Worksheet: Basic Knowledge of Deep Learning

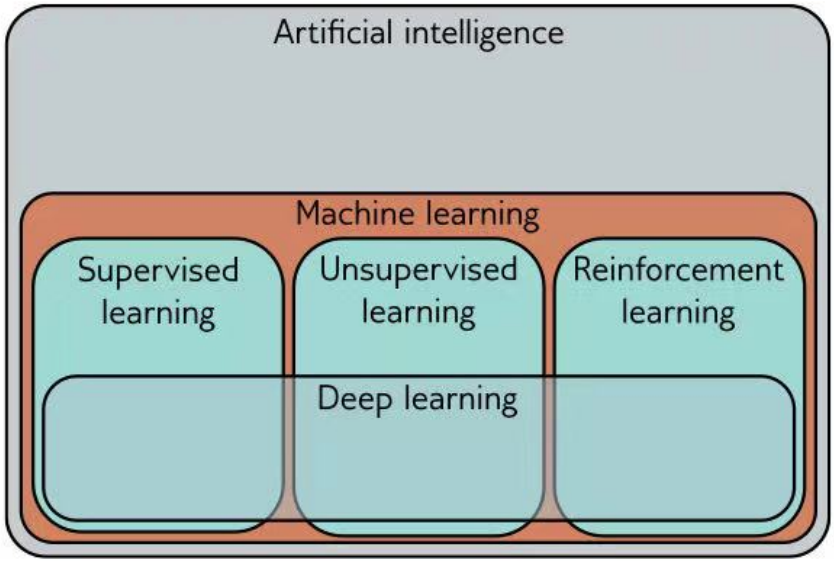
-

March 2024

The content of this worksheet is all from the book *Understanding Deep Learning* by Simon J.D. Prince. The most recent version of this book can be found at http://udlbook.com.

1 Introduction

Machine learning can be coarsely divided into **supervised**, unsupervised, and reinforcement learning.



1.1 Superivised learning

Supervised learning models define a mapping from input data to an output prediction. The method to find the map is called "training" a model.

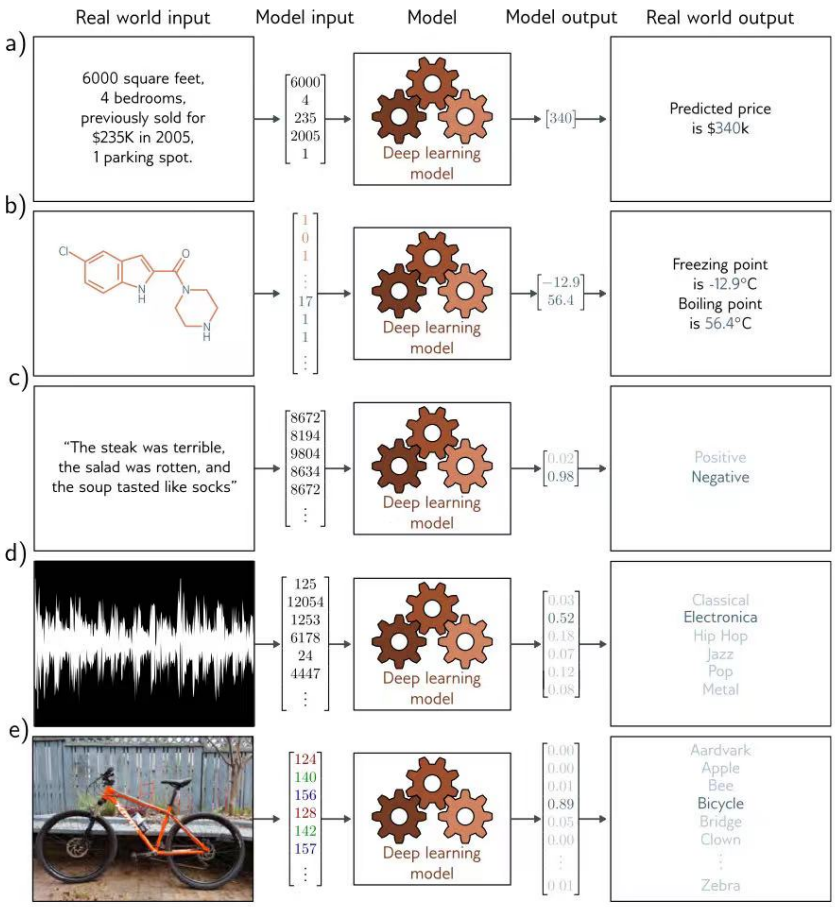


Figure 1.2 Regression and classification problems. a) This *regression* model takes a vector of numbers that characterize a property and predicts its price. b) This *multivariate regression* model takes the structure of a chemical molecule and predicts its melting and boiling points. c) This *binary classification* model takes a restaurant review and classifies it as either positive or negative. d) This *multiclass classification* problem assigns a snippet of audio to one of N genres. e) A second multiclass classification problem in which the model classifies an image according to which of N possible objects it might contain.

In Figure 1.2, we can see different types of problems. There are regression problems (model returning a continuous number) and classification problems (model assigning the input to different categories). It is worth mentioning that there can be more than one output and can be more than one input, which requires our models to be able to take in an output matrix.

1.2 Unsupervised Learning

Unsupervised learning is constructing a model from input data without corresponding output labels. Examples are generative unsupervised models.

1.3 Reinforcement learning

Reinforcement learning is when the machine takes actions and gets rewards to maximize the reward.

2 Shallow Neural Network and Deep Neural Network

We mainly focus on supervised learning. As shown in the picture before, the model we want to achieve is to map the input x to the output y.

2.1 Key concepts

Inputs: vector denoted as x.
Outputs: vector denoted as y.

For simplicity, we assume that input x and output y are vectors of a predetermined and fixed size that the elements of each vector are always ordered in the same way. This is termed *structured* or *tabular* data.

Inference: the output computed by the model (mathematical equation).

Let the model be f[·], then y=f[x].

Parameters: the numbers we find to describe the true relationship between inputs and outputs.

Different parameters change the outcome of the computation and the model equation describes a family of possible relationships between input and output.

When we train or learn the model, we use the training set of input and output pairs. If the model works well for the training set, we hope that it will make good predictions for new input when the true output is unknown.

(However it might not be the case.)

Therefore, the model should also contain parameter ϕ.

The ultimate function should be $y = f[x, \phi]$.

Loss: the deviation from the truth output y and the prediction $f[x_i, \phi]$.

After training the model, we use test data to assess its performance.

2.2 Simple example: Linear Regression Example: 1D linear regression model

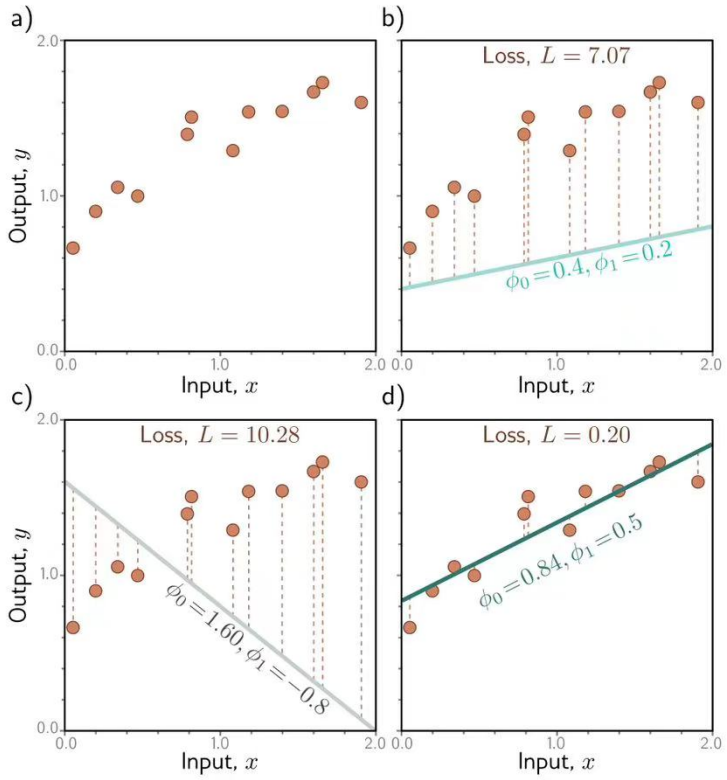


Figure 2.2 Linear regression training data, model, and loss. a) The training data (orange points) consist of $I = 12$ input/output pairs $\{x_i, y_i\}$. b–d) Each panel shows the linear regression model with different parameters. Depending on the choice of y-intercept and slope parameters $\boldsymbol{\phi} = [\phi_0, \phi_1]^T$, the model errors (orange dashed lines) may be larger or smaller. The loss L is the sum of the squares of these errors. The parameters that define the lines in panels (b) and (c) have large losses $L = 7.07$ and $L = 10.28$, respectively because the models fit badly. The loss $L = 0.20$ in panel (d) is smaller because the model fits well; in fact, this has the smallest loss of all possible lines, so these are the optimal parameters.

