2.1.4.1 Neural Networks

The idea of neural networks is to generate nonlinearity through a so-called *activation function* $\sigma : \mathbb{R} \to \mathbb{R}$. Popular activation functions include $\mathrm{ReLU} \sigma(t) = \max(t, 0), \mathrm{Sigmoid}\ \sigma(t) = \frac{1}{1+e^{-t}}$ (squashes values to the range $[0, 1]$ which is useful for probabilities), the Heaviside function $\sigma(t) = 1_{t>0}$, or the hyperbolic tangent $\sigma(t) = \tanh(t)$ (squashes values to $[-1, 1]$, similar to sigmoid but centered at 0).
Now consider the basis function $x \mapsto \sigma(\langle w, x \rangle + b)$, which is also called a neuron. (takes an input x applies a weight w, adds bias b and passes the result through an activation function σ). This function is nonlinear but corresponds only to a shift and stretch of the activation function. (The dot product $\langle w, x \rangle + b$ shifts and stretches the input linearly). The activation function σ then bends this linear combination into a **nonlinear** output. To obtain a better basis function, we form a linear combination of n of these basis functions or neurons:

2.1.21

$$\hat{f}(x) := \sum_{j=1}^{n} a_j \sigma(\langle w_j, x \rangle + b_j)$$

Here:

- n: Number of neurons (the **width** of the network)
- a_j: Coefficient for the j-th neurons output
- w_j, b_j: Weight and bias for the j-th neuron

Key terminology

1. **Neuron**: The function $x \mapsto \sigma(\langle w_j, x \rangle + b_j)$
2. **Width** (n): The number of neurons in the hidden layer
3. **Weights** (w_j): Parameters controlling how inputs are combined
4. **Biases** (b_j): Parameters shifting the activation threshold
5. **Layers**:
   1. **Input layer**: Raw data x
   2. **Hidden layer**: Neurons ($j = 1, \ldots, n$) processing inputs
   3. **Output layer**: The final prediction $\hat{f}(x)$
   Each neuron contributes d weights ($w_j \in \mathbb{R}^d$ for d-dimensional input), 1 bias (b_j) and 1 coefficient (a_j).
   The function $\hat{f}$ now depends on $(2 + d) \cdot n$ many parameters that need to be determined.

Example

For $n = 10$ neurons and $d = 3$ inputs, parameters = $(2 + 3) \cdot 10 = 50$

Back to the script

Equation 2.1.21 is referred to as a neural network with two layers (or also with one hidden layer). The following terminology is very common:

1. The function $x \mapsto \sigma(\langle w_j, x \rangle + b_j)$ is referred to as the j-th neuron
2. $n \in \mathbb{N}$ is the number of neurons or the width of the neural network
3. $\{w_j\}_{j=1,\dots,n}$ are referred to as weights, and $\{b_j\}_{j=1,\dots,n}$ are referred to as bias terms or biases

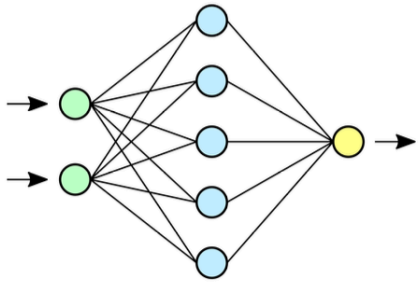

Figure 2.5: Illustration of a neural network with one hidden layer.

Universal Approximation

A special feature of neural networks is that they have to so-called universal approximation property. It means that it can approximate any continuous function on compact subset of $\mathbb{R}^d$.

