CS 760: Machine Learning Fall 2020

Topic 12: Neural Networks

Instructor: Daniel L. Pimentel-Alarcón © Copyright 2020

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

12.1 Introduction

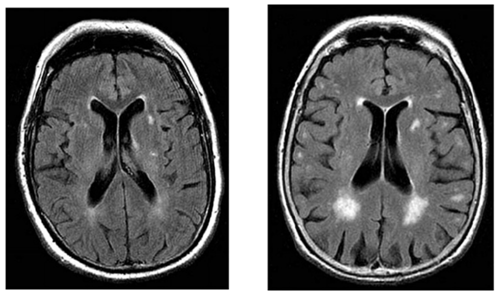
Neural networks are one of the main reasons why Machine Learning is making a big splash. Similar to logistic regression and support vector machines (SVMs), their main purpose is prediction/classification. Classical applications of neural networks include:

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.**

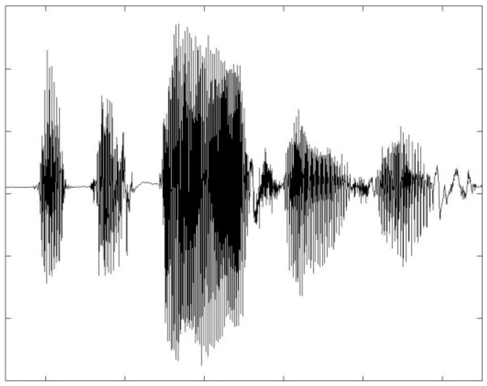
Each of these problems can be rephrased mathematically as finding an unknown (possibly multivariate) function $\boldsymbol{f}^\star$, such that given certain data $\mathbf{x} \in \mathbb{R}^{\mathrm{D}}$, $\boldsymbol{f}^\star(\mathbf{x})$ gives the desired classification/prediction $\mathbf{y}$. In our examples:

1. **Image classification.** In digit classification $\mathbf{x}$ is the vector containing the pixels in the image of a digit, and $\mathrm{y} \in \{0, \dots, 9\}$ is the digit depicted in such image:

   In medical diagnostics, $\mathbf{x}$ could contain the pixels in an MRI, and $\mathrm{y} \in \{0, 1\}$ would be the diagnostic, 0 corresponding to healthy, and 1 to Alzheimers.

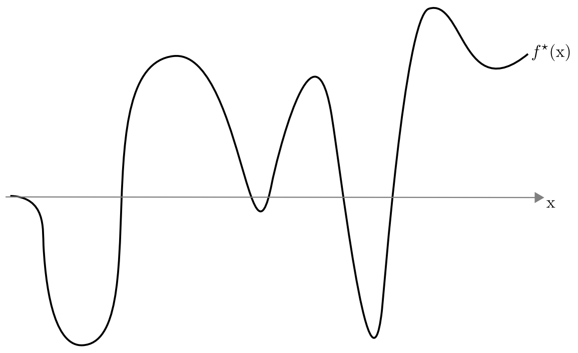


Natural language processing. Here $\mathbf{x}$ is a vector containing the values in a voice signal, and $\mathbf{y}$ is the interpretation, in this case "turn off the t.v.":



Stock market prediction. Here $\mathbf{x}$ could be a sequence of stock market prices at times $\mathrm{t} = 1, \dots, \mathrm{T}$, and y 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. Consequently, $\boldsymbol{f}^{\star}$ could be quite complex.



Neural networks arise as an alternative to approximate complex functions like these. 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 $\boldsymbol{f}^{\star}$ with arbitrary precision.