As is shown in the figure, we try to describe the relationship between input x and output y by a linear function.

$$\begin{aligned} y =& f[x, \phi] \\ =& \phi_0 + \phi_1 x \end{aligned}$$

And the loss can be:

$$\begin{aligned} L[\phi] =& \sum_{i=1}^{I} (f[x_i, \phi] - y_i])^2 \\ =& \sum_{i=1}^{I} (\phi_0 + \phi_1 x_i - y_i)^2 \end{aligned}$$

Then, our goal is to find the parameters $\widehat{\phi}$ that minimize the loss function:

$$\begin{aligned}\widehat{\phi} &= \underset{\phi}{\arg\min}[L[\phi]] \\ &= \underset{\phi}{\arg\min}[\sum_{i=1}^{I}(f[x_i,\phi]-y_i)^2] \\ &= \underset{\phi}{\arg\min}[\sum_{i=1}^{I}(\phi_0+\phi_1 x_i-y_i)^2]\end{aligned}$$

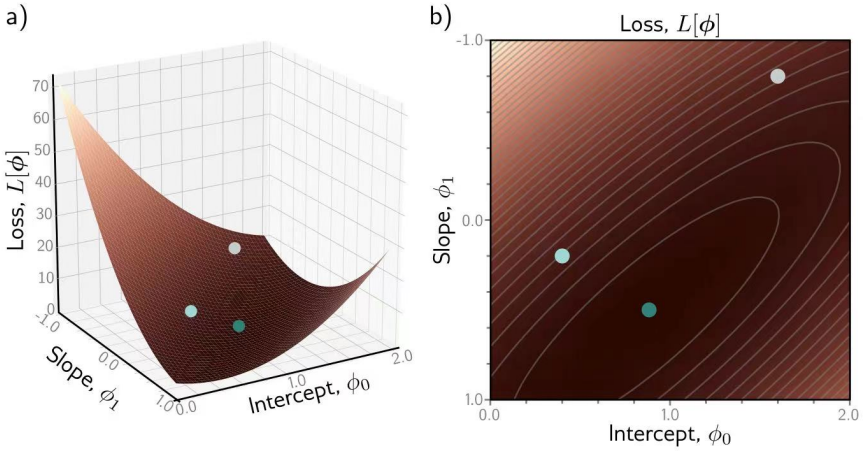


Figure 2.3 Loss function for linear regression model with the dataset in figure 2.2a. a) Each combination of parameters $\boldsymbol{\phi} = [\phi_0,\phi_1]^T$ has an associated loss. The resulting loss function $L[\boldsymbol{\phi}]$ can be visualized as a surface. The three circles represent the lines from figure 2.2b–d. b) The loss can also be visualized as a heatmap, where brighter regions represent larger losses; here we are looking straight down at the surface in (a) from above and gray ellipses represent isocontours. The best fitting line (figure 2.2d) has the parameters with the smallest loss (green circle).

Model fitting, training or learning: the process of finding the best parameters from the initial parameters. From the graph, we can imagine that it is a process of walking down the hill. To specify, we measure the gradient of the surface at the current position and take a step in the direction that is most steeply downhill unless the gradient is flat.

2.3 Shallow neural networks

Since the relationship between input x and output y is unlikely to be a simple linear function, we need to use the combination of piecewise linear functions to approximate the arbitrarily complex relationship between multi-dimensional inputs and outputs.

2.3.1 Terminology

activation function a[·]: change a linear function to a piecewise linear function. One common choice is the rectified linear unit of ReLU. When a unit is clipped, we refer to it as **inactive**, and when it is not clipped, we refer to it as **active**.

$$\mathrm{a}[z] = \mathrm{ReLU}[z] = \begin{cases} 0 & z < 0 \\ z & z \geq 0 \end{cases}.$$

In order to approximate arbitrary complex relationships, we need to add up different piecewise functions.
β **bias**: The parameters represent the slope.
Ω **weight**: The offset parameters.
h_1, h_2, h_3: **hidden units**.

$$h_1 = a[\theta_{10} + \theta_{11}x]$$
$$h_2 = a[\theta_{20} + \theta_{21}x]$$
$$h_3 = a[\theta_{30} + \theta_{31}x]$$

Layers: The associate jargon of neural networks. There can be **input layer**, **hidden layer**, and **output layer**.
Neurons: The hidden units.

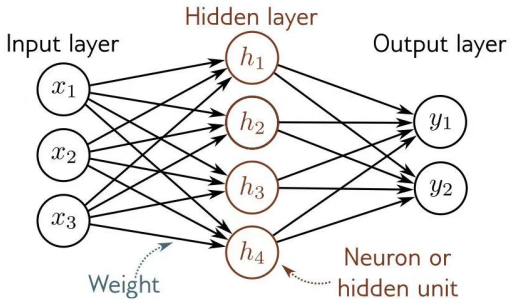


Figure 3.12 Terminology. A shallow network consists of an input layer, a hidden layer, and an output layer. Each layer is connected to the next by forward connections (arrows). For this reason, these models are referred to as feed-forward networks. When every variable in one layer connects to every variable in the next, we call this a fully connected network. Each connection represents a slope parameter in the underlying equation, and these parameters are termed weights. The variables in the hidden layer are termed neurons or hidden units. The values feeding into the hidden units are termed pre-activations, and the values at the hidden units (i.e., after the ReLU function is applied) are termed activations.

Pre-activations: The values of the input to the hidden layers (values before the activation function applied).
Activations: The values at the hidden layer (values after the activation function is applied).
Multi-layer perception(MLP): Any neural network with at least one hidden layer.
Shallow neural networks: The networks with one hidden layer.
Deep neural networks: The network with multiple hidden layers.
Feed-forward networks: Neural networks in which the connections form an acyclic graph.
Fully connected networks: The network that every element in one layer connects to every element in the next.

2.3.2 Univariate inputs and ouputs

We add bias and weight to the linear function and use the activation function to clip it. Then we can get several hidden units can get the outcome from it.

$$y = \phi_0 + \phi_1 h_1 + \phi_2 h_2 + \phi_3 h_3$$

And the whole process can be represented in figures like this.

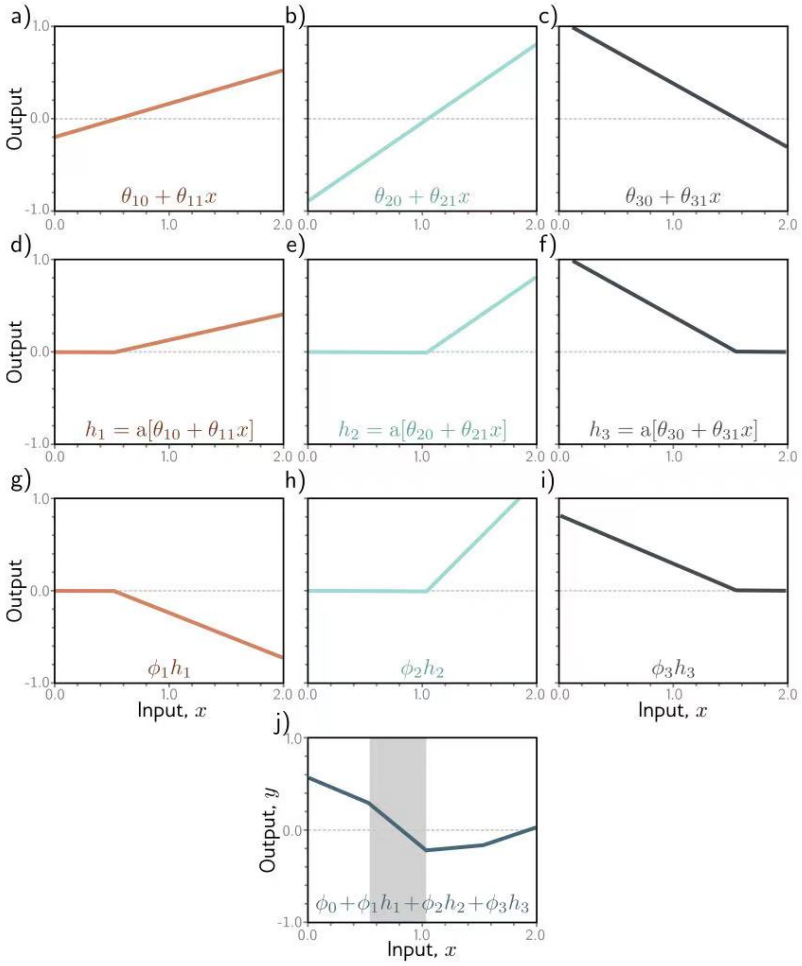


Figure 3.3 Computation for function in figure 3.2a. a–c) The input x is passed through three linear functions, each with a different y-intercept $\theta_{\bullet 0}$ and slope $\theta_{\bullet 1}$. d–f) Each line is passed through the ReLU activation function, which clips negative values to zero. g–i) The three clipped lines are then weighted (scaled) by ϕ_1, ϕ_2, and ϕ_3, respectively. j) Finally, the clipped and weighted functions are summed, and an offset ϕ_0 that controls the height is added. Each of the four linear regions corresponds to a different activation pattern in the hidden units. In the shaded region, h_2 is inactive (clipped), but h_1 and h_3 are both active.

As is shown in the figure, this function has more regions (separated by the joint) and thus can simulate more complex functions. Each linear region in this figure corresponds to a different activation pattern in the hidden units. For example, the shaded region receives contributions from h_1 and h_3, which are active, but not from h_2, which is inactive.

We can also use graphs to depict neural networks.