Theorem 2.1.3

Let the activation function $\sigma : \mathbb{R} \to \mathbb{R}$ be piecewise continuous and not a polynomial, and let $f : [0, 1]^d \to \mathbb{R}$ be any continuous function. Then for every $\epsilon > 0$, there exists $n \in \mathbb{N}$ along with weights $\{w_j\}_{j=1,\dots,n}$ and biases $\{b_j\}_{j=1,\dots,n}$ such that for the corresponding neural network of the form 2.1.21, it holds that

$$\max_{x \in [0,1]^d} |f(x) - \hat{f}(x)| < \epsilon$$

for any $\epsilon > 0$, there exists a neural network $\hat{f}$ such that the network $\hat{f}$ can approximate f **uniformly** over $[0, 1]^d$ with arbitrary small error

The proof is relatively simple but requires some profound theorems from functional analysis. It is clear that for polynomial activation functions σ, the theorem cannot hold. If σ is a polynomial of degree p, then the same holds for $\hat{f}$ regardless of n, w_i or b.
An intuition for why the approximation property holds is the following argument, which we

formulate for simplicity in the case $d = 1$: Every continuous function f can partition $[0, 1]$ into $n \in \mathbb{N}$ intervals of the form $I_j := [\frac{j-1}{n}, \frac{j}{n}]$ for $j = 1, \dots, n$. We define the step function $f_n : [0, 1] \to \mathbb{R}$ by

$$f_n(x) := f\left(\frac{j}{n}\right)$$

where $j \in \{1, \dots, n\}$ is chosen such that $x \in I_j$. Now let $\epsilon > 0$ be given. Since f is continuous on $[0, 1]$ and therefore uniformly continuous, there exists $n \in \mathbb{N}$ such that for all $x, y \in [0, 1]$ with $|x - y| \leq \frac{1}{n}$, it holds that $|f(x) - f(y)| < \epsilon$. Let $x \in [0, 1]$ be arbitrary and let $j \in \{1, \dots, n\}$ be such that $x \in I_j$. Then it follows that

$$|f(x) - f_n(x)| = |f(x) - f\left(\frac{j}{n}\right)| < \epsilon$$

Since x in $[0, 1]$ was arbitrary, we obtain

$$\sup_{x \in [0,1]} |f(x) - f_n(x)| < \epsilon$$

It remains to show that each step function can be approximated by neural networks of the form 2.1.21. Here, we also restrict ourselves to a special case and consider the Heaviside activation $\sigma(t) = 1_{t>0}$. It suffices to show that such neural networks can approximate a step function. Let $f(x) = 1_{s<x\leq t}$ for $s < t$ be a step function. Then choose
$n = 2, a_1 = 1, w_1 = 1, b_1 = -s, a_2 = -1, w_2 = 1, b_2 = -t$ and obtain according to 2.1.21

$$\sum_{j=1}^{2} a_j \sigma(\langle w_j, x \rangle + b_j)$$

$$= 1\sigma(\langle 1, x \rangle - s) + (-1\sigma(\langle 1, x \rangle - t))$$

$$= \sigma(x - s) - \sigma(x - t)$$

Its the same as written below (copied from script)

$$\hat{f}(x) = \sigma(x - s) - \sigma(x - t) = 1_{s<x\leq t} = f(x)$$

Training Neural Networks

As with linear regression, we can use the least squares method to determine the parameters of $\hat{f}$. For this, we consider the problem

2.1.22

$$\min_{\substack{a \in \mathbb{R}^n, \\ W \in \mathbb{R}^{d \times n} \\ b \in \mathbb{R}^n}} \left\{ \mathcal{L}(a, W, b) := \frac{1}{2N} \sum_{i=1}^{N} |a^T \sigma(W^T x_i + b) - y_i|^2 \right\}$$

where we abbreviate $a = (a_1, \dots, a_n), W = (w_1, \dots, w_n)$ and $b = (b_1, \dots, b_n)$ and we used the fact that

$$\sum_{j=1}^{n} a_j \sigma(\langle w_j, x \rangle + b_j) = a^T \sigma(W^T x + b)$$

To determine the parameters a, W, and b, we use a gradient descent method.
Let $L : \mathbb{R}^P \to \mathbb{R}$ be a continuously differentiable function. Then we denote

$$\begin{cases} \text{Initialize } x^0 \in \mathbb{R}^P \\ x^k := x^{k-1} - \tau \nabla L(x^{k-1}), \quad k \in \mathbb{N} \end{cases}$$

