances of class 0 (blue dots) lie on the other. In other words, we must find the hyperplane $\sum_i^{N+1} w_i x_i = 0$ that perfectly separates the two groups of points.

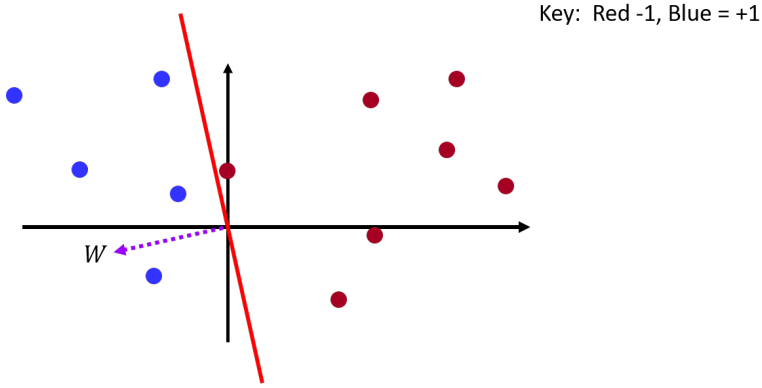
Before we continue, let us see how the classification rule works. Here we note



Restating the perceptron by adding another dimension to X. The boundary is now a hyperplane through the origin.

Figure 3.22: Restating the perceptron: adding bias as input

that $\mathbf{W} = [w_1, w_2, \ldots, w_N, w_{N+1}]^T$ is a vector that is orthogonal to the hyperplane. The component-wise product of vectors $\mathbf{W}$ and X, $\sum_{i=1}^{N+1} w_i x_i$, is the innerproduct $\mathbf{W}^\top X$ between the two, and this is 0 when $\mathbf{W}$ and X are orthogonal to one another. In fact the equation for the hyperplane ($\sum_i^{N+1} w_i x_i = \mathbf{W}^\top X = 0$) itself means "the set of all X's that are orthogonal to $\mathbf{W}$." The set of all such X's forms a hyperplane (which would be a line in two-dimensions), and represents the classification boundary.



Find the weights vector $\mathbf{W}$ such that $\mathbf{W}^T X = 0$ is positive for all blue dots and negative for all red ones.

Figure 3.23: Requirement for learning the perceptron [label red: 1, blue: -1]

For vectors X that do *not* lie on this hyperplane, we note that the inner product $\mathbf{W}^\top X$ is simply proportional to the cosine of the angle θ between $\mathbf{W}$ and X: $\mathbf{W}^\top X = |X||\mathbf{W}| \cos\theta$. When the two vectors point in the same direction, i.e. into the same half of the space across the boundary (line) in Fig. 3.23, θ lies in the range $-\pi/2 < \theta < \pi/2$, and $\cos(\theta) > 0$. Consequently, the product $\mathbf{W}^\top X$ too will be positive. When the two vectors point in opposite directions, into different halves of the space, i.e. $\pi/2 < \theta < 3\pi/2$, we have $\cos(\theta) < 0$, and hence inner product $\mathbf{W}^\top X$ will be negative. And, of course, when X is exactly at 90^o to $\mathbf{W}$, the product will be 0 and it will lie exactly on the hyperplane (indicated by the red line in the figure).

Thus our problem of finding the perceptron given these data samples, is that of finding the hyperplane that exactly separates the two sets of sample points. To find this hyperplane, we must find the weight vector that is exactly orthogonal to this plane. Additionally, the weight vector must point into the positive class – that way, for the positive (red) instances, $\mathbf{W}$ and X are aligned.

The algorithm for this is summarized below:

Procedure 1.1: Finding the classification boundary for one perceptron

Cycle through the training instances
Only update $\mathbf{W}$ on misclassified instances
If instance X is misclassified:
If X belongs to the positive class (positive misclassified as negative),
$\mathbf{W} = \mathbf{W} + X$
If X belongs to the negative class (negative misclassified as positive),
$\mathbf{W} = \mathbf{W} - X$

The perceptron algorithm given above is based on a simple idea: if we had only one positive instance, the weight that puts it on the positive side of the hyperplane with maximum clearance would be perfectly aligned with the instance such that the angle between the two is 0, i.e. $\theta = 0$ (and $\cos\theta = 1$)

If we had only one negative instance, by the same reasoning, the weight that would give us a hyperplane that puts the point on the negative side of it, with maximum clearance would be the negative of the instance: $\mathbf{W} \propto -X$.

The perceptron learning rule is an iterative learning algorithm that builds on this idea. We initialize the set of weights $\mathbf{W}$ with some value, such as 0. Then we cycle through the training instances and test all of them. If any positive instance is misclassified as negative, we then add that input vector to the current weight vector – i.e. we "pull' the weight vector towards the ideal direction for that instance: $\mathbf{W} = \mathbf{W} + X$. If any negative-class instance is misclassified (as positive), we subtract that input vector from the weight: $\mathbf{W} = \mathbf{W} - X$.

More formally, the algorithm may be written as follows. We are given N training instances $(X_1, y_1), (X_2, y_2), \ldots, (X_N, y_N)$. Note here that X_i is the i^{th} training sample, and y_i its corresponding class label. For the perceptron learning algorithm, we will set our labels for the positive and negative classes to be +1 and -1 (rather than 1 and 0) to simplify the notation for the algorithm.

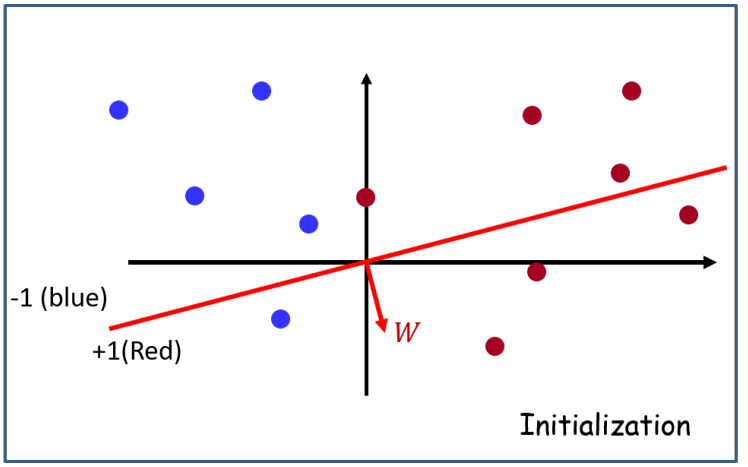
Algorithm 1: Training pseudocode

```
Initialize weights;
repeat
    for (X, y) in training set do
        O(X) = sign(W^T X);
        if O(X) ≠ y then
            W = W + yX;
        end
    end
until no more classification errors;
```

In the algorithm given above, we first initialize the weight vector $\mathbf{W}$ (the initial value could simply be 0). For each training instance, we compute the sign of the inner product between the weight vector and the input. If this sign matches the label of the input, the classification is correct. Otherwise it is wrong. If the classification is wrong we simply add the product of the label and the input X to

the current weight. Note that this *adds* positive instances, for which $y = 1$, and subtracts negative instances,for which $y = -1$. We continue to do this until there are no more misclassifications and therefore no more updates to be made.

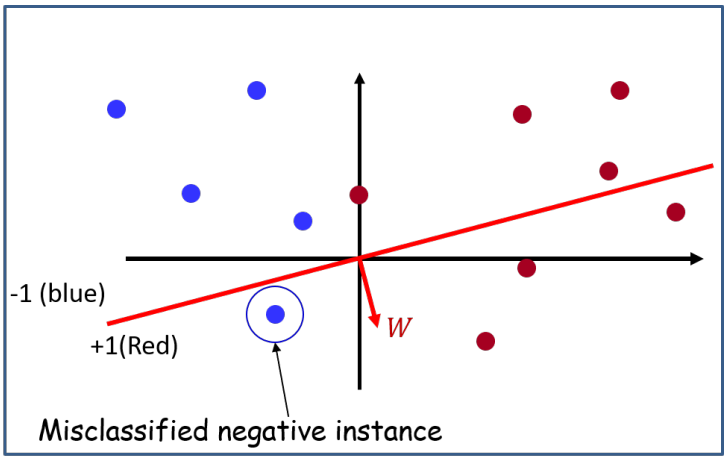
The figures that follow illustrate this process schematically. Consider first the the figure in Fig. 3.24. The red dots are training instances from the positive class. The blue dots are training instances from the negative class. We initialize the weights vector, say as shown. The classification boundary is perpendicular to it and is shown by the long red line. The instances on the same side as the weight vector are classified as the positive class the instances on the other side are classified as the negative class. We then check each of our training points to see if it is correctly classified by this boundary



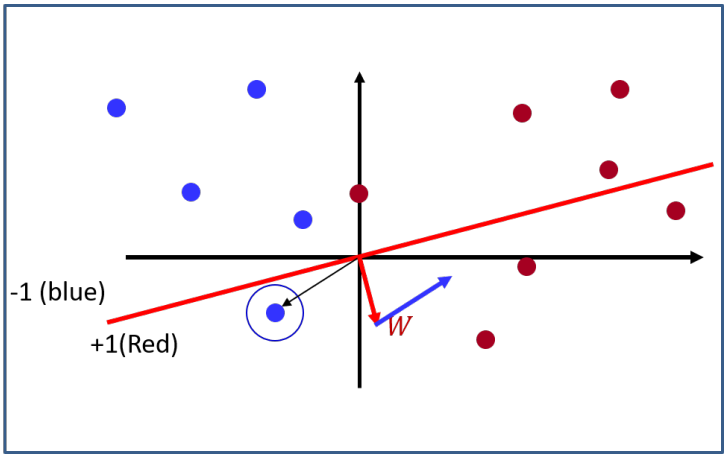
The classification boundary is perpendicular to the initialized weight vector and is shown by the long red line.

Figure 3.24: Illustration of the perceptron training algorithm: 1

In Fig. 3.25(a), we see a negative class instance that has fallen on the positive side of the plane and is misclassified. Since this is a negative class instance, we will subtract it from the current weight vector. This is the same as flipping the vector as shown in Fig. 3.25(b), and adding it to the weight.



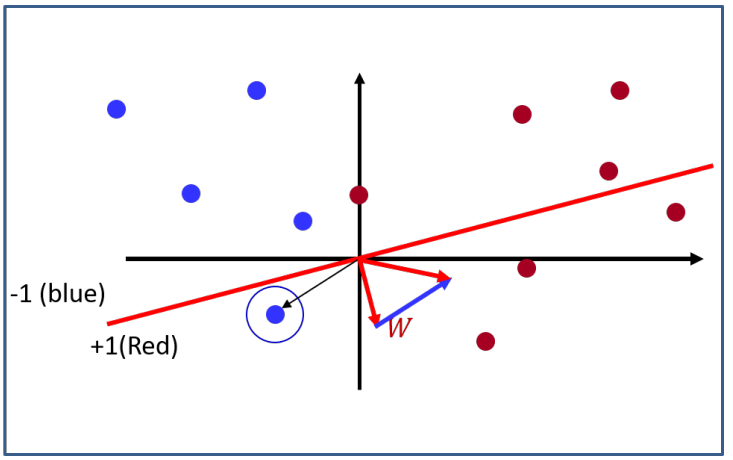
(a) A negative class instance that has fallen on the positive side of the plane and is misclassified.



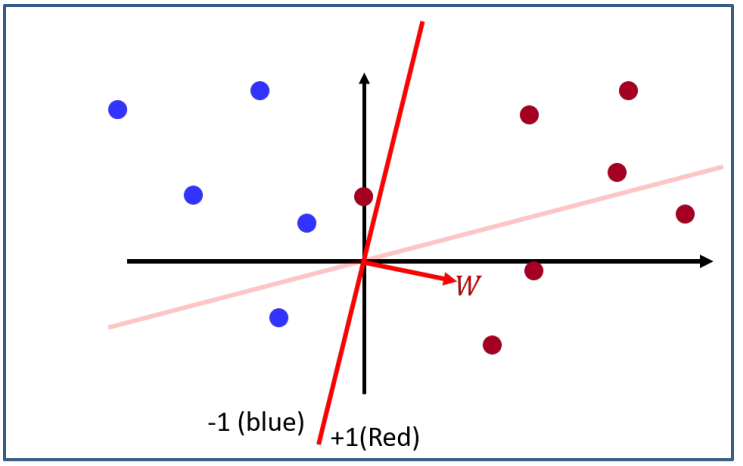
(b) The misclassified instance is subtracted from the weight vector.

Figure 3.25: Illustration of the perceptron training algorithm: 2

This gives us a new updated weight vector as shown in Fig. 3.26(a). This in turn gives us a new decision boundary, as shown in Fig. 3.26(b).



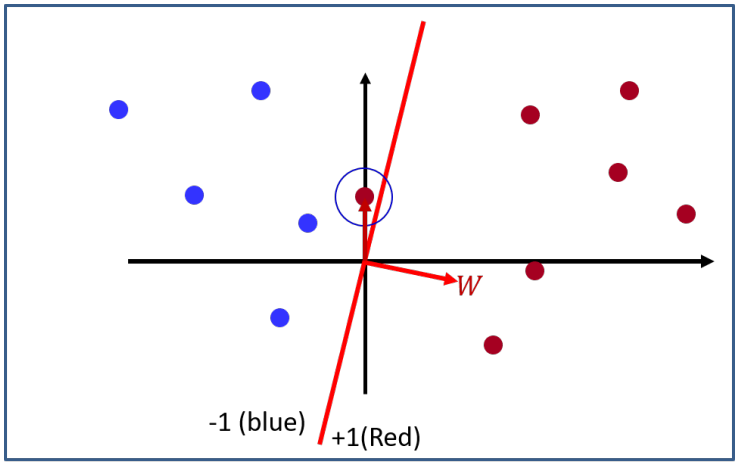
(a) The new updated weight vector.



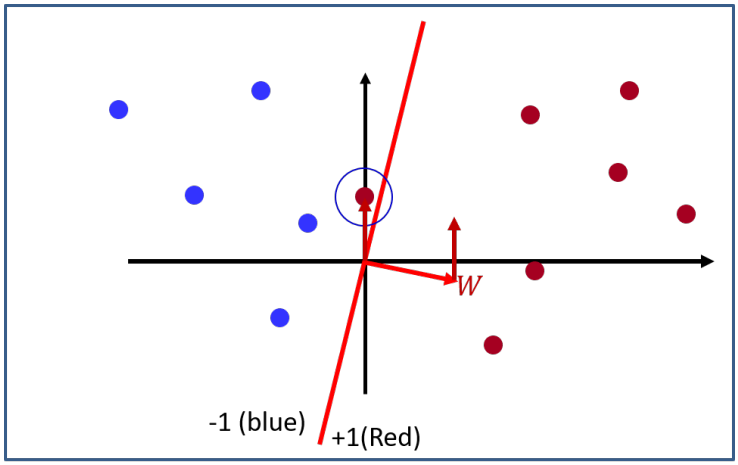
(b) The updated weight vector defines an updated decision boundary.

Figure 3.26: Illustration of the perceptron training algorithm: 3

However, now we find that we have a different misclassified (positive) instance, as shown in Fig. 3.27(a). Since this is an instance from the positive class, we *add* it to **W**, as shown in Fig. 3.27(b).



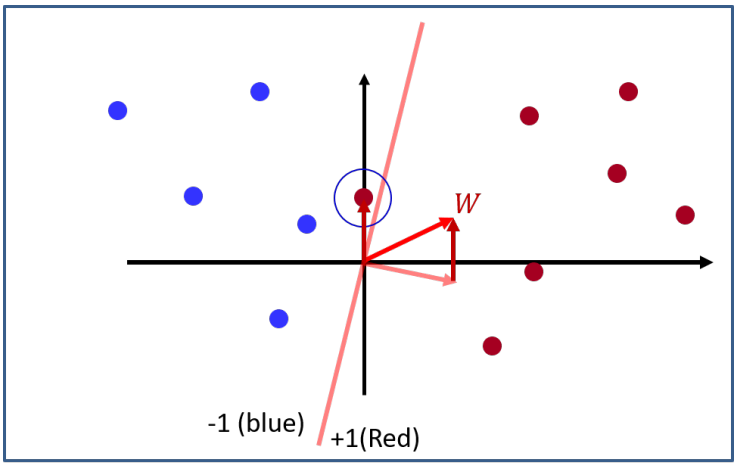
(a) The positive instance shown is now misclassified with the new decision boundary.



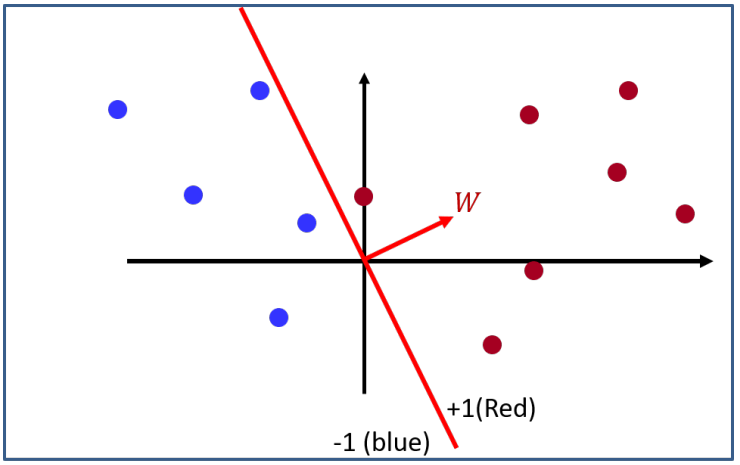
(b) The misclassified positive instance is *added* to **W**.

Figure 3.27: Illustration of the perceptron training algorithm: 4

This in turn gives us a new weight as shown in Fig. 3.28(a). We now obtain a new updated decision boundary, as shown in Fig. 3.28(b). At this juncture all training instances are being correctly classified and there are no more updates to be made. This completes the training of the perceptron in this example.



(a) The new weight vector.