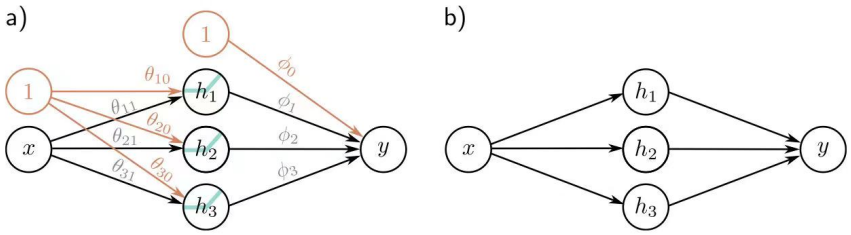


Figure 3.4 Depicting neural networks. a) The input x is on the left, the hidden units h_1, h_2, and h_3 in the center, and the output y on the right. Computation flows from left to right. The input is used to compute the hidden units, which are combined to create the output. Each of the ten arrows represents a parameter (intercepts in orange and slopes in black). Each parameter multiplies its source and adds the result to its target. For example, we multiply the parameter ϕ_1 by source h_1 and add it to y. We introduce additional nodes containing ones (orange circles) to incorporate the offsets into this scheme, so we multiply ϕ_0 by one (with no effect) and add it to y. ReLU functions are applied at the hidden units. b) More typically, the intercepts, ReLU functions, and parameter names are omitted; this simpler depiction represents the same network.

From this, we can derive a universal approximation theorem. Consider the case with D hidden units where the d^{th} hidden units is

$$h_d = a[\theta_{d0} + \theta_{d1}x]$$

and combining them we get

$$y = \phi_0 + \sum_{d=1}^{D} \phi_d h_d$$

Network capacity: The number of hidden units in a shallow network.
When there are D hidden units, there are at most D joints and D+1 regions. If we can add more hidden units, the model can approximate more complex functions.

2.3.3 Multivariate inputs and outputs

2.3.3.1 multivariate outputs

We simply use a different linear function of the hidden units for each output.

$$\begin{aligned}
h_1 &= a[\theta_{10} + \theta_{11}x] \\
h_2 &= a[\theta_{20} + \theta_{21}x] \\
h_3 &= a[\theta_{30} + \theta_{31}x] \\
h_4 &= a[\theta_{40} + \theta_{41}x]
\end{aligned}$$

$$\begin{aligned}
y &= \phi_{10} + \phi_{11}h_1 + \phi_{12}h_2 + \phi_{13}h_3 + \phi_{14}h_4 \\
y &= \phi_{20} + \phi_{21}h_1 + \phi_{22}h_2 + \phi_{23}h_3 + \phi_{24}h_4
\end{aligned}$$

Illustrated by the graph, we can get

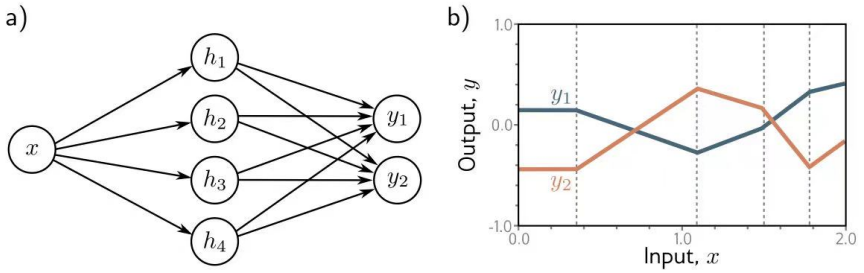


Figure 3.6 Network with one input, four hidden units, and two outputs. a) Visualization of network structure. b) This network produces two piecewise linear functions, $y_1[x]$ and $y_2[x]$. The four "joints" of these functions (at vertical dotted lines) are constrained to be in the same places since they share the same hidden units, but the slopes and overall height may differ.

2.3.3.2 multivariate inputs

For multivariate inputs, we extend the linear relationship between the inputs and the hidden units, then the equations can be

$$h_1 = a[\theta_{10} + \theta_{11}x + \theta_{12}x_2]$$
$$h_2 = a[\theta_{20} + \theta_{21}x + \theta_{22}x_2]$$
$$h_3 = a[\theta_{30} + \theta_{31}x + \theta_{32}x_2]$$
$$y = \phi_0 + \phi_1 h_1 + \phi_2 h_2 + \phi_3 h_3$$

This can be illustrated in figure 3.7 and 3.8.

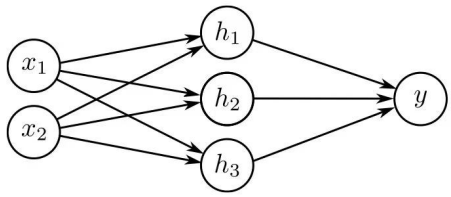


Figure 3.7 Visualization of neural network with 2D multivariate input $\mathbf{x} = [x_1, x_2]^T$ and scalar output y.

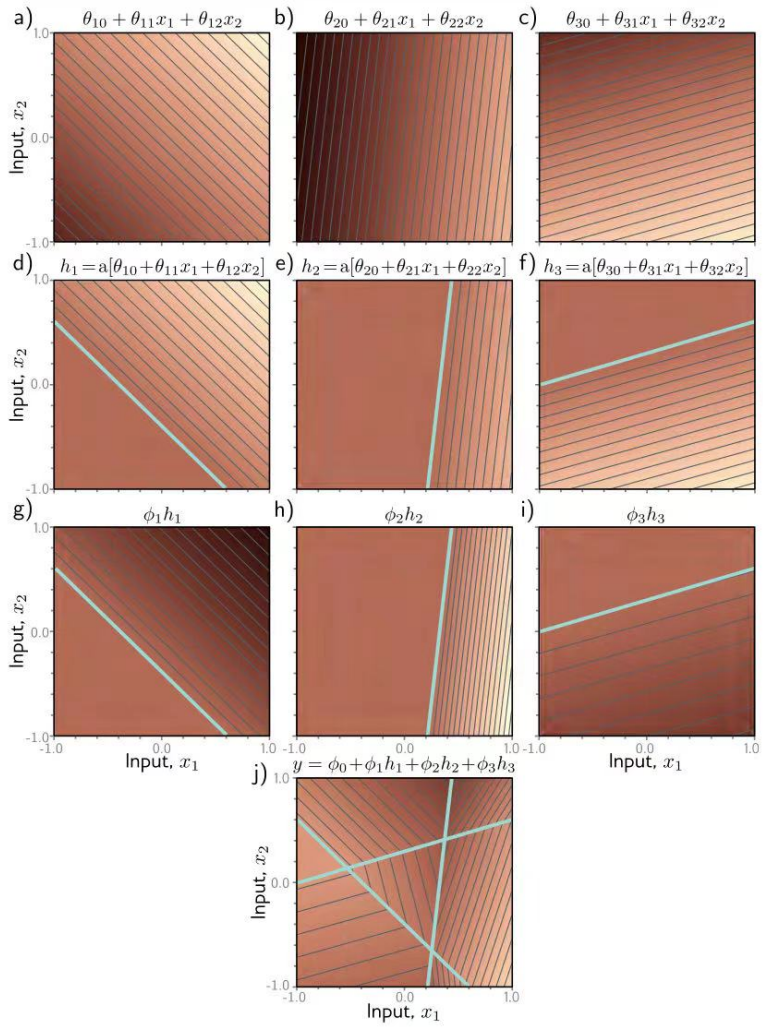


Figure 3.8 Processing in network with two inputs $\mathbf{x} = [x_1, x_2]^T$, three hidden units h_1, h_2, h_3, and one output y. a–c) The input to each hidden unit is a linear function of the two inputs, which corresponds to an oriented plane. Brightness indicates function output. For example, in panel (a), the brightness represents $\theta_{10} + \theta_{11}x_1 + \theta_{12}x_2$. Thin lines are contours. d–f) Each plane is clipped by the ReLU activation function (cyan lines are equivalent to "joints" in figures 3.3d–f). g-i) The clipped planes are then weighted, and j) summed together with an offset that determines the overall height of the surface. The result is a continuous surface made up of convex piecewise linear polygonal regions.

As is shown in 3.8, these hidden units form an oriented plane in the 3D input/output space, and the activated function clipped the planes and the final result is a continuous piecewise linear surface consisting of convex polygonal regions.

2.3.4 General case of Shall Neural Network

Define a general equation for a shallow neural network $y = f[x, \phi]$ that maps a multi-dimensional input $x \in \mathbb{R}^{D_i}$ to a multi-dimensional output $y \in \mathbb{R}^{D_o}$ using $h \in \mathbb{R}^D$ hidden units. Each hidden unit is computed as:

$$h_d = a[\theta_{d0} + \sum_{i=1}^{D_i} \theta_{di} x_i]$$

and the combination is

$$y_j = \phi_{j0} + \sum_{d=1}^{D} \phi_{jd} h_d$$

where $a[\cdot]$ is a nonlinear activation function.

2.4 Deep neural Network

Although we can use shallow neural networks to approximate complex continuous functions, sometimes we need a huge number of hidden units. By using a deep neural network, we can produce more linear regions than a shallow network for a given number of parameters. Therefore, deep neural networks can be used to describe a broader family of functions.