12.2 The Building Blocks of Neural Networks

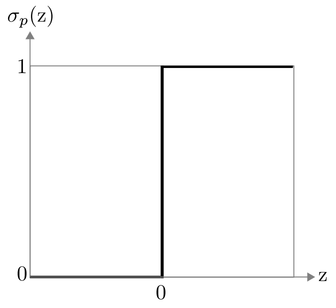
The *perceptron* was the first building block used in neural networks, inspired by the behavior of neurons in the brain. This function would have the form

$$g_p(\mathbf{x}) \;=\; \sigma_p(\boldsymbol{\theta}^{\mathsf{T}}\mathbf{x} - \mathrm{b}) \tag{12.1}$$

where $\boldsymbol{\theta} \in \mathbb{R}^{\mathrm{D}}$ and $\mathrm{b} \in \mathbb{R}$ were the parameters of the perceptron, and the perceptron's *activation function* σ_p is given by

$$\sigma_p(\mathrm{z}) \;=\; \begin{cases} 1 & \text{if } \mathrm{z} > 0, \\ 0 & \text{otherwise.} \end{cases}$$

If you recall from SVMs, this is essentially the function that defines a hyperplane. The main intuition here is that a perceptron would *fire up*, like a neuron, if it receives the right input.



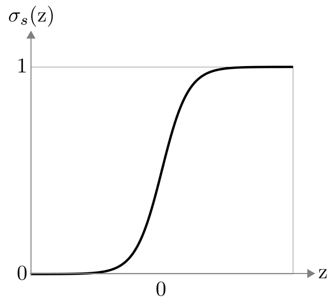
Unfortunately, such a discontinuous activation function can be quite unstable, as a tiny change in the input can produce a massive change in the output, which is a behavior we often don't want. For example, imagine this type of function were used to decide whether a self-driving car should stop or move. It sounds a bit unreasonable that if $\boldsymbol{\theta}^{\mathsf{T}}\mathbf{x} - \mathrm{b} = -0.0001$, then the car stops, and if $\boldsymbol{\theta}^{\mathsf{T}}\mathbf{x} - \mathrm{b} = 0.0001$, then the car moves. Often we would like a softer transition, which is one of the main reasons people decided to use the *sigmoid neuron*

$$g_s(\mathbf{x}) \;=\; \sigma_s(\boldsymbol{\theta}^{\mathsf{T}}\mathbf{x} - \mathrm{b})$$

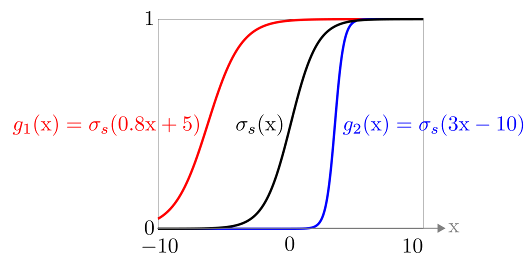
which has the same form as (12.1), only instead of σ_p, it uses the *logistic sigmoid* activation function

$$\sigma_s(\mathrm{z}) \;:=\; \frac{1}{1 + e^{-\mathrm{z}}},$$

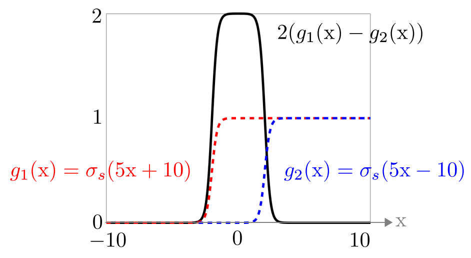
which looks like this:



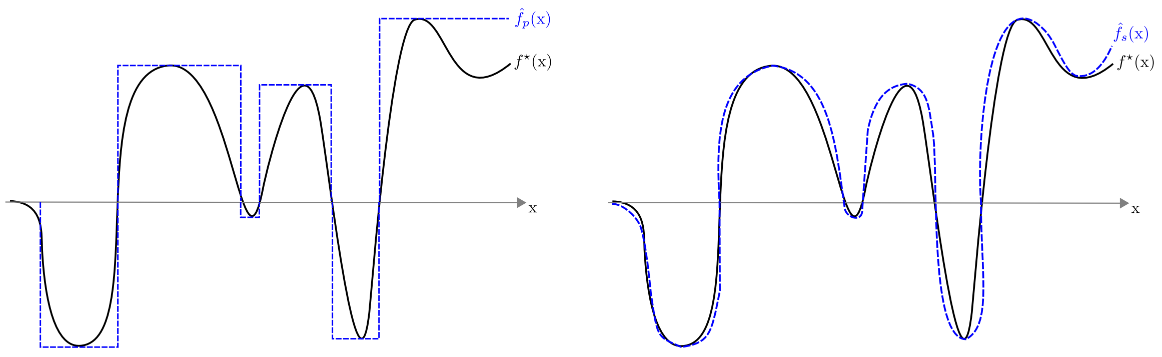
Notice that $g(\mathrm{x}) = \sigma(\theta\mathrm{x} - \mathrm{b})$ *shifts* and *squeezes* the function $\sigma(\mathrm{x})$ by b and θ, for example:



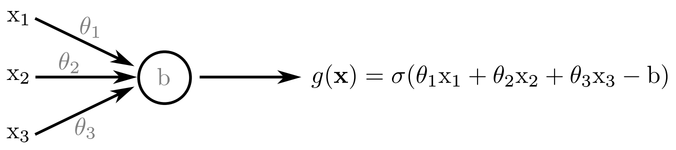
So by adding and scaling bunch of these functions g, each with appropriate parameters θ and b, we can create *soft step* functions, for example:



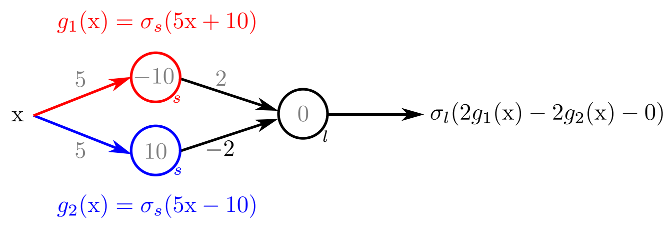
Hence, by adding and scaling enough of these functions g (either perceptrons g_p or sigmoid neurons g_s), we can approximate any function $\boldsymbol{f}^\star$ to arbitrary precision, kind of like a Riemann-type approximation:



The composition of functions g is often represented through a network, where each function g is depicted by a node (a.k.a. neuron) like this:

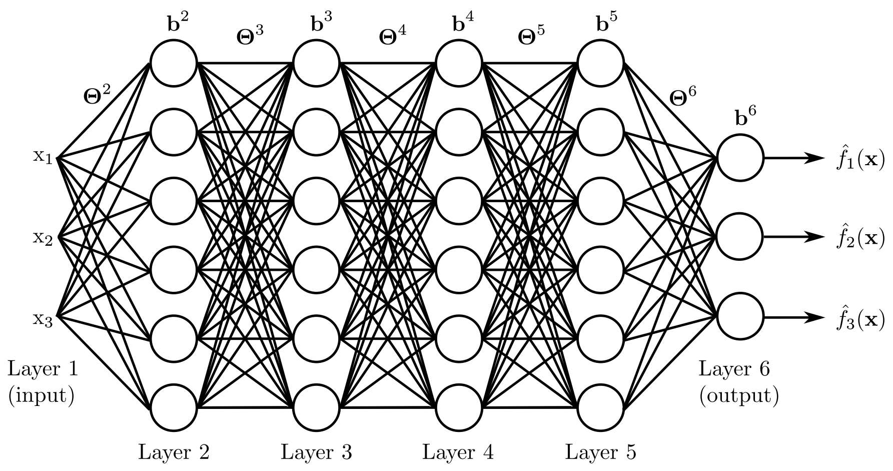

x_1
x_2
x_3
θ_1
θ_2
θ_3
b
$g(\mathbf{x}) = \sigma(\theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 - b)$

Notice that the activation function σ does not have to be restricted to σ_p or σ_s. One can use other activation functions, such as the *linear* activation function $\sigma_l(z) := z$, or the so-called *rectified linear unit* (ReLU) $\sigma_r(z) := \max(0, z)$, or the *tanh* $\sigma_h(z) := 2\sigma_s(2z) - 1$, or combinations. For example, a representation of the soft step function above would look like:



$g_1(x) = \sigma_s(5x + 10)$
x
5
5
-10
s
10
s
2
-2
0
l
$\sigma_l(2g_1(x) - 2g_2(x) - 0)$
$g_2(x) = \sigma_s(5x - 10)$

where the neurons on the left have sigmoid activation functions, and the neuron on the right has a linear activation function. More generally, one can add more neurons to obtain more powerful networks, capable of approximating more complex functions (the downside of larger networks is that they will have more parameters that will have to be learnt, and one runs the risk of overfitting). In general, we consider networks like the following:



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

$$\boldsymbol{g}^2(\mathbf{x}) = \sigma_2(\mathbf{\Theta}^2\mathbf{x} - \mathbf{b}^2). \tag{12.2}$$

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

$$\boldsymbol{g}^\ell(\mathbf{x}) = \sigma_\ell(\mathbf{\Theta}^\ell \boldsymbol{g}^{\ell-1}(\mathbf{x}) - \mathbf{b}^\ell), \tag{12.3}$$

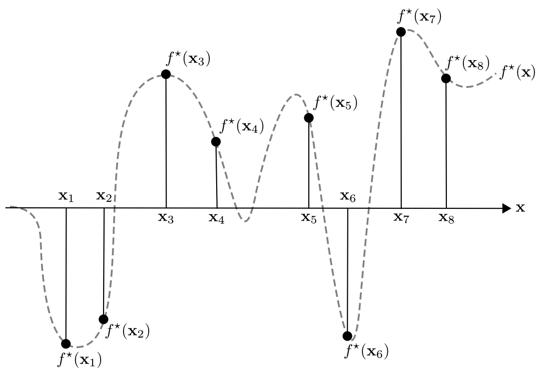
and the final output of the network, $\boldsymbol{g}^L(\mathbf{x})$, is also denoted as:

$$\hat{\boldsymbol{f}}(\mathbf{x}) \;:=\; \underbrace{\sigma_L(\mathbf{\Theta}^L \underbrace{\sigma_{L-1}(\mathbf{\Theta}^{L-1} \cdots \underbrace{\sigma_3(\mathbf{\Theta}^3 \underbrace{\sigma_2(\mathbf{\Theta}^2\mathbf{x} - \mathbf{b}^2)}_{\boldsymbol{g}^2(\mathbf{x})} - \mathbf{b}^3)}_{\boldsymbol{g}^3(\mathbf{x})} \cdots - \mathbf{b}^{L-1})}_{\boldsymbol{g}^{L-1}(\mathbf{x})} - \mathbf{b}^L)}_{\boldsymbol{g}^L(\mathbf{x})}, \tag{12.4}$$

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

12.3 Learning $\hat{\boldsymbol{f}}$

By the arguments above, with the right parameters $\boldsymbol{\Theta} := \{\boldsymbol{\Theta}^\ell, \mathbf{b}^\ell\}_{\ell=2}^{\mathrm{L}}$, the function $\hat{\boldsymbol{f}}$ in (12.4) can approximate any function $\boldsymbol{f}^\star$ with arbitrary accuracy. The challenge is to find the right parameters $\{\boldsymbol{\Theta}^\ell, \mathbf{b}^\ell\}_{\ell=2}^{\mathrm{L}}$. The process of finding such parameters is often known as *learning*. To this end, we use *training* data, that is, 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. Since $\mathbf{y}_{\mathrm{i}} = \boldsymbol{f}^\star(\mathbf{x}_{\mathrm{i}})$ for every $\mathrm{i} = 1, 2, \ldots, \mathrm{N}$, this means that we don't get to see *all* of $\boldsymbol{f}^\star$, but we get to see N *snapshots* of $\boldsymbol{f}^\star$ at the samples $\mathbf{x}_{\mathrm{i}}$:



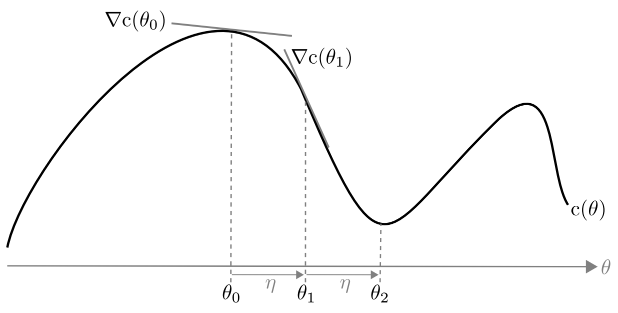
Our goal is to exploit this information to find the parameters $\{\boldsymbol{\Theta}^\ell, \mathbf{b}^\ell\}_{\ell=2}^{\mathrm{L}}$ such that $\hat{\boldsymbol{f}} \approx \boldsymbol{f}^\star$, such that the function (network) can reproduce the response $\mathbf{y}$ whenever a new vector $\mathbf{x}$ 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

$$\min_{\{\boldsymbol{\Theta}^\ell, \mathbf{b}^\ell\}_{\ell=2}^{\mathrm{L}}} \sum_{\mathrm{i}=1}^{\mathrm{N}} \|\mathbf{y}_{\mathrm{i}} - \hat{\boldsymbol{f}}(\mathbf{x}_{\mathrm{i}})\|^2. \tag{12.5}$$

Notice that the dependency of $\{\boldsymbol{\Theta}^\ell, \mathbf{b}^\ell\}_{\ell=2}^{\mathrm{L}}$ is hidden in $\hat{\boldsymbol{f}}$ (see (12.4)).

12.4 Backpropagation

The most widely used technique to minimize the *cost* c in (12.5) is through *stochastic gradient descent* and *backpropagation*. Recall that gradient descent iteratively moves a step of size η in the negative direction of the gradient:



This, of course, requires computing the gradient of each parameter. The key insight to achieve this is that the gradient of the parameters at the ℓ^{th} layer can be computed *backwards* in terms of the gradients of the parameters 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 $\ell = 2, 3, \dots, \mathrm{L}$, recursively define $\mathbf{z}_{\mathrm{i}}^{\ell} := \boldsymbol{\Theta}^{\ell}\mathbf{y}_{\mathrm{i}}^{\ell-1} - \mathbf{b}^{\ell}$, where $\mathbf{y}_{\mathrm{i}}^{\ell} := \boldsymbol{g}^{\ell}(\mathbf{x}_{\mathrm{i}}) = \sigma_{\ell}(\mathbf{z}_{\mathrm{i}}^{\ell})$ 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

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

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

$$\nabla_{\mathrm{i}}\boldsymbol{\Theta}^{\ell} := \frac{\partial\|\mathbf{c}_{\mathrm{i}}\|^2}{\partial\boldsymbol{\Theta}^{\ell}} = -2\ \boldsymbol{\delta}_{\mathrm{i}}^{\ell}\ (\mathbf{y}_{\mathrm{i}}^{\ell-1})^{\mathsf{T}}, \qquad \nabla_{\mathrm{i}}\mathbf{b}^{\ell} := \frac{\partial\|\mathbf{c}_{\mathrm{i}}\|^2}{\partial\mathbf{b}^{\ell}} = -2\ \boldsymbol{\delta}_{\mathrm{i}}^{\ell}, \qquad 2 \leq \ell \leq \mathrm{L}. \tag{12.6}$$

With these results, one can use gradient descent, or, to reduce computational burden an increase convergence speed, *stochastic* gradient descent, which at each training time t selects a random subsample $\Omega_{\mathrm{t}} \subset \{1, 2, \dots, \mathrm{N}\}$ of the training data (hence the term *stochastic*), and updates the parameters according to this subsample:

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

These iterations are repeated until convergence to obtain the *final* parameters $\{\hat{\boldsymbol{\Theta}}^{\ell}, \hat{\mathbf{b}}^{\ell}\}_{\ell=2}^{\mathrm{L}}$.

