CS 4850/6850: Introduction to Machine Learning **Fall 2018**

Topic 3: Neural Networks

Instructor: Daniel L. Pimentel-Alarcón

3.1 Introduction

Neural networks are arguably the main reason why Machine Learning is making a big splash. Similar to logistic regression and random forests, their main purpose is prediction/classification. Classical applications of neural networks include:

1. Image classification. For example, determining the digit contained in an image, as in Figure 3.1.
2. Natural language processing. For example, interpreting voice commands, like Siri and Alexa do.
3. Stock market prediction.

Each of these problems can be rephrased mathematically as finding an unknown (and highly complex) function $f^\star$, such that given certain data $\mathbf{x}$, $f^\star(\mathbf{x})$ gives the desired classification/prediction. In our examples:

1. Image classification. In our example, $\mathbf{x}$ is a vectorized image of a digit, and $f^\star(\mathbf{x}) \in \{0, \dots, 9\}$ is the digit depicted in such figure. For the images $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_5$ in Figure 3.1, $f^\star(\mathbf{x}_1) = 5, f^\star(\mathbf{x}_2) = 0, \dots, f^\star(\mathbf{x}_5) = 9$.
2. Natural language processing. In our example, $\mathbf{x}$ is a vector containing the values in a voice signal, as in Figure 3.2, and $f^\star(\mathbf{x})$ is the interpretation, in this case "turn off the t.v.".
3. Stock market prediction. Here $\mathbf{x}$ could be a sequence of stock market prices at times $\mathrm{t} = 1, \dots, \mathrm{T}$, and $f^\star(\mathbf{x})$ could be the prediction of such stock market price at time $\mathrm{T} + 1$.

All of these tasks can be tremendously challenging. For example, whether an image contains a 0, or a 1, or any other digit, depends not only on the values of isolated pixels, but on the way that pixels *interact* with one an other in complex manners.

The main idea behind ANNs is to use a sequence of simpler functions that interact with one another in a networked way, so that combined, they approximate $f^\star$ with arbitrary precision.

Figure 3.1: Digit images from the MNIST dataset. The goal is to automatically determine the digit contained in each image, in this case $\{5, 0, 4, 1, 9\}$.

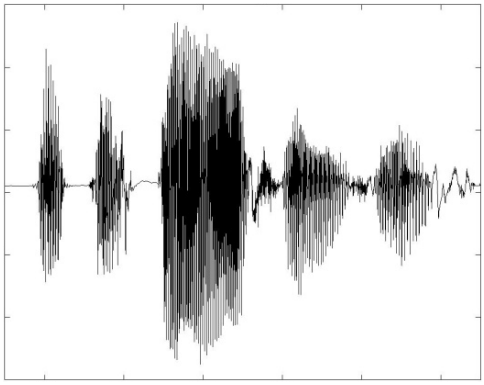


Figure 3.2: Voice signal corresponding to the command "turn off the t.v.".

3.2 Intuition

To build some intuition, suppose we want to estimate the following function $f^\star$:

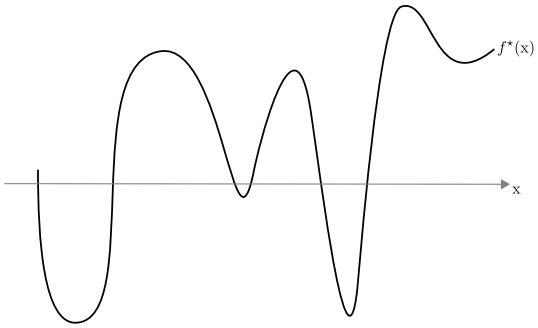
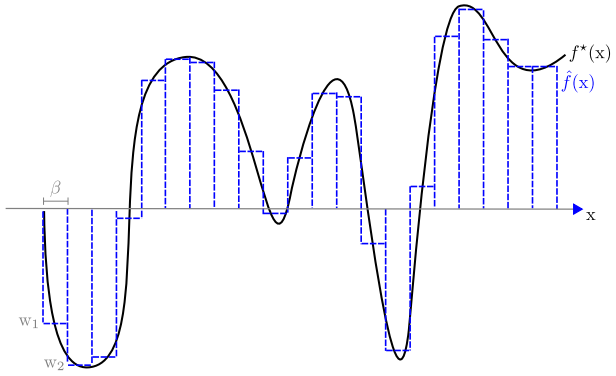


Figure 3.3: Function $f^\star(\mathbf{x})$ that we want to estimate.

One option is to use a Riemann-type approximation $\hat{f} = \sum_{\ell=1}^{\mathrm{L}} g_\ell$ where g_ℓ are step functions of the form:

$$g_\ell(\mathrm{x}) \;=\; \begin{cases} \mathrm{w}_\ell & \text{if } \beta(\ell-1) \le \mathrm{x} < \beta\ell \\ 0 & \text{otherwise.} \end{cases}$$

Here w_ℓ indicates the height of $f^\star$ in the ℓ^{th} interval $[\beta(\ell-1), \beta\ell)$. This type of approximation would look something like:

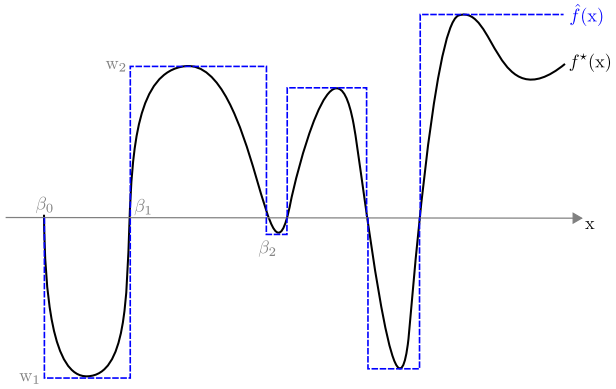


Notice that the parameters $\mathrm{w}_1, \ldots, \mathrm{w}_\ell$, together with the interval width β determine $\hat{f}$. Unfortunately, estimating this type of function would require to observe *at least* one sample x on each interval $[\beta(\ell-1), \beta\ell)$. For the figure above, this would require at least 21 samples. This may not sound like much. However, as we move to higher dimensions, this number scales exponentially, and quickly becomes prohibitively large: if $f^\star$ is a multivariate function of $\mathbf{x} \in \mathbb{R}^{\mathrm{d}}$ (instead of a single variable function of $\mathrm{x} \in \mathbb{R}$), then a similar approximation would require 21^{d} samples. With d as little as 10, this would mean more than 1 trillion samples. As we will see, in most of the big data applications that we are interest, d is often in the order of hundreds, and often thousands or millions, whence this approach is completely infeasible.

Another option is to allow varying size intervals, such that

$$g_\ell(\mathrm{x}) \;=\; \begin{cases} \mathrm{w}_\ell & \text{if } \beta_{\ell-1} \le \mathrm{x} < \beta_\ell \\ 0 & \text{otherwise.} \end{cases}$$

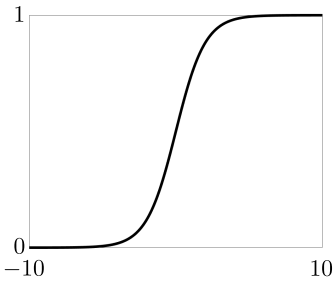
In this case, the parameters are the weights $\mathrm{w}_1, \ldots, \mathrm{w}_{\mathrm{L}}$ and the interval boundaries $\beta_0, \ldots, \beta_{\mathrm{L}}$. This type of approximation would look more like:



and will potentially require far fewer samples. The downside is that now $\hat{f}$ may be quite inaccurate (on top of being discontinuous). To address this, we can make each g_ℓ a simple nonlinear yet continuous function. For example, a sigmoid:

$$\sigma(\mathrm{x}) \;=\; \frac{1}{1+e^{-\mathrm{x}}}, \tag{3.1}$$

which looks as follows:

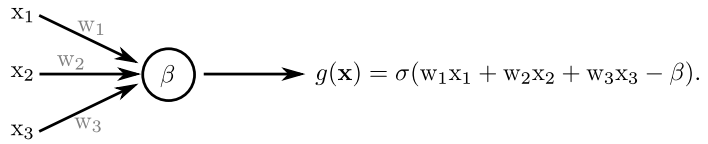


Notice that $\sigma(\mathrm{wx}-\beta)$ *shifts* the function by β, and *squeezes* the function by w. So if we let $g(\mathrm{x}) \;=\; \sigma(\mathrm{wx}-\beta)$, and we choose the right parameters w and β, and add a bunch of these functions, then we can approximate

$f^\star$ much more accurately. Moreover, if $f^\star$ is a function of more than one variable, say of a vector $\mathbf{x} \in \mathbb{R}^{\mathrm{d}}$, then we can generalize g in a very natural way:

$$g(\mathbf{x}) \;=\; \sigma(\mathbf{w}^{\mathsf{T}}\mathbf{x} - \beta), \tag{3.2}$$

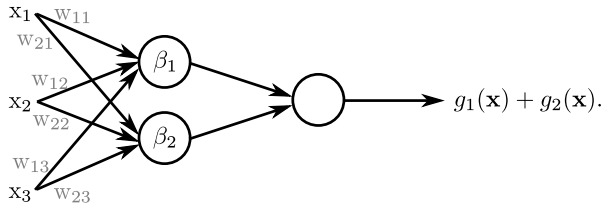
where now $\mathbf{w} \in \mathbb{R}^{\mathrm{d}}$ is a parameter vector. Just as the step function is the building block of the Riemann-type approximation above, this type of function g is precisely the building block (often called *neuron*) of a neural network, usually depicted as follows:



The main insight of a neural network is that by adding a bunch of these building blocks, one can approximate $f^\star$ with arbitrary accuracy. For example, with two neurons we would obtain the following:

$$\hat{f}(\mathbf{x}) \;=\; g_1(\mathbf{x}) + g_2(\mathbf{x}),$$

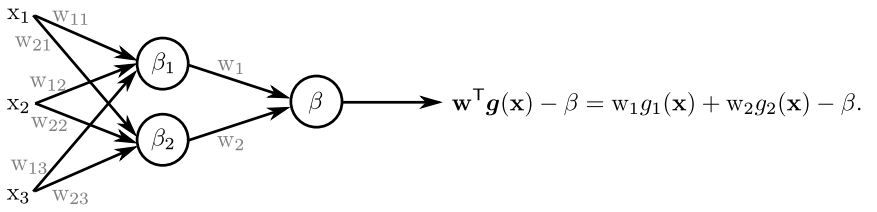
which we can depict as follows:



There is no reason why we cannot include weights to each g, nor a final shift, to obtain:

$$\hat{f}(\mathbf{x}) \;=\; \mathrm{w}_1 g_1(\mathbf{x}) + \mathrm{w}_2 g_2(\mathbf{x}) - \beta \;=\; \begin{bmatrix}\mathrm{w}_1 & \mathrm{w}_2\end{bmatrix}\begin{bmatrix} g_1(\mathbf{x}) \\ g_2(\mathbf{x})\end{bmatrix} - \beta, \;=:\; \mathbf{w}^{\mathsf{T}}\boldsymbol{g}(\mathbf{x}) - \beta,$$

where w_1, w_2, and β are new parameters. This function would be represented like:



There is no reason to stop there. We can add more neurons, and more *layers* to obtain more powerful networks, capable of approximating more complex functions. Let L be the number of layers, and let n_ℓ denote the number of neurons in the ℓ^{th} layer. For $\ell = 2, 3, \ldots, \mathrm{L}$, let $\mathbf{W}^\ell \in \mathbb{R}^{\mathrm{n}_\ell \times \mathrm{n}_{\ell-1}}$ be the matrix formed by transposing and stacking the weight vectors of the n_ℓ neurons in the ℓ^{th} layer, and let $\boldsymbol{\beta}^\ell \in \mathbb{R}^{\mathrm{n}_\ell}$ be the vector containing the shifting coefficients (often called *bias* coefficients) of the neurons in the ℓ^{th} layer, such that the output at the second layer is given by:

$$\boldsymbol{g}^2(\mathbf{x}) = \sigma(\mathbf{W}^2\mathbf{x} - \boldsymbol{\beta}_2),$$

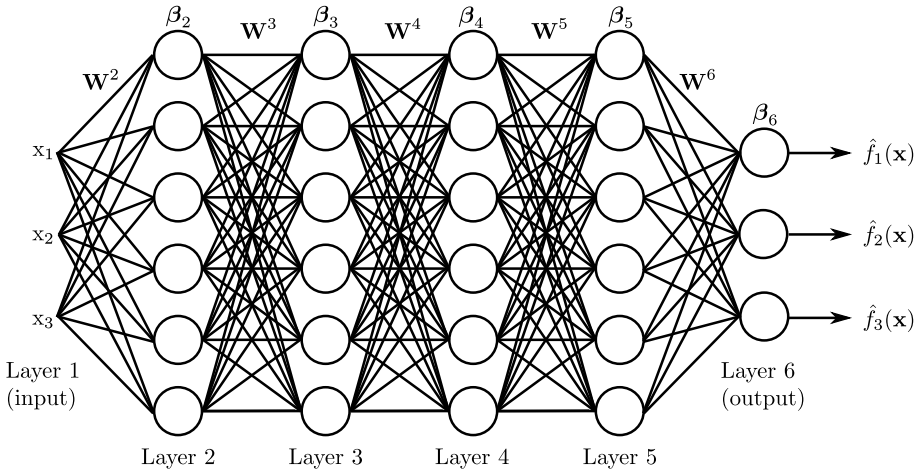


Figure 3.4: Components of a neural network.

and for $\ell = 3, \ldots, \mathrm{L}$, the output at the ℓ^{th} layer is given by:

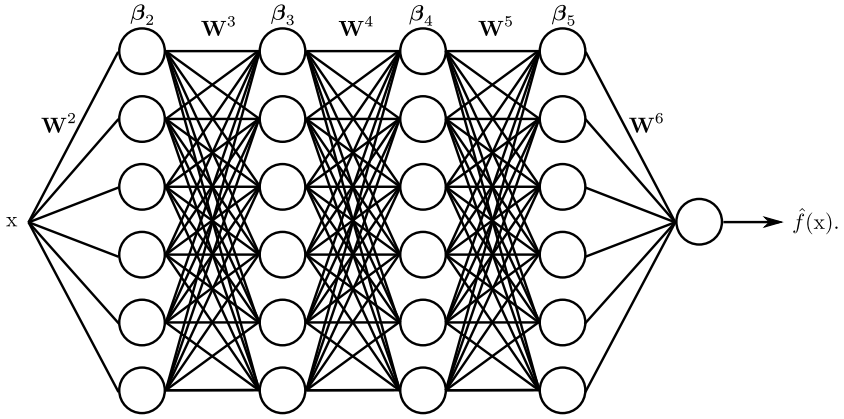
$$\boldsymbol{g}^{\ell}(\mathbf{x}) = \sigma(\mathbf{W}^{\ell}\boldsymbol{g}^{\ell-1}(\mathbf{x}) - \boldsymbol{\beta}_{\ell}),$$

where $\boldsymbol{g}^{\mathrm{L}}(\mathbf{x})$ is the final output of the network, also denoted as:

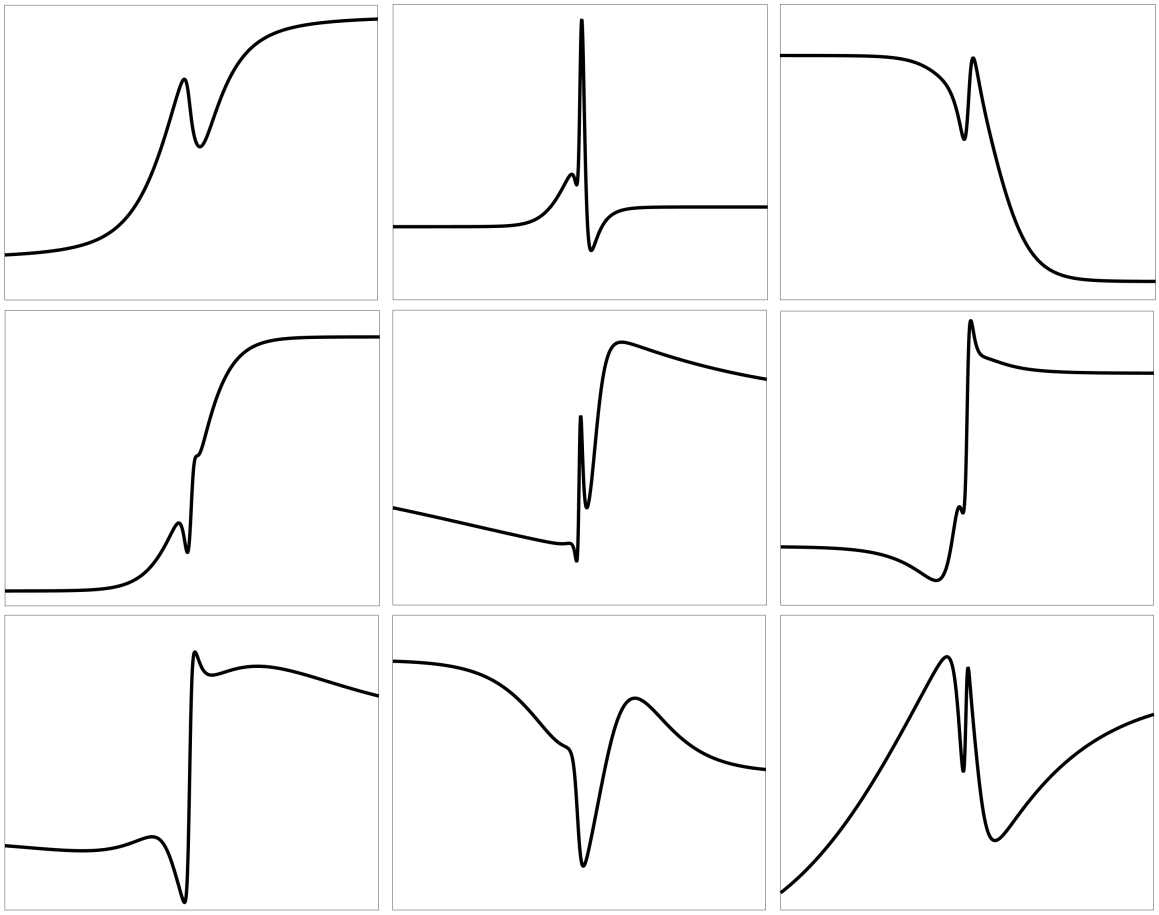
$$\hat{\boldsymbol{f}}(\mathbf{x}) \;:=\; \sigma(\mathbf{W}^{\mathrm{L}}\sigma(\mathbf{W}^{\mathrm{L}-1} \;\cdots\; \sigma(\mathbf{W}^{3}\sigma(\mathbf{W}^{2}\mathbf{x} - \boldsymbol{\beta}^{2}) - \boldsymbol{\beta}^{3}) \cdots - \boldsymbol{\beta}^{\mathrm{L}-1}) - \boldsymbol{\beta}^{\mathrm{L}}). \tag{3.3}$$

Notice that $\hat{\boldsymbol{f}}(\mathbf{x}) \in \mathbb{R}^{\mathrm{n_L}}$ may be a vector (if $\mathrm{n_L} > 1$), as opposed to a scalar (if $\mathrm{n_L} = 1$), and so neural networks allow to infer vector functions. See Figure 3.4 to build some intuition.