2.4.1 From Composing network to Deep networks

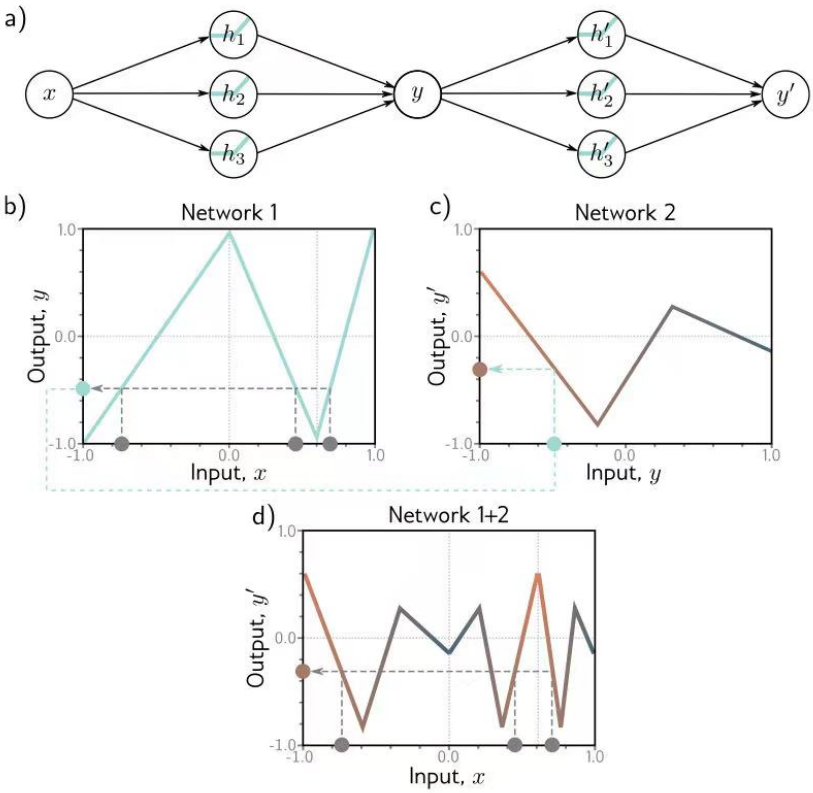


Figure 4.1 Composing two single-layer networks with three hidden units each. a) The output y of the first network constitutes the input to the second network. b) The first network maps inputs $x \in [-1, 1]$ to outputs $y \in [-1, 1]$ using a function comprising three linear regions that are chosen so that they alternate the sign of their slope (fourth linear region is outside range of graph). Multiple inputs x (gray circles) now map to the same output y (cyan circle). c) The second network defines a function comprising three linear regions that takes y and returns y' (i.e., the cyan circle is mapped to the brown circle). d) The combined effect of these two functions when composed is that (i) three different inputs x are mapped to any given value of y by the first network and (ii) are processed in the same way by the second network; the result is that the function defined by the second network in panel (c) is duplicated three times, variously flipped and rescaled according to the slope of the regions of panel (b).

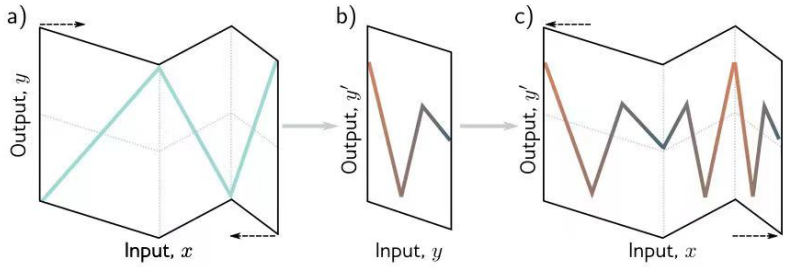


Figure 4.3 Deep networks as folding input space. a) One way to think about the first network from figure 4.1 is that it "folds" the input space back on top of itself. b) The second network applies its function to the folded space. c) The final output is revealed by "unfolding" again.

The first idea we get might be: why not making the output y as another input, thus we can make "fold" the graph and create more regions.
Then we can construct two networks as is shown in figure 4.1.a.
The equations of the first network is:

$$\begin{aligned} h_1 &= a[\theta_{10} + \theta_{11}x] \\ h_2 &= a[\theta_{20} + \theta_{21}x] \\ h_3 &= a[\theta_{30} + \theta_{31}x] \end{aligned}$$

$$y = \phi_0 + \phi_1 h_1 + \phi_2 h_2 + \phi_3 h_3$$

The equation of the second network is:

$$\begin{aligned} h'_1 &= a[\theta'_{10} + \theta_{11}y] \\ h'_2 &= a[\theta'_{20} + \theta_{21}y] \\ h'_3 &= a[\theta'_{30} + \theta_{31}y] \end{aligned}$$

$$y' = \phi'_0 + \phi'_1 h'_1 + \phi'_2 h'_2 + \phi'_3 h'_3$$

Then, we can show that this equation is actually a special case of a deep network with two hidden layers.

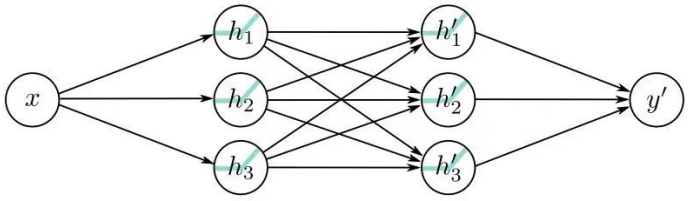


Figure 4.4 Neural network with one input, one output, and two hidden layers, each containing three hidden units.

We can unfold y in the second layer of hidden units and arrive at equations as:

$$\begin{aligned} h'_1 &= a[\theta'_{10} + \theta_{11}y] \\ &= a[\theta'_{10} + \theta'_{11}\phi_0 + \theta'_{11}\phi_1 h_1 + \theta'_{11}\phi_2 h_2 + \theta'_{11}\phi_3 h_3] \\ h'_2 &= a[\theta'_{20} + \theta_{21}y] \\ &= a[\theta'_{10} + \theta'_{11}\phi_0 + \theta'_{11}\phi_1 h_1 + \theta'_{11}\phi_2 h_2 + \theta'_{11}\phi_3 h_3] \\ h'_3 &= a[\theta'_{30} + \theta_{31}y] \\ &= a[\theta'_{10} + \theta'_{11}\phi_0 + \theta'_{11}\phi_1 h_1 + \theta'_{11}\phi_2 h_2 + \theta'_{11}\phi_3 h_3] \end{aligned}$$

which we can rewrite as:

$$\begin{aligned} h'_1 =& a[\psi_{10} + \psi_{11}h_1 + \psi_{12}h_2 + \psi_{13}h_3] \\ h'_2 =& a[\psi_{20} + \psi_{21}h_1 + \psi_{22}h_2 + \psi_{23}h_3] \\ h'_3 =& a[\psi_{30} + \psi_{31}h_3 + \psi_{32}h_3 + \psi_{33}h_3] \end{aligned}$$

where $\psi_{10} = \theta'_{10} + \theta'_{11}\phi_0$, $\psi_{11} = \theta'_{11}\phi_1$, $\psi_{12} = \theta'_{11}\phi_2$, and so on. The result is corresponding to the network with two layers.
For deep network, in general case, we can write things like:
The first layer is defined by:

$$\begin{aligned} h_1 &= a[\theta_{10} + \theta_{11}x] \\ h_2 &= a[\theta_{20} + \theta_{21}x] \\ h_3 &= a[\theta_{30} + \theta_{31}x] \end{aligned}$$

The second layer is defined by:

$$\begin{aligned} h'_1 =& a[\psi_{10} + \psi_{11}h_1 + \psi_{12}h_2 + \psi_{13}h_3] \\ h'_2 =& a[\psi_{20} + \psi_{21}h_1 + \psi_{22}h_2 + \psi_{23}h_3] \\ h'_3 =& a[\psi_{30} + \psi_{31}h_3 + \psi_{32}h_3 + \psi_{33}h_3] \end{aligned}$$

and the output by:

$$y' = \phi'_0 + \phi'_1 h'_1 + \phi'_2 h'_2 + \phi'_3 h'_3$$

2.4.2 Hyparameters

Width of the network: The number of hidden units in each layer.
Depth of the network: The number of the hidden layers.
Capacity of the network: The total number of the hidden units.
We denote the number of layers as K and the number of hidden units in each layer as $D_1, D_2, ..., D_k$.

2.4.3 Matrix notation

In practical, we can use matrix to represent the example before as:

$$\begin{bmatrix} h_1 \\ h_2 \\ h_3 \end{bmatrix} = \mathbf{a}\left[\begin{bmatrix} \theta_{10} \\ \theta_{20} \\ \theta_{30} \end{bmatrix} + \begin{bmatrix} \theta_{11} \\ \theta_{21} \\ \theta_{31} \end{bmatrix} x\right],$$

$$\begin{bmatrix} h'_1 \\ h'_2 \\ h'_3 \end{bmatrix} = \mathbf{a}\left[\begin{bmatrix} \psi_{10} \\ \psi_{20} \\ \psi_{30} \end{bmatrix} + \begin{bmatrix} \psi_{11} & \psi_{12} & \psi_{13} \\ \psi_{21} & \psi_{22} & \psi_{23} \\ \psi_{31} & \psi_{32} & \psi_{33} \end{bmatrix} \begin{bmatrix} h_1 \\ h_2 \\ h_3 \end{bmatrix}\right],$$

$$y' = \phi'_0 + \begin{bmatrix} \phi'_1 & \phi'_2 & \phi'_3 \end{bmatrix} \begin{bmatrix} h'_1 \\ h'_2 \\ h'_3 \end{bmatrix},$$

And in a more compact way as:

$$\begin{aligned} \mathbf{h} &= \mathbf{a}[\theta_{\mathbf{0}} + \theta x] \\ \mathbf{h}' &= \mathbf{a}[\psi_{\mathbf{0}} + \mathbf{\Psi}\mathbf{h}] \\ y' &= \phi'_0 + \phi'\mathbf{h}' \end{aligned}$$

where the function $\mathbf{a}[\cdot]$ applies the activation function separately to every element of its vector input. Then, we can arrive at the **general formulation**.

