BMI/CS 567: Medical Image Analysis **Spring 2020**

Week 11: Neural Networks

Instructor: Daniel L. Pimentel-Alarcón

GO GREEN. AVOID PRINTING, OR PRINT 2-SIDED MULTI-PAGE.

11.1 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, digit classification (determining the digit contained in an image), or medical diagnostics (deciding whether a patient is healthy or not based on an MRI).
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 digit classification $\mathbf{x}$ is a vectorized image of a digit, and $f^\star(\mathbf{x}) \in \{0, \dots, 9\}$ is the digit depicted in such image. For the images $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_5$ in Figure 11.1, $f^\star(\mathbf{x}_1) = 5, f^\star(\mathbf{x}_2) = 0, \dots, f^\star(\mathbf{x}_5) = 9$. In medical diagnostics, $\mathbf{x}$ could be a vectorized MRI, as in Figure 11.2, and $f^\star(\mathbf{x}) \in \{0, 1\}$ would be the diagnostic, 0 corresponding to healthy, and 1 to Alzheimers.

Figure 11.1: Digit images from the MNIST dataset. The goal is to automatically determine the digit contained in each image, in this case $\{5, 0, 4, 1, 9\}$.

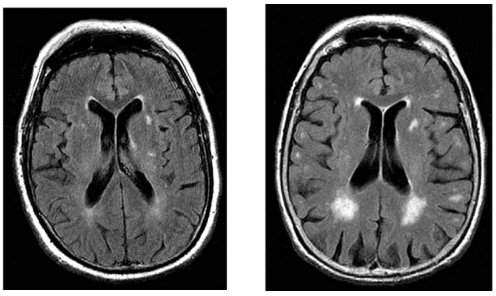


Figure 11.2: Two brain MRIs. The goal is to automatically determine whether they are healthy or not.

2. **Natural language processing.** Here $\mathbf{x}$ is a vector containing the values in a voice signal, as in Figure 11.3, and $f^\star(\mathbf{x})$ is the interpretation, in this case "turn off the t.v.".

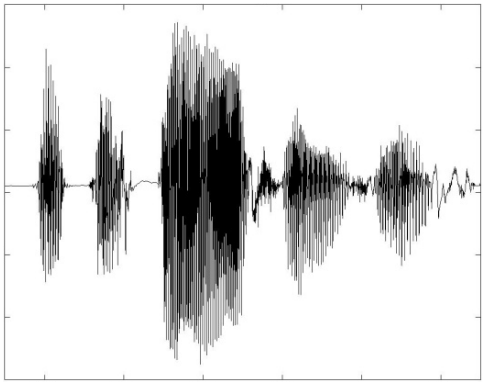


Figure 11.3: Voice signal corresponding to the command "turn off the t.v.".

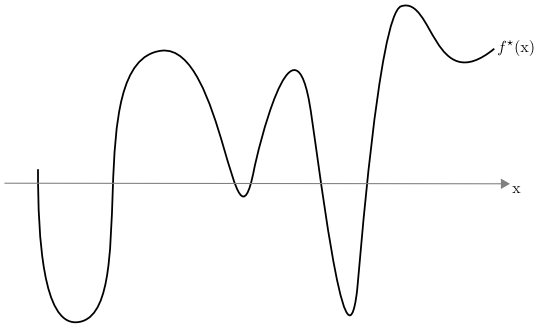
3. **Stock market prediction.** Here $\mathbf{x}$ could be a sequence of stock market prices at times $\mathrm{t} = 1, \ldots, \mathrm{T}$, and $f^\star(\mathbf{x})$ could be the prediction of such stock market price at time $\mathrm{T} + 1$.

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 Neural Networks 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.