This is the gradient descent method for L with step size $\tau > 0$. We want to apply this function $\mathcal{L}(a, W, b)$ and compute the gradients with respect to the three groups of variables.
To do this, we define the residual

$$\text{res}(x_i) := a^T \sigma(W^T x_i + b) - y_i$$

and we obtain

$$\nabla_a \mathcal{L}(a, W, b) = \frac{1}{N} \sum_{i=1}^{N} [\text{res}(x_i) \sigma(W^T x_i + b)] \in \mathbb{R}^n$$

$$\nabla_W \mathcal{L}(a, W, b) = \frac{1}{N} \sum_{i=1}^{N} [\text{res}(x_i) a \odot \sigma'(W^T x + b) x_i^T]^T \in \mathbb{R}^{d \times n}$$

$$\nabla_b \mathcal{L}(a, W, b) = \frac{1}{N} \sum_{i=1}^{N} [\text{res}(x_i) a \odot \sigma'(W^T x + b)] \in \mathbb{R}^n$$

where $\odot$ defines the Hadamard-Product (element-wise product)

Definition (Hadamard product)

For two matrices A and B of the same dimension $m \times n$, the Hadamard product $A \odot B$ is a matrix of the same dimension as the operands, with elements given by

$$(A \odot B)_{ij} = (A)_{ij} (B)_{ij}$$

For matrices of different dimensions the Hadamard product is undefined.
For example, the Hadamard product for two arbitrary 2×3 matrices is:

$$\begin{bmatrix} 2 & 3 & 1 \\ 0 & 8 & -2 \end{bmatrix} \odot \begin{bmatrix} 3 & 1 & 4 \\ 7 & 9 & 5 \end{bmatrix} = \begin{bmatrix} 2 \cdot 3 & 3 \cdot 1 & 1 \cdot 4 \\ 0 \cdot 7 & 8 \cdot 9 & -2 \cdot 5 \end{bmatrix} = \begin{bmatrix} 6 & 3 & 4 \\ 0 & 72 & -10 \end{bmatrix}$$

Properties:

- $A \odot B = B \odot A$

- $A \odot (B \odot C) = (A \odot B) \odot C$
- $A \odot (B + C) = A \odot B + A \odot C$
- $(kA) \odot B = A \odot (kB) = k(A \odot B)$
- $A \odot 0 = 0$

Explanation how the gradients are derived

Remember the Loss function is the MSE:

$$\mathcal{L}(a, W, b) = \frac{1}{2N} \sum_{i=1}^{N} |a^T \sigma(W^T x_i + b) - y_i|^2$$

Explanation for gradient w.r.t. a

Remember the chain rule and set $f(x) = x^2$ and $g(x) = a^T \sigma(W^T x_i + b) - y_i$
By applying $f'(g(x)) \cdot g'(x)$
Focus on single weight a_j (the j-th component of a):

$$\frac{\partial \mathcal{L}}{\partial a_j} = \frac{1}{N} \sum_{i=1}^{N} (a^T \sigma(W^T x_i + b) - y_i) \cdot \sigma(\langle w_j, x_i \rangle + b_j)$$

where $(a^T \sigma(W^T x_i + b) - y_i)$ is the residual (prediction error for x_i)
Alternatively we could do it like this:

$$\mathcal{L}(a, W, b) = \frac{1}{2N} \sum_{i=1}^{N} \text{res}(x_i)^2$$

Now compute $\nabla_a \mathcal{L}$ by differentiating $\mathcal{L}$ with respect to each component a_j of a
Applying the **chain rule**
For each term $\text{res}(x_i)^2$:

$$\frac{\partial}{\partial a_j} \text{res}(x_i)^2 = 2 \cdot \text{res}(x_i) \cdot \frac{\partial}{\partial a_j} \text{res}(x_i)$$

The derivative of $\text{res}(x_i)$ with respect to a_j is:

$$\frac{\partial}{\partial a_j} [a^T \sigma(W^T x_i + b) - y_i] = \sigma(w_j^T x_i + b_j)$$