We describe the vectors of hidden units at later k as $\mathbf{h}_k$, the vectors of biases(intercepts) that contribute to the hidden layer k+1 as η_k, and weights(slopes) that are applied to the k^{th} layer and contribute to the $(k+1)^{th}$ layer as $\mathbf{\Omega}_k$. A general network $y = f[x, \phi]$ with K layers can be written as:

$$\begin{aligned}
\mathbf{h_1} &= \mathbf{a}[\beta_\mathbf{0} + \mathbf{\Omega_0 x}] \\
\mathbf{h_1} &= \mathbf{a}[\beta_\mathbf{1} + \mathbf{\Omega_1 h_1}] \\
\mathbf{h_1} &= \mathbf{a}[\beta_\mathbf{2} + \mathbf{\Omega_2 h_2}] \\
&..... \\
\mathbf{y} &= \beta_\mathbf{k} + \mathbf{\Omega_k h_k}
\end{aligned}$$

And the parameters ϕ of this model comprise all of there weight matrices and bias vectors

$$\phi = \{\beta_\neg, \not\leqslant_\neg\}_{k=0}^{K}$$

3 Loss function

After constructing the neural network to approximate the complex function, we need to verify whether it can represent the original function or how much is the mismatch. To deal with this, we need to use probability. We need to figure out the parameters that achieve the maximum likelihood of the function.

3.1 Maxumum likelihood criterion

We consider the model as computing a conditional probability distribution $Pr(\mathbf{y}|\mathbf{x})$ over the possible outputs $\mathbf{y}$ given input $\mathbf{x}$. The loss encourages each training output $\mathbf{y_i}$ to have a high probability under the distribution $Pr(\mathbf{y_i}|\mathbf{x_i})$ computed from the corresponding input $\mathbf{x_i}$. We need to select a prediction model according to the distribution of the data and then use the model $f[x, \phi]$ to predict the possibility of y. We with to get a higher possibility since the occurrence of y is the ground truth.
Here, we assume the output given the input is independent. So the total likelihood of the training data decomposes as:

$$\begin{aligned}
\widehat{\phi} &= \arg\max_{\phi}[\prod_{i=1}^{I} Pr(\mathbf{y_i}|\mathbf{x_i})] \\
&= \arg\max_{\phi}[\prod_{i=1}^{I} Pr(\mathbf{y_i}|\theta_\mathbf{i})] \\
&= \arg\max_{\phi}[\prod_{i=1}^{I} Pr(\mathbf{y_i}|\mathbf{f}[\mathbf{x_i}, \phi])]
\end{aligned}$$

Since each term of $Pr(\mathbf{y_i}|\mathbf{f}[\mathbf{x_i}, \phi])$ can be very small, the product of so many of them is not practical. We can use logarithm to replace it:

$$\begin{aligned}
\widehat{\phi} &= \arg\max_{\phi}[\prod_{i=1}^{I} Pr(\mathbf{y_i}|\mathbf{f}[\mathbf{x_i}, \phi])] \\
&= \arg\max_{\phi}[\log[\prod_{i=1}^{I} Pr(\mathbf{y_i}|\mathbf{f}[\mathbf{x_i}, \phi])] \\
&= \arg\max_{\phi}[\sum_{i=1}^{I} \log[Pr(\mathbf{y_i}|\mathbf{f}[\mathbf{x_i}, \phi])]]
\end{aligned}$$

Finally, by convention, the model fitting problem are framed in terms of minimizing a loss. Then we use the negative log-likelihood criterion to satisfy this:

$$\begin{aligned}
\widehat{\phi} &= \arg\min_{\phi}[-\sum_{i=1}^{I} \log[Pr(\mathbf{y_i}|\mathbf{f}[\mathbf{x_i}, \phi])]] \\
&= \arg\min_{\phi}[L[\phi]]
\end{aligned}$$

3.2 The selection of a suitable probability distribution

Data Type	Domain	Distribution	Use
univariate, continuous, unbounded	$y \in \mathbb{R}$	univariate normal	regression
univariate, continuous, unbounded	$y \in \mathbb{R}$	Laplace or t-distribution	robust regression
univariate, continuous, unbounded	$y \in \mathbb{R}$	mixture of Gaussians	multimodal regression
univariate, continuous, bounded below	$y \in \mathbb{R}^+$	exponential or gamma	predicting magnitude
univariate, continuous, bounded	$y \in [0, 1]$	beta	predicting proportions
multivariate, continuous, unbounded	$\mathbf{y} \in \mathbb{R}^K$	multivariate normal	multivariate regression
univariate, continuous, circular	$y \in (-\pi, \pi]$	von Mises	predicting direction
univariate, discrete, binary	$y \in \{0, 1\}$	Bernoulli	binary classification
univariate, discrete, bounded	$y \in \{1, 2, \ldots, K\}$	categorical	multiclass classification
univariate, discrete, bounded below	$y \in [0, 1, 2, 3, \ldots]$	Poisson	predicting event counts
multivariate, discrete, permutation	$\mathbf{y} \in \text{Perm}[1, 2, \ldots, K]$	Plackett-Luce	ranking

Figure 5.11 Distributions for loss functions for different prediction types.

3.2.1 Example: multiclass classification

Goal: Assign an input data example x to one of K¿2 classes, so $y \in \{1, 2, ..., K\}$.
Choosing distribution: Categorical distribution.

$$Pr(y = k) = \lambda_k$$

The parameters are constrained to take values between zero and one, and they must collectively sum to one to ensure a valid probability distribution.

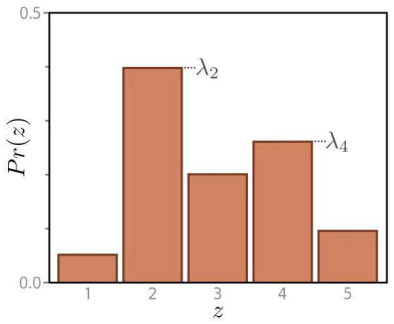


Figure 5.9 Categorical distribution. The categorical distribution assigns probabilities to $K > 2$ categories, with associated probabilities $\lambda_1, \lambda_2, \ldots, \lambda_K$. Here, there are five categories, so $K = 5$. To ensure that this is a valid probability distribution, each parameter λ_k must lie in the range $[0, 1]$, and all K parameters must sum to one.

However, when we use the network function $\mathbf{f}[\mathbf{x}, \phi]$ with K outputs to compute the K parameters from the input x, we can't ensure it obeys these constraints. Therefore, we need to put the vector

$\mathbf{f}[\mathbf{x}, \phi]$ into a function called **softmax function** to fulfill the constraints.

$$softmax_k[\mathbf{z}] = \frac{exp[z_k]}{\sum_{k'=1}^{K} exp[z_{k'}]}$$

where the exponential function ensures positivity and the sum ensures that the sum of all the K numbers is one.

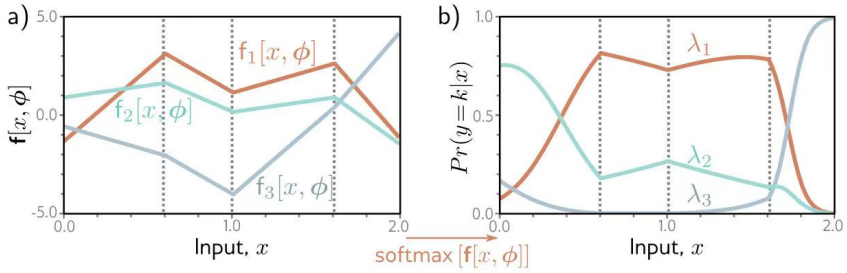


Figure 5.10 Multiclass classification for $K = 3$ classes. a) The network has three piecewise linear outputs, which can take arbitrary values. b) After the softmax function, these outputs are constrained to be non-negative and sum to one. Hence, for a given input $\mathbf{x}$, we compute valid parameters for the categorical distribution: any vertical slice of this plot produces three values sum to one and would form the heights of the bars in a categorical distribution similar to figure 5.9.

The likelihood that input x has label y=k is hence:

$$Pr(y = k|x) = softmax_k[\mathbf{f}[\mathbf{x}, \phi]]$$

The loss function therefore is:

$$\begin{aligned} L[\phi] &= -\sum_{i=1}^{I} \log[softmax_{y_i}[\mathbf{f}[\mathbf{x_i}, \phi]]] \\ &= -\sum_{i=1}^{I} (f_{y_i}[\mathbf{x_i}, \phi] - \log[\sum_{k'=1}^{K} \exp[f_{k'}[\mathbf{x_i}, \phi]]]) \end{aligned}$$

4 Fitting models

The way of finding the parameter values that minimize the loss is known as *learning* the networks' parameters or simply as *training or fitting* the model. The process is to choose the initial parameter values can then iterate the following two steps:
i) compute the derivatives(gradients) of the loss with respect to the parameters;
ii) adjust the parameters based on the gradients to decrease the loss.

