Lecture 4
Training Neural Networks: Some Basics

By
Dr. Muhammad Alrabeiah
Electrical Engineering Dept., KSU
Spring 2022

February 18th, 2022

Disclaimer These notes are still under development, and they have not been subjected to proper review and revision.

Contents

Chapter 1

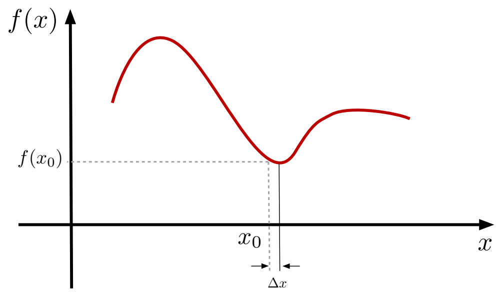
Iterative Training

Moving to neural networks gives us the ability to represent more complex functions, as the universal approximation theorem suggests. Nevertheless, this ability comes with a price; we no longer have the ability to derive a closed-form solution for the network parameters that achieves optimal performance, like we do in linear regression. This could be seen as a consequence of the layered architecture that contains several non-linearities. Try to write the input-output relation for a three-layer neural network with breadths greater than 3, and see how complex it gets!

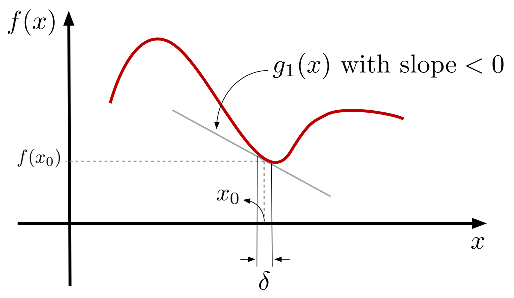
Now if we cannot get a close-form solution, how can we learn the target function and find the optimal set of parameters? This is the core of this chapter. The short answer is "we follow an iterative training approach". It is an implementation of some calculus principles. We will start with a brief review of relevant concepts, like derivatives, chain rule, gradient, and so forth. Then, we will move from there to discuss gradient descent which is the core for most iterative training approaches.

1 A Blast from The Past

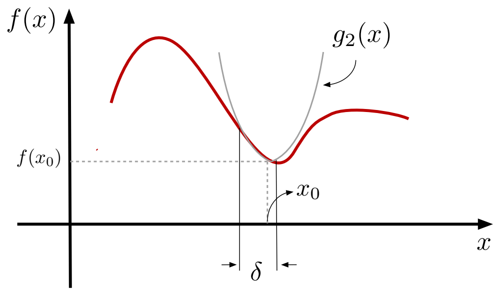
Below, we review a few concepts from calculus that we will use when we discuss iterative training and training neural networks.



(a)



(b)



(c)

1.1 Taylor Series Approximation

Let $f(.)$ be a continuous function defined as $f : x \in \mathbb{R} \to y \in \mathbb{R}$ and is infinitely differentiable, i.e., the derivatives $f^{(1)}(x), f^{(2)}(x), \ldots, f^{(k)}(x)$ for $k \to \infty$ all exists. Then, at any point $x = x_0$, the function can be approximated within a neighborhood around x_0, $f(x_0 + \Delta x)$, using Taylor series as follows

$$f(x_0 + \Delta x) = f(x_0) + f^{(1)}(x_0)\Delta x + f^{(2)}(x_0)\frac{(\Delta x)^2}{2} + \cdots + f^{(k)}(x_0)\frac{(\Delta x)^k}{k!}. \tag{1.1}$$

The series in Equation 1.1 says that a function can be approximated to an arbitrary accuracy by increasing the order of the derivatives. If we are interested in a small neighborhood Δx around some point x_0, then the magnitude of the high order terms will rapidly decay, leaving us with a small number of terms. For instance, we could approximate $f(x_0 + \Delta x)$ using the linear part of the series (called first-order Taylor approximation)

$$f(x_0 + \Delta x) = f(x_0) + f^{(1)}(x_0)\Delta x, \tag{1.2}$$

which is illustrated in Figure 1.1b. We can even go further and use the quadratic part of the series to approximate the function (called second-order Taylor approximation)

$$f(x_0 + \Delta x) = f(x_0) + f^{(1)}(x_0)\Delta x + f^{(2)}(x_0)\frac{(\Delta x)^2}{2}, \tag{1.3}$$

see Figure 1.1c. Adding more terms can obviously result in better approximation, but in the same time, it brings about an increase in computations. In this lecture, we will focus on the linear approximation, for it is the foundation of the famous *gradient descent algorithm.*