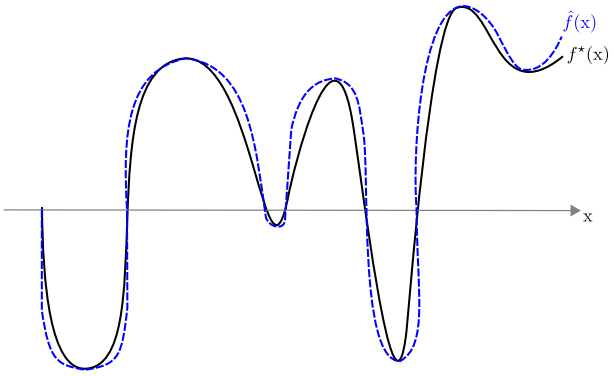
Going back to estimating the function in Figure 3.3, we can construct the following network:



Depending on the parameters $\{\mathbf{W}^{\ell}, \boldsymbol{\beta}^{\ell}\}_{\ell=2}^{\mathrm{L}}$ that we choose, may produce functions like the following:



An approximation of the function in Figure 3.3 with a neural network like this would look like:



However, this will require to identify the right parameters $\{\mathbf{W}^{\ell}, \boldsymbol{\beta}^{\ell}\}_{\ell=2}^{\mathrm{L}}$ that produce this approximation.

3.3 Identifying the Right Parameters

As discussed before, with the right parameters $\{\mathbf{W}^\ell, \boldsymbol{\beta}^\ell\}_{\ell=2}^{\mathrm{L}}$, the function $\hat{\boldsymbol{f}}(\mathbf{x})$ in (3.3) can approximate any function $\boldsymbol{f}^\star(\mathbf{x})$ with arbitrary accuracy. The challenge is to find the right parameters $\{\mathbf{W}^\ell, \boldsymbol{\beta}^\ell\}_{\ell=2}^{\mathrm{L}}$.

Hence, our goal can be summarized as follows: find the parameters $\{\mathbf{W}^\ell, \boldsymbol{\beta}^\ell\}_{\ell=2}^{\mathrm{L}}$ of the function $\hat{\boldsymbol{f}}$ in (3.3), such that $\hat{\boldsymbol{f}}(\mathbf{x}) \approx \boldsymbol{f}^\star(\mathbf{x})$. Recall that we do not know $f^\star$. However, we do have *training* data. More precisely, we have N samples $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_{\mathrm{N}}$ of N, as well as their *response* $\mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_{\mathrm{N}}$. For example, if we were studying diabetes, $\mathbf{x}_{\mathrm{i}}$ could contain demographic information about the i^{th} person in our study, and $\mathbf{y}_{\mathrm{i}}$ could be a binary variable indicating whether this person is diabetic or not.

In other words, we know that $\boldsymbol{f}^\star(\mathbf{x}_{\mathrm{i}}) = \mathbf{y}_{\mathrm{i}}$ for every $\mathrm{i} = 1, 2, \ldots, \mathrm{N}$. Intuitively, this means that we get to see N snapshots of $\boldsymbol{f}^\star$ at points $\mathbf{x}_{\mathrm{i}}$. Our goal is to exploit this information to find the parameters $\{\mathbf{W}^\ell, \boldsymbol{\beta}^\ell\}_{\ell=2}^{\mathrm{L}}$ such that $\hat{\boldsymbol{f}} \approx \boldsymbol{f}^\star$, such that the network can reproduce the response $\mathbf{y}$ whenever a new vector $\mathbf{x}$ is fed to the network. This can be done by minimizing the error (over all training data) between the network's prediction $\hat{\boldsymbol{f}}(\mathbf{x}_{\mathrm{i}})$ and its corresponding observation $\mathbf{y}_{\mathrm{i}}$. Mathematically, we can achieve this by solving the following optimization

$$\min_{\{\mathbf{W}^\ell, \boldsymbol{\beta}^\ell\}_{\ell=2}^{\mathrm{L}}} \quad \sum_{\mathrm{i}=1}^{\mathrm{N}} \|\mathbf{y}_{\mathrm{i}} - \hat{\boldsymbol{f}}(\mathbf{x}_{\mathrm{i}})\|^2. \tag{3.4}$$

The most widely used technique to solve (3.4) is through stochastic gradient descent and back-propagation. The key insight is to observe that the gradient of each weight matrix can be computed *backwards* in terms of the gradients of the weight matrices of subsequent layers. To see this, first define $\mathbf{z}_{\mathrm{i}}^1 = \mathbf{y}_{\mathrm{i}}^1 = \mathbf{x}_{\mathrm{i}}$, and then for $\ell = 2, 3, \ldots, \mathrm{L}$, recursively define $\mathbf{z}_{\mathrm{i}}^\ell := \mathbf{W}^\ell \mathbf{y}_{\mathrm{i}}^{\ell-1} - \boldsymbol{\beta}^\ell$, where $\mathbf{y}_{\mathrm{i}}^\ell := \boldsymbol{g}^\ell(\mathbf{x}_{\mathrm{i}}) = \sigma(\mathbf{z}_{\mathrm{i}}^\ell)$ denotes the output at the ℓ^{th} layer, so that $\hat{\boldsymbol{f}}(\mathbf{x}_{\mathrm{i}}) = \mathbf{y}_{\mathrm{i}}^{\mathrm{L}}$. Defining the *cost* of the i^{th} sample as $\mathbf{c}_{\mathrm{i}} := \mathbf{y}_{\mathrm{i}} - \hat{\boldsymbol{f}}(\mathbf{x}_{\mathrm{i}})$, and

$$\begin{aligned} \boldsymbol{\delta}_{\mathrm{i}}^{\mathrm{L}} &:= \mathbf{c} \odot \boldsymbol{f}'(\mathbf{z}_{\mathrm{i}}^{\mathrm{L}}), \\ \boldsymbol{\delta}_{\mathrm{i}}^{\ell} &:= \left[(\mathbf{W}^{\ell+1})^{\mathsf{T}}\, \boldsymbol{\delta}^{\ell+1}\right] \odot \boldsymbol{f}'(\mathbf{z}_{\mathrm{i}}^{\ell}), \qquad 2 \le \ell \le \mathrm{L}-1, \end{aligned}$$

where $\odot$ represents the Hadamard product, with a simple chain rule we obtain the following gradients:

$$\nabla_{\mathrm{i}}\mathbf{W}^\ell := \frac{\partial \|\mathbf{c}_{\mathrm{i}}\|^2}{\partial \mathbf{W}^\ell} = -2\, \boldsymbol{\delta}_{\mathrm{i}}^\ell\, (\mathbf{y}_{\mathrm{i}}^{\ell-1})^{\mathsf{T}}, \qquad \nabla_{\mathrm{i}}\boldsymbol{\beta}^\ell := \frac{\partial \|\mathbf{c}_{\mathrm{i}}\|^2}{\partial \boldsymbol{\beta}^\ell} = -2\, \boldsymbol{\delta}_{\mathrm{i}}^\ell, \qquad 2 \le \ell \le \mathrm{L}. \tag{3.5}$$

Training the neural network is equivalent to identifying the parameters $\{\mathbf{W}^\ell, \boldsymbol{\beta}^\ell\}_{\ell=2}^{\mathrm{L}}$ such that $\hat{\boldsymbol{f}} \approx \boldsymbol{f}^\star$. This can be done using stochastic gradient descent, which iteratively updates the parameters of interest according to (3.5).

More precisely, at each training time t we select a random subsample $\Omega_{\mathrm{t}} \subset \{1, 2, \ldots, \mathrm{N}\}$ of the training data (hence the term *stochastic*), and we update our parameters as follows

$$\begin{aligned} \mathbf{W}_{\mathrm{t}}^\ell &= \mathbf{W}_{\mathrm{t}-1}^\ell - \eta \sum_{\mathrm{i}\in\Omega_{\mathrm{t}}} \nabla_{\mathrm{i}}\mathbf{W}_{\mathrm{t}-1}^\ell, \\ \boldsymbol{\beta}_{\mathrm{t}}^\ell &= \boldsymbol{\beta}_{\mathrm{t}-1}^\ell - \eta \sum_{\mathrm{i}\in\Omega_{\mathrm{t}}} \nabla_{\mathrm{i}}\boldsymbol{\beta}_{\mathrm{t}-1}^\ell, \end{aligned}$$

where η is the gradient *step* parameter to be tuned. This iteration is repeated until convergence to obtain the *final* parameters $\{\widehat{\mathbf{W}}^\ell, \widehat{\boldsymbol{\beta}}^\ell\}_{\ell=2}^{\mathrm{L}}$.

3.4 Neural Networks Flavors

At this point you may be wondering whether $\boldsymbol{g}^\ell(\mathbf{x}) = \sigma(\mathbf{W}^\ell \boldsymbol{g}^{\ell-1}(\mathbf{x}) - \boldsymbol{\beta})$ is the only option for $\boldsymbol{g}$, where σ is the sigmoid function in (3.1). The answer is: no. For instance, if you use

$$\sigma(\mathrm{x}) \;=\; \mathrm{x},$$

then you obtain a *linear* layer/network. If you use

$$\sigma(\mathrm{x}) \;=\; \max(0, \mathrm{x}),$$

then you obtain the so-called *rectified linear unit* (ReLU) layer/network. σ is usually called *activation function*. Similarly, if instead of $\boldsymbol{g}^\ell(\mathbf{x}) = \sigma(\mathbf{W}^\ell \boldsymbol{g}^{\ell-1}(\mathbf{x}) - \boldsymbol{\beta})$ you use

$$\boldsymbol{g}^\ell(\mathbf{x}) \;=\; \sigma(\mathbf{W}^\ell * \boldsymbol{g}^{\ell-1}(\mathbf{x}) - \boldsymbol{\beta}),$$

where $*$ denotes the convolution operator, then you obtain a *convolutional* layer/network.

The *topology* (shape) of the network is defined by the number of layers (L) and the number of neurons in each layer $(\mathrm{n}_1, \mathrm{n}_2, \ldots, \mathrm{n}_\mathrm{L})$. Networks with large L are often called *deep*, as opposed to *shallow* networks that have small L.

Depending on the problem at hand, one choice of L, $\mathrm{n}_1, \ldots, \mathrm{n}_\mathrm{L}$, σ, and $\boldsymbol{g}$ (or combinations) may be better than another. For example, for image classification, people usually like to mix cascades of one linear layer, followed by a ReLU layer, followed by a convolutional layer, with large n_ℓ in each case, and sometimes with fixed (predetermined) $\mathbf{W}$'s in the convolutional layers. The possibilities are endless, and deciding on the best choice of topology remains more of an art than science.

3.5 A Word of Warning

We have mentioned before that using neural networks we can approximate any function $\boldsymbol{f}^\star$ with arbitrary accuracy. Put another way, for every $\boldsymbol{f}^\star$, and every $\epsilon > 0$ there exists a function $\hat{\boldsymbol{f}}_{\boldsymbol{\theta}}$ with parameters $\boldsymbol{\theta} := \{\mathbf{W}^\ell, \boldsymbol{\beta}^\ell\}_{\ell=2}^{\mathrm{L}}$ such that,

$$\|\boldsymbol{f}^\star(\mathbf{x}) - \hat{\boldsymbol{f}}_{\boldsymbol{\theta}}(\mathbf{x})\|_2 \;<\; \epsilon \qquad \text{for every } \mathbf{x} \text{ in the domain of } \mathbf{f}^\star. \tag{3.6}$$

The challenge is to find the parameter $\boldsymbol{\theta}$ for which (3.6) is true, which we aim to do using gradient descent to minimize the following *cost* function:

$$\mathbf{c}(\boldsymbol{\theta}) \;:=\; \sum_{\mathrm{i}=1}^{\mathrm{N}} \|\boldsymbol{f}^\star(\mathbf{x}_\mathrm{i}) - \hat{\boldsymbol{f}}_{\boldsymbol{\theta}}(\mathbf{x}_\mathrm{i})\|_2.$$

The wrinkle is that the cost function that we are trying to minimize may be non-convex, which implies that we may *never* find the right parameter $\boldsymbol{\theta}$. In other words, for every $\boldsymbol{f}^\star$ there will always exist *a* neural network (parametrized by $\boldsymbol{\theta}$) that approximates $\boldsymbol{f}^\star$ arbitrarily well. However, we may be unable to find such network.