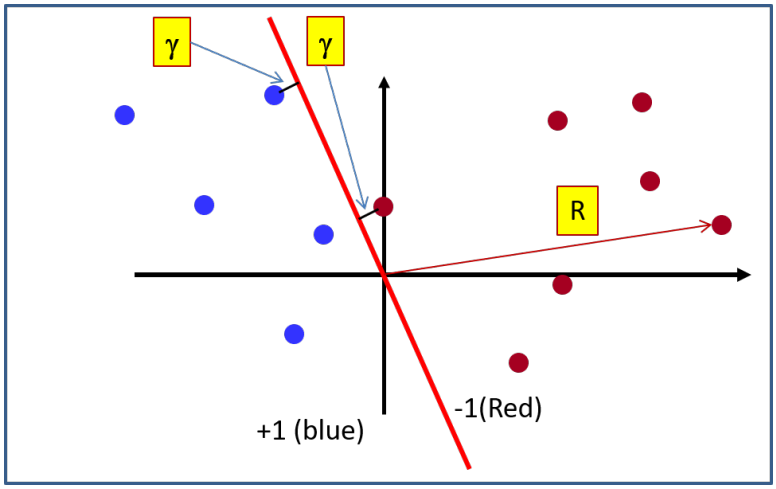
(b) The new decision boundary yields perfect classification. No more updates are needed.

Figure 3.28: Illustration of the perceptron training algorithm: 5

3.1.3 Convergence of the perceptron training algorithm

The perceptron algorithm is guaranteed to converge to a solution, provided the training data are linearly separable (i.e., if there really is a hyperplane that perfectly separates the positive instances from the negative instances). In fact,it will converge in a *finite* number of steps. It will converge in no more than $\left(\frac{R}{\gamma}\right)^2$ updates, specifically if we initialize $\mathbf{W}$ as 0, where R is the distance of the farthest training point from origin (vector length of the farthest training sample) and γ is the distance of the closest training point from the boundary (which is the same as the *margin* in a support vector machine (SVM)). Intuitively this can be interpreted to mean that it would take many increments of size γ to undo an error resulting from a step of size R.

Fig. 3.29 helps visualize R and γ. R is the length of the farthest/longest training vector from the boundary. If we initially make a mistake and set $\mathbf{W}$ to R, then we have to make $\left(\frac{R}{\gamma}\right)^2$ corrections to undo it.



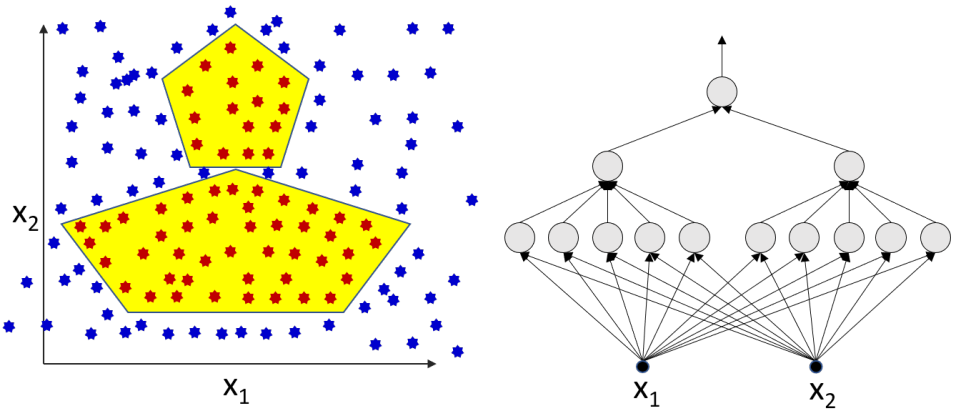
R is the length of the longest training vector. If we draw the most conservative decision boundary, so that the training instances are all maximally distant from it, then γ is the distance of the closest training point to this conservative boundary. γ is thus the best-case margin.

Figure 3.29: Entities that govern the convergence of perceptron training

3.2 Learning networks that model complex decision boundaries

So far, we have seen that we can learn the parameters of a single perceptron in a reasonable amount of time if the data are separable. We have also reasoned in earlier chapters that more complex decision boundaries require *networks* of perceptrons to model. Let us now consider how we can learn the parameters of networks that model complex decision boundaries.

Consider the decision boundary of Fig. 3.30 – we have seen this earlier in Chapter 2. This requires a network – a multi-layer perceptron – to model. Our goal is to learn an MLP to model this classification boundary (on 2-dimensional inputs). The network must be such that it outputs a 1 for instances within the yellow regions of Fig. 3.30, and 0 outside. As in the previous example, we want to automatically learn this network using just the given training samples, shown by the red and blue dots. From our discussion in Chapter 2, we know that the decision boundary shown in Fig. 3.30 can be perfectly represented by the MLP with the structure shown in this figure.

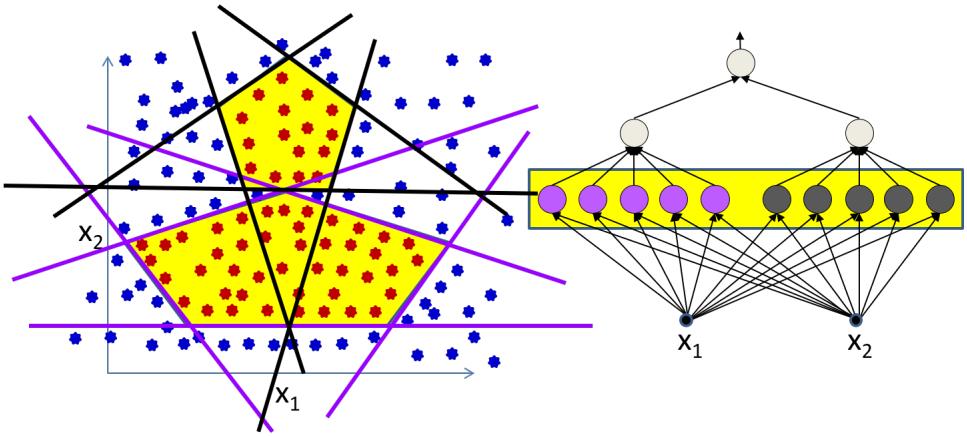


Left: A more complex decision boundary. **Right:** An MLP that can represent the decision boundary on the left perfectly. We must learn its parameters. Even using the perfect architecture, can we use the perceptron algorithm and learn its parameters by making incremental corrections every time we encounter an error?

Figure 3.30: Learning the parameters of a perceptron network: 1

Given a network with the perfect architecture for this problem (or in general the more complex problems that his example represents), let us consider how we can apply the perceptron algorithm, which incrementally tests training instances and makes adjustments to network parameters when a training instance is wrongly classified, to train this network.

In order to understand how, we first consider what happens at the lower levels (first hidden layer of the network). To learn the pattern represented by the function perfectly, each of the neurons in the first hidden layer must first learn the decision boundaries shown by the lines in Fig. 3.31. One set of first-layer neurons, shown as purple circles in this figure, must learn the purple decision boundaries, so that we can compose the lower pentagon with them. The second set of first-layer neurons, shown as the dark grey circles, must learn the boundaries shown by the black lines. Only if we learn these boundaries can we subsequently construct the pentagonal decision regions and the overall classification function. Let us see how this can be done.



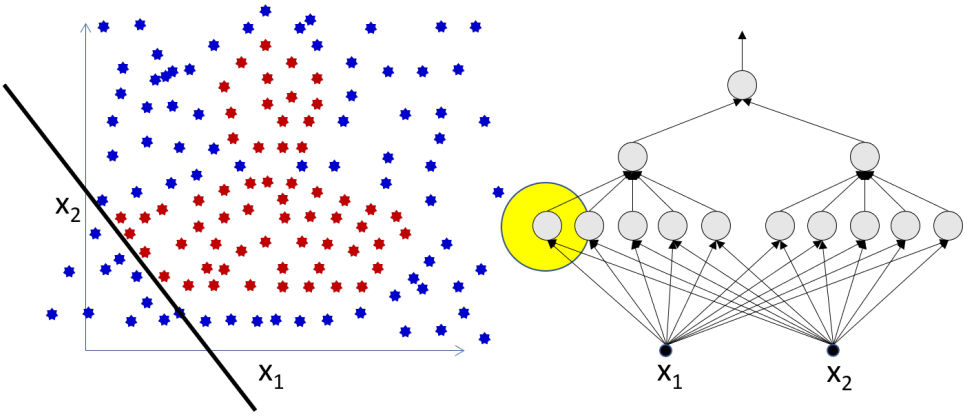
The lower-level neurons are linear classifiers. They require linearly separated labels to be learned. The labels actually provided are not linearly separated. The challenge here is that we must also learn the labels for the lowest units.

Figure 3.31: Learning the parameters of a perceptron network: 2

For simplicity we assume that just the one neuron highlighted in Fig. 3.32 must learn the boundary shown by the line in this figure, and that the parameters of the rest of the neurons are already perfectly learned. This one remaining neuron

must be learned using the training data, and the only labels we have are shown by the colors of the dots in the figure.

With some thought we see that we cannot learn the line shown from the training data shown using the perceptron algorithm, since the dots are not linearly separable. We can only learn a linear classifier perfectly if the data are linearly separable.



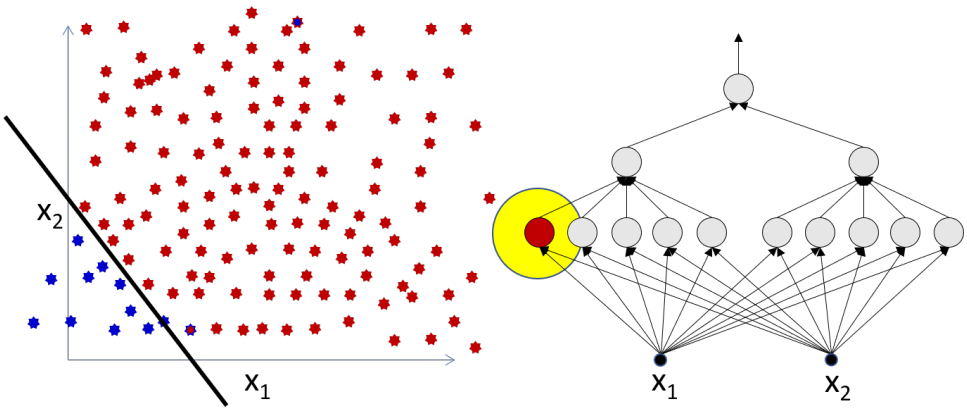
Consider a single linear classifier that must be learned from the training data. The dots shown are not linearly separable if we use the labels that were given to us, and a linear classifier cannot be learned for this reason.

Figure 3.32: Learning the first hidden layer of the network: 1

To learn the line shown in Fig. 3.33, we require linearly separable labels of the kind shown in the figure (red on one side, blue on the other of the line). However, these are not given to us.

To learn the neuron under consideration, we must *relabel* some of the training instances. This must be part of the learning process: we must also learn which of the samples (dots in the figure) that were labeled as 0 should be flipped to 1, and which that were labeled 1 must be flipped to 0, in order to learn the line as shown in Fig. 3.34.

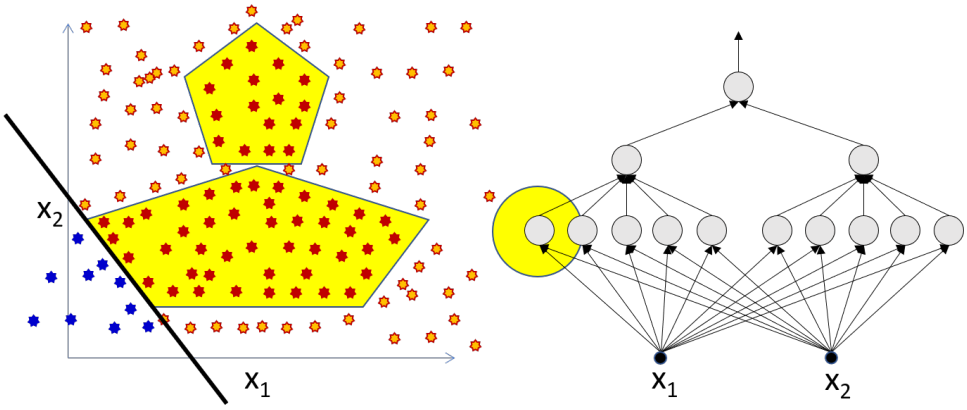
However, we would not know *a-priori* which labels must be flipped. To deal with this situation, we must consider and test *every way* of relabeling the data until we find the one set such that when we learn the line with it, the resulting decision



To learn the line shown, linearly separable lables are required as shown in the figure.

Figure 3.33: Learning the first hidden layer of the network: 2

boundary correctly completes the pattern of the decision boundary, and we get perfect classification for all the training points.

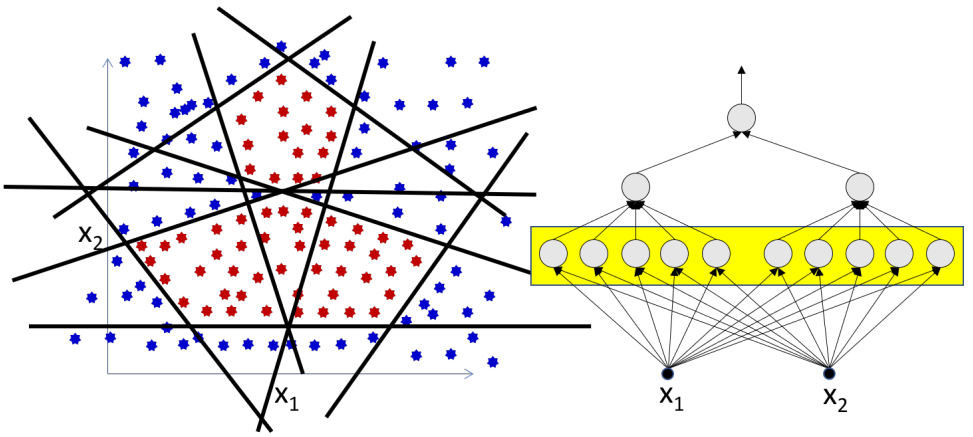


In the learning process some of the blue samples should be flipped red, and vice-versa, in order for the samples to be linearly separable, so that the line shown can be learned. For a single line, we try out every possible way of relabeling the dots such that we can learn a line that correctly completes the double-pentagon decision boundary.

Figure 3.34: Learning the first hidden layer of the network: 3

Thus, even if the parameters of the rest of the network are learned, and we only have to learn one more linear boundary, we must still try out every possible way of relabeling the training instances such that the line separates the two classes, while also giving us the correct overall function.

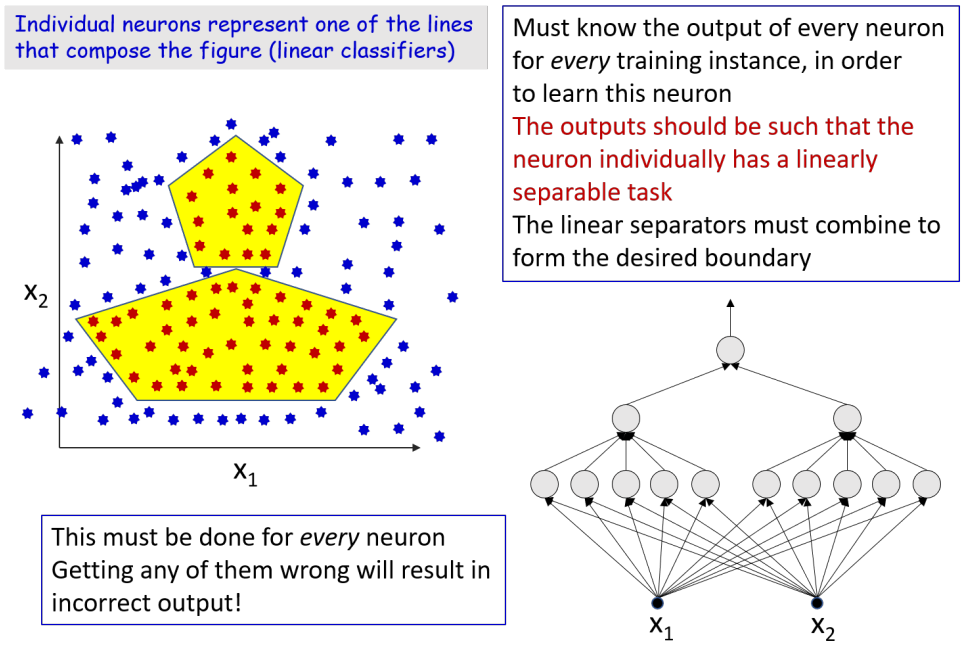
In a real situation, the perceptrons in the network are not already trained. We must perform such an exhaustive search (of all possible ways of relabeling the data) for every one of the neuron (perceptrons) in the network, such that the network as a whole computes the desired classification pattern. For each of the neurons this is an exponential search over inputs. For the entire network, the overall search is exponential in both the number of inputs *and* the size of the network.



Relabeling must be done for each of the lines (perceptrons) such that, when all of them are combined by the higher-level perceptrons, we get the desired pattern Basically an exponential search over inputs.

Figure 3.35: Learning the first hidden layer of the network: 4

These seemingly insurmountable requirements (i.e. the situation we would face if we tried to learn this entire network using the perceptron algorithm) are summarized in Fig. 3.36. Even if our network has the perfect architecture for the problem, we would have to know the appropriate label for every training instance, for every neuron, in order to learn the neuron. The neuron-specific label reassignments should be such that each neuron by itself has a linearly separable task to learn, and the learned linear separators combine to form the desired decision boundary. If even one of the training instances is relabeled wrongly for even one of the neurons, the entire network we learn will be wrongly learned and will produce incorrect output.