1.2 Gradient and Jacobian

The discussion about Taylor series above could be extended to scalar-valued and vector-valued functions of multiple variables $f : \mathbf{x} \in \mathbb{R}^N \to y \in \mathbb{R}$ and $\mathbf{f} : \mathbf{x} \in \mathbb{R}^N \to \mathbf{y} \in \mathbb{R}^M$, respectively. To do so, we need two new mathematical operator, which are the *gradient* and *Jacobian.*

For a scalar-valued function, the gradient is given by

$$\bigtriangledown f(\mathbf{x}) = \left[\frac{\partial f(\mathbf{x})}{\partial x_1}, \dots, \frac{\partial f(\mathbf{x})}{\partial x_N}\right]^T. \tag{1.4}$$

It is important at this point to remember two facts about the derivative of 1-D variable: (i) the absolute value of the derivative at a point reflects the steepness of the function, and (ii) the sign of the derivative indicates the direction along which the function increases. Both are illustrated in Figure 1.1b. Similar to the derivative w.r.t. one variable, the magnitude of the gradient encodes information about the function rate of change (steepness), and its direction points to the direction along which $f(\mathbf{x})$ increases.

For a vector-valued function, on the other hand, the first order derivative is given by the

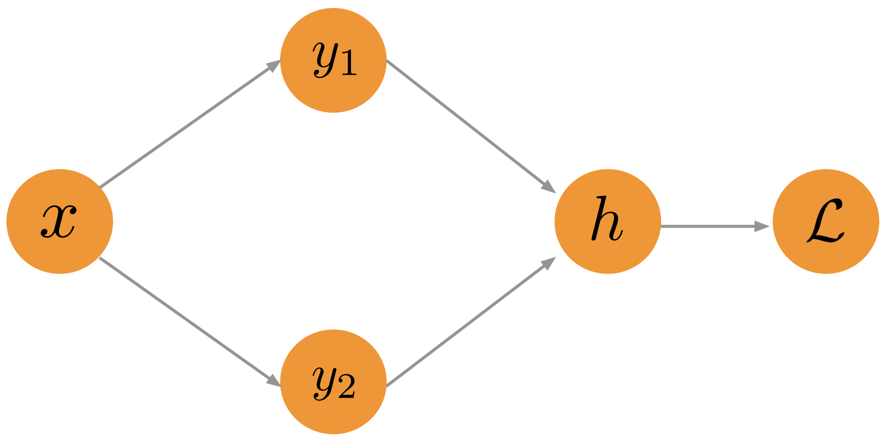


Figure 1.2: An example of an augmented function represented as a *graphical model*.

Jacobian matrix

$$\mathbf{J}(\mathbf{x}) = \frac{\partial \mathbf{f}(\mathbf{x})}{\partial \mathbf{x}} = \left[\frac{\partial \mathbf{f}(\mathbf{x})}{\partial \mathbf{x}_1}, \ldots, \frac{\partial \mathbf{f}(\mathbf{x})}{\partial \mathbf{x}_N}\right] \tag{1.5}$$

$$= \begin{bmatrix} \bigtriangledown f_1^T(\mathbf{x}) \\ \bigtriangledown f_2^T(\mathbf{x}) \\ \vdots \\ \bigtriangledown f_M^T(\mathbf{x}) \end{bmatrix} \tag{1.6}$$

$$= \begin{bmatrix} \frac{\partial f_1(\mathbf{x})}{\partial x_1} & \cdots & \frac{\partial f_1(\mathbf{x})}{\partial x_N} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_M(\mathbf{x})}{\partial x_1} & \cdots & \frac{\partial f_M(\mathbf{x})}{\partial x_N} \end{bmatrix}. \tag{1.7}$$