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 $\hat{\boldsymbol{f}}$ with parameters $\{\boldsymbol{\Theta}^{\ell}, \mathbf{b}^{\ell}\}_{\ell=2}^{\mathrm{L}}$ such that,

$$\|\boldsymbol{f}^{\star}(\mathbf{x}) - \hat{\boldsymbol{f}}(\mathbf{x})\|_2 < \epsilon \qquad \text{for every } \mathbf{x} \text{ in the domain of } \boldsymbol{f}^{\star}. \tag{12.7}$$

The challenge is to find the parameters $\{\boldsymbol{\Theta}^{\ell}, \mathbf{b}^{\ell}\}_{\ell=2}^{\mathrm{L}}$ for which (12.7) is true. We aim to find such parameters by minimizing the cost function in (12.5). The wrinkle is that such cost may be non-convex, which implies that we may *never* find the right parameters $\{\boldsymbol{\Theta}^{\ell}, \mathbf{b}^{\ell}\}_{\ell=2}^{\mathrm{L}}$. In other words, for every $\boldsymbol{f}^{\star}$ there will always exist *a* neural network (parametrized by $\{\boldsymbol{\Theta}^{\ell}, \mathbf{b}^{\ell}\}_{\ell=2}^{\mathrm{L}}$) that approximates $\boldsymbol{f}^{\star}$ arbitrarily well. However, we may be unable to find such network.

12.5 Neural Networks Flavors

In practice, a specific network may work be better than another for each problem. Unfortunately, options (number of layers, number of neurons in each layer, type of layers, connectivity, etc.) are endless, and deciding on the best choice remains more of an art than science. However, people have empirically identified certain strategies that tend to work well for certain tasks.

Starting from encoding, there are several creative and popular strategies. For example:

- **One-hot-encoding.** Nominal features are represented with binary vectors with a single 1. For example, for nucleotides, we could have:

$$A \mapsto \begin{bmatrix}1\\0\\0\\0\end{bmatrix}, \qquad C \mapsto \begin{bmatrix}0\\1\\0\\0\end{bmatrix}, \qquad G \mapsto \begin{bmatrix}0\\0\\1\\0\end{bmatrix}, \qquad T \mapsto \begin{bmatrix}0\\0\\0\\1\end{bmatrix}.$$

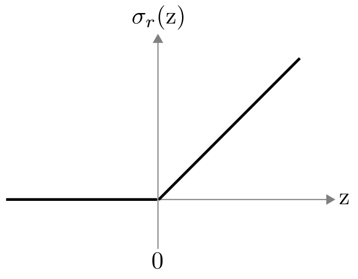
- **Thermometer encoding.** Ordinal features are represented with binary vectors that have an increasing number of ones. For example, for size we could have:

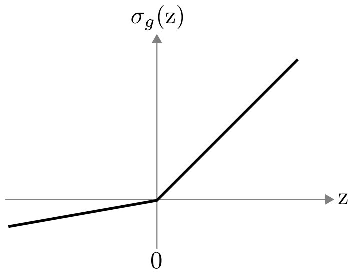
$$\textit{small} \mapsto \begin{bmatrix}1\\0\\0\end{bmatrix}, \qquad \textit{medium} \mapsto \begin{bmatrix}1\\1\\0\end{bmatrix}, \qquad \textit{large} \mapsto \begin{bmatrix}1\\1\\1\end{bmatrix}.$$

Often, the performance of a network depends on choosing the right feature encoding. Similarly, performance heavily depends on the choice of activation functions for each layer. Popular choices for *intermediate* layers (that is, all but the first and last, a.k.a. *hidden* layers) include combinations of the aforementioned linear, ReLU, sigmoid, tanh activation functions, and variants of these, for example the *generalized ReLU* with parameter α:

$$\sigma_g(\mathrm{z}) \ := \ \max(0,\mathrm{z}) + \alpha \min(0,\mathrm{z}).$$

In particular, when $\alpha = 0$ we recover the standard ReLU, and when α is small (e.g., 0.01), we obtain the so-called *Leaky ReLU*:





As to the output layer, arguably the most popular choice in classification problems is the *softmax* function:

$$\boldsymbol{\sigma}_{sm}(\mathbf{z}) \ = \ \frac{e^{\mathbf{z}}}{\sum_{\mathrm{j}} e^{\mathrm{z_j}}}$$