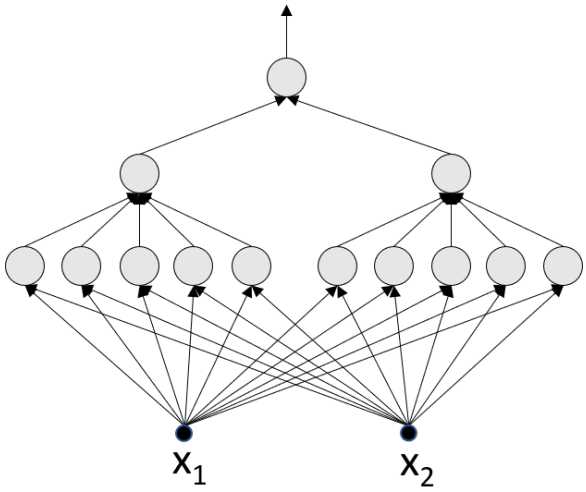


Leraning an entire MLP to model a complex function using the perceptron learning algorithm described earlier in this chapter is an insurmountable problem.

Figure 3.36: Learning the first hidden layer of the network: 5

3.3 Learning a *multi-layer* perceptron

As we can see now from our example, training the network shown in Fig. 3.37 using the perceptron rule is a combinatorial optimization problem. We don't know the outputs of the individual intermediate neurons in the network for any training input *a-priori*. So to use the perceptron rule, we must must also determine the correct output for each neuron for every training instance as part of the training. This is an NP-complete problem (it is equivalent to the knapsack problem) and has exponential-time complexity.



Training samples only specify the input and output of network. Intermediate outputs (outputs of individual neurons) are not specified.

Figure 3.37: Training a multi-layer perceptron: 1

3.3.1 Greedy algorithms: ADALINE and MADALINE

It is clear to us by now that the perceptron learning algorithm cannot directly be used to learn an MLP – it incurs the exponential complexity of assigning intermediate labels, even if the classes are separable and our network structure is perfect for it. It gets even worse when classes are not actually separable.

As a workaround, in 1963 Bernie Widrow [1] proposed a greedy algorithm that used greedy search to learn the perceptrons. The algorithm to learn a single neuron was called **ADALINE**, and the MLP extension of it was called **MADALINE**.

Fun facts 3.1: Bernie Widrow

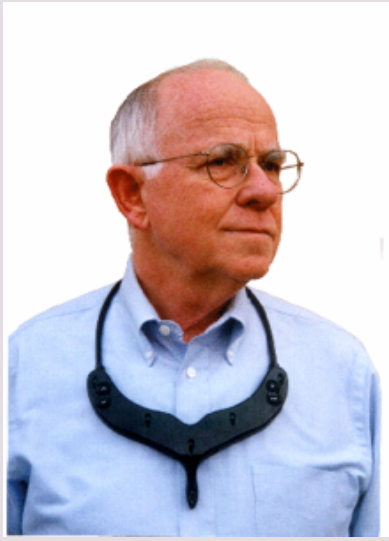


Figure 3.38: Bernie Widrow

Bernie Widrow is a professor of Electrical Enginnering at Stanford University. He (with his student Ted Hoff) invented the Least Mean Squares (LMS) filter adaptive algorithm, a staple in Electrical Engineering and related fields. The algorithm is now famous as the LMS algorithm [2] and is widely used. It is also known as the "delta rule."

His work on perceptrons in the early 1960's included the first known attempt at an analytical solution to training the perceptron and the MLP.

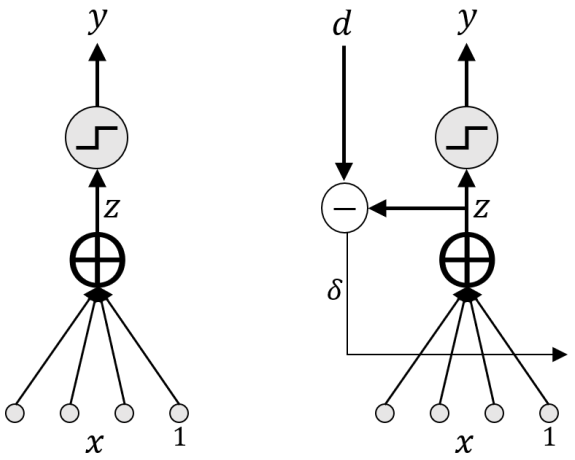
3.3.2 ADALINE

Let us first discuss ADALINE. Consider the single perceptron in Fig. 3.39. For this, using the 1-extended vector notation to account for bias, we can write:

$$z = \sum_t w_i x_i \tag{3.7}$$

which implies that

$$y = \begin{cases} 0, & z < 0 \\ 1, & z \geq 0 \end{cases} \tag{3.8}$$



Left: a perceptron. **Right:** The gradient descent update rule if we have a single training input.

Figure 3.39: ADALINE for a single perceptron: 1

ADALINE is an acronym for **Adaptive Linear Element** as introduced by Widrow, and actually refers to just a regular perceptron: a weighted sum on inputs and bias passed through a thresholding function. However, ADALINE differs in the learning rule. Unlike the perceptron algorithm, which attempts to adjust the model parameters to minimize the error between the desired output d (in response to an input X) and the actual, thresholded output y, ADALINE minimizes the squared error between d and the *pre-activation* affine value z. Since the desired output is still binary, the error for a single input is:

$$Err(x) = \frac{1}{2}(d - z)^2 \tag{3.9}$$

$$\frac{dErr(x)}{dw_i} = -(d - z)x_i \tag{3.10}$$

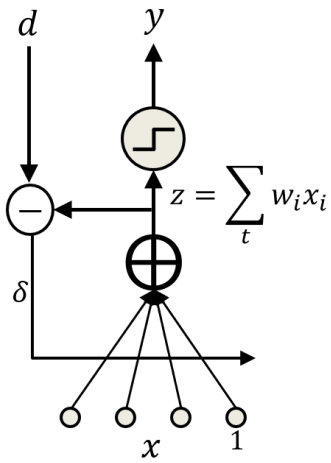
For a single training input, we can update the weights w_i through a *gradient descent* update rule as follows:

$$w_i = w_i + \eta(d - z)x_i \tag{3.11}$$

This is illustrated in the right panel in Fig. 3.39. We can modify this into an *online learning* rule, which updates the parameters after each input, as illustrated in 3.40. In this, for each input X that has target (binary) output d, we compute the error δ and update parameters as:

$$\delta = d - z \tag{3.12}$$
$$w_i = w_i + \eta\delta x_i \tag{3.13}$$

Observe the similarity to Rosenblatt's perceptron update rule: $w_i = w_i + \eta(d - y)x_i$. ADALINE differs from the Rosenblatt's update rule only in the fact that it considers the pre-activation value z rather than the post-activation output y.



Online learning rule for ADALINE.

Figure 3.40: ADALINE for a single perceptron: 2

Equation 3.13 is the widely used "delta rule," which is also called the "LMS update rule."

In fact both the perceptron learning rule and ADALINE are variants of the delta

rule. For the perceptron, the output used in the delta rule to estimate weights is y: $\delta = d - y$, and for ADALINE, the output used in the delta rule to estimate weights is z: $\delta = d - z$. In both, weights are re-estimated as:

$$w_i = w_i + \eta \delta x_i \tag{3.14}$$

The **generalized delta rule**, also widely known as the **Widrow-Hoff update rule**, is illustrated in Fig. 3.41. For any differentiable activation function this update rules is:

$$\begin{aligned} \delta &= d - y & (3.15)\\ w_i &= w_i + \eta \delta f'(z) x_i & (3.16) \end{aligned}$$

We will see later that this is exactly back-propagation in multi-layer nets if we let $f(z)$ represent the entire network between z and y. The generalized delta rule is possibly the most-used update rule in machine learning and signal processing. Variants of it appear in almost every problem. It must be immediately clear that the perceptron update rule is not an instance of the generalized delta rule, since $f'(z)$ is either 0 or undefined for the threshold activation. By the same token, ADALINE too is, oddly enough, not an instance of the generalized delta rule (if we consider $f(z)$ to be the output of the perceptron).

3.3.3 MADALINE

The ADALINE algorithm, like the perceptron update rule, gives us a learning rule to learn the parameters of a single perceptron. Like the perceptron rule, it needs the target output of the neuron for each training instance. As as result, like the perceptron rule, it cannot be directly applied to learn the parameters of MLPs, since the target output labels of the individual neurons will not be known for every training instance.

MADALINE (which stands for "Multiple ADALINE") is an extension of the ADALINE algorithm, which uses a greedy approach to assign greedily finds tar-

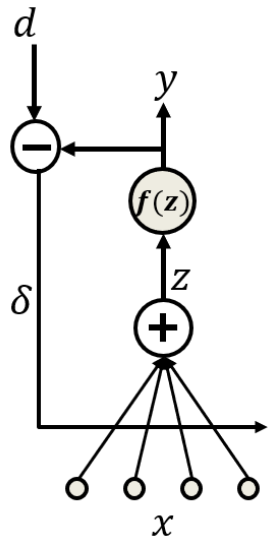


Figure 3.41: ADALINE for a single perceptron: 3

get outputs for individual neurons in the network. It uses the following approach:

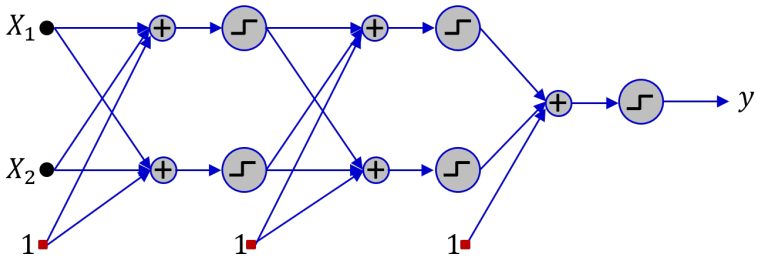
- It only updates network parameters on misclassified (training) instances.
- For each misclassified training instance, it finds the neuron which requires the smallest perturbation to change its output, such that changing the neuron's output changes the output of the network.
- It updates the parameters of this neuron using ADALINE.

Algorithm 2 outlines the algorithm.

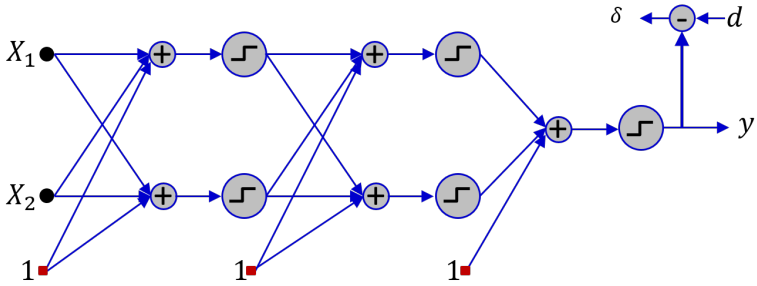
Algorithm 2: MADALINE algorithm. n represents a neuron, y_n represents the output of the neuron, z_n represents the pre-activation affine value for the neuron. All neurons are assume to output binary 0/1 values.

for *Each training instance* (X, d) **do**
  Evaluate $y = f(X)$;
  if *instance is misclassified, i.e.* $y \neq d$ **then**
    Tag all all neurons as untested ;
    while *untested neurons remain* **do**
      Find untested neuron n with smallest affine value z ;
      Mark n as tested;
      Flip the output of y_n of the neuron to $\hat{y}_n = 1 - y_n$;
      Re-evaluate the network with the output of n set to $\hat{y}_n$, to obtain the new network output $\hat{y}$;
      if $\hat{y} == d$*, i.e. flipping the neuron has corrected the misclassification* **then**
        Update the parameters of n using ADALINE, with the target output $\hat{y}_n$;
        break;
      end
    end
  end
end

The MADALINE algorithm is illustrated in Figures 3.42–3.43. In MADALINE, updates are performed on only when there is an error, i.e. when the output $y(X)$ of the network for any input differs X differs from the desired target output d for that input. Upon encountering a misclassified instance, we find the neuron n with smallest affine value z_n for that instance. We then flip the output y_n of the neuron n (from 1 to 0, if the current output is 1, or from 0 to 1 if it is currently 0) to obtain a flipped output $\hat{y}_n$. and recompute the network output using the flipped neuron output. If this results in a change in the overall output of the net, resulting in correct classification of the previously misclassified instance, then we surmise that the correct out of neuron n, when X is the input to the net, is $\hat{y}_n$. We then apply ADALINE updates to the neuron with target output $\hat{y}_n$.



(a) Classify an input.

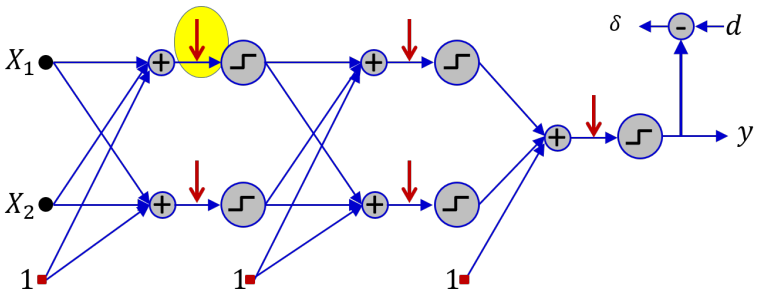


(b) Compute the error.

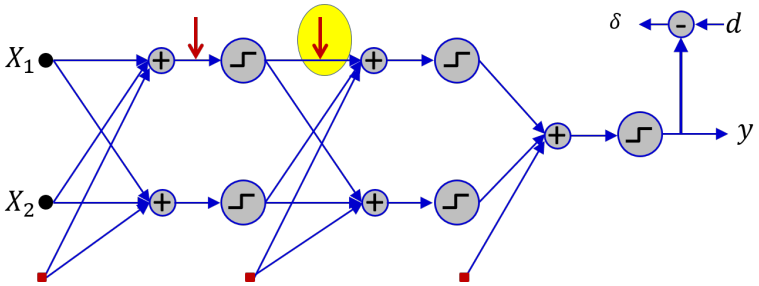
Figure 3.42: MADALINE for MLPs: 1

MADELINE terminates when a stopping criterion is met (an error that the MLP makes in classifying input data that is “acceptable”), or when no such “critical” neuron can be found for any input.

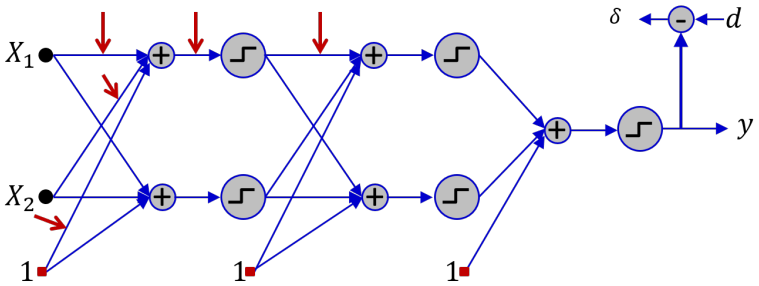
MADALINE is however a greedy algorithm, effective only for small networks. It



(a) If error, find the z that is closest to 0.



(b) Flip the output of corresponding unit and compute new output.



(c) If error reduces, set the desired output of the unit to the flipped value. Apply ADALINE rule to update weights of the unit.

Figure 3.43: MADALINE for MLPs: 2

is not very useful for large MLPs – it is too computationally expensive for them.

Recap 3.1

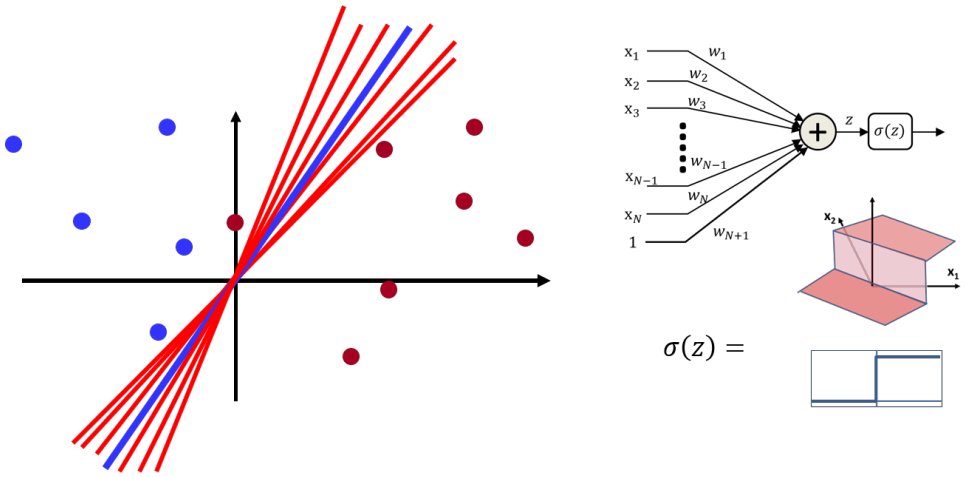
1. "Learning" a network is the same as learning the weights and biases to compute a target function
2. It requires a network with sufficient "capacity"
3. In practice, we learn networks by "fitting" them to match the input-output relation of "training" instances drawn from the target function
4. A linear decision boundary can be learned by a single perceptron (with a threshold-function activation) in linear time if classes are linearly separable
5. Non-linear decision boundaries require networks of perceptrons
6. Training an MLP with threshold-function activation perceptrons will require knowledge of the input-output relation for every training instance, for every perceptron in the network
   - These must be determined as part of training
   - For threshold activations, this is an NP-complete combinatorial optimization problem

3.3.4 The problem of non-differentiability

Once people discovered that training a multi-layer perceptron using the perceptron rule was an essentially intractable combinatorial optimization problem, all research on the topic stalled for over a decade. It was many years before the next breakthrough happened.

The reason for the intractability was that the perceptron with the threshold activation is a flat function with a zero derivative everywhere, except at 0 where it is non-differentiable. As a result, it is possible to change the weights by a lot

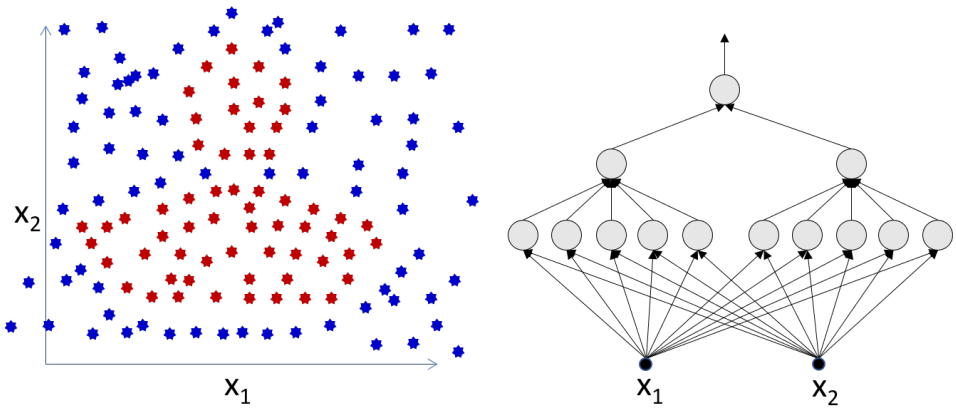
without changing the error. For example, in Fig. 3.44, if our current weights give us the boundary shown by the blue line, if we were to modify the weights of the perceptron such that the boundary changes to of the red lines, the error will remain unchanged. In this example, of course, the correct thing to do is to rotate that boundary left, until it crosses that misclassified red dot, but if we simply perturb the boundary somewhat while checking the error, we cannot tell whether we are perturbing it in the correct direction (such that continuing to perturb the boundary in the same direction will lead to reduced error) or not.



The perceptron is a flat function with zero derivative everywhere, except at 0 where it is non-differentiable. We can vary the weights a lot without changing the error. There is no indication of which direction to change the weights to reduce error.

Figure 3.44: Uncertainties in applying the perceptron training rule

This only compounds in larger problems in general, such as the one of classifying the data points shown in Fig. 3.45, where the decision boundary is more complex. In learning a classifier for this example, the individual neurons' weights can change significantly without changing the overall error (each line that defines the complex decision boundary can be rotated to some extent without changing the error, as in the example above). This happens because the overall MLP itself is a flat, non-differentiable function. The output is a constant 1 in some regions and a constant 0 in other regions, so that it has 0 derivative everywhere except at class boundaries, where the derivative is undefined.



Individual neurons' weights can change significantly without changing overall error The simple MLP is a flat, non-differentiable function – actually a function with 0 derivative nearly everywhere, and no derivatives at the boundaries.

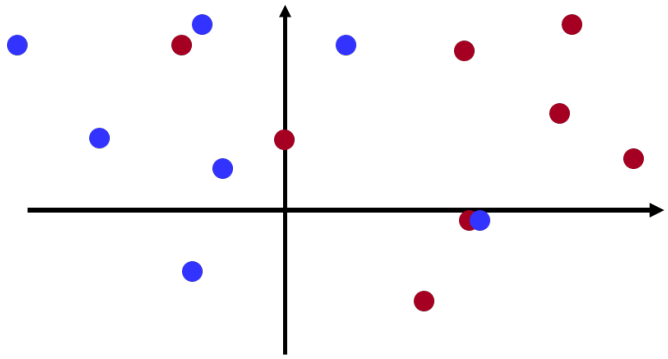
Figure 3.45: Compounding of training uncertainties for a complex problem

3.3.4.1 The problem of real world data

There is a second, even more pervasive problem in this context. In our examples we assumed that the classes are separable, and the training data are also actually separable. So we assumed that there are no red dots in the blue region and no blue dots in the red region, as in Fig. 3.45. At the level of the individual perceptrons, we assumed that the data would be linearly separable. Real-life data are rarely so clearly separable. For example, there could be red dots in the blue region and blue dots in the red region, as illustrated in Fig. 3.46. Rosenblatt's perceptron algorithm wouldn't even work in this situation, since it assumes that the data are linearly separable.

3.4 The differentiable activation function

The solution to the problems with ADALINE and MADALINE mentioned above was to make the neurons *differentiable*, with non-zero derivatives over much of the input space. To achieve this, instead of using the threshold activation function

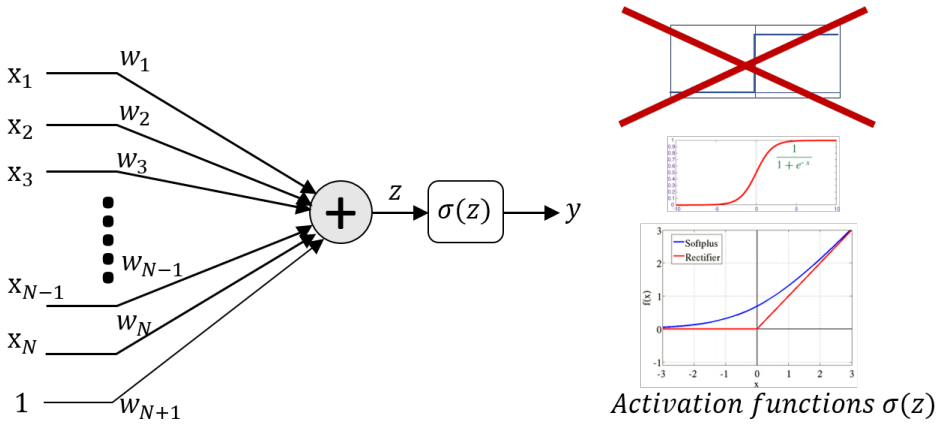


Real-life data are rarely clean and are not linearly separable. Rosenblatt's perceptron would not work in the first place.

Figure 3.46: Modeling uncertainties: wrong assumptions about training data

(the output of which is flat everywhere across the threshold), we use other activation functions that are continuously varying, such as those shown in Fig. 3.47. Doing so gives us a significant advantage: now when we change the weights even slightly, it results in a change in the output, even if the classification boundary itself does not cross a training instance. This allows us to evaluate whether a particular incremental change to the weights has moved them in a direction that, if continued, is likely to improve classifier accuracy, or whether it will make it worse. This in turn allows us to use *gradient descent* methods to train the network.

To understand this better, let us consider an example of 1-dimensional data, shown in Fig. 3.48. In this figure, the input to an activation function is a single scalar x. The target output, which is the class label, is either 0 or 1. The blue dots are the 0 class and the red dots are the 1 class. We can plot all the data in two dimensions, as shown in the figure. In Fig. 3.48(a), we compare two different threshold activations. In the left panel, the threshold is T_1. In the right panel, it is T_2. In both cases we have one red dot that is misclassified, and so the error is 1. However, let us assume that we had an initial guess of T_1 for the threshold, and then adjusted it to T_2. By simply counting the error, we cannot tell if we have shifted the threshold in the correct direction or not – the error is still 1 in both cases. The error does not inform us that if we continue shifting the threshold towards the left, we will eventually correctly classify the misclassified red dot



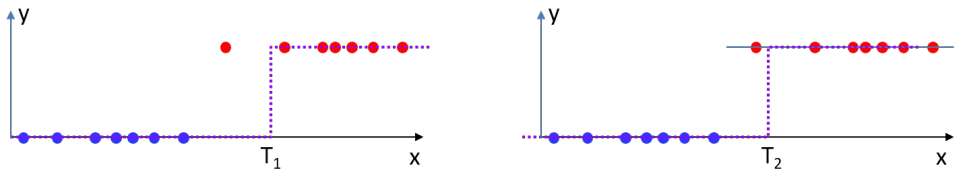
Using differentiable functions such as the two shown on the bottom right gives us the advantage that we have non-zero derivatives over much of the input space. Small changes in weight can therefore result in non-negligible changes in output. This enables us to estimate the parameters using gradient descent techniques.

Figure 3.47: Using differentiable activation functions

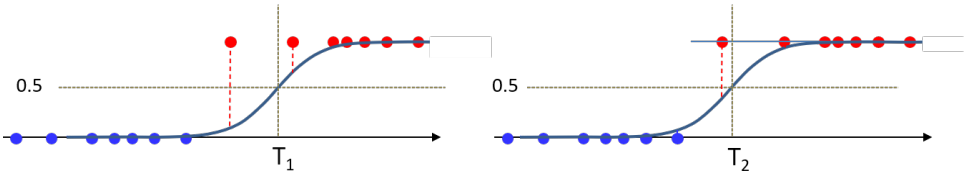
shown, and will be able to improve the classifier.

On the other hand, in Fig. 3.48(b), we have a different kind of activation, shown by the continuously varying curve which changes smoothly from 0 to 1 from the left to right. For classification with this function, let us assign a class of 1 if the activation value is greater than 0.5 and 0 if it is less than 0.5. In the right panel the cross-over from class 0 to class 1 happens when the activation value goes from being less than 0.5 to greater than 0.5 (this occurs at $x = T_1$). In the right panel we have shifted the curve left and the crossover occurs at T_2. So again, in both cases, the number of classification errors is 1.

But now, instead of simply counting whether an instance has been classified as class 0 or 1 correctly, let us consider *how much* the target value deviates from the curve that represents the function. This is given by the lengths of the dotted lines shown. We can now clearly see that in the panel to the right the total length is much smaller than in the panel to the left. In fact we will be able to compute/infer that the total length decreased as the curve was shifted left: from the classification boundary of T_1 to T_2. Thus we would know that continuing to shift it left is a good move that will improve the classification performance (reduce the classification



(a) **Threshold activation:** shifting the threshold from T_1 to T_2 does not change the classification error. As a result we do not know whether moving the threshold left was good or not.



(b) **Smooth, continuously varying activation:** Classification based on whether the output is greater than 0.5 or less. We can now quantify how much the output differs from the desired target value (0 or 1). Moving the function left or right changes this quantity, even if the classification error itself does not change.

Figure 3.48: The advantage of using a differentiable activation function

error).

In terms of derivatives, we would find that the derivative of this difference with respect to a shift of the threshold is positive, i.e., if we shift the threshold left, the difference (in the respective deviations from the fuction curve) *decreases*. This is a consequence of the fact that the activation function itself has non-zero derivative. At any x, we can compute whether the deviation of the miscalssified instances from the function decreases or increases if we shift the function in any direction.

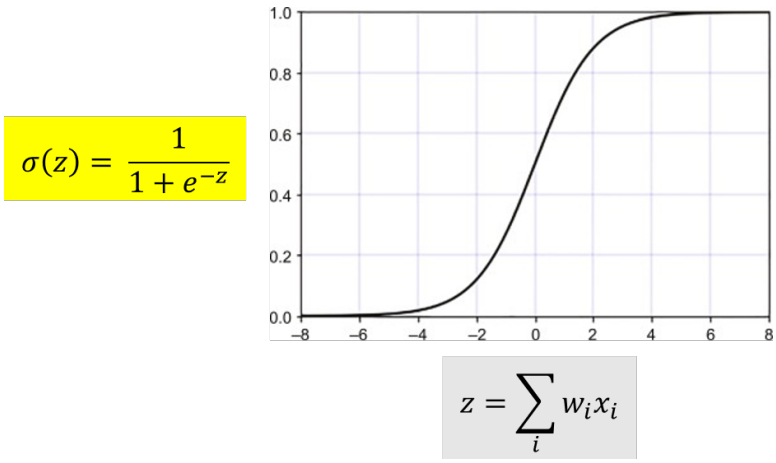
3.4.1 The special nature and significance of the Sigmoid activation function

There are many different continuously-varying activation functions that we can use instead of the threshold activation function, all of which have the essential

property of having non-zero derivatives. However, of all of these, one function stands out in particular as being especially interpretable: the **sigmoid** activation function. It is given by:

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{3.17}$$

As the affine input z varies from $-\infty$ to ∞, the sigmoid function varies smoothly from 0 to 1. This is easy to understand: if z is $-\infty$ then e^{-z} is ∞, and $1/\infty$ is 0. If z is ∞, then e^{-z} is 0 and the denominator is 1, so the function takes the value 1. Between these two extremum values, the variation of function values is continuous. The shape of the function is the curve shown in Fig. 3.49.



This sigmoid function can also be interpreted as the *probability* function $P(y = 1|x)$.

Figure 3.49: The sigmoid activation function as a probability function

The sigmoid has a very useful interpretation: the output of the sigmoid activation can be viewed as the *a posteriori probability* that the class of the input is 1, i.e. the probability $P(y = 1|x)$. When this is low, we can assume that the input belongs to class 0. If it is high, we can assume that the class is 1.

3.4.1.1 The behavior of the sigmoid function

The figures that follow give more of intuition into the behavior of the sigmoid function function. Let us take the example of linearly *in*separable input data. Fig. 3.50 shows a two-dimensional example of such data. In the figure, there are blue dots on the "red" side and red dots on the "blue" side of an unclear boundary between the two. The boundary is not linear – no line will cleanly separate the two colors.

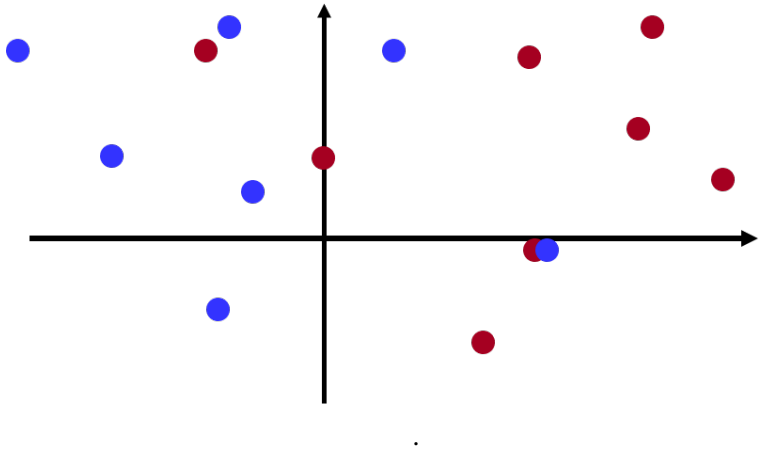


Figure 3.50: Non-linearly separable data: 2-D example

Fig. 3.51 is a similar one-dimensional example. All (red) dots at $Y = 1$ represent instances of class 1. All (blue) dots at $Y = 0$ are from class 0. The data are not linearly separable. In this 1-D example, a linear separator is a threshold, and we see that no threshold will cleanly separate red and blue dots.

Let us view this example differently. At each point x, we consider a small window around that point, as shown in Fig. 3.52, and plot the average value of y for all the instances within the window. This is an *approximation* of the probability of y being 1 ($P(y = 1|x)$) at that point.

We perform this computation at all points in the training data. A couple of intermediate stages of this process are shown in Figs. 3.53(a), (b) and (b).

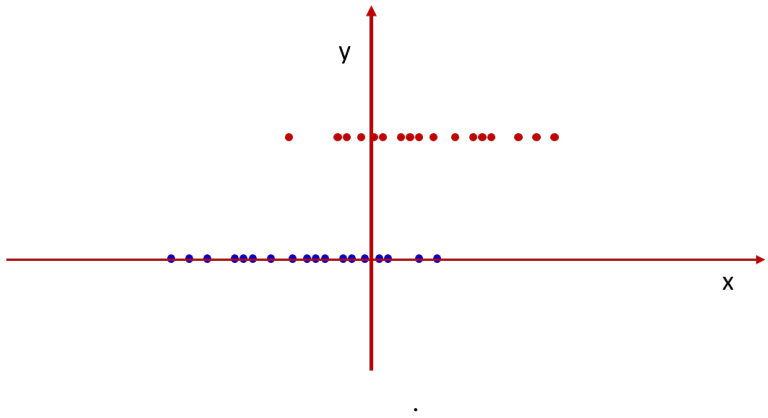
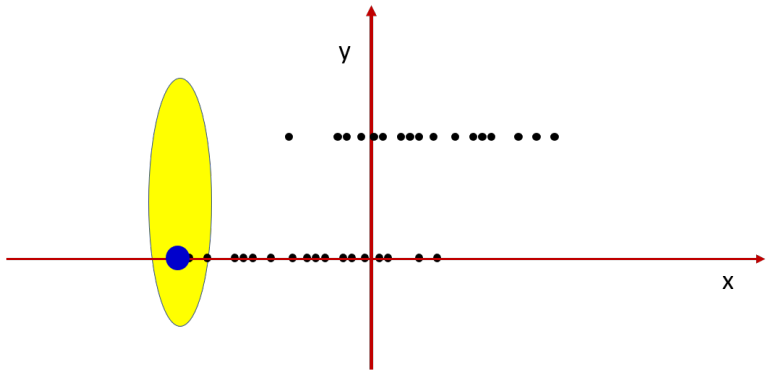


Figure 3.51: Non-linearly separable data: 1-D example



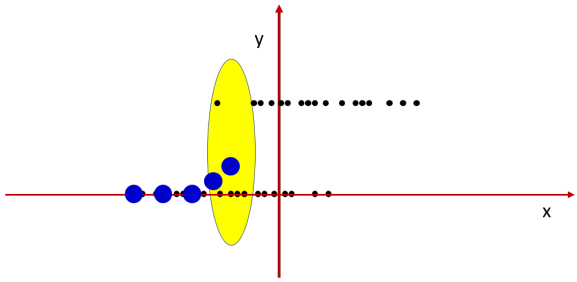
The average value within the elliptical window shown is the probability $P(y = 1|x)$.

Figure 3.52: The *a posteriori* probability of class 1 for a data point

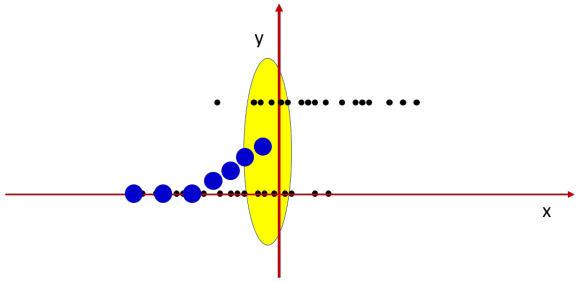
This process of successive averaging gives us the function illustrated in Fig. 3.54(a). It is well modeled by the sigmoid function shown for reference in Fig. (b), justifying the claim we made about sigmoid activations earlier.

The sigmoid function for bivariate inputs (2-D inputs) is shown in Fig. 3.55. *This is, in fact, a perceptron with a sigmoid activation* and actually computes the probability that the input belongs to class 1.