Here, w_j is the j-th column of W, and $\sigma(W^T x_i + b)$ evaluated element-wise
Now combining those components we get:

$$\nabla_a \mathcal{L} = \frac{1}{N} \sum_{i=1}^{N} \text{res}(x_i) \cdot \sigma(W^T x_i + b)$$

Explanation for gradient w.r.t. W

Remember,

$$\mathcal{L}(a, W, b) = \frac{1}{2N} \sum_{i=1}^{N} \text{res}(x_i)^2, \quad \text{res}(x_i) = a^T \sigma(W^T x_i + b) - y_i$$

Expanding $\text{res}(x_i)$ using the columns w_j of W we get:

$$\text{res}(x_i) = \sum_{j=1}^{n} a_j \sigma(w_j^T x_i + b_j) - y_i$$

Now computing the gradient for the derivative of $\mathcal{L}$ w.r.t one column w_j

$$\frac{\partial \mathcal{L}}{\partial w_j} = \frac{1}{N} \sum_{i=1}^{N} \text{res}(x_i) \cdot \frac{\partial}{\partial w_j} (a^T \sigma(W^T x_i + b) - y_i)$$

Now focus on the last term where we have to compute the gradient for the residual:

$$\frac{\partial}{\partial w_j} (a^T \sigma(W^T x_i + b) - y_i) = \frac{\partial}{\partial w_j} (a^T \sigma(W^T x_i + b)) - \frac{\partial}{\partial w_j} y_i$$

Notice that y_i is a constant and does not depend on w_j thus it disappears
So we have to focus on

$$\frac{\partial}{\partial w_j} (a^T \sigma(W^T x_i + b)) = \frac{\partial}{\partial w_j} (a_j \sigma(w_j^T x_i + b_j)) = a_j \sigma'(w_j^T x_i + b_j) \cdot \frac{\partial}{\partial w_j} (w_j^T x_i + b_j)$$

The derivative of

$$\frac{\partial}{\partial w_j} (w_j^T x_i + b_j) = x_i$$

since w_j is a vector and $w_j^T x_i$ is linear in w_j
Combining the results we get:

$$\frac{\partial}{\partial w_j} (a^T \sigma(W^T x_i + b)) = a_j \sigma'(w_j^T x_i + b_j) x_i$$

The **almost finished gradient** $\nabla_W \mathcal{L}$ aggregates derivatives for all columns $w_1, \ldots, w_n$. For each data point x_i:

$$\nabla_W \mathcal{L} = \frac{1}{N} \sum_{i=1}^{N} \text{res}(x_) \cdot [a \odot \sigma'(W^T x_i + b)] x_i^T$$

where $[a \odot \sigma'(W^T x_i + b)]$ is a vector with components $a_j \sigma'(w_j^T x_i + b_j)$
The outer product $x_i^T \in \mathbb{R}^{1 \times d}$ ensures the gradient $\nabla_W \mathcal{L}$ matches W's dimensions.
$[a \odot \sigma'(W^T x_i + b)] \in \mathbb{R}^{n \times 1}$. BUT this doesn't match the dimensions of W ($d \times n$) for now, the

gradient gives us a $n \times d$ matrix because of the dimensions described above. Because of this we have to transpose everything, so the **FULL** gradient is:

$$\nabla_W \mathcal{L} = \frac{1}{N} \sum_{i=1}^{N} [\text{res}(x_i) a \odot \sigma'(W^T x + b)] x_i^T]^T, \quad \mathbb{R}^{d \times n}$$

Explanation for gradient w.r.t. b

Remember

$$\mathcal{L}(a, W, b) = \frac{1}{2N} \sum_{i=1}^{N} \text{res}(x_i)^2, \quad \text{res}(x_i) = a^T \sigma(W^T x_i + b) - y_i$$

$$\frac{\partial}{\partial b} = \frac{1}{N} \sum_{i=1}^{N} \text{res}(x_i) \cdot \frac{\partial}{\partial b} (a^T \sigma(W^T x_i + b) - y_i)$$