Using the two definitions above, the first-order approximation of a scalar-valued function is given by

$$f(\mathbf{x}_0 + \Delta\mathbf{x}) = f(\mathbf{x}_0) + \bigtriangledown f(\mathbf{x}_0)^T \Delta\mathbf{x}, \tag{1.8}$$

and similarly the first-order approximation of a vector-valued function looks something like the following

$$\mathbf{f}(\mathbf{x}_0 + \Delta\mathbf{x}) = \mathbf{f}(\mathbf{x}_0) + \mathbf{J}(\mathbf{x}_0)\Delta\mathbf{x}, \tag{1.9}$$

1.3 Chain-Rule of Derivatives

For $x \in \mathbb{R}$, augmented functions $(f \circ g \circ h)(x)$ could be differentiated with respect to a variable x using the chain rule of derivatives as follows

$$\frac{\mathrm{d}f}{\mathrm{d}x} = \frac{\mathrm{d}f}{\mathrm{d}g}\frac{\mathrm{d}g}{\mathrm{d}h}\frac{\mathrm{d}h}{\mathrm{d}x}. \tag{1.10}$$

This rule is quite important and very handy to our work in deep learning, for it constitutes the foundation of *the back-propagation algorithm*, which we will explore in Lecture 5. This rule could be used with more complex augmented functions, especially those with vector variables. Let's consider the example in Figure 1.2. It is a simplified case to what we will face later with neural networks. The derivative of $\mathcal{L}$ w.r.t. x is given by

$$\frac{\partial \mathcal{L}}{\partial x} = \frac{\partial \mathcal{L}}{\partial h}\left[\frac{\partial h}{\partial y_1}\frac{\partial y_1}{\partial x} + \frac{\partial h}{\partial y_2}\frac{\partial y_2}{\partial x}\right]. \tag{1.11}$$

Notice how the derivative of h w.r.t. y_1 and y_2 is computed and related to x. We can think of this example as an illustration of how to differentiate function that have scalar and vector variables. Equation 1.11 could be re-written as follows

$$\frac{\partial \mathcal{L}}{\partial x} = \frac{\partial \mathcal{L}}{\partial h}\begin{bmatrix}\frac{\partial h}{\partial y_1} & \frac{\partial h}{\partial y_2}\end{bmatrix}\begin{bmatrix}\frac{\partial y_1}{\partial x} \\ \frac{\partial y_2}{\partial x}\end{bmatrix}. \tag{1.12}$$

The variables y_1 and y_2 form a vector $\mathbf{y}$. Hence, the second factor is the derivatives of $\frac{\partial h}{\partial \mathbf{y}}$, which is the gradient of h w.r.t. $\mathbf{y}$. On the other hand, the third factor is the derivative $\frac{\partial \mathbf{y}}{\partial x}$, which represents the Jacobian of a vector-valued function, a.k.a. $\mathbf{y}$, of one independent variable, a.k.a. x.

2 Iterative Training: Gradient Descent Algorithm

As we discussed at the beginning of this chapter, training neural networks requires an iterative training approach, for we cannot derive a close-form solution for the optimal parameters. To learn how to train neural networks, let's start by posing and answering the following question: *what do we mean by iterative training?*

2.1 Motivation for Iterative Training

Let's digress a bit and point out a couple of interesting observations As we learned from Lecture 3, a neural network could be mathematically represented as a function composed of a sequence of augmented functions like this

$$\mathbf{y} = \mathbf{f}(\mathbf{x}) = (\varphi_L \circ \varphi_{L-1} \circ \cdots \circ \varphi_1)(\mathbf{x}), \tag{1.13}$$

all of which are parameterize by its own matrix $\mathbf{W}$ and $\mathbf{b}$. This function $\mathbf{f}$ has two types of independent variables. The first type represents all the parameters of the network, which could be put in one set named Θ, and the second is the input variables to the network $\mathbf{x}$, which we have been calling observed variables. Equation 1.13 emphasizes that neural networks are functions in their inputs as well as their parameters. The latter will be utilized to explain stochastic gradient descent while the former is the subject of the training process.