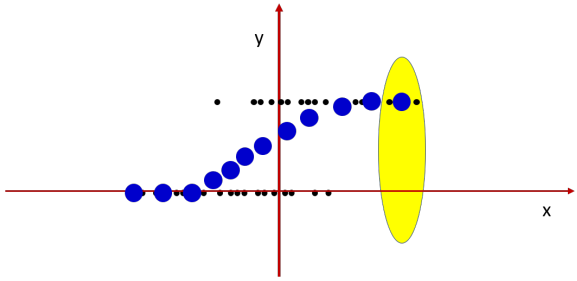
We will return to the fact that perceptrons with sigmoidal activations actually model class probabilities later in this book.



(a) The elliptical window is successively applied to each data point on the x-axis, and the y values of all points within it are averaged. Each average value is represented by the corresponding large blue dot within the ellipse.

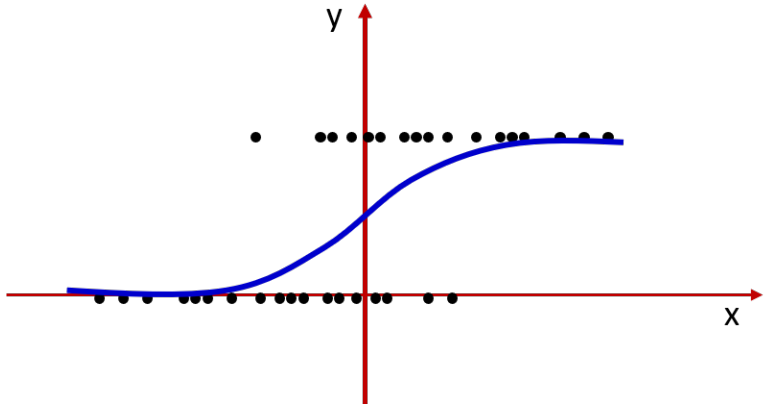


(b) Another intermediate stage in the process.

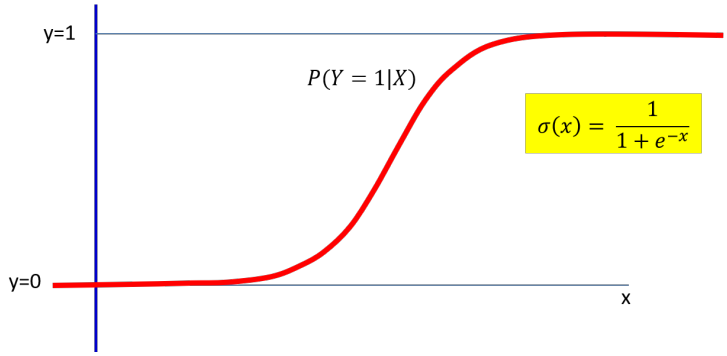


Another intermediate stage in the process..

Figure 3.53: Stages in the process of averaging data points within a window

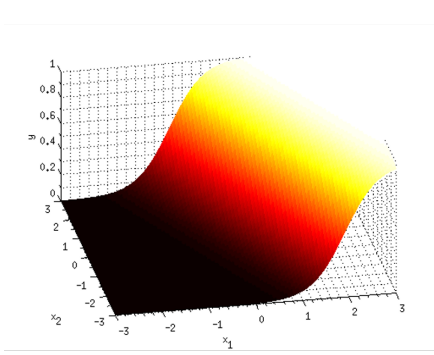


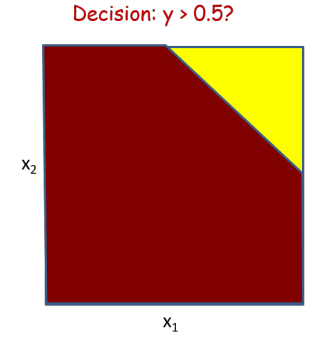
(a) The *a posteriori* probability of $y = 1$ varies smoothly and monotonically from 0 to 1 as shown by the blue curve.



(b) The *sigmoid* function, also known as a logistic regression

Figure 3.54: The logistic regression model





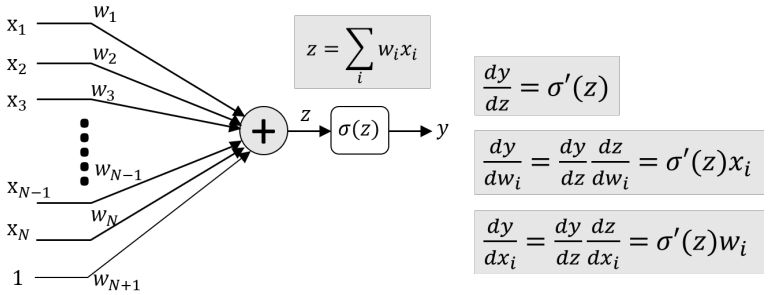
$$P(Y = 1|X) = \frac{1}{1 + \exp\left(-\sum_i w_i x_i - b\right)}$$

Sigmoid activation over two variables

Figure 3.55: Logistic regression over two variables

For now, consider the great significance of the fact that the activation $\sigma(z)$ is a differentiable function of z, with a derivative $\sigma'(z)$ that is well-defined almost everywhere, has non-zero values, and is bounded.

As a result, the output of the neuron, y, is differentiable w.r.t. the affine combination z. Since z is simply the sum of the products of weights and inputs, using the chain rule, y is also differentiable with respect to both the weights w_i of the neuron, and the inputs x_i to the neuron. This means that we can compute the change in the output for small changes in either the input or the weights.

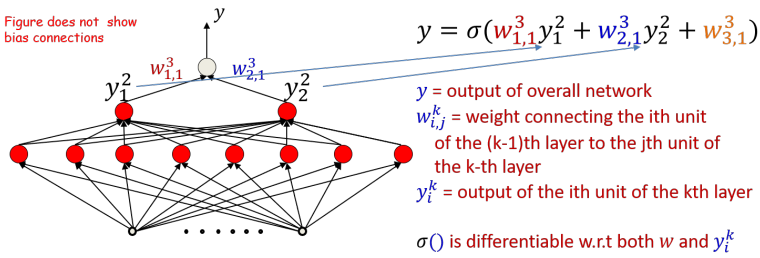


y is differentiable with respect to the affine combination z, the weights w_i and the inputs x_i. Changes in the output y can be computed for the smallest changes in $\mathbf{W}$ or X.

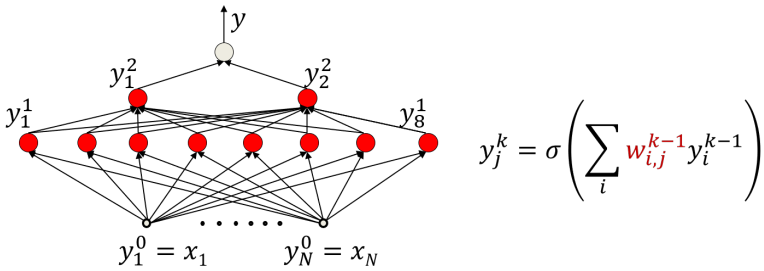
Figure 3.56: Perceptron with differentiable activation function

In an MLP that uses differentiable activations, each perceptron is differentiable w.r.t. each of its weights and biases, and its inputs. This also means is that the *overall network is differentiable*, and we can explicitly compute how the output of the MLP changes for small perturbations of its input and weights.

For instance, consider the MLP, shown in Fig. 3.57. Small changes in the weight of the any connection into a lower-layer neuron (first layer), e.g. the first neuron (shown to the far left in Fig. 3.57(a)) will result in small changes to its output y_1^1. This in turn will result in perturbations of the outputs of the two neurons in the next layer, namely y_1^2 and y_2^2, and these will cumulatively cause perturbations in the network output y. We can hence now explicitly compute how changing any weight will change y by computing the derivative of the output of the network w.r.t. it. This holds for every parameter in the network. The overall function



(a) Every individual perceptron is differentiable w.r.t its inputs and its weights (including "bias" weight). By the chain rule, the overall function is differentiable w.r.t every parameter (weight or bias). Small changes in the parameters result in measurable changes in output.



(b) The overall function is differentiable w.r.t every parameter. We can compute how small changes in the parameters change the output. For non-threshold activations the derivatives are finite and generally non-zero.

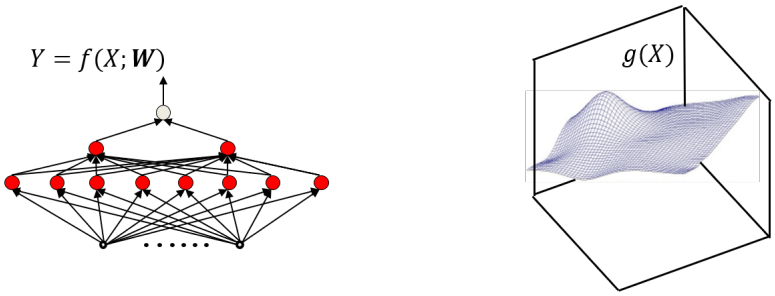
Figure 3.57: The overall network is differentiable

represented by the network is differentiable w.r.t. every parameter in the network, using the chain rule. Also, in the case of non-thresholding activation functions, the derivatives are generally non-zero for most inputs. We will derive the actual derivatives using the chain rule later. Based on the derivatives, we can also find out how to adjust a weight to modify the output of the network in a specific manner (e.g. increase or decrease it),

Note the slight modification in the notation we have introduced here: y is the output of the network, w^k_i, j is the weight of the connection from the i^{th} neuron in the $k-1^{th}$ layer to the j^{th} neuron of the k^{th} layer. y^k_i is the output of the i^{th} neuron in the k^{th} layer. We will use this notation in other instances going forward.

3.4.2 Learning an MLP with differentiable activations

Having set up the network as a differentiable function, let us now return to the problem of *learning* a network to model a function: given a target function $g(X)$, we must learn a network to model it. The target function itself may represent any classification, prediction or regression task. We continue to assume that the architecture of the network is given, and we must only learn its parameters (as illustrated in Fig. 3.58).



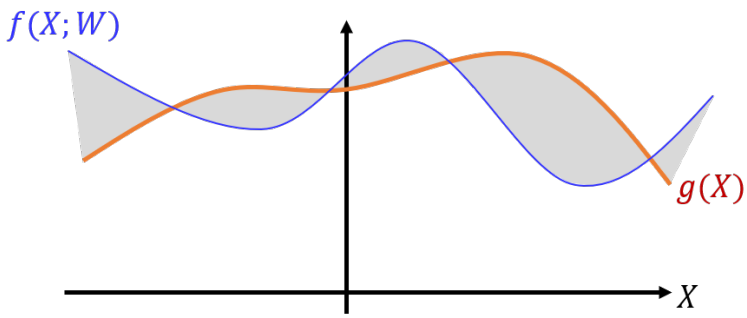
The parameters of the MLP shown must be learned for it to model the function on the right accurately.

Figure 3.58: Learning an MLP to model a function (recalled)

The problem, specifically, is that of learning the network parameters such that the function $f(X; \mathbf{W})$ embodied by the network matches the target function $g(X)$ as closely as possible.

Recall that we set up this problem as that of learning network parameters such that the area (or volume) of the error between the two is minimal. This is illustrated again by Figure **??** – our objective now is to estimate network parameters $\mathbf{W}$ such that the volume of the shaded region, which reflects the discrepancy between $f(X; \mathbf{W})$ and $g(X)$, is minimized.

Recall again that as a first step, to quantify this discrepancy, we define a divergence function $div(f(X; \mathbf{W}), g(X))$ between the target function $g(X)$ and the network output $f(X; \mathbf{W})$ for any input X. The divergence is a non-negative function that goes to 0 when $f(X; \mathbf{W}) = g(X)$ and gives us a measure of the discrepancy between the two. The volume of discrepancy (shaded area of Fig-



This is a repetition of Fig. 3.16, for reference. The shaded area quantifies the discrepancy between $f(X;\mathbf{W})$ and $g(X)$.

Figure 3.59: The expected divergence as an area

ure **??**) is obtained by integrating the divergence over the entire input space. We estimate the network parameters to minimize this integral:

$$\hat{\mathbf{W}} = \underset{\mathbf{W}}{\operatorname{argmin}} \int_X div\left(f(X;\mathbf{W}), g(X)\right) dX \tag{3.18}$$

The minimization of error can be done in a more conservative fashion. In general, since we are only guaranteed to be approximating the true function and not modeling it exactly through this process, we can just focus on the input values of X that really matter. To understand what this means, consider that not all inputs are even possible – for example, infinite valued inputs never happen, or (as is common in some settings) inputs can never be negative. Attempting to minimize the error at values of X that we will never see is not particularly useful. We can identify the X values that "matter" in many ways, but the most effective strategy is based on the fact that in general, some values of X are more frequent than others, and so it is more important to minimize the error at more frequent values of X than at less frequent ones. Therefore, instead of simply minimizing the error, we minimize the *weighted* error, where the weight given to the error at any X is the probability of obtaining that input X.

So, more generally, assuming X is a random variable, the modified discrepancy measure is now:

$$E\left[div(f(X;\mathbf{W}),g(X))\right] = \int_X div(f(X;\mathbf{W}),g(X))P(X)dX \tag{3.19}$$

The integral in Eq. 3.19 is the statistical expectation, or the *expected value* of the divergence (recall that for any $f(X)$, $\int_X f(X)P(X)dX$, where $P(X)$ is the probability of X, is simply the expected value of $f(X)$). It represents the average value of $f(X;\mathbf{W})$ over all possible values of X that we can obtain. It is (an instance of) what is sometimes referred to as the *risk* – the penalty of using $f(X;\mathbf{W})$ instead of $g(X)$.

Thus, in order to estimate the parameters of the network shown in Fig. 3.58, we wish find the $\mathbf{W}$ that minimizes the risk, i.e. *expected divergence* between the output of the network and the target function we want it to model.

$$\hat{\mathbf{W}} = \underset{\mathbf{W}}{\operatorname{argmin}} \int_X div(f(X;\mathbf{W}),g(X))P(X)dX \tag{3.20}$$

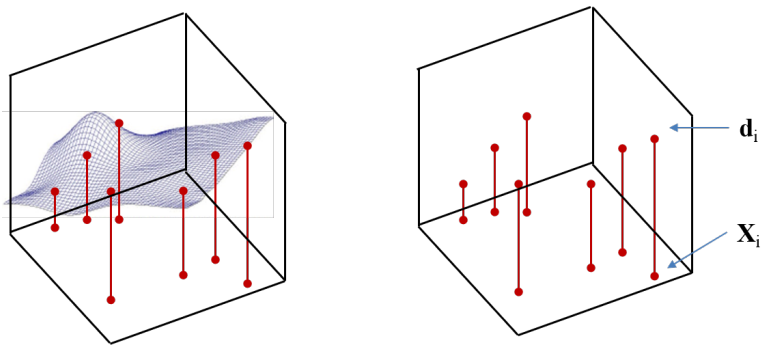
$$= \underset{\mathbf{W}}{\operatorname{argmin}}\, E\left[div(f(X;\mathbf{W}),g(X)\right] \tag{3.21}$$

This is our *true* objective.

However, as we explained earlier, in reality we cannot really compute the expectation in Eq. 3.19 because we do not have a complete specification of the target function $g(X)$ everywhere – this is a function we are learning after all. So this optimization cannot be performed.

We recall that our solution was to *sample* $g(X)$ – collect a number of *snapshots* of the function, where in each snapshot we record an input X_i and the corresponding value of the function $g(X_i)$, as shown in Fig. 3.60. These are our "training data", or training samples. We will learn our model to "fit" the input-output relationship at these samples, instead. We will try to ensure that for every input X_i in the training data, the network accurately computes the output as $g(X_i)$, i.e. that $f(X_i;\mathbf{W}) \approx g(X_i)$. A good sampling strategy is to draw more samples of input that are naturally more probable – to draw the input samples of X from its probability distribution $P(X)$. That way, when we estimate the function we will make the estimate more accurate for input values that are more likely to be seen,

and not focus on inputs that are unlikely or impossible to happen anyway.



A schematic representing the sampling of $g(X)$. We obtain input-output pairs for a number of samples of input X_i. In a good sampling scheme, the samples of X will be drawn from $P(X)$. We then estimate the desired function from the samples.

Figure 3.60: Sampling a function for learning

Note that this is only an approximation to our true objective (Equation 3.20). Let us now formulate this approximation.

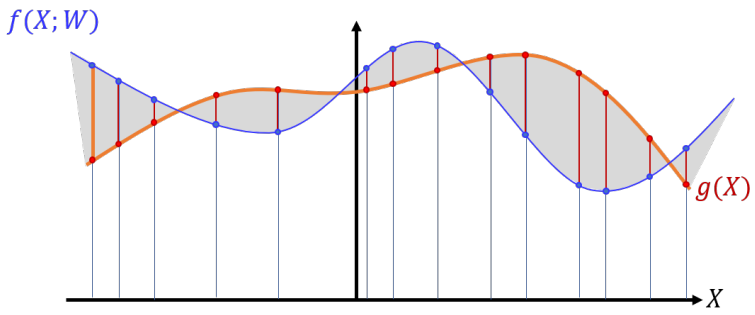
3.4.2.1 The empirical risk

As an approximation for the *expected* divergence between $f(X;\mathbf{W})$ and $g(X)$, we will use the *average* divergence computed over the set of training instances, as illustrated by Figure 3.61, and estimate $\mathbf{W}$ to minimize that term instead.

$$E\left[div(f(X;\mathbf{W}),g(X)\right] \approx \frac{1}{N}\sum_{i=1}^{N} div(f(X_i;\mathbf{W}),d_i) \tag{3.22}$$

The right hand side of Equation 3.22 is the *empirical estimate* of the expected divergence over the samples, or the *empirical risk*. If we sample the training instances of X from the actual distribution of $P(X)$, then this will be an unbiased estimate of the expected risk. This means that the expected value of the empirical risk is the actual expected risk, as we see below.

The expected value of the empirical risk is:



The shaded area is approximated as the average of the lengths of the red lines. Each line represents a training instance for which X and $g(X)$ are both known. The divergence quantifies the lengths of these lines.