Compute the derivative of the activation function using the chain rule

$$\frac{\partial}{\partial b} \sigma(W^T x_i + b) = \sigma'(W^T x_i + b)$$

The gradient of $\frac{\partial}{\partial x}(a^T \sigma(x)) = a \odot \sigma'(x)$, since this is out outer function. Remember the chain rule $f'(g(x)) \cdot g'(x)$
The $\odot$ symbol represents the Hadamard product (element-wise multiplication) between two vectors. The reason the Hadamard is taken is that when you differentiate the scalar output $a^T \sigma(W^T x_i + b)$ with respect to b is essentially a vector where each element corresponds to the gradient of the respective output dimension. The Hadamard product allows you to combine this vector with the residual, where element-wise multiplication is necessary to correctly propagate the gradient.

The final gradient is thus:

$$= \frac{1}{N} \sum_{i=1}^{N} \text{res}(x_i) \cdot a \odot \sigma'(W^T x_i + b)] \in \mathbb{R}^n$$

Now back to the script

Thus, the gradient descent procedure in this case is given by

$$\begin{cases} \text{Initialize } a \in \mathbb{R}^n, W \in \mathbb{R}^{d \times n}, b \in \mathbb{R}^n \\ \text{For } k \in \mathbb{N} \text{ iterate :} \\ \\ \quad a^k := a^{k-1} - \tau \nabla_a \mathcal{L}(a^{k-1}, W^{k-1}, b^{k-1}) \\ W^k := W^{k-1} - \tau \nabla_W \mathcal{L}(a^{k-1}, W^{k-1}, b^{k-1}) \\ \quad b^k := b^{k-1} - \tau \nabla_b \mathcal{L}(a^{k-1}, W^{k-1}, b^{k-1}) \end{cases}$$

a, W and b start with random values (e.g. sampled from a normal distribution)

Since the number of data points $N \in \mathbb{N}$ is typically very large, a stochastic gradient method with batch size $B << N$ is usually employed:

$$\begin{cases} \text{Initialize } a \in \mathbb{R}^n, W \in \mathbb{R}^{d \times n}, b \in \mathbb{R}^n \\ \text{For } k \in \mathbb{N} \text{ iterate :} \\ \text{Choose a random } B\text{-element subset } \mathcal{I} \text{ from } \{1, \dots, N\} \\ \\ a^k := a^{k-1} - \tau \hat{\nabla}_a \mathcal{L}(a^{k-1}, W^{k-1}, b^{k-1}) \\ W^k := W^{k-1} - \tau \hat{\nabla}_W \mathcal{L}(a^{k-1}, W^{k-1}, b^{k-1}) \\ b^k := b^{k-1} - \tau \hat{\nabla}_b \mathcal{L}(a^{k-1}, W^{k-1}, b^{k-1}) \end{cases}$$

where

$$\hat{\nabla}_a \mathcal{L}(a, W, b) := \frac{1}{B} \sum_{i \in \mathcal{I}} [\text{res}(x_i) \sigma(W^T x_i + b)]$$

and analogously for the W and b variables. (just change N with the batch size B)

$$\hat{\nabla}_W \mathcal{L}(a, W, b) = \frac{1}{B} \sum_{i=1}^{N} [\text{res}(x_i) a \odot \sigma'(W^T x + b) x_i^T]^T$$

$$\hat{\nabla}_b \mathcal{L}(a, W, b) = \frac{1}{B} \sum_{i=1}^{N} [\text{res}(x_i) a \odot \sigma'(W^T x + b)]$$

The random subset $\mathcal{I}$ changes in each iteration
Although neural networks of the form 2.1.21 have the universal approximation property, deep neural networks are often used in practice. These are obtained by nesting individual layers as shown in 2.1.21
To this end, we choose a depth $L \in \mathbb{N}$, activation functions $\sigma_l : \mathbb{R} \to \mathbb{R}$ (activation function for layer l, bias terms $b_l \in \mathbb{R}^{n_l}$ and weight matrices $W^{d_l \times n_l}$ (weight matrix or layer l). Here $d_l \in \mathbb{N}$ is the input dimension and $n_l \in \mathbb{N}$ is the number of neurons in the l-th layer. We define the l-th layer of the neural network as

$$\Phi_l(x) := \sigma_l(W_l^T x + b_l), \quad x \in \mathbb{R}^{d_l}$$

and define a neural network as

$$\hat{f}(x) := \Phi_L \circ \Phi_{L-1} \circ \cdots \circ \Phi_1(x), \quad x \in \mathbb{R}^d$$

where we set $d_1 := d$ (the first layer takes the raw input $x \in \mathbb{R}^d$ and also assume that $d_{l+1} = n_l$ (the output of layer l becomes the input to layer $l+1$) for $l \in \{1, \dots, L-1\}$. The output dimension n_L is equal to 1 for the scalar-valued neural networks but can also be greater than 1 in the case of multidimensional regression.
Example for $L = 3$:

$$\hat{f}(x) = \sigma_3(W_3^T\sigma_2(W_2^T\sigma(W_1^Tx + b_1) + b_2) + b_3)$$

Typically, the activation functions $\sigma_1, \dots, \sigma_{L-1}$ are chosen to be the same (e.g. ReLU or $\mathrm{Sigmoid}$). However, for σ_L, it is usually chosen as $\sigma_L(x) = x$ to preserve the universal approximation property of the inner layers
We now consider a generalization of the least squares problem 2.1.22 by using more general loss functions

2.1.24

$$\min_{\substack{W_l \in \mathbb{R}^{d_l \times n_l} \\ b_l \in \mathbb{R}^{n_l} \\ l=1,\dots,L}} \left\{ \mathcal{L}(a, W, b) := \frac{1}{N} \sum_{i=1}^{N} \ell(\hat{f}(x_i), y_i) \right\}$$

where $\ell : \mathbb{R}^{n_L \times n_L}$ denotes a loss function (e.g. $\ell(\tilde{y}, y) = \frac{1}{2}|\tilde{y} - y|^2$), and we suppress in our notation that $\hat{f}$ depends on the sought parameters W_l, b_l (all weight matrices and biases across layers)

Backpropagation

$$\frac{d\ell(\hat{f}(x), y)}{d(W_l)_{ij}} = (z_{l-1})_i(\delta_l)_j$$

$$\frac{d\ell(\hat{f}(x), y)}{d(b_l)_{ij}} = (\delta_l)_j$$

Where z_{l-1} is the output from the previous layer!
Backpropagation computes gradients of the loss $\ell(\hat{f}(x), y)$ with respect to weights W_l and biases b_l. It works by:

- Forward Pass: Compute predictions $\hat{f}(x)$ and intermediate layer outputs z_l
- Backward Pass: Propagate the "error" δ_l backward through the network to compute the gradients

$\delta_l \in \mathbb{R}^n_l$: The "error" at layer l, defined as:

$$\delta_l = \frac{\delta_l}{\delta a_l}$$

This quantifies how sensitive the loss is to changes in the pre-activation a_l
Now on to explaining $\frac{d\ell(\hat{f}(x), y)}{d(W_l)_{ij}} = (z_{l-1})_i(\delta_l)_j$

- $(z_{l-1})_i$: Activation from neuron i in layer $l-1$
- $(\delta_l)_j$: Error at neuron j in layer l
- **Why?**

- The weight $(W_l)_{ij}$ connects neuron i in layer $l-1$ to neuron j in layer l. The gradient depends on:
  - How much neuron i in layer $l-1$ was activated (z_{l-1})
  - How much error is attributed to neuron j in layer l (δ_l)

Now on to explaining $\frac{d\ell(\hat{f}(x),y)}{d(b_l)_{ij}} = (\delta_l)_j$

- This is the error at neuron j in layer l
- **Why?**
  - The bias $(b_l)_j$ directly offsets the pre-activation a_l. Its gradient is purely the error at neuron j