The objective of training a neural network is to minimize some loss function w.r.t. the *parameters of the network*, not its inputs. This is because the parameters are what we can control (remember the input is just some observed variables). Since it is difficult to express the relation between the output and input in neural networks, the relation between the loss function and the parameters is also difficult to express. This complicates the differentiation of the loss w.r.t. the parameters[1], and it makes solving for the optimal parameters almost impossible. In other words, we cannot have a closed-form solution for the parameters like we have in linear regression and classification.

2.2 The Gradient Descent Algorithm

Let's get back to our question. Instead of seeking a closed-form solution for neural networks, we will follow an iterative approach to find those parameters (i.e., optimal Θ) that minimize the loss. Form the name, we can guess that iterative training is a repetitive approach, which is true. What we do in this approach is that we implement the following steps:

1. Pass the input through the network and get the predictions.

2. Use the groundtruth desired response to measure the error of the predictions.

[1]This complexity will be alleviated by the chain rule of derivative as we will see in Lecture 5, inshallah

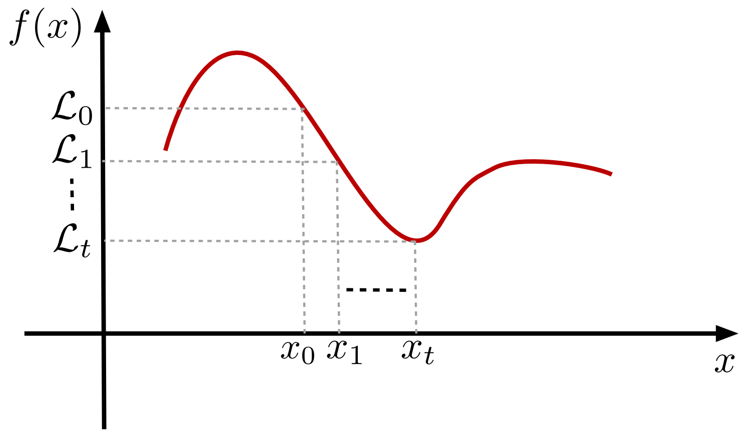


Figure 1.3: Example of iteratively minimizing $\mathcal{L}$

3. Utilize the error to *adjust* the parameters of the network so that the predictions improves,

over and over, until optimal parameters are found (equivalently the loss is minimized). The three steps above are the ballpark of iterative training, and we will learn how to implement all of them in Lecture 5. However, for now, we are interested in laying the groundwork for training neural networks. As such, we will look at the problem of iteratively minimizing some function $\mathcal{L}$ w.r.t some variable x.

Assume the function relating $\mathcal{L}$ and x is that depicted in Figure 1.3, and say that we are at point $x = x_0$ where $\mathcal{L} = f(x_0)$. Iteratively minimizing $\mathcal{L}$ means we need to find the sequence of points $s_{min} = \{x_0, x_1, x_2, \dots, x_t\}$ such that the following inequality holds

$$\mathcal{L}_t \leq \cdots \leq \mathcal{L}_2 \leq \mathcal{L}_1 \leq \mathcal{L}_0. \tag{1.14}$$

To do so, we will start with point x_0 and write the first-order Taylor approximation

$$f(x_0 + \Delta x) = f(x_0) + f^{(1)}(x_0)\Delta x, \tag{1.15}$$

We want to find Δx that takes us to $x_1 = x_0 + \Delta x$ such that $f(x_0) > f(x_1)$ (i.e., $\mathcal{L}_0 > \mathcal{L}_1$). Recall from Section 1.2 that the sign of the first derivative at any point, say x_0, indicates the direction along which the function increases. Hence, let's say we want our Δx to be a scaled version of the derivative at point x_0 but with a negative sign. More specifically, let

$\Delta x = -\eta f^{(1)}(x_0)$ where $\eta > 0$, and substitute back into Equation 1.15 to get

$$f(x_0 - \eta f^{(1)}(x_0)) = f(x_0) + f^{(1)}(x_0)(-\eta f^{(1)}(x_0)) \tag{1.16}$$

$$f(x_0 + \eta f^{(1)}(x_0)) - f(x_0) = -\eta \left(f^{(1)}(x_0)\right)^2 \leq 0 \tag{1.17}$$