Figure 3.61: The empirical estimate of the average divergence

$$E[\frac{1}{N}\sum div(f(X,\mathbf{W}),g(X))] = \frac{1}{N}\sum E[div(f(X;\mathbf{W}),g(X)] \quad (3.23)$$

where the summation is over N training samples. Since the $E[div(f(X;\mathbf{W}),g(X)]$ is, by definition, not a function of X, we have

$$\begin{aligned}\frac{1}{N}\sum E[div(f(X;\mathbf{W}),g(X)] &= \tfrac{1}{N}N\ E[div(f(X;\mathbf{W}),g(X)] \\ &= E[div(f(X;\mathbf{W}),g(X)] \end{aligned} \quad (3.24)$$

which is the true expected risk.

This implies is that if we were to repeat the following experiment many times:

- draw a fresh batch of training samples, and
- compute the empirical risk in Eq. 3.22 from the batch,

then the *average* of the empirical risk values computed across all attempts will be approximately the same as the true risk.

This gives us a mathematical justification for minimizing the empirical estimate of risk, instead of the actual risk. Our hope is that if we minimize that, we will also be minimizing Eq. 3.20. We will refer to the empirical risk as the

loss and denote it by $Loss(\mathbf{W})$, where the notation $Loss(\mathbf{W})$, which only shows dependence on $\mathbf{W}$ reflects the fact that the empirical the loss is a function of $\mathbf{W}$.

The empirical risk minimization that we actually perform in the context of an MLP is thus described as follows:

- We obtain a training set of input-output pairs $(X_1, d_1), (X_2, d_2), \ldots, (X_N, d_N)$, where $d_i = g(X_i)$ is the *desired output* of the network in response to input X_i;
- For each instance X_i, we define a divergence $div(f(X_i; \mathbf{W}), d_i)$ between the desired output d_i and the actual network output $f(X_i; \mathbf{W})$.
- We define our loss (empirical risk) as

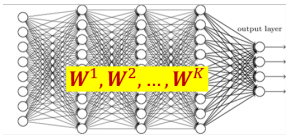
$$Loss(\mathbf{W}) = \frac{1}{N} \sum_i div(f(X_i; \mathbf{W}), d_i) \tag{3.25}$$

- We estimate the network parameters to minimize the empirical estimate of expected divergence (empiricial risk) as follows:

$$\widehat{\mathbf{W}} = \underset{\mathbf{W}}{\operatorname{argmin}}\, Loss(\mathbf{W}) \tag{3.26}$$

ERM for neural networks is summarized in Fig. 3.62. The exact form of $div()$ will be discussed later. To do this, we optimize the network parameters to minimize the total error over all training inputs.

Thus our final solution for learning the neural network is this: We are given a training set of input-output pairs $(X_1, d_1), (X_2, d_2), \ldots, (X_N, d_N)$. We construct our loss function as the average divergence over the training set. For a given training set, the loss is actually a function of the network parameters $\mathbf{W}$. We then minimize the loss function w.r.t. its argument $\mathbf{W}$. This is an instance of function minimization, or more generally, of function optimization.



Actual output of network:

$$Y_i = net(X_i; \{w_{i,j}^k \forall i, j, k\}) = net(X_i; W^1, W^2, \ldots, W^K)$$

Desired output of network: d_i

Error on i-th training input: $Div(Y_i, d_i; W^1, W^2, \ldots, W^K)$

Average training error(loss):

$$Loss(W^1, W^2, \ldots, W^K) = \frac{1}{N}\sum_{i=1}^{N} Div(Y_i, d_i; W^1, W^2, \ldots, W^K)$$

.

Figure 3.62: Empirical risk minimization (ERM) for neural networks

Recap 4.1

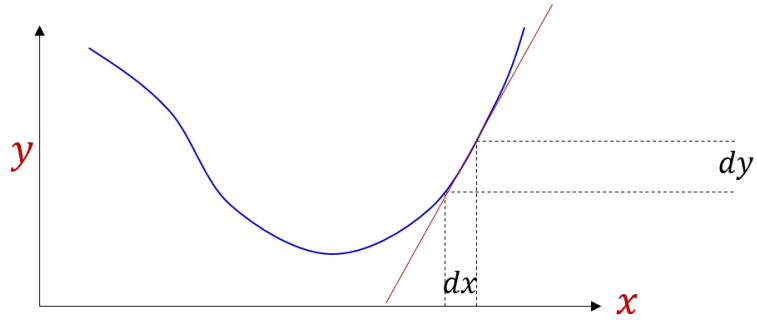
1. We learn networks by "fitting" them to training instances drawn from a target function.
2. Learning networks of threshold-activation perceptrons requires solving a hard combinatorial-optimization problem, because we cannot compute the influence of small changes to the parameters on the overall error.
3. Instead we use continuous activation functions with non-zero derivatives to enables us to estimate network parameters
   - This makes the output of the network differentiable w.r.t every parameter in the network
   - The logistic activation perceptron actually computes the a posteriori probability of the output given the input
4. We define differentiable divergence between the output of the network and the desired output for the training instances, and a total error, which is the average divergence over all training instances.
5. We optimize network parameters to minimize this error (Empirical risk minimization).
6. This is an instance of function optimization

3.5 A crash course on function optimization

Before we proceed, let us review some key concepts in function optimization.

3.5.1 A note on derivatives

As a first step, let us begin by understanding what a derivative of a function really means. We will do this with the help of Fig. 3.63. The derivative of a function at any point tells us how much a minute increment to the argument of the function will increment the value of the function.



A derivative of a function at any point tells us how much a very small increment to the argument of the function will increment the value of the function. For any $y = f(x)$, the derivative is expressed as a multiplier α to a tiny increment Δx to obtain the increments dy to the output. Thus $\Delta y = \alpha dx$. This is based on the fact that at a fine enough resolution, any smooth, continuous function can be considered to be locally linear at any point.

Figure 3.63: Relating increments in x and y through the derivative

For any function $Y = f(X)$, the derivative is a multiplicative factor by which we have to multiply the small increment in X (denoted by dX, to obtain the corresponding increment in Y. Thus if α is the derivative at any point, it means that the increment dY in $f(X)$ is given by $dY = \alpha dX$ (we should ideally write this as $\alpha(X)$, since the expression is a function of X). This relation is based on the fact that at a fine enough resolution, any smooth, continuous function can be

considerd to be locally linear at any point. Figure 3.63 illustrates this relationship for a scalar function of a scalar variable ($y = f(x)$).

It is also important to note here that the only generically correct way to write the derivative α is as $dY = \alpha dX$. The popular alternative $\alpha = dY/dX$ is not generalizable – it does not work when X is a vector. In the discussion below we will consider the situation where the value Y computed by the function is a scalar, as that is most relevant to the discussion here. Extension to vector Y will be discussed in the next chapter.

3.5.1.1 Scalar function of a scalar argument

For a scalar function of a scalar variable, y and x are scalar. $y = f(x)$ and its derivative $dy = \alpha dx$, often represented as dy/dx. To reiterate, the derivative α is the value by which we must multiply the increment dx to get the corresponding increment dy. In the scalar case, it is reasonable to represent the derivative as dy/dx. But the notation $f'(x)$ is more reasonable, and we will use this often in this book.

3.5.1.2 Multivariate scalar function: scalar function of a vector argument

An example of a multivariate scalar function is given in Fig. 3.64.

When the argument to the function is a vector, then X is a vector and dX is also a vector. Once again,the derivative α is the multiplier α that we multiply dX by to compute the increment dy:

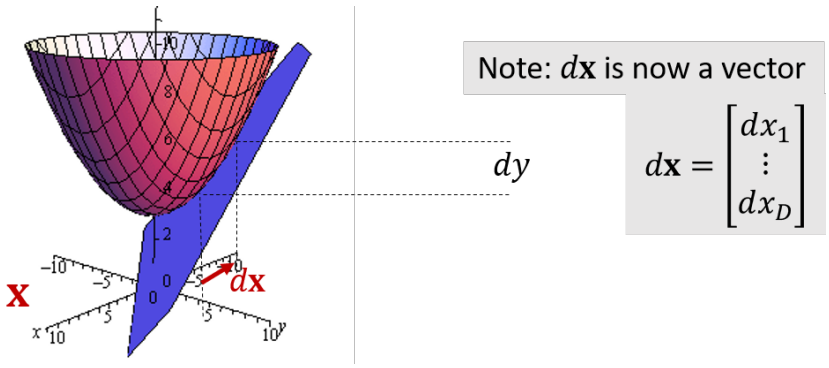
$$dy = \alpha dX \tag{3.27}$$

If X is a column vector $[x_1\ x_2\ \ldots\ x_i\ \ldots\ x_D]^\top$, then dX, which a vector of increments to X, is also a column vector $[dx_1\ dx_2\ \ldots\ dx_i\ \ldots\ dx_D]^\top$. Since dy is scalar, this that implies α is a row vector $\alpha = [\alpha_1, \alpha_2, \ldots, \alpha_D]$, and

$$dy = \alpha_1 dx_1 + \alpha_2 dx_2 + \ldots + \alpha_D dX \tag{3.28}$$

Each of the individual α components, α_i, tells us how much y increments when only x_i is incremented and all the other dX terms are 0 ($X = [0\ 0\ \cdots\ dx_i\ 0\ 0\ \cdots 0]^\top$). This value is termed the *partial* derivative of y w.r.t. x_i and is represented as $\alpha_i = \partial y / \partial x_i$. We obtain the overall increment in y as:

$$dy = \frac{\partial y}{\partial x_1} dx_1 + \frac{\partial y}{\partial x_2} dx_2, \ldots, \frac{\partial y}{\partial x_D} dx_D \tag{3.29}$$



.

Figure 3.64: A multivariate scalar function

Thus, we can write the derivative equation for a multivariate scalar function as

$$dy = \nabla_X y dX \tag{3.30}$$

where

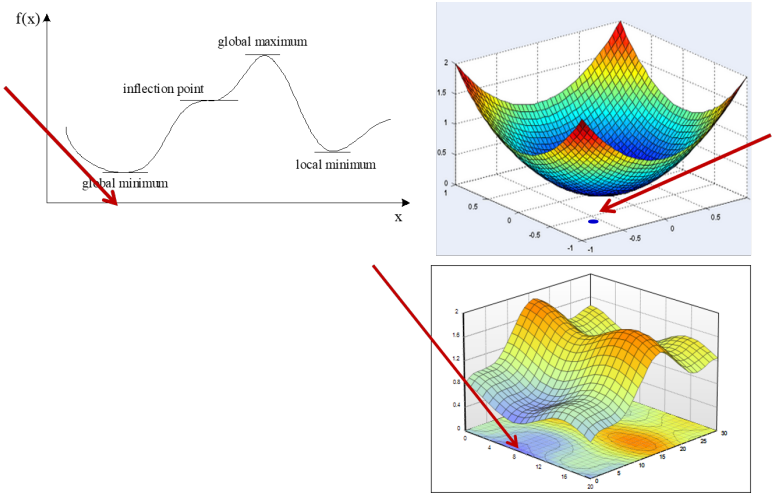
$$\nabla_X y = \left[\frac{\partial y}{\partial x_1}, \frac{\partial y}{\partial x_2}, \ldots, \frac{\partial y}{\partial x_D}\right] \tag{3.31}$$

Here, instead of α, we represent the derivative by the inverted triangle ∇. The notation $\nabla_X y$ stands for the derivative of y w.r.t the argument X. We will be using this symbol for vector and matrix derivatives henceforth. Of course, the final derivative is the row vector in Eq. 3.31. The widely used term "gradient" in this context is actually defined as the transpose of the derivative.

3.5.1.3 From derivatives to optimization

We now discuss the subject of optimizing a function w.r.t a variable x. Note that what follows applies to the *general* problem of optimizing a function of any variable – which we call the variable x in this case. In our actual network optimization problems, we would be optimizing w.r.t. the network weights $\mathbf{W}$ and the discussion below would apply to that case. Note especially that for the purpose of this discussion only, x does *not* denote the input to a neural network, but rather it represents any variable that we are optimizing a function over.

The general problem of optimization in our setting is illustrated in Fig. 3.65: We are given a function $f(x)$ of a variable x. x could be a scalar, like in the example on the left, or a vector as in the panels to the right (where x is a 2-dimensional vector). Our problem is to find the value of x where $f(x)$ is minimum, i.e. the locations shown by the arrows in the figure.



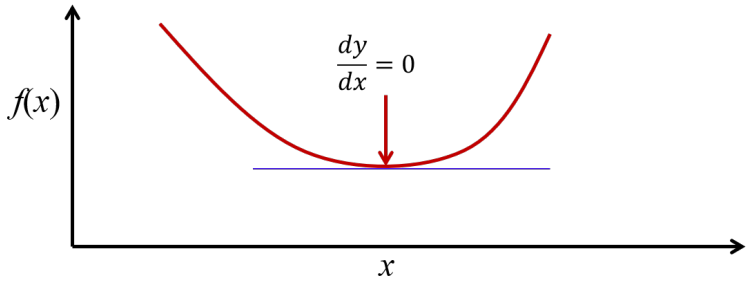
Finding the minimum of a function over x, which is a scalar in the panel to the left, and a 2-D vector in the panels to the right. The red arrows point to the value of x where $f(x)$ is minimum.

Figure 3.65: Illustrating the problem of optimizing a function over a variable

Fig. 3.66 illustrates how this can be done, with a simple example. We recall from elementary calulus that for any function of a scalar, the derivative is generally 0 at the location of a minimum. A minimum is also a "turning point" – the function

value is lowest at the minimum, but is higher all around it. Thus no matter which direction we approach the minimum from, the function first decreases, reaches a minimum, and then begins to increase, i.e. it "turns" at the minimum. At the minimum itself we expect the function to be locally flat and its derivative to be 0.

Thus to identify the minimum, we must find values of x at which the derivative is 0, for which we solve $\frac{df(x)}{dx} = 0$ for x. However, a zero derivative alone is not sufficient to ensure that it is a minimum.

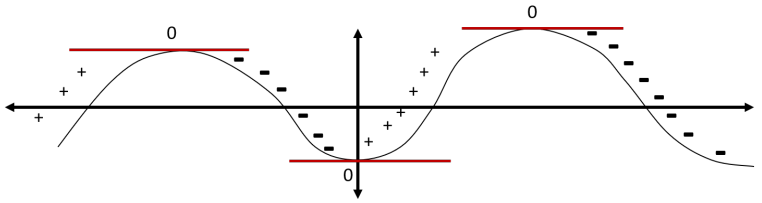


To find the value x at which $f'(x) = 0$, we solve $(d(f(x))/dx = 0$. The solution is a "turning point". Derivatives go from positive to negative or vice versa at this point. This is however not sufficient to determine if it is a minimum point. For it to be a minimum, the second derivative must be positive.

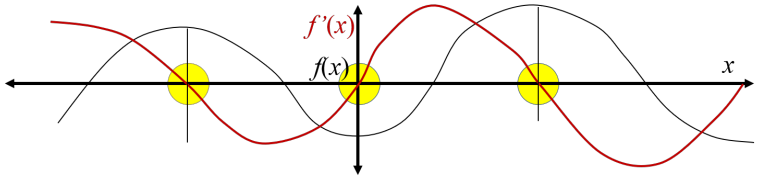
Figure 3.66: Finding the point where a function is minimum

To understand why, we must revisit the properties of inflexion points of functions. Both maxima and minima are turning points, as shown in the example in Fig. 3.67. In the case of maxima, as we approach the maximum along x from the left, the function initially increases, and has a positive slope, or derivative. At the maximum it flattens out and the derivative goes to 0. Once we cross the maximum the function begins to decrease, with negative derivative. For minima, as we approach it from the left along x, the function initially decreases, with negative derivative, then flattens out at the minimum, and then begins to increase with positive derivative. At both maxima and minima, the function has zero derivative.

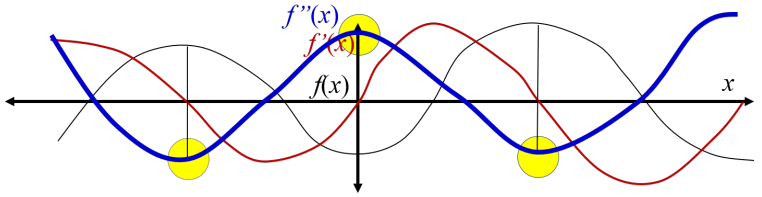
The derivative of the function shown in Fig. 3.67(a) is plotted in Fig. 3.67(b). At the maximum, the derivative goes from being positive to negative, and at the



(a) Both maxima *and* minima have zero derivative. Both are turning points.



(b) Both maxima and minima are turning points. Both maxima and minima have zero derivative.



(c) The second derivative $f''(x)$ is negative at maxima and positive at minima.

Figure 3.67: Behavior of the derivative around inflexion points

minimum it goes from being negative to positive.

The *derivative of the derivative* of the function is shown in Fig. 3.67(c). For the derivative curve, at maxima the slope goes from being positive to being negative, so the slope of *its* derivative is negative. At minima the derivative goes from being negative to being positive, and the slope of *its* derivative is positive. Thus the second derivative $f''(x)$ is -ive at maxima and +ve at minima.

The solution to finding the minimum or maximum of a function comes from this observation. We first find the value x at which $f'(x) = 0$. The solution x_{soln} is a turning or inflexion point. To verify if the turning point is a minimum or maximum, we compute the second derivative at that point. If the second derivative is positive, the solution is a minimum, if it's negative it is a maximum. Thus we

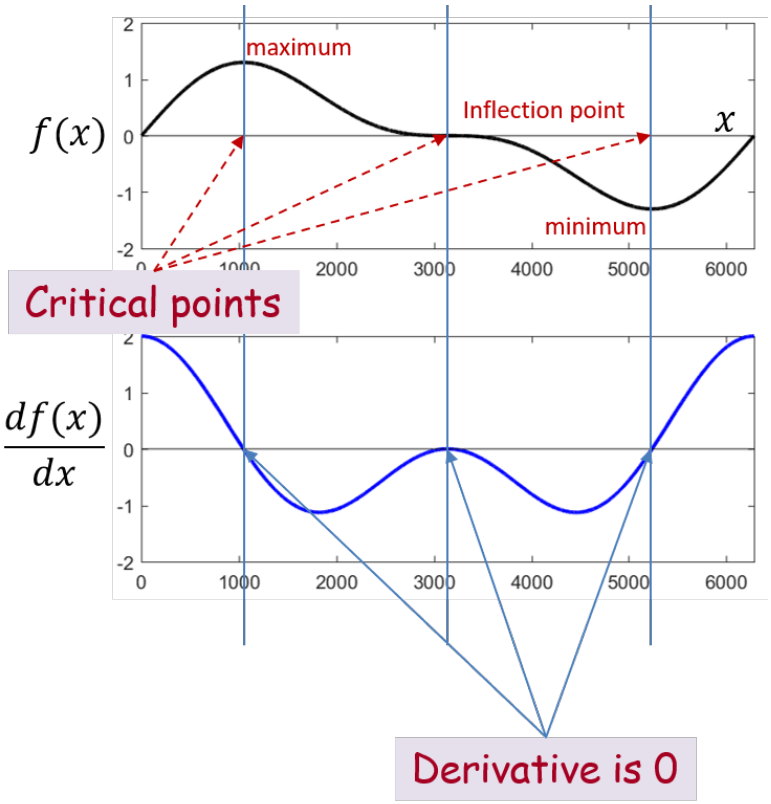
compute the second derivative at x_{soln}: $f''(x_{soln}) = (df'(x_{soln})/dx$. If $f''(x_{soln})$ is positive, x_{soln} is a minimum, otherwise it is a maximum.

3.5.2 On derivatives of functions of a single variable

Not all locations where the derivative is 0 are turning points. Locations where the derivative goes to zero are *critical* points, and these can be local minima or local maxima, which are turning points, but they can also be inflection points as shown in Fig. 3.68. An inflection point is a location where the function's behavior is of one kind, say decreasing as in the example shown in the figure, and then the function flattens briefly, and then continues behaving in the same fashion – here it continues to decrease.

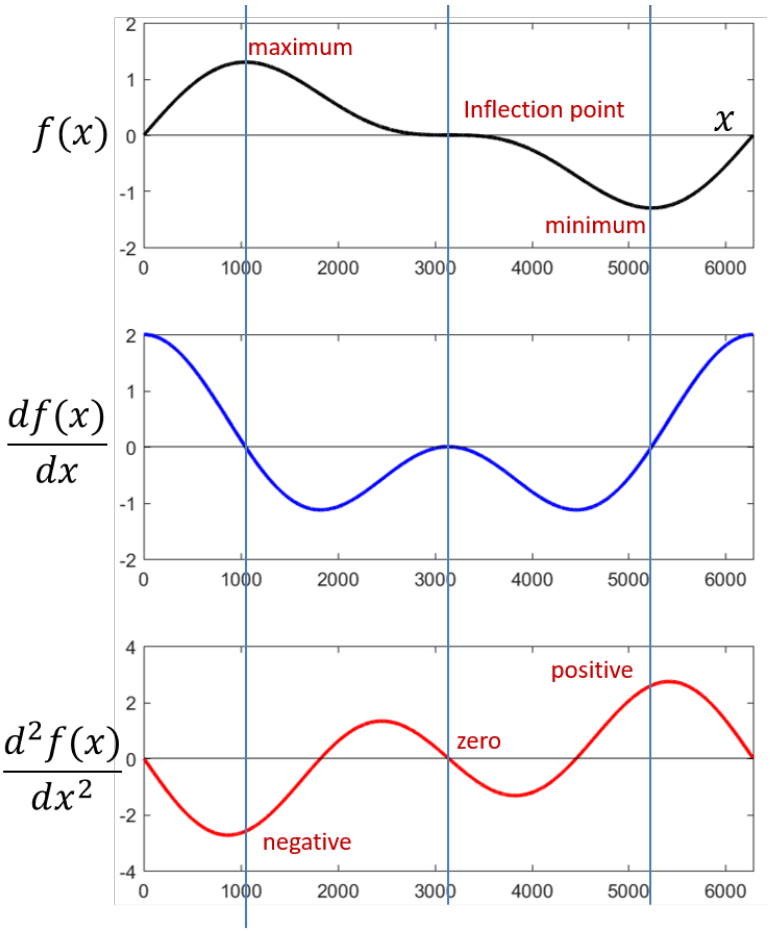
In Fig. 3.68, the second panel shows the derivative of the first curve, and we can see it goes to 0 at the maximum, the minimum, AND at the inflection point. For the inflection point where the function is decreasing, then flattens, and then continues to decrease, the derivative will be negative, increase to 0, and become negative again. If an inflection point is one where the function briefly flattens while increasing, the derivative will be positive, decrease to 0 and then go back to being positive.

Fig. 3.69 further illustrates some important points about derivatives of functions of a single variable. The second derivative, shown for the example by the red curve, will generally be negative at maxima and positive at minima. At the inflection points, the derivative itself becomes flat, so the second derivative is zero at inflection points. For functions of multiple variables, this is somewhat more complicated, as we will see next.



All locations with zero derivative are critical points. These can be local maxima, local minima, or inflection points.

Figure 3.68: Critical points of functions of a single variable

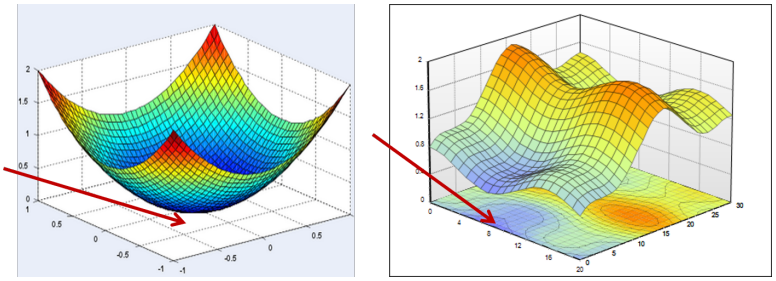


Upper: A function of a single variable. **Middle:** All locations with zero derivative are critical points. These can be local maxima, local minima, or inflection points. **Lower:** The second derivative is ≥ 0 at minima, ≤ 0 at maxima and 0 at inflection points.

Figure 3.69: Derivatives of functions of a single variable

3.5.3 On derivatives of multivariate functions

Fig. 3.70 illustrates the case for functions of a multiple variables. For multivariate functions (functions of multiple variables), minima are still "turning" points. As we approach the minimum from any direction, the function will decrease, arrive at the minimum, and then start to increase. So, shifting from the minimum in any direction by a significant amount will increase the value of the function. At the minimum itself, for smooth functions, the function will locally be flat – infinitesimal shifts in any direction will not result in any change at all. To find a minimum, or any turning point, we must find a point where shifting in any direction by a very small amount will not change the value of the function.



Functions of 3 variables. The optimum (minimum in this example, indicated by the arrows) point is still a "turning" point. Shifting away from it in any direction will increase the value of the function. For smooth functions, very small shifts will not result in any change at all. To find a critical point, we must find a point where shifting in any direction by an infinitesimal amount will not change the value of the function.

Figure 3.70: Functions of multiple variables

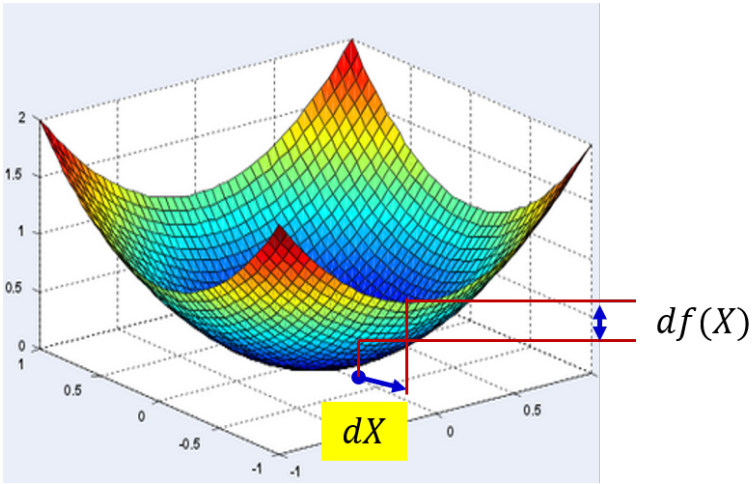
3.5.3.1 The gradient

Derivatives of functions of multiple variables are often computed in terms of **gradients**. Fig. 3.71 shows a scalar multivariate function $f(X)$ of a variable X (a vector), and its increments with increments in the variable X. Recall that the derivative $\nabla_X f(X)$ of a scalar function $f(X)$ of a multi-variate input X is a multiplicative factor that gives us the change in $f(X)$ for very small variations

in X. Thus, if dX is the increment in X, then the resulting increment in $f(X)$ is given by

$$df(X) = \nabla_X f(X) dX \tag{3.32}$$

As mentioned earlier, since X is a vector, dX too is a vector (a column vector in our notation). $f(X)$ is a scalar and hence so is $df(X)$. Consequently, $\nabla_X f(X)$, the derivative of $f(X)$ with respect to X, must be a row vector of the same size as the *transpose* of X. The term *gradient* is used in this context: The gradient is the *transpose* of the derivative: $\nabla_X f(X)^\top$. It has the same dimensionality as X.



$df(X)$ is the increment of $f(X)$ that results from an increment dX in X. The gradient specifies the multiplicative relationship between the two.

Figure 3.71: Increment in a multivariate scalar function with a small change in variable

At the risk of being unnecessarily repetitive, we remind the reader again that the derivative $\nabla_X f(X)$ is a row vector comprising the partial derivatives of $f(X)$ w.r.t. the components of X, i.e.:

$$\nabla_X f(X) = \left[\frac{\partial f(X)}{\partial x_1} \; \frac{\partial f(X)}{\partial x_2} \; \ldots \; \frac{\partial f(X)}{\partial x_n} \right] \tag{3.33}$$

This value depends on the location X where we compute it. Thus, the *gradient*

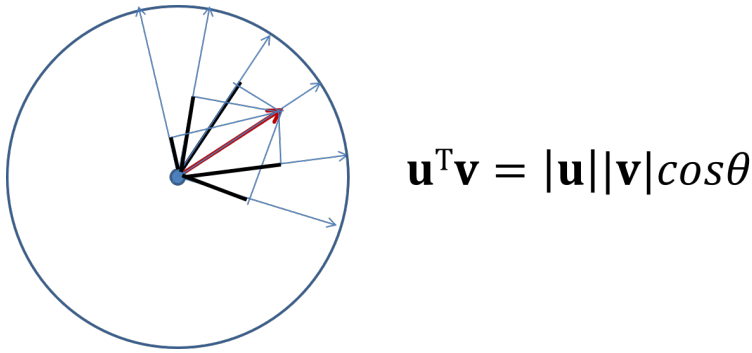
$\nabla_X f(X)^\top$, which is the transpose of the derivative, is a column vector that too is a function of X.

The relationship between an incremental change dX from any current input X and the corresponding incremental change $df(X)$ in the output is given by the relation:

$$df(X) = \nabla_X f(X) dX = \frac{\partial f(X)}{\partial x_1} dx_1 + \frac{\partial f(X)}{\partial x_2} dx_2 + \ldots + \frac{\partial f(X)}{\partial x_n} dx_n \quad (3.34)$$

The derivative here is a row vector. dX is a column vector. So $df(X)$ is the scalar product of 2 vectors: it is simply the vector inner product of the gradient and dX. To understand its behavior, let us consider a well-known property of inner products.

The inner product of any two vectors u and v, as shown in Fig. 3.72, is the product of their lengths and the cosine of the angle between them. If we keep their lengths fixed, and if we rotate v w.r.t. u, we find that the inner product between them is maximum when the two vectors are aligned, i.e. when the angle between them, $\theta = 0$



The inner product between two vectors of fixed lengths is maximum when the two vectors are aligned, i.e. when the angle θ between them is 0.

Figure 3.72: A well-known vector property

This helps in understanding the behavior of the gradient. $df(X) = \nabla_X f(X) dX$ is the inner product between the gradient $\nabla_X f(X)^T$ and the increment dX.

Again, dX is a vector representing an incremental step in the input space. If we fix the length of dX, e.g. $|dX| = 1$, but allow the direction of dX to change, we will find that the function increment $df(X)$ is maximum when dX is aligned with $\nabla_X f(X)^T$, i.e. when the angle between dX and the gradient is 0:

$$\angle(\nabla_X f(X)^T, dX) = 0 \tag{3.35}$$

In other words, the function $f(X)$ increases most rapidly if the input increment dX is perfectly aligned to the gradient $\nabla_X f(X)^T$. The gradient is thus the direction of fastest increase in $f(X)$

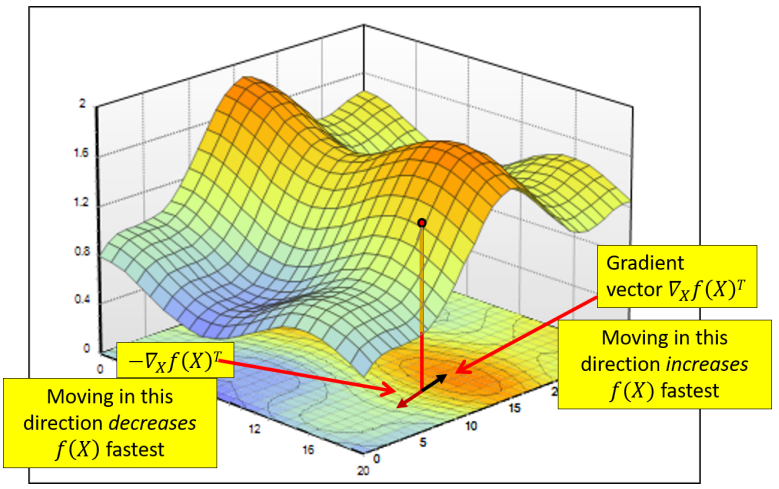
Fig. 3.73 shows a function as an example. The gradient at any point (e.g. the one shown in the figure), is a vector of partial derivatives. Since it is a vector, it has a direction and a magnitude. Moving in the direction of the gradient will result in the fastest increase in $f(X)$. The length of the gradient gives us the rate of increase of $f(X)$ if we move in the direction shown by the arrows in the figure.

Conversely, the exact opposite direction to the gradient, i.e. in the direction of the negative of the gradient, is the direction in which the function $f(X)$ decreases fastest.

At local minima of the function, the gradient is 0. It doesn't matter which direction we make an incremental shift starting from there, the function will remain flat. This is illustrated in Fig. 3.74.

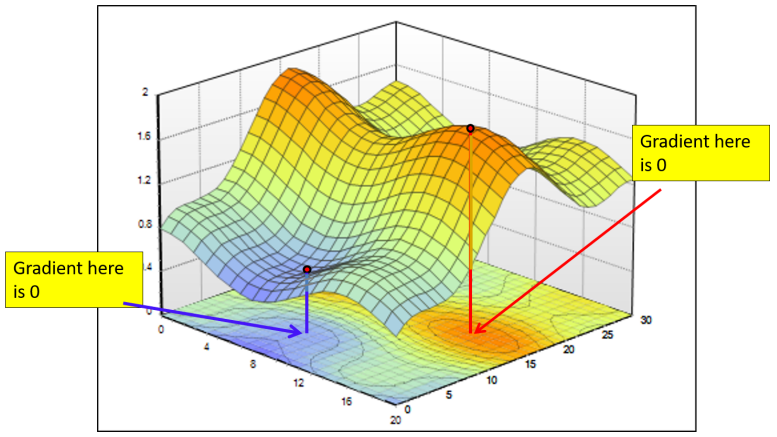
A second useful property of the gradient is illustrated in Fig. 3.75. If we slice the function horizontally at any vertical height in the figure, we obtain an edge where the function value $f(X)$ is the same for every location on the edge. The corresponding X values are called its *level set*. The bottom plane of the example in the figure shows the contours of multiple level sets.

As a specific example, let us focus on the dot on top shown in Fig. 3.75. If we sliced the function horizontally through the dot on top, we would obtain an edge shown by the red curve. The height of the edge will be uniform across the edge. The shadow of this edge on the X plane, shown by the red curve at the bottom,



The length of the gradient gives us the rate of increase of $f()$ if we move in the direction of the gradient vector, and it gives the rate of decrease of $f()$ if we move in the direction opposite to the gradient vector.

Figure 3.73: Behavior of the gradient of a scalar multi-variate function: 1



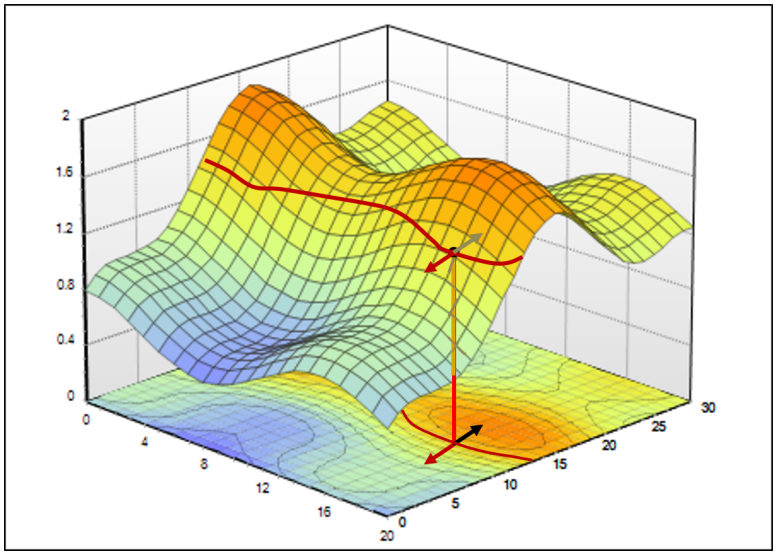
At local minima of the function, the gradient is 0..

Figure 3.74: Behavior of the gradient of a scalar multi-variate function: 2

is the level set for that edge.

The gradient of the function at any point will always be perpendicular to the level set for the function that goes through that point. In the figure, the black arrow that is at right angles to the red curve represents the gradient. The red arrow is

the negative of the gradient, and is the direction of maximum *decrease* of the function, and naturally this too will be orthogonal to the level set.



The gradient vector $\nabla_X f(X)^T$ is perpendicular to the level curve, and is shown by the black arrow.

Figure 3.75: Level sets and the gradient of a scalar multi-variate function

3.5.3.2 The Hessian

We can also compute the second derivative of a function of a vector, with respect to its argument. This is called the Hessian. It is a matrix, where the i, j^{th} component is the partial derivative of $f(X)$ w.r.t to the i^{th} and j^{th} components of X.

The Hessian of a function $f(X) = f(x_1, x_2, \ldots, x_n)$ is given by the second derivative:

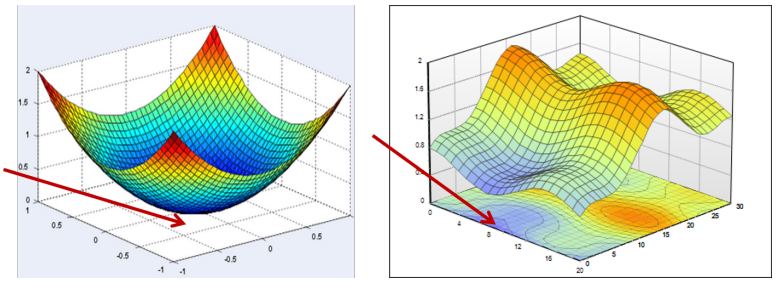
$$\nabla_X^2 f(x_1, x_2, \ldots, x_n) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \cdots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix} \tag{3.36}$$

This matrix, which too is a function of X, is positive definite for minima (we will revisit these terms later with explanation). This statement simply implies that no matter which direction (of X) we move from a minimum, the second derivative of the function will be positive.

Let us now return to our problem of direct optimization of multivariate functions.

3.5.4 Finding the minimum of scalar functions of multi-variate input

As we have seen, the minimum for a function is a turning point, where the gradient is 0, as indicated in Fig. 3.76 by the red arrows.



The arrows point to the minimum of the two functions shown.

Figure 3.76: The minimum of a scalar function of multi-variate input

3.5.4.1 Unconstrained minimization of a multivariate function

To find the minimum, we solve for the X where the derivative (or gradient) equals to zero (i.e., $\nabla_X f(X)^T = 0$. This gives us the turning points. To find out if a turning point is a minimum or a maximum, we compute the Hessian matrix $\nabla^2_X f(X)$ at the candidate solution.

If the Hessian is positive definite (i.e., if its eigenvalues are positive), the turning point a local minimum; if it is negative definite, the given turning point a local

maximum. Otherwise, it is likely to be an inflection point. A specific example of finding a minimum is given below.

Procedure 5.1: An exercise: Finding the minimum of a multivariate function

Minimize:

$$f(x_1, x_2, x_3) = (x_1)^2 + x_1(1 - x_2) + (x_2)^2 - (x_2x_3) + (x_3)^2 + x_3$$

Gradient:

$$\nabla_X f^T = \begin{bmatrix} 2x_1 + 1 - x_2 \\ -x_1 + 2x_2 - x_3 \\ -x_2 + 2x_3 + 1 \end{bmatrix}$$

Set the gradient to null:

$$\nabla_X f^T = 0 \implies \begin{bmatrix} 2x_1 + 1 - x_2 \\ -x_1 + 2x_2 - x_3 \\ -x_2 + 2x_3 + 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

Solving this system of 3 equations with 3 unknowns:

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} -1 \\ -1 \\ -1 \end{bmatrix}$$

Compute the Hessian matrix:

$$\nabla_X f^T = \begin{bmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 2 \end{bmatrix}$$

Evaluate the eigenvalues of the Hessian matrix:

$$\lambda_1 = 3.414, \lambda_2 = 0.586, \lambda_3 = 2$$

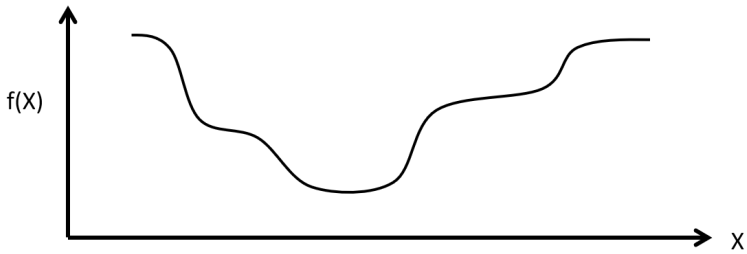
All the eigenvalues are positives $\implies$
the Hessian matrix is positive definite

The point

$$x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} -1 \\ -1 \\ -1 \end{bmatrix}$$

is therefore a minimum.

Closed-form solutions as in the exercise above are not always available for all functions, though. Often it is not possible to even solve $\nabla_X f(X)^T = 0$. The solution, or even the *equation for the gradient* may not have a closed form that we can equate to 0 and solve. Fig. 3.77 shows an example.

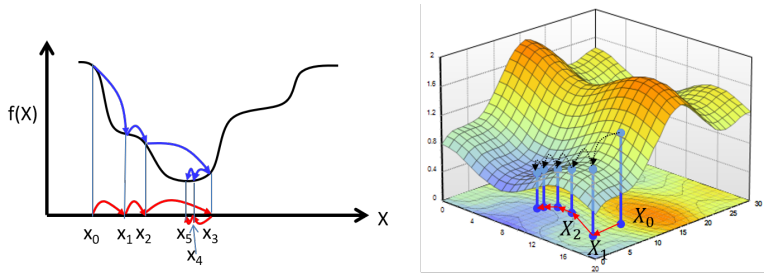


The function to minimize/maximize may have an intractable form.

Figure 3.77: A example of a function that may not have a closed form

In these situations, we use an iterative procedure instead of the one described above. In this iterative procedure, we begin with a "guess" for the optimal X, and refine it iteratively until the correct value is obtained.

The procedure is illustrated by the example in Fig. 3.78). We start with an initial guess, or initial estimate X_0 for the optimal X. We check if it is actually a minimum; if it is not, we update the estimate to a new location where the function value is lower. We repeat the process at the new estimate, and if it isn't a minimum, we update it again, and keep doing this until $f(X)$ no longer decreases.



Iterative solutions: a) Start from an initial guess X_0 for the optimal X. Update the guess towards a (hopefully) "better" value of $f(X)$. Stop when $f(X)$ no longer decreases. The problems are: a) Which direction to step in, b) How large or small our step must be.

Figure 3.78: Iterative solution to finiding the minimum of a function

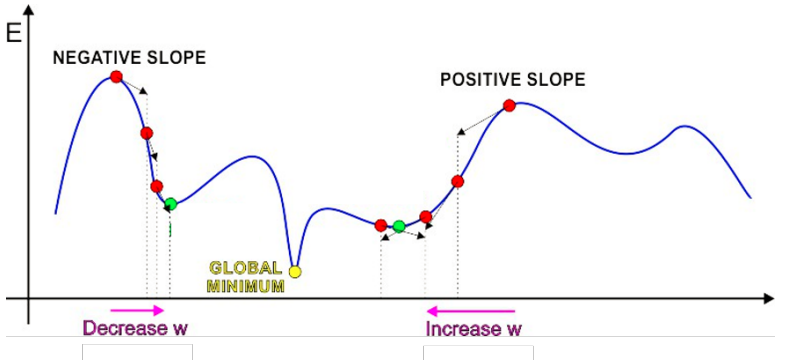
To follow this approach, we must address two issues: if we are at some current estimate which is not a minimum, it is hard to determine whether we must increase or decrease X, and by how much in to get the next estimate. Loosely speaking, it is hard to decide which "direction" we must step in, and how large our step must be. These problems are addressed by the approach of *gradient descent*, which is illustrated in Fig. 3.79.

We want each step (each change in X that we decide to make) to decrease the function. We start with our initial estimate. Then, from that estimate, we check which direction we must move in, in order to decrease the function. *This information can be found from the derivative*.

3.5.4.2 Minimization of a function of a scalar variable

Consider a function of a scalar variable x. If the derivative at the current estimate is negative, then it means that a small increase in x will decrease the function. This is illustrated in Fig. 3.79: if we are at the point indicated by the first (leftmost) red dot, the slope there is negative – increasing x, i.e. moving right decreases the function. Thus if we want to move in a direction where the function decreases, we must take a positive step and move right.

Similarly if the derivative is positive, as in the case of the rightmost red dot shown in Fig. 3.79, then a small increase in x will increase the function. So if we are at this point, then to move in a direction where the function decreases, we must take a negative step and move towards the left. So, based on the derivative at the current estimate, we decide to move left or right to get the new estimate, and repeat this process.



Iterative solution: Start at some point, and find direction in which to shift this point to decrease error (this can be found from the derivative of the function). A negative derivative $\rightarrow$ moving right decreases error. A positive derivative $\rightarrow$ moving left decreases error. Shift point in this direction.

Figure 3.79: Illustrating gradient descent for a function of a scalar variable

This gives us our first trivial iterative solution to minimize $f(x)$:

Algorithm 3: Iterative solution to minimize $f(x)$ based on the sign of the derivative

```
Initialize x^0;
while f'(x^k) ≠ 0 do
    if sign(f'(x^k)) is positive then
        x^{k+1} = x^k − step
    else
        x^{k+1} = x^k + step
    end
end
```

where $step$ is a positive value. We can write the algorithm more compactly as

follows:

Algorithm 4: Trivial solution simplified

Initialize x^0;
while $f'(x^k) \neq 0$ **do**
  $x^{k+1} = x^k - sign(f'(x^k)) \times step$;
end

This is identical to the previous algorithm. Note that by directly multiplying the step by the sign of the derivative, we get the desired behavior. If the derivative is positive, the correction is negative (i.e. a "backward" step), otherwise, if the derivative is negative, it is a positve step.

We can improve the above algorithm further by considering the derivative itself, not just its sign. This gives us the following algorithm:

Algorithm 5: Gradient descent for a function of a scalar variable

Initialize x^0;
while $f'(x^k) \neq 0$ **do**
  $x^{k+1} = x^k - \eta(f'(x^k))$;
end

Here η is the "step size." This is a slightly updated version of the Algorithm 4 with the modification that instead of using only the sign of the derivative, we now also consider the magnitude of the derivative. This gives us the update rule above, where we multiply the step size by the derivative itself, which naturally also takes up the sign of the derivative.

In a reasonably shaped function, the minima will typically appear like a (multi-dimensional) bowl. When we are far away from the minimum, the slope will be large. Near it, the slope will be small. It is therefore to our advantage to take larger steps when we are farther away form the minimum, but smaller ones as we approach the minimum. By setting our step size to be proportional to the derivative, we automatically achieve this desired property.

3.5.4.3 Gradient descent/ascent for a multivariate function

Let us now discuss gradient ascent/descent for multivariate functions. The procedure we have used in the case of univariate functions in our examples so far generalizes easily to multi-variate functions too. In this case, X is now a vector. We take a step in the direction of steepest ascent to find a maximum, and in the direction of steepest descent to find a minimum.

Thus, if we want to find the maximum of a multi-variate function, at each iteration we move in the direction of the gradient, i.e. we add a term that is proportional to the gradient at the current estimate:

$$X^{k+1} = X^k + \eta \nabla_X f(X^k)^T \tag{3.37}$$

To find a minimum we move in a direction that is exactly opposite to the direction of the gradient, i.e. we subtract a term that is proportional to the gradient at the current estimate:

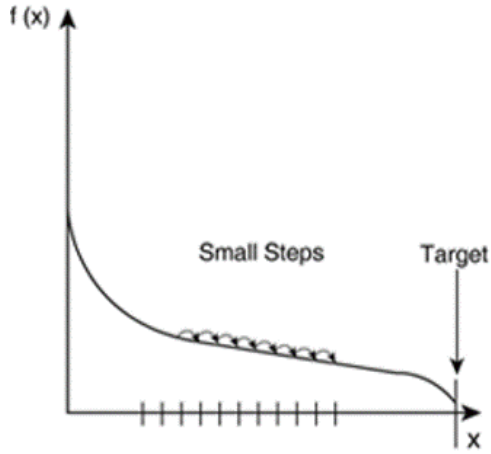
$$X^{k+1} = X^k - \eta \nabla_X f(X^k)^T \tag{3.38}$$

We call this procedure gradient *ascent* when maximizing the function, and gradient *descent* when minimizing it. As before, the multiplicative constant η is the step size parameter.

3.5.4.4 Choosing the step size

How do we choose the step size parameter η? There are many solutions. The first one is to simply set η to a fixed size, as in the algorithms above, and as shown in Fig. 3.80.

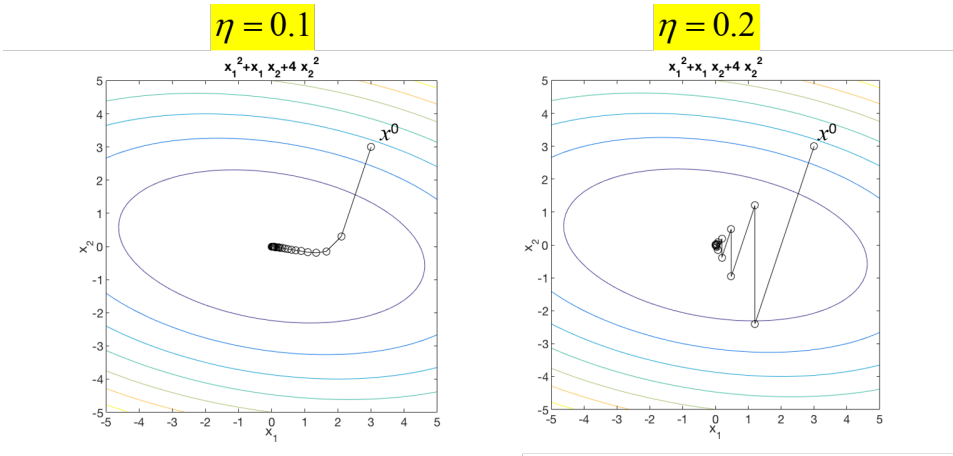
The value we choose for the fixed step size determines how fast (or slowly, in terms of number of computations required) we converge to the desired optimum. For a larger step size, convergence is faster but the chance of overshooting the



Here we use fixed value for the step parameter η^k.

Figure 3.80: Choosing a constant step size for gradient ascent/descent

optimum is also greater. Fig. 3.81 illustrates the effect of the step size on convergence to an optimum in a specific example case.



With smaller step size (left) the convergece is slower and more computations are required to determine the optimum point.

Figure 3.81: Influence of step size on convergence

A more generic setting is to make the step size η depend on the iteration itself, i.e. we would have a step size parameter η^k rather than a simple η in the above algorithms. The problem of how to determine the optimal step size in a optimization procedure will be addressed later. Step size is critical for fast optimization, and

this is an important topic. For now, however, we simply assume a potentially-iteration-dependent step size η^k.

In an iterative solution setting, we however also have to determine the stopping criteria – how do we determine when and if the process of oprimization has converged? We discuss this briefly next.

3.5.4.5 Gradient descent convergence criteria

Starting from an initial estimate, the iterative process of gradient descent continues until either additional updates don't decrease the function any more (i.e. the difference in function value between subsequent iterations becomes negligibly small), or if the gradient becomes negligibly small (which implies we are close to an minimum).

The gradient descent algorithm converges when one of the following criteria is satisfied:

$$|f(X^{k+1}) - f(X^k)| \leq \epsilon_1 \tag{3.39}$$

or

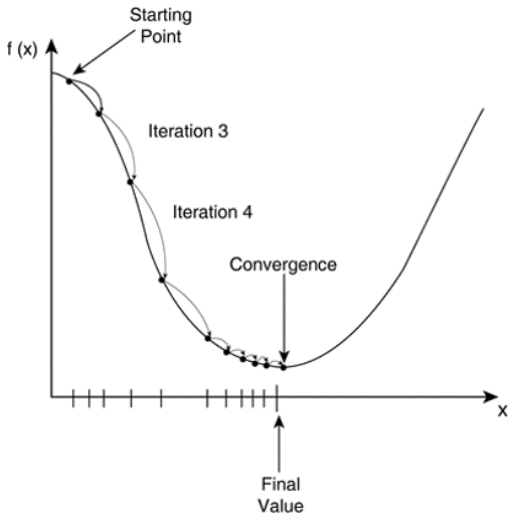
$$\| \nabla_X f(X^k)\| \leq \epsilon_2 \tag{3.40}$$

The overall gradient descent algorithm can now be written as

Algorithm 6: The general gradient decent algorithm

Initialize X^0, $k = 0$;
while $|f(X^{k+1}) - f(X^k)| > \epsilon$ **do**
  $X^{k+1} = X^k - \eta^k \nabla_X f(X^k)^T$;
  $k = k + 1$;
end

In the algorithm above, we first initialize the variable. Then we iteratively compute the gradient at the current estimate and take a step against it until the change



The gradient descent algorithm stops either when subsequent steps do not result in significant change in the function, or when the derivative of the function goes to 0.

Figure 3.82: Convergence criteria for gradient descent

in the value of the function in subsequent iterations becomes very small.

With this, we are now set to deal with the problem of training neural networks. In the chapters that follow, we will learn how to use gradient descent to train them. In the next chapter will discuss back propagation – the algorithm to compute the gradients.

Bibliography

[1] Widrow, B., 1987, June. Adaline and madaline – 1963. In Proc. IEEE 1st Int. Conf. on Neural Networks (Vol. 1, pp. 143-57).

[2] Widrow, B., McCool, J. and Ball, M., 1975. The complex LMS algorithm. Proceedings of the IEEE, 63(4), pp.719-720.

[3] Widrow, B., 2005. Thinking about thinking: the discovery of the LMS algorithm. IEEE Signal Processing Magazine, 22(1), pp.100-106