4.1 The optimization of the gradient descent

We might omit some optimization due to the time limit, but I will put the formula and the figures here.
When doing the gradient descent, we are walking down the hill.

Step 1. Compute the derivatives of the loss with respect to the parameters:

$$\frac{\partial L}{\partial \boldsymbol{\phi}} = \begin{bmatrix} \frac{\partial L}{\partial \phi_0} \\ \frac{\partial L}{\partial \phi_1} \\ \vdots \\ \frac{\partial L}{\partial \phi_N} \end{bmatrix}. \tag{6.2}$$

Step 2. Update the parameters according to the rule:

$$\boldsymbol{\phi} \longleftarrow \boldsymbol{\phi} - \alpha \cdot \frac{\partial L}{\partial \boldsymbol{\phi}}, \tag{6.3}$$

where the positive scalar α determines the magnitude of the change.

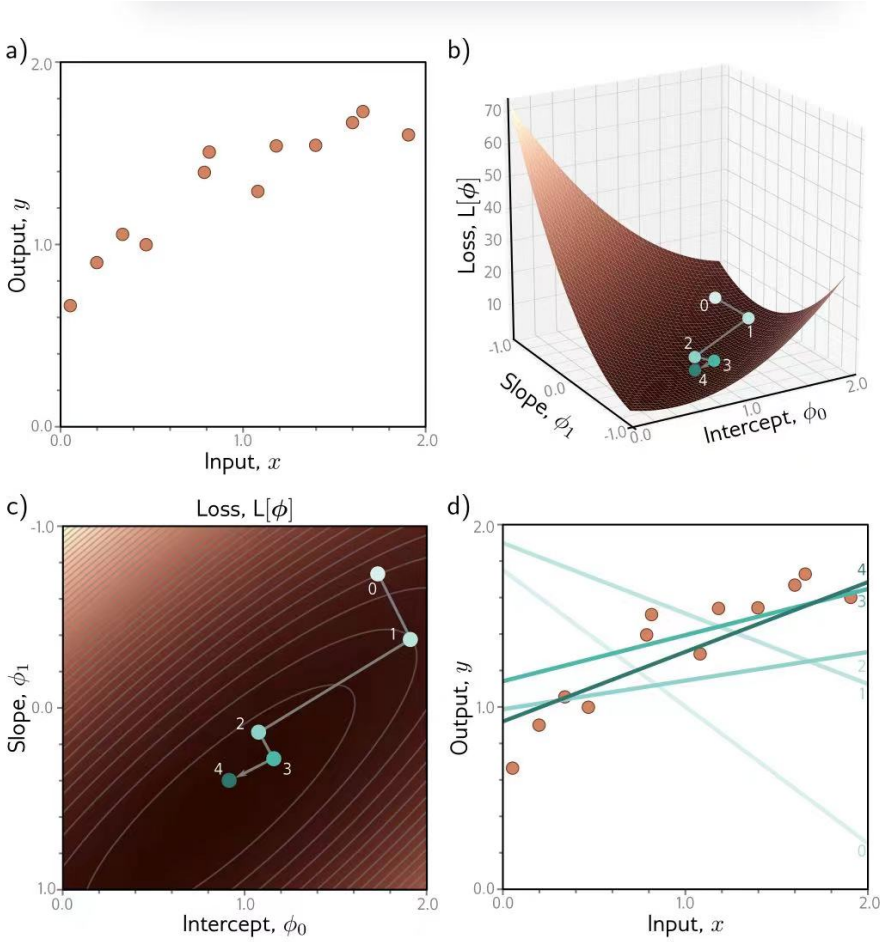


Figure 6.1 Gradient descent for the linear regression model. a) Training set of $I = 12$ input/output pairs $\{x_i, y_i\}$. b) Loss function showing iterations of gradient descent. We start at point 0 and move in the steepest downhill direction until we can improve no further to arrive at point 1. We then repeat this procedure. We measure the gradient at point 1 and move downhill to point 2 and so on. c) This can be visualized better as a heatmap, where the brightness represents the loss. After only four iterations, we are already close to the minimum. d) The model with the parameters at point 0 (lightest line) describes the data very badly, but each successive iteration improves the fit. The model with the parameters at point 4 (darkest line) is already a reasonable description of the training data.

However, problems might exist since not all the functions are convex. Therefore, the gradient descent might stuck in local minima and saddle points. To tackle this, we add noises to the gradient descent to create **Stochastic gradient descent (SGD)**.

Now, at each iteration, the algorithm choose a random subset of the training data and computes the gradient from these examples alone. This subset is known as a **minibatch** or **batch** for short.
Now the iteration and graph should be:

$$\boldsymbol{\phi}_{t+1} \longleftarrow \boldsymbol{\phi}_t - \alpha \cdot \sum_{i \in \mathcal{B}_t} \frac{\partial \ell_i[\boldsymbol{\phi}_t]}{\partial \boldsymbol{\phi}},$$

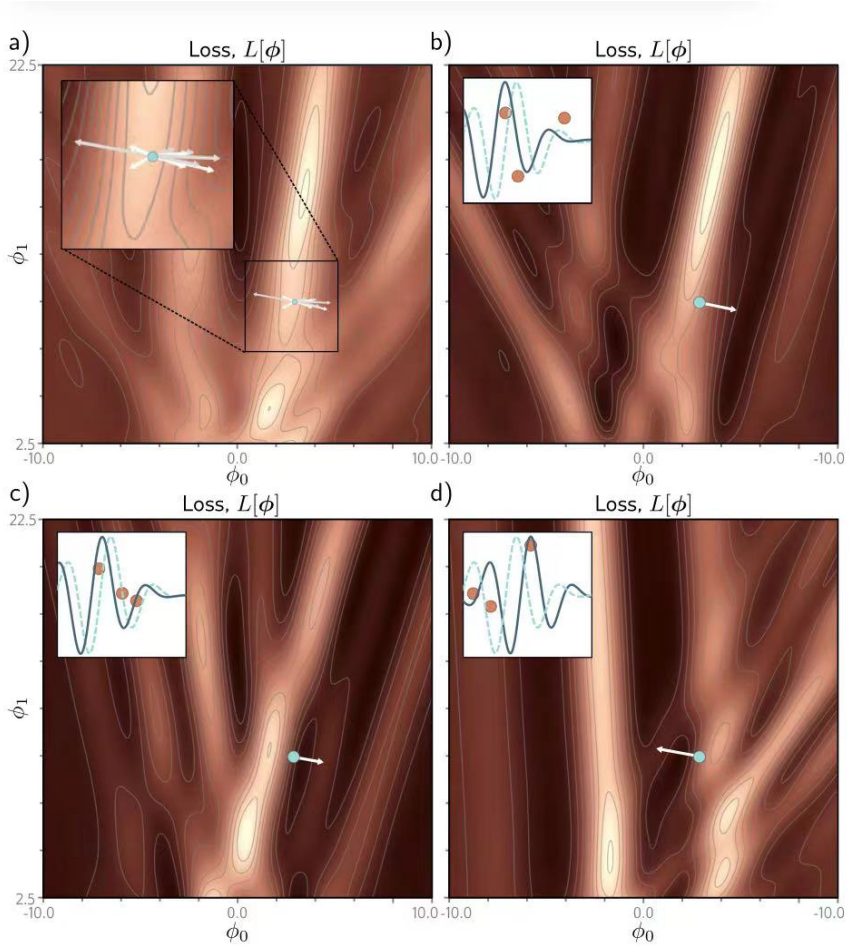


Figure 6.6 Alternative view of SGD for the Gabor model with a batch size of three. a) Loss function for the entire training dataset. At each iteration, there is a probability distribution of possible parameter changes (inset shows samples). These correspond to different choices of the three batch elements. b) Loss function for one possible batch. The SGD algorithm moves in the downhill direction on this function for a distance that is determined by the learning rate and the local gradient magnitude. The current model (dashed function in inset) changes to better fit the batch data (solid function). c) A different batch creates a different loss function and results in a different update. d) For this batch, the algorithm moves *downhill* with respect to the batch loss function but *uphill* with respect to the global loss function in panel (a). This is how SGD can escape local minima.

Then, we want to make the iteration smoother and predict what the next step should be based on the previous steps. We add **momentum term** to the gradient descent.
We get equations and figures like this:

$$\begin{aligned}\mathbf{m}_{t+1} &\leftarrow \beta \cdot \mathbf{m}_t + (1-\beta)\sum_{i\in\mathcal{B}_t}\frac{\partial \ell_i[\boldsymbol{\phi}_t]}{\partial \boldsymbol{\phi}} \\ \boldsymbol{\phi}_{t+1} &\leftarrow \boldsymbol{\phi}_t - \alpha \cdot \mathbf{m}_{t+1},\end{aligned}$$

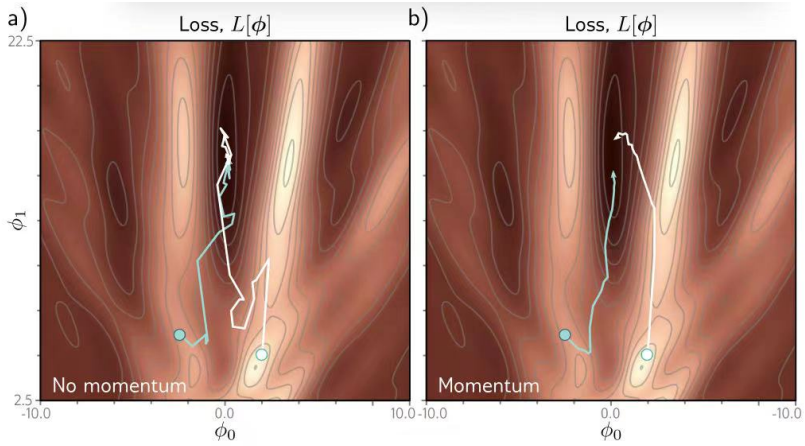


Figure 6.7 Stochastic gradient descent with momentum. a) Regular stochastic descent takes a very indirect path toward the minimum. b) With a momentum term, the change at the current step is a weighted combination of the previous change and the gradient computed from the batch. This smooths out the trajectory and increases the speed of convergence.