11.2 Intuition

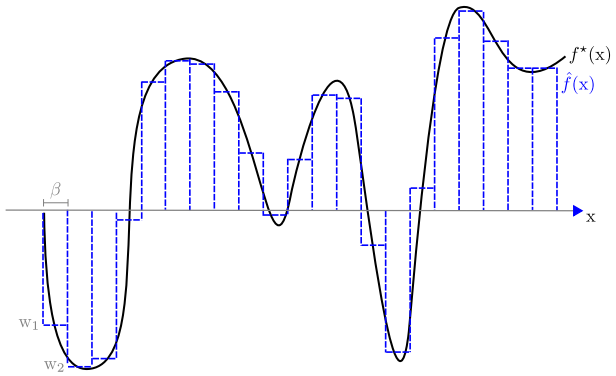
To build some intuition, suppose we want to estimate the following function $f^\star$:



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:

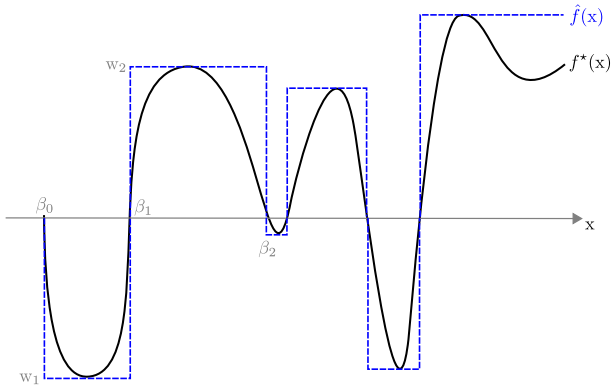


Notice that the parameters $\mathrm{w}_1, \ldots, \mathrm{w}_\ell$, together with the interval width β determine $\hat{f}$. Unfortunately, estimating this type of function would require to observe *at least* one sample x on each interval $[\beta(\ell-1), \beta\ell)$. For the figure above, this would 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

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

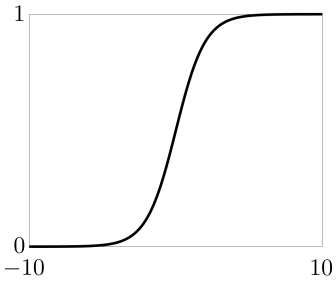
In this case, the parameters are the weights $\mathrm{w}_1, \ldots, \mathrm{w}_{\mathrm{L}}$ and the interval boundaries $\beta_0, \ldots, \beta_{\mathrm{L}}$. This type of approximation would look more like:



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:

$$\sigma(\mathrm{x}) \;=\; \frac{1}{1+e^{-\mathrm{x}}}, \tag{11.1}$$

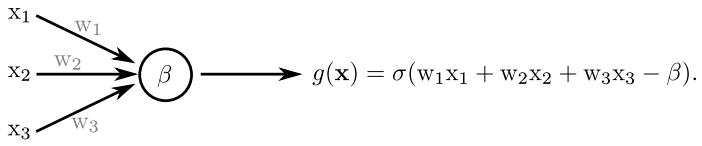
which looks as follows:



Notice that $\sigma(\mathrm{w}\mathrm{x}-\beta)$ *shifts* the function by β, and *squeezes* the function by w. So if we let $g(\mathrm{x}) = \sigma(\mathrm{w}\mathrm{x}-\beta)$, 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:

$$g(\mathbf{x}) \;=\; \sigma(\mathbf{w}^{\mathsf{T}}\mathbf{x} - \beta), \tag{11.2}$$

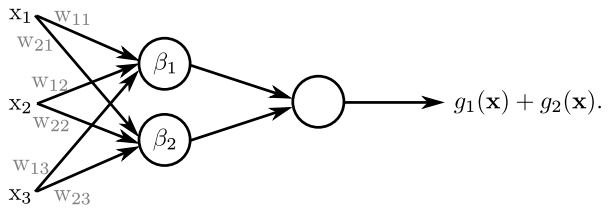
where now $\mathbf{w} \in \mathbb{R}^{\mathrm{d}}$ is a parameter vector. Just as the step function is the building block of the Riemann-type approximation above, this type of function g is precisely the building block (often called *neuron*) of a neural network, usually depicted as follows:



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:

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

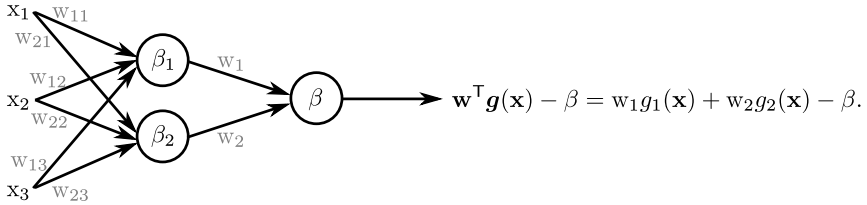
which we can depict as follows:



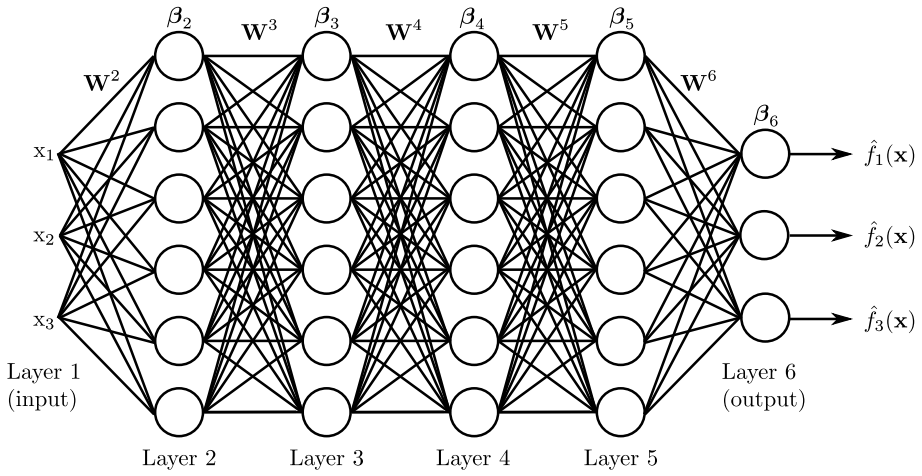
There is no reason why we cannot include weights to each g, nor a final shift, to obtain:

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

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, like the following, capable of approximating more complex functions:



Here we use L to denote the number of layers (in the figure above $\mathrm{L} = 6$), and n_ℓ to denote the number of neurons in the ℓ^{th} layer. For $\ell = 2, 3, \ldots, \mathrm{L}$, $\mathbf{W}^\ell \in \mathbb{R}^{\mathrm{n}_\ell \times \mathrm{n}_{\ell-1}}$ is the matrix formed by transposing and stacking the weight vectors of the n_ℓ neurons in the ℓ^{th} layer, and $\boldsymbol{\beta}^\ell \in \mathbb{R}^{\mathrm{n}_\ell}$ is the vector containing the shifting coefficients (often called *bias* coefficients) of the neurons in the ℓ^{th} layer. This way the output at the second layer is given by:

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

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

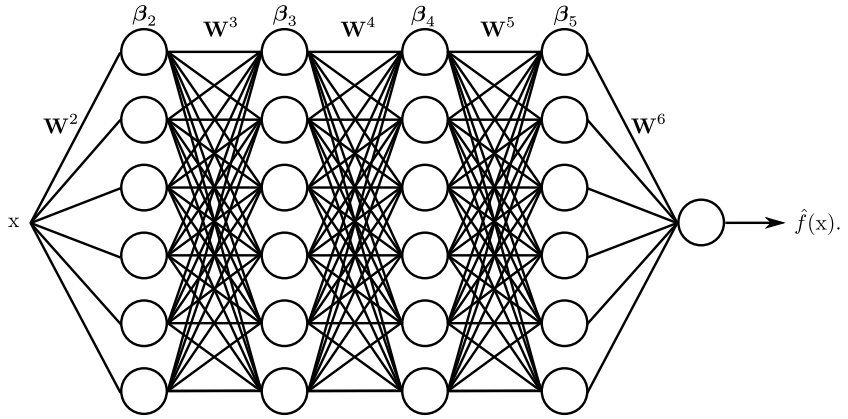
$$\boldsymbol{g}^\ell(\mathbf{x}) = \sigma(\mathbf{W}^\ell \boldsymbol{g}^{\ell-1}(\mathbf{x}) - \boldsymbol{\beta}_\ell),$$