Equation 1.17 shows that setting $\Delta x = -\eta f^{(1)}(x_0)$ guarantees that we are moving in the direction reducing $f(x_0 + \Delta x)$. This choice for the increment Δx can be repeated at every point $x_i \in s_{min}$ to find a new point and get a smaller $\mathcal{L}_i = f(x_i)$. Therefore, we can derive a general rule for minimizing a function, which is

$$x_i = x_{i-1} - \eta f^{(1)}(x_{i-1}). \tag{1.18}$$

The algorithm that repeatedly implements Equation 1.18 or a similar rule relying on the gradient until the loss is minimized is called *the gradient descent algorithm*.

Now, the discussion above has focused on functions of a single variable, but the final algorithm is also applicable for functions of multiple variables. In machine learning we commonly pick loss functions of the form $f : \mathbf{y} \in \mathbb{R}^M \to \mathcal{L} \in \mathbb{R}$. Therefore, the rule in Equation 1.18 is adjusted using the gradient to become

$$\mathbf{x}_i = \mathbf{x}_{i-1} - \eta \bigtriangledown f(\mathbf{x}_i), \tag{1.19}$$

and the resulting minimizing sequence is $s_{min} = \{\mathbf{x}_0, \dots, \mathbf{x}_t\}$. Something similar to Equation 1.19 could be derived for minimizing vector-valued functions.

Remark: the scalar η is referred to as the *learning rate* in the field of machine learning and the *step size* in the field of optimization theory. It is a hyper-parameter for the training algorithm, and it determines how large a step the descent algorithm is taking. Since we are relying on first-order approximations, it is justifiable to expect η to be quite small.

3 Mini-Batch Training

Training neural networks usually requires tremendous amount of data, i.e., large training dataset. For example, the popular ImageNet [1] dataset used for training image recognition networks contains an excess of 1.4 million images occupying more than 100 Giga-Byte (GB) of storage space. Therefore, when training a neural network, we cannot feed the whole dataset into the network to get the predictions, step 1 in the list of Section 2.2. We instead *randomly select a subset of data points* from the dataset, $\mathcal{B} \subset \mathcal{D}$ such that $|\mathcal{B}| << |\mathcal{D}|$, and feed it to the network. This subset of data-point is commonly referred to as the *mini-batch* and training with such subsets is called *mini-batch training. Every training iteration in mini-batch training consists of applying the three steps in Section 2.2 with a single mini-batch.*

The reason behind the random selection is rooted in statistics and probability theory. A sample from a population could convey all important information about the population given that its size is sufficient. We are not going to dig deeper here, and discuss what is a sufficient size. We will, instead, conclude by saying "the bigger the mini-batch size is, the better," and we will see a justification for that in the next Section.

4 Stochastic Gradient Descent

As we discussed earlier, the neural network is a function of two types of variables, observed variables and parameters. In training, we are interested in minimizing the loss function w.r.t. the parameters given a set of data points. Assuming the use of gradient descent to train the network, the update rule for the any parameter is given by

$$w_i^{(t)} = w_i^{(t-1)} - \eta \frac{1}{U} \sum_{u=1}^{U} \frac{\partial \ell_u}{\partial w_i} |_{w_i = w_i^{(t-1)}} \tag{1.20}$$

where $w_i^{(t)}$ is the value of the i-th parameter in the network at the t-th iteration, $\frac{1}{U}\sum_{u=1}^{U}\frac{\partial \ell}{\partial w_i}$ is the derivative of the loss $\mathcal{L}$ w.r.t. w_i, and $\mathcal{L} = (1/U)\sum_{u=1}^{U}\ell_u$ with an error metric ℓ. Notice how the gradient in Equation 1.20 is computed as a sum over all data points in the training set. This is an important observation when it is paired with what we have learned about mini-batch training; the actual gradient cannot really be computed in mini-batch training, for we do not feed all data points. Instead, we compute an approximation for that gradient from the samples of the mini-batch

$$w_i^{(t)} = w_i^{(t-1)} - \eta \frac{1}{|\mathcal{B}|} \sum_{u=1}^{|\mathcal{B}|} \frac{\partial \ell_u}{\partial w_i} |_{w_i = w_i^{(t-1)}} \tag{1.21}$$

This approximation results in what is commonly referred to as *stochastic gradient descent*. The name refers to the fact that the mini-batch is randomly sampled from the training dataset, which makes the estimate of the gradient itself random.

Chapter 2

Machine Learning Basics

In Lectures 1 and 2, we have learned about two fundamental supervised learning tasks, regression and classification, and we have focused on developing machine learning algorithms that rely on linear models. With neural networks, we are expanding our pool of models to include non-linear ones. This allows us to address complex real-world problems, which is the main skill we aim to develop in this course. Before we embark upon that journey with neural networks, we have now some basic understanding of supervised machine learning such that we can discuss a few important machine learning concepts. Those are model capacity, overfitting, underfitting, and generalization.

1 Hypothesis Space and Model Capacity

What does a learning algorithm do exactly? This is a quite broad question, but it is equally important. We have addressed it somehow from a practical perspective—within the realm of supervised learning, of course; it learns a parameterized function approximating a target function, a decision boundary or a relation between two sets of variables. Anyhow, we will attempt to develop a more abstract and formal answer here.

1.1 Hypothesis Space

Let's recall how we addressed both the regression and classification tasks. We started with an assumption that our task could be handled with linear models. Then, we proceeded to

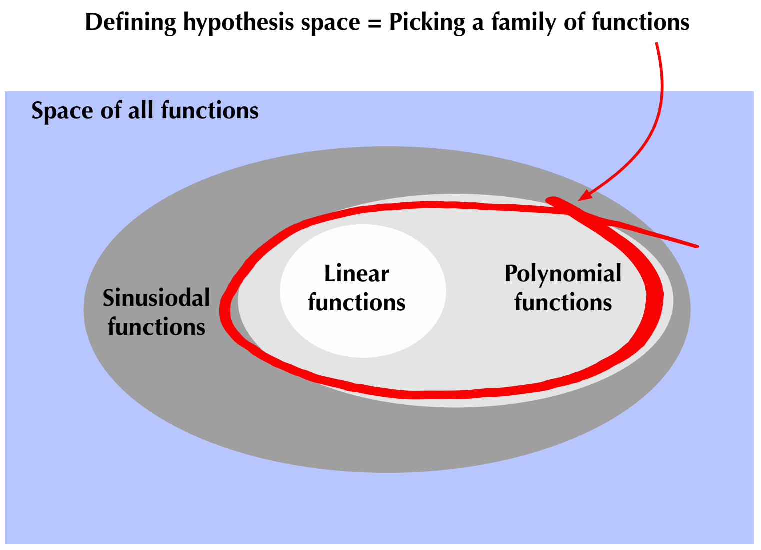


Figure 2.1: Illustration of the hypothesis space.

develop a learning algorithm around that choice of model. We commonly refer to the step of choosing a type of model as *defining the hypothesis space* [2]. We will not dig into the concept of hypothesis spaces. Rather we will try to develop an intuitive understanding of what it means. This will set the stage for our discussion in the remaining sections of this chapter.

To understand the hypothesis space, let's take a look at Figure 2.1. It shows the pool of all known functions and a few examples of popular function families, like linear, polynomial, and sinusoidal. When we, in Lecture 2, assumed a linear model to handle our regression problem, we selected one family from the pool to be our hypothesis space. This selection means that we restrict our learning algorithm to pick a function from the set of all possible linear functions. Such choice implies something about our understanding of the problem; it encodes a belief that the function governing the relation between the observed variables and desired responses is linear of quasi-linear in nature.

Let's take the discussion one step forward and bring neural networks into the picture. When dealing with neural networks, there is no explicit assumption on the function families making up the hypothesis space. However, from the universal approximation theorem [3], we know that varying the number of neurons in the input layer enables the network to approximate increasing number of functions. This means despite the fact that we do not know the families making up the hypothesis space, its size could be controlled by the number of neurons we add.

1.2 Model Capacity

Not all function families have the same size, of course. Some families envelop others (e.g., the polynomial family includes the linear family), and some families do not intersect at all. This simple observation points to an important fact; function families have different sizes! This change in size of the space have a direct relation to the number of parameters in a machine learning algorithm. To see that, consider an observed variable $x \in \mathbb{R}$ and a desired response $y \in \mathbb{R}$. We can choose to model the relation between the two using a linear model and a polynomial model as follows:

$$y = wx + b \quad \text{Model.1} \tag{2.1}$$

$$y = w_1x^2 + w_2x + b \quad \text{Model.2} \tag{2.2}$$

Going from one hypothesis space to another in this case results in an increase in the number of parameters. This increase in number of parameters is said to have changed the *model's capacity* or *model's representational capacity*. This increase means our learning algorithm has more functions to pick from and, therefore, its ability to represent the relation between x and y has increased.

2 Overfitting, Underfitting, and Generalization

When developing a machine learning algorithm, in general, our objective is to obtain a model that can perform well not only on the training dataset but also on unseen data. This objective is referred to as *model generalization*. It is common when training machine learning algorithm to have three datasets. One is the training dataset, the second is the validation dataset, and the third is called the test dataset. The first is self-explanatory, and the last two are used in general to measure the generalization performance of the algorithm. More specifically, validation data are used in parallel with the training data to assess the training progress, i.e., get some feedback on how well the algorithm is learning and fine-tune the training hyper-parameters—more about them later. The test dataset, on the other hand, is used to measure how well the trained algorithm (the model) generalizes. Hence, it is only used at when the algorithm training and validation is done.

Getting to the point where the model generalizes means the algorithm needs to navigate

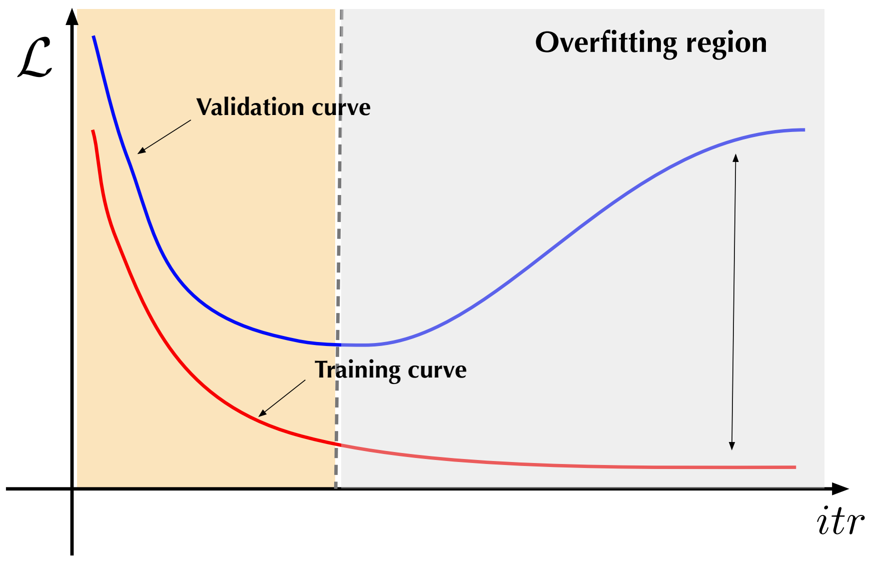


Figure 2.2: Illustration of the overfitting issue in machine learning

and avoid several pitfalls during training, the most important of which are *overfitting* and *undersfitting*. We discuss them below

- **Overfitting:** in cases where the capacity of a machine learning model is too large for the task and dataset at hand, the training of the algorithm runs into the so-called overfitting problem. Overfitting is illustrated in Figure 2.2. In simple words, the algorithm is said to overfit when its training loss rapidly shrinks while its validation loss increases. There are various approaches to deal with overfitting, increasing the training dataset size, applying regularization techniques, and reducing model capacity to name three popular examples.
- **Underfitting:** this problem could be thought of as the opposite to overfitting. It is happens when an algorithm is unable to learn from the dataset. This happens when the model capacity is too small for the task the algorithm is learning.

Bibliography

[1] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015.

[2] I. J. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning.* Cambridge, MA, USA: MIT Press, 2016, http://www.deeplearningbook.org.

[3] K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural networks*, vol. 2, no. 5, pp. 359–366, 1989.