We can also optimize this equation more to compute the gradients at this predicted point rather than at the current point, using **Nesterov accelerated momentum**:

$$\begin{aligned}\mathbf{m}_{t+1} &\leftarrow \beta \cdot \mathbf{m}_t + (1-\beta)\sum_{i\in\mathcal{B}_t}\frac{\partial \ell_i[\boldsymbol{\phi}_t - \alpha\beta\cdot\mathbf{m}_t]}{\partial \boldsymbol{\phi}} \\ \boldsymbol{\phi}_{t+1} &\leftarrow \boldsymbol{\phi}_t - \alpha \cdot \mathbf{m}_{t+1},\end{aligned}$$

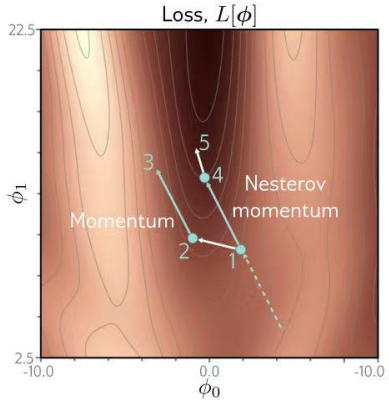


Figure 6.8 Nesterov accelerated momentum. The solution has traveled along the dashed line to arrive at point 1. A traditional momentum update measures the gradient at point 1, moves some distance in this direction to point 2, and then adds the momentum term from the previous iteration (i.e., in the same direction as the dashed line), arriving at point 3. The Nesterov momentum update first applies the momentum term (moving from point 1 to point 4) and then measures the gradient and applies an update to arrive at point 5.

However, problems still exist since at each step, we move fast when the gradient is steep and move slowly when the gradient is small. We want to move in the middle instead.
Then we try to normalize the gradients to move a fixed distance:

$$\begin{aligned}\mathbf{m}_{t+1} &\leftarrow \frac{\partial L[\boldsymbol{\phi}_t]}{\partial \boldsymbol{\phi}} \\ \mathbf{v}_{t+1} &\leftarrow \left(\frac{\partial L[\boldsymbol{\phi}_t]}{\partial \boldsymbol{\phi}}\right)^2.\end{aligned}$$

Then we apply the update rule:

$$\boldsymbol{\phi}_{t+1} \leftarrow \boldsymbol{\phi}_t - \alpha \cdot \frac{\mathbf{m}_{t+1}}{\sqrt{\mathbf{v}_{t+1}} + \epsilon},$$

This simple algorithm makes good progress in both directions but will not converge unless it happens to land exactly at the minimum.
Then, the **Adaptive moment estimation**, **Adam**, takes this idea and adds momentum to both estimate the gradient and the squared gradient:

$$\begin{aligned}\mathbf{m}_{t+1} &\leftarrow \beta \cdot \mathbf{m}_t + (1-\beta)\frac{\partial L[\boldsymbol{\phi}_t]}{\partial \boldsymbol{\phi}} \\ \mathbf{v}_{t+1} &\leftarrow \gamma \cdot \mathbf{v}_t + (1-\gamma)\left(\frac{\partial L[\boldsymbol{\phi}_t]}{\partial \boldsymbol{\phi}}\right)^2,\end{aligned}$$

$$\begin{aligned}\tilde{\mathbf{m}}_{t+1} &\leftarrow \frac{\mathbf{m}_{t+1}}{1-\beta^{t+1}} \\ \tilde{\mathbf{v}}_{t+1} &\leftarrow \frac{\mathbf{v}_{t+1}}{1-\gamma^{t+1}}.\end{aligned}$$

$$\boldsymbol{\phi}_{t+1} \leftarrow \boldsymbol{\phi}_t - \alpha \cdot \frac{\tilde{\mathbf{m}}_{t+1}}{\sqrt{\tilde{\mathbf{v}}_{t+1}} + \epsilon}.$$

$$\begin{aligned}\mathbf{m}_{t+1} &\leftarrow \beta \cdot \mathbf{m}_t + (1-\beta)\sum_{i \in \mathcal{B}_t} \frac{\partial \ell_i[\boldsymbol{\phi}_t]}{\partial \boldsymbol{\phi}} \\ \mathbf{v}_{t+1} &\leftarrow \gamma \cdot \mathbf{v}_t + (1-\gamma)\sum_{i \in \mathcal{B}_t} \left(\frac{\partial \ell_i[\boldsymbol{\phi}_t]}{\partial \boldsymbol{\phi}}\right)^2,\end{aligned}$$

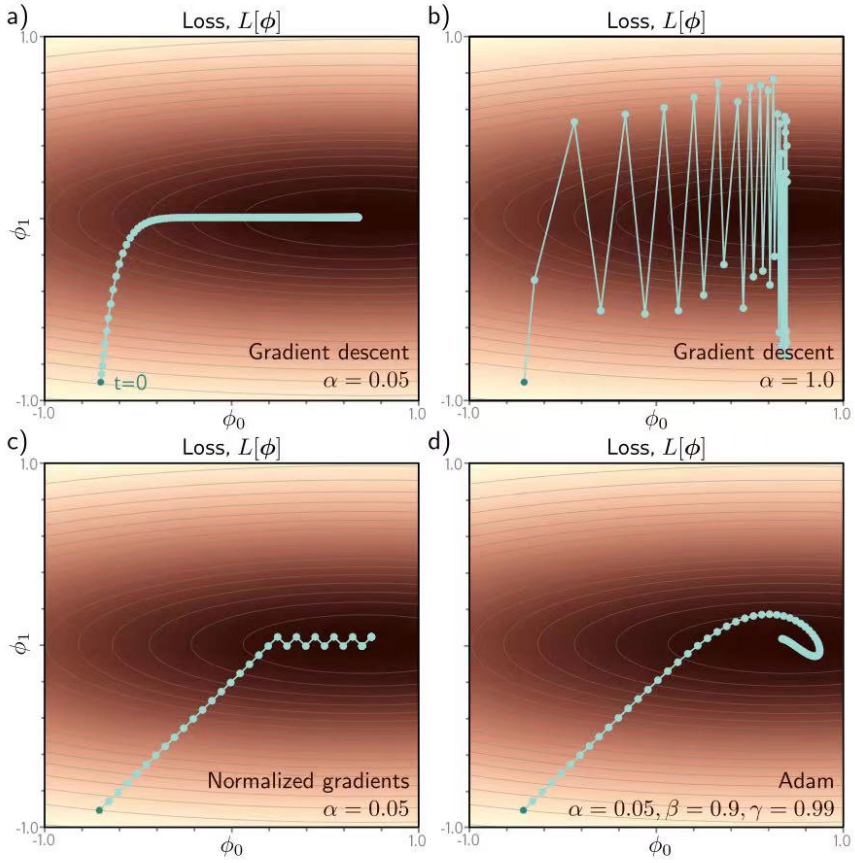


Figure 6.9 Adaptive moment estimation (Adam). a) This loss function changes quickly in the vertical direction but slowly in the horizontal direction. If we run full-batch gradient descent with a learning rate that makes good progress in the vertical direction, then the algorithm takes a long time to reach the final horizontal position. b) If the learning rate is chosen so that the algorithm makes good progress in the horizontal direction, it overshoots in the vertical direction and becomes unstable. c) A straightforward approach is to move a fixed distance along each axis at each step so that we move downhill in both directions. This is accomplished by normalizing the gradient magnitude and retaining only the sign. However, this does not usually converge to the exact minimum but instead oscillates back and forth around it (here between the last two points). d) The Adam algorithm uses momentum in both the estimated gradient and the normalization term, which creates a smoother path.

Notice: The most commonly used optimation algorithm for training neural networks is SGD, as is shown before.

4.2 The computation of the gradient

We use the **backpropagation algorithm** to comput the gradient efficiently.
Since there are two kind of parameters $\beta_{\mathbf{k}}$ and $\mathbf{\Omega_k}$, we need to compute $\{\beta_{\mathbf{k}}, \mathbf{\Omega_k}\}$ at every layer. We can have two observations:
i) Since each hidden unit connect the next layer, be weighted and adds something. Then we can store the previous one to facilitate the calculation, which is known as the **forward pass**.
ii) As we move backward the network, we see that most of the terms we need were already calculate in the previous steps, so we don't need to re-compute them. This is known as the **backward pass**.
We use a simpler equations to better illustrate this:

Forward pass: We treat the computation of the loss as a series of calculations:

$$\begin{aligned} f_0 &= \beta_0 + \omega_0 \cdot x_i \\ h_1 &= \sin[f_0] \\ f_1 &= \beta_1 + \omega_1 \cdot h_1 \\ h_2 &= \exp[f_1] \\ f_2 &= \beta_2 + \omega_2 \cdot h_2 \\ h_3 &= \cos[f_2] \\ f_3 &= \beta_3 + \omega_3 \cdot h_3 \\ \ell_i &= (f_3 - y_i)^2. \end{aligned} \tag{7.8}$$

We compute and store the values of the intermediate variables f_k and h_k (figure 7.3).

Backward pass #1: We now compute the derivatives of ℓ_i with respect to these intermediate variables, but in reverse order:

$$\frac{\partial \ell_i}{\partial f_3}, \quad \frac{\partial \ell_i}{\partial h_3}, \quad \frac{\partial \ell_i}{\partial f_2}, \quad \frac{\partial \ell_i}{\partial h_2}, \quad \frac{\partial \ell_i}{\partial f_1}, \quad \frac{\partial \ell_i}{\partial h_1}, \quad \text{and} \quad \frac{\partial \ell_i}{\partial f_0}. \tag{7.9}$$

The first of these derivatives is straightforward:

$$\frac{\partial \ell_i}{\partial f_3} = 2(f_3 - y_i). \tag{7.10}$$

The next derivative can be calculated using the chain rule:

$$\frac{\partial \ell_i}{\partial h_3} = \frac{\partial f_3}{\partial h_3}\frac{\partial \ell_i}{\partial f_3}. \tag{7.11}$$

$$\begin{aligned} \frac{\partial \ell_i}{\partial f_2} &= \frac{\partial h_3}{\partial f_2}\left(\frac{\partial f_3}{\partial h_3}\frac{\partial \ell_i}{\partial f_3}\right) \\ \frac{\partial \ell_i}{\partial h_2} &= \frac{\partial f_2}{\partial h_2}\left(\frac{\partial h_3}{\partial f_2}\frac{\partial f_3}{\partial h_3}\frac{\partial \ell_i}{\partial f_3}\right) \\ \frac{\partial \ell_i}{\partial f_1} &= \frac{\partial h_2}{\partial f_1}\left(\frac{\partial f_2}{\partial h_2}\frac{\partial h_3}{\partial f_2}\frac{\partial f_3}{\partial h_3}\frac{\partial \ell_i}{\partial f_3}\right) \\ \frac{\partial \ell_i}{\partial h_1} &= \frac{\partial f_1}{\partial h_1}\left(\frac{\partial h_2}{\partial f_1}\frac{\partial f_2}{\partial h_2}\frac{\partial h_3}{\partial f_2}\frac{\partial f_3}{\partial h_3}\frac{\partial \ell_i}{\partial f_3}\right) \\ \frac{\partial \ell_i}{\partial f_0} &= \frac{\partial h_1}{\partial f_0}\left(\frac{\partial f_1}{\partial h_1}\frac{\partial h_2}{\partial f_1}\frac{\partial f_2}{\partial h_2}\frac{\partial h_3}{\partial f_2}\frac{\partial f_3}{\partial h_3}\frac{\partial \ell_i}{\partial f_3}\right). \end{aligned}$$

Backward pass #2: Finally, we consider how the loss ℓ_i changes when we change the parameters $\beta_\bullet$ and $\omega_\bullet$. Once more, we apply the chain rule (figure 7.5):

$$\begin{aligned}\frac{\partial \ell_i}{\partial \beta_k} &= \frac{\partial f_k}{\partial \beta_k}\frac{\partial \ell_i}{\partial f_k} \\ \frac{\partial \ell_i}{\partial \omega_k} &= \frac{\partial f_k}{\partial \omega_k}\frac{\partial \ell_i}{\partial f_k}.\end{aligned} \tag{7.13}$$

In each case, the second term on the right-hand side was computed in equation 7.12. When $k > 0$, we have $f_k = \beta_k + \omega_k \cdot h_k$, so:

$$\frac{\partial f_k}{\partial \beta_k} = 1 \qquad \text{and} \qquad \frac{\partial f_k}{\partial \omega_k} = h_k. \tag{7.14}$$

And here is the figure to help to understand:

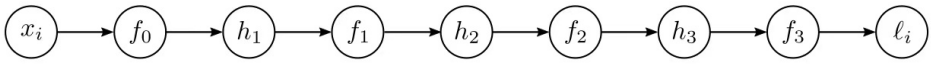


Figure 7.3 Backpropagation forward pass. We compute and store each of the intermediate variables in turn until we finally calculate the loss.

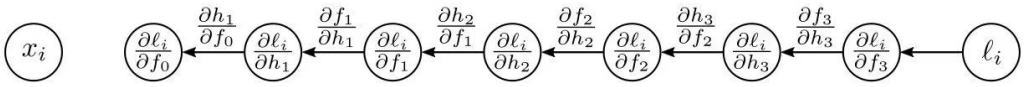


Figure 7.4 Backpropagation backward pass #1. We work backward from the end of the function computing the derivatives $\partial \ell_i / \partial f_\bullet$ and $\partial \ell_i / \partial h_\bullet$ of the loss with respect to the intermediate quantities. Each derivative is computed from the previous one by multiplying by terms of the form $\partial f_k / \partial h_k$ or $\partial h_k / \partial f_{k-1}$.

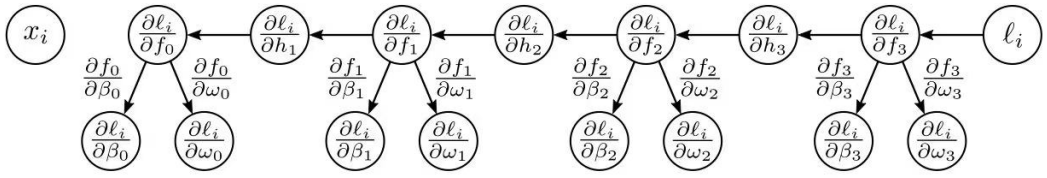


Figure 7.5 Backpropagation backward pass #2. Finally, we compute the derivatives $\partial \ell_i / \partial \beta_\bullet$ and $\partial \ell_i / \partial \omega_\bullet$. Each derivative is computed by multiplying the term $\partial \ell_i / \partial f_k$ by $\partial f_k / \partial \beta_k$ or $\partial f_k / \partial \omega_k$ as appropriate.

However, in reality, it is the matrix calculus and it won't be further explained here. I'll only put the general formula and you can read the book if you like.

Forward pass: We compute and store the following quantities:

$$\begin{aligned}
\mathbf{f}_0 &= \boldsymbol{\beta}_0 + \boldsymbol{\Omega}_0 \mathbf{x}_i \\
\mathbf{h}_k &= \mathbf{a}[\mathbf{f}_{k-1}] \qquad\qquad k \in \{1, 2, \ldots, K\} \\
\mathbf{f}_k &= \boldsymbol{\beta}_k + \boldsymbol{\Omega}_k \mathbf{h}_k. \qquad\qquad k \in \{1, 2, \ldots, K\}
\end{aligned} \tag{7.23}$$

Backward pass: We start with the derivative $\partial \ell_i / \partial \mathbf{f}_K$ of the loss function ℓ_i with respect to the network output $\mathbf{f}_K$ and work backward through the network:

$$\begin{aligned}
\frac{\partial \ell_i}{\partial \boldsymbol{\beta}_k} &= \frac{\partial \ell_i}{\partial \mathbf{f}_k} \qquad\qquad k \in \{K, K-1, \ldots, 1\} \\
\frac{\partial \ell_i}{\partial \boldsymbol{\Omega}_k} &= \frac{\partial \ell_i}{\partial \mathbf{f}_k} \mathbf{h}_k^T \qquad\qquad k \in \{K, K-1, \ldots, 1\} \\
\frac{\partial \ell_i}{\partial \mathbf{f}_{k-1}} &= \mathbb{I}[\mathbf{f}_{k-1} > 0] \odot \left(\boldsymbol{\Omega}_k^T \frac{\partial \ell_i}{\partial \mathbf{f}_k} \right), \qquad k \in \{K, K-1, \ldots, 1\}
\end{aligned} \tag{7.24}$$

where $\odot$ denotes pointwise multiplication, and $\mathbb{I}[\mathbf{f}_{k-1} > 0]$ is a vector containing ones where $\mathbf{f}_{k-1}$ is greater than zero and zeros elsewhere. Finally, we compute the derivatives with respect to the first set of biases and weights:

$$\begin{aligned}
\frac{\partial \ell_i}{\partial \boldsymbol{\beta}_0} &= \frac{\partial \ell_i}{\partial \mathbf{f}_0} \\
\frac{\partial \ell_i}{\partial \boldsymbol{\Omega}_0} &= \frac{\partial \ell_i}{\partial \mathbf{f}_0} \mathbf{x}_i^T.
\end{aligned} \tag{7.25}$$

We calculate these derivatives for every training example in the batch and sum them together to retrieve the gradient for the SGD update.

5 Initialization and Performance

Actually, things like initializing the initial parameter and the discussion about the performance of the model are also important. However, due to the time limit, they will not be explored further here.

6 Reference

The content of this worksheet is all from the book ¡Understanding Deep Learning¿ by Simon J.D. Prince. The most recent version of this book can be found at http://udlbook.com.