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

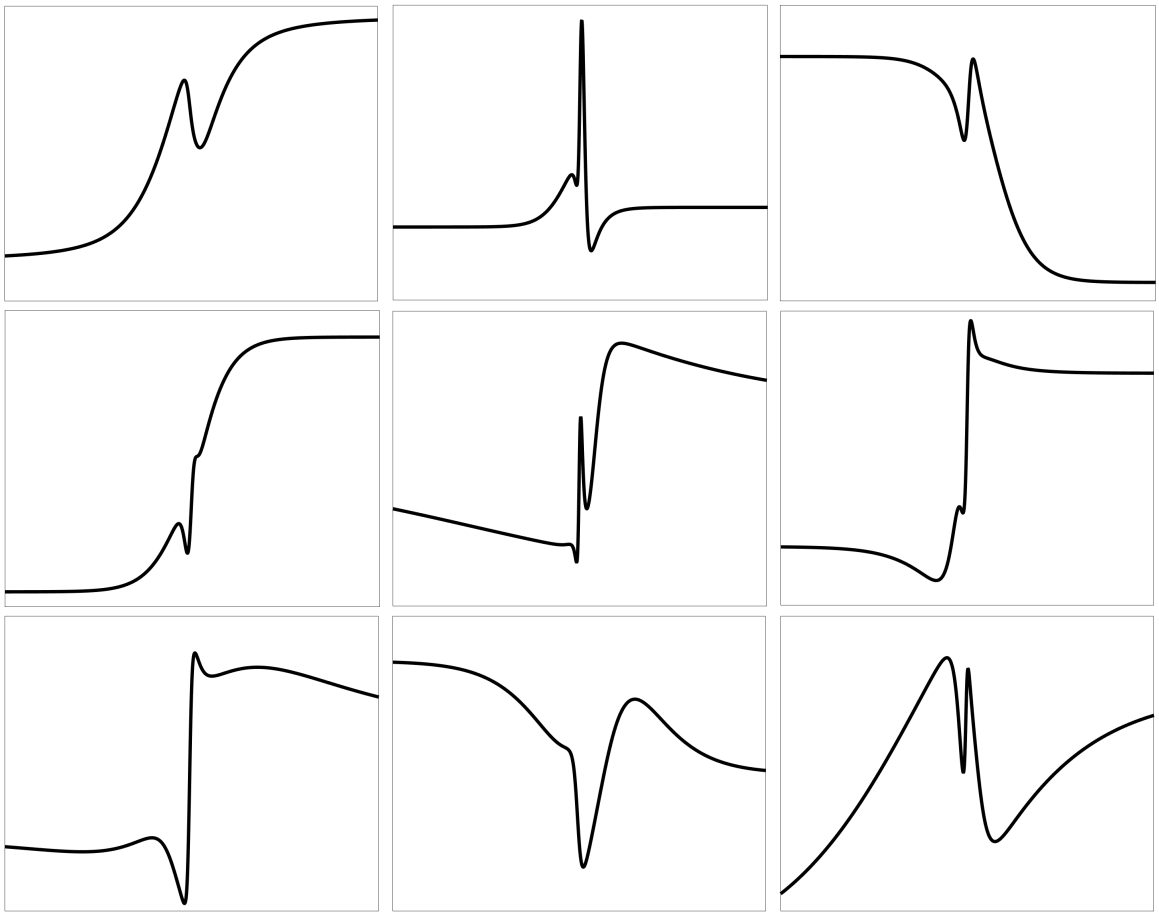
$$\hat{\boldsymbol{f}}(\mathbf{x}) \;:=\; \sigma(\mathbf{W}^{\mathrm{L}}\sigma(\mathbf{W}^{\mathrm{L}-1} \;\cdots\; \sigma(\mathbf{W}^3\sigma(\mathbf{W}^2\mathbf{x} - \boldsymbol{\beta}^2) - \boldsymbol{\beta}^3)\cdots - \boldsymbol{\beta}^{\mathrm{L}-1}) - \boldsymbol{\beta}^{\mathrm{L}}). \tag{11.3}$$

Notice that $\hat{\boldsymbol{f}}(\mathbf{x}) \in \mathbb{R}^{\mathrm{n}_{\mathrm{L}}}$ may be a vector (if $\mathrm{n}_{\mathrm{L}} > 1$), as opposed to a scalar (if $\mathrm{n}_{\mathrm{L}} = 1$), and so neural networks allow to infer vector functions.

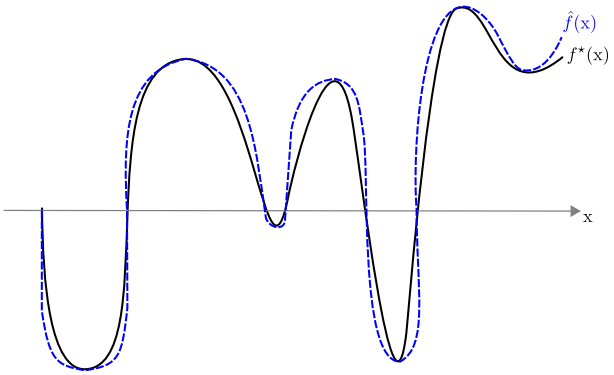
Going back to estimating the function above, we can construct the following network:



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 above with a neural network like this would look like:



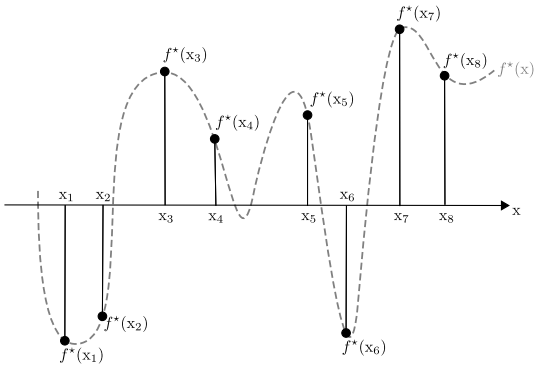
The trick lies in finding the parameters $\{\mathbf{W}^\ell, \boldsymbol{\beta}^\ell\}_{\ell=2}^{\mathrm{L}}$ that produce such approximation.

11.3 Identifying the Right Parameters (aka Training)

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 (11.3) can approximate any function $\boldsymbol{f}^\star(\mathbf{x})$ with arbitrary accuracy. The challenge is to find the right parameters $\{\mathbf{W}^\ell, \boldsymbol{\beta}^\ell\}_{\ell=2}^{\mathrm{L}}$. The process of finding such parameters is often known as *training*.

Recall that we do not know $f^\star$. However, we do have *training* data. More precisely, we have N samples $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_{\mathrm{N}}$, 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 don't get to see *all* of $f^\star$, but 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 function (network) can reproduce the response $\mathbf{y}$ whenever a new vector $\mathbf{x}$ is fed to the function (network). This can be done by minimizing the error over all training data (often called *cost function*) 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

$$\min_{\{\mathbf{W}^\ell, \boldsymbol{\beta}^\ell\}_{\ell=2}^{\mathrm{L}}} \quad \sum_{\mathrm{i}=1}^{\mathrm{N}} \|\mathbf{y}_{\mathrm{i}} - \hat{\boldsymbol{f}}(\mathbf{x}_{\mathrm{i}})\|^2. \tag{11.4}$$

The most widely used technique to solve (11.4) is through stochastic gradient descent and backpropagation.

11.4 Gradient Descent

One often wants to find the *minimizer* of a cost function $c(\mathbf{W})$, that is, the value $\mathbf{W}^\star$ such that $c(\mathbf{W}^\star) \leq c(\mathbf{W})$ for every $\mathbf{W}$ in the domain of c. If c is convex and *simple* enough, $\mathbf{W}^\star$ can be determined using our elemental calculus recipe:

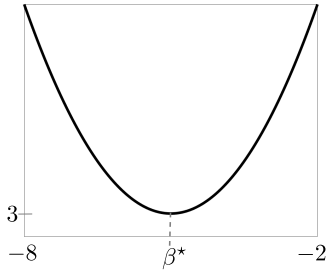
1. Take derivative of $c(\mathbf{W})$
2. Set derivative to zero, and solve for the minimizer.

Example 11.1. Consider $c(w) = (w+5)^2 + 3$. We can follow our recipe to find its minimizer:

1. The derivative of c is $\nabla c(w) = 2(w+5)$.
2. Setting the derivative to zero and solving for w we obtain:

$$\begin{aligned} 2(w+5) &= 0 \\ w &= -5. \end{aligned}$$

Since c is convex (can you show this?), we conclude that its minimizer is $w^\star = -5$, as depicted below:



Some functions, however, are either not convex, or too complex that we cannot solve for $\mathbf{W}$ in step 2. For example, can you compute the gradient of (11.3), set to zero, and solve for $\mathbf{W}$?

For cases where our calculus 101 recipe does not work, we use *optimization*, which is the field of mathematics that deals with finding minimums (and maximums). In particular, we will use one of the most elemental tools of optimization: gradient descent.

The setting is is this: you have a function $c(\mathbf{W})$. You want to find its minimum. You cannot solve for it directly using the derivative trick, so what can you do? You can *test* the value of c for different values of $\mathbf{W}$. For example, you can test $c(0)$, then maybe $c(1)$, then maybe $c(-1)$, then maybe $c(1.5)$, and so on, until you find the minimizer. Of course, depending on the domain of c, there could be infinitely many options, so testing them all would be infeasible.

As the name suggests, the main idea of gradient descent is to test some initial value $\mathbf{W}_0$ (for example 0), and iteratively use the gradient (another name for derivative) to determine which value of $\mathbf{W}$ to test next, such that each new value $\mathbf{W}_{t+1}$ produces a lower value for c, until we find the minimum. The main intuition is that the gradient $\nabla c(\mathbf{W})$ tells us the slope of c at $\mathbf{W}$. More precisely, if c is a function of a matrix $\mathbf{W} \in \mathbb{R}^{m\times n}$, then the gradient $\nabla c(\mathbf{W}) \in \mathbb{R}^{m\times n}$ gives the slope of c in each of the $m \times n$ components of $\mathbf{W}^\ell$. For each component (see Figure 11.4 to build some intuition):

- If this slope is positive, then we know that c is increasing around $\mathbf{W}$, and we should try a smaller value of $\mathbf{W}$, say $\mathbf{W}_{t+1} = \mathbf{W}_t - \eta$, where η is often referred to as *step-size*.
- If the slope is negative, then we know that c is decreasing, and we should try a larger value of $\mathbf{W}$, say $\mathbf{W}_{t+1} = \mathbf{W}_t + \eta$.

Both of these insights can be summarized multiplying the step-size by the gradient: $\mathbf{W}_{t+1} = \mathbf{W}_t - \eta\nabla c(\mathbf{W}_t)$. Hence gradient descent can be summarized as in Algorithm 1.

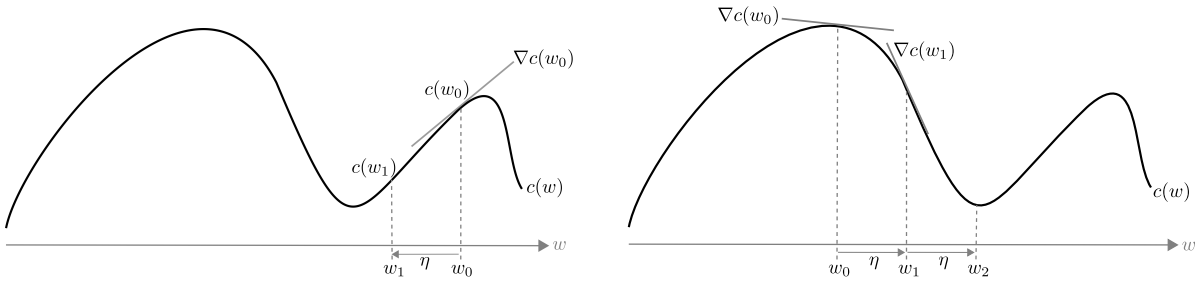


Figure 11.4: Start at some point w_0. If the gradient is positive (left figure), try a smaller value of w, say $w_1 = w_0 + \eta$. If the gradient is negative (right figure), try a larger value of w, say $w_1 = w_0 - \eta$. Repeat this until convergence.

Algorithm 1: Gradient Descent

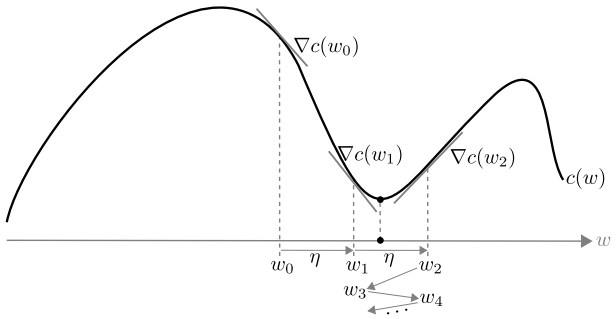
Input: Function c, step-size parameter $\eta > 0$.
Initialize $\mathbf{W}_0$. For example, $\mathbf{W}_0 = \mathbf{0}$.
Repeat until convergence: $\mathbf{W}_{t+1} = \mathbf{W}_t - \eta \nabla c(\mathbf{W}_t)$.
Output: $\mathbf{W}^\star = \mathbf{W}_t$.

11.4.1 Step-size η

The keen reader will be wondering, what if we move too far? In our example of Figure 11.4, we could run into an infinite loop, where

$$\begin{aligned} w_1 &= w_3 = w_5 = w_7 = \cdots \\ w_2 &= w_4 = w_6 = w_8 = \cdots, \end{aligned}$$

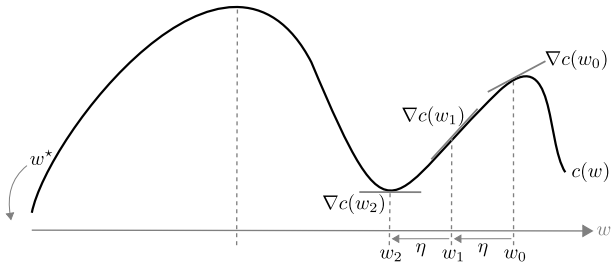
without ever achieving the minimizer, as depicted below:



How would you fix this problem?

11.4.2 Initialization

The keen reader will also be wondering: what if we start at the wrong place, as depicted below:



In cases like these we could run into a so-called local minimum, that is, a point that is smaller than all other points in its vicinity, but not necessarily the minimum over the whole domain of c. In the figure above, w_2 is a local minimizer.

How would you fix this problem?

11.5 Backpropagation

Back to neural networks, recall that it all boils down to finding the right parameters $\{\mathbf{W}^\ell, \boldsymbol{\beta}^\ell\}_{\ell=2}^{\mathrm{L}}$ such that $\hat{\boldsymbol{f}} \approx \boldsymbol{f}^\star$. As discussed above, this can be done by solving (11.4), which we will do using gradient descent, which in turn requires computing the gradient of each parameter. 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 $\hat{\boldsymbol{f}}(\mathbf{x}_{\mathrm{i}}) = \mathbf{y}_{\mathrm{i}}^{\mathrm{L}}$. Define the *cost* of the i^{th} sample as $\mathbf{c}_{\mathrm{i}} := \mathbf{y}_{\mathrm{i}} - \hat{\boldsymbol{f}}(\mathbf{x}_{\mathrm{i}})$, and

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

where $\odot$ represents the Hadamard product, and σ' represents the derivative of σ. Then with a simple chain rule we obtain the following gradients:

$$\nabla_{\mathrm{i}} \mathbf{W}^\ell := \frac{\partial \|\mathbf{c}_{\mathrm{i}}\|^2}{\partial \mathbf{W}^\ell} = -2\, \boldsymbol{\delta}_{\mathrm{i}}^\ell\, (\mathbf{y}_{\mathrm{i}}^{\ell-1})^{\mathsf{T}}, \qquad \nabla_{\mathrm{i}} \boldsymbol{\beta}^\ell := \frac{\partial \|\mathbf{c}_{\mathrm{i}}\|^2}{\partial \boldsymbol{\beta}^\ell} = -2\, \boldsymbol{\delta}_{\mathrm{i}}^\ell, \qquad 2 \le \ell \le \mathrm{L}. \tag{11.5}$$

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 (11.5).

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 $\{\widehat{\mathbf{W}}^\ell, \widehat{\boldsymbol{\beta}}^\ell\}_{\ell=2}^{\mathrm{L}}$.

11.6 Neural Networks Flavors

At some point you may wonder whether the sigmoid function in (11.1) is the only option to use as the function σ in $\boldsymbol{g}^\ell(\mathbf{x}) = \sigma(\mathbf{W}^\ell \boldsymbol{g}^{\ell-1}(\mathbf{x}) - \boldsymbol{\beta})$. 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.

11.7 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,

$$\|\boldsymbol{f}^\star(\mathbf{x}) - \hat{\boldsymbol{f}}_{\boldsymbol{\theta}}(\mathbf{x})\|_2 \;<\; \epsilon \qquad \text{for every } \mathbf{x} \text{ in the domain of } \mathbf{f}^\star. \tag{11.6}$$

The challenge is to find the parameter $\boldsymbol{\theta}$ for which (11.6) 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.

11.8 Neural Nets in Practice

So, in the end neural networks are just a family of functions of the form in (11.3) that receive an input $\mathbf{x}$ (e.g., image), and depending on its parameters $\{\mathbf{W}^\ell, \boldsymbol{\beta}^\ell\}_{\ell=2}^{\mathrm{L}}$, will produce an output $\hat{\boldsymbol{f}}(\mathbf{x})$. Using training

data we can tweak the parameters of the network to make it approximate any desired function $\boldsymbol{f}^\star$ (e.g., the function diagnosing whether an MRI corresponds to a healthy or sick person), so that the network *learns* to replicate the function $\boldsymbol{f}^\star$.

In practice, this can be summarized in several easy steps:

1. **Collect training data**, i.e., pairs $\{(\mathbf{x}_\mathrm{i}, \mathbf{y}_\mathrm{i})\}_{\mathrm{i}=1}^{\mathrm{N}}$ (recall that here $\mathbf{x}_\mathrm{i}$ is your input data, e.g., an image, and $\mathbf{y}_\mathrm{i}$ is your output, e.g., *healthy* or *sick* labels).
2. **Choose your network architecture**, essentially: number of layers, number of neurons in each layer, type of layers (e.g., ReLU, convolutional, maxpool, etc.).
3. **Choose your cost function**, i.e., how you will measure your network's accuracy. Common examples include the cost in (11.4), or regularizations such as

$$\sum_{\mathrm{i}=1}^{\mathrm{N}} \|\mathbf{y}_\mathrm{i} - \hat{\boldsymbol{f}}(\mathbf{x}_\mathrm{i})\|^2 \; + \; \sum_{\ell=2}^{\mathrm{L}} \|\mathbf{W}^\ell\|_1,$$

   which favors sparse matrices $\mathbf{W}^\ell$, to avoid overfitting as much as possible.
4. **Train your network** (e.g., using stochastic gradient descent; notice that the gradient will change depending on the choice of the cost function) to obtain the *final* parameters $\{\widehat{\mathbf{W}}^\ell, \widehat{\boldsymbol{\beta}}^\ell\}_{\ell=2}^{\mathrm{L}}$.
5. **Use your network.** Given a new data point $\mathbf{x}$, compute its network output as (11.3).