The intuition here is that the softmax function produces a vector with nonnegative entries that add up to 1, and can thus be interpreted as the probability that the given input belongs to each class. Possibilities do not end with the choice of activation functions. One can also play around with the form of $\boldsymbol{g}$. For example, rather than the linear product in (12.2) and (12.3), one can use a convolution:

$$\begin{aligned} \boldsymbol{g}^2(\mathbf{x}) &= \sigma_2(\boldsymbol{\Theta}^2 * \mathbf{x} - \mathbf{b}^2), \\ \boldsymbol{g}^\ell(\mathbf{x}) &= \sigma_\ell(\boldsymbol{\Theta}^\ell * \boldsymbol{g}^{\ell-1}(\mathbf{x}) - \mathbf{b}^\ell), \qquad\qquad 3 \leq \ell \leq \mathrm{L}, \end{aligned}$$

resulting in the popular *convolution layers*. In addition, one can also play around with the cost function that one aims to optimize. For example, instead of minimizing the squared error in (12.5), one could try to maximize a data likelihood model:

$$\max_{\{\boldsymbol{\Theta}^\ell, \mathbf{b}^\ell\}_{\ell=2}^{\mathrm{L}}} \ \mathbb{P}\Big(\mathbf{Y}, \mathbf{X} \Big| \{\boldsymbol{\Theta}^\ell, \mathbf{b}^\ell\}_{\ell=2}^{\mathrm{L}}\Big),$$

or even a posterior probability:

$$\max_{\{\mathbf{\Theta}^\ell,\mathbf{b}^\ell\}_{\ell=2}^{\mathrm{L}}} \mathbb{P}\Big(\{\mathbf{\Theta}^\ell,\mathbf{b}^\ell\}_{\ell=2}^{\mathrm{L}}\Big|\mathbf{Y},\mathbf{X}\Big),$$

which (by Bayes rule) is equivalent to maximizing the log-likelihood with a prior regularization term:

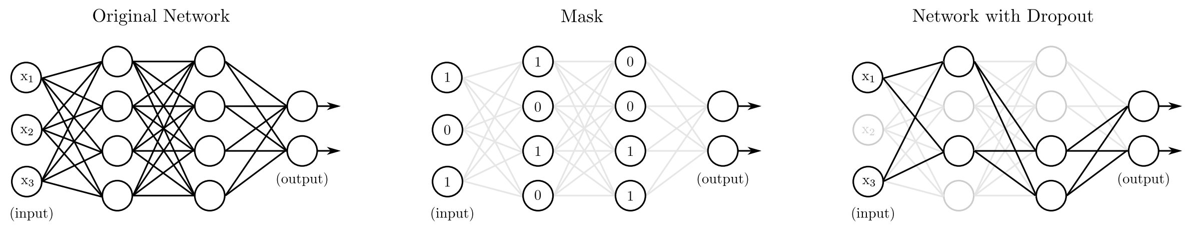
$$\max_{\{\mathbf{\Theta}^\ell,\mathbf{b}^\ell\}_{\ell=2}^{\mathrm{L}}} \log\mathbb{P}\Big(\mathbf{Y},\mathbf{X}\Big|\{\mathbf{\Theta}^\ell,\mathbf{b}^\ell\}_{\ell=2}^{\mathrm{L}}\Big)+\log\mathbb{P}\Big(\{\mathbf{\Theta}^\ell,\mathbf{b}^\ell\}_{\ell=2}^{\mathrm{L}}\Big),$$

resulting in a bayesian framework. Similarly, one can play around with the cost function, adding regularization terms in the interest of favoring certain solutions, or achieving certain goals, like minimizing overfitting. For example, in addition to the squared error in (12.5), one could add a regularization term that favors sparse solutions:

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

where λ is a regularization parameter. Other alternatives to minimize overfitting include:

- **Early Stopping.** The main idea is to stop training as soon as the error on the validation data stops decreasing.
- **Dropout.** The main idea is to pretend at each update step, that a fraction of neurons don't exist (typically 20% of the input layer, and 50% of the intermediate layers). This is typically done by creating a random binary mask whose entries correspond to the input and intermediate neurons, then multiplying each neuron by its corresponding mask entry, and then proceed to do the update as usual.



Data Augmentation. The main idea is to generate additional samples to be used for training, by performing transformations on the original ones. For example, in image processing, typical approaches rotate, flip, translate, scale, crop, and add noise to the original samples: