Foundations of Deep Learning
Lecture Notes

Aurelien Lucchi
Department of Mathematics and Computer Science
University of Basel

September 13, 2025

Contents

Preface

These lecture notes were prepared for a lecture delivered at the Department of Mathematics and Computer Science, University of Basel, Switzerland. They are designed for Master's level students with prior knowledge in machine learning and a solid understanding of mathematics, particularly in algebra and calculus.

Acknowledgment Parts of these lecture notes are based on resources from other researchers and teachers whom I would like to acknowledge:

- **Francois Chollet**: The presentation of Chapter 1 (as well as many figures) is inspired by the book "Deep Learning with Python" by Francois Chollet.
- **Thomas Hofmann**: Portions of Chapters 2, 3, and 8 are derived from the Deep Learning lecture taught by Thomas Hofmann at ETH Zurich.
- **Mihai Nica**: Chapter 6 (NTK) is inspired by the insightful lecture notes authored by Mihai Nica.
- **Matus Telgarsky**: Sections of Chapters 2 and 3 are directly based on the excellent lecture notes available here.

Additionally, many figures are sourced from Wikipedia and occasionally other references. If any uncredited material is discovered, please notify me so that I can properly credit the sources.

Lastly, I want to extend my gratitude to the teaching assistants (Tin Sum Cheng, Navish Kumar, Enea Monzio Compagnoni, Jim Zhao) for their ongoing contributions to enhancing the course material over the years.

Chapter 1

Prerequisites

1.1 Notations & Math symbols

1.1.1 Vectors

A column vector is a $d \times 1$ array that is, an array consisting of a single column of d elements, denoted as

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{bmatrix} .$$

Similarly, a row vector is a $1 \times d$ array that is, an array consisting of a single row of d elements, denoted as

$$\mathbf{x} = \begin{bmatrix} x_1 & x_2 & \dots & x_d \end{bmatrix} .$$

Throughout, boldface is used for the row and column vectors. The transpose (indicated by $\top$) of a row vector is a column vector. We also recall that the inner product between two vectors $\mathbf{u} \in \mathbb{R}^d$ and $\mathbf{v} \in \mathbb{R}^d$ is defined as

$$\mathbf{u}^\top \mathbf{v} = \sum_{i=1}^{d} u_i v_i.$$

The **span** of a set of vectors is the set of all possible linear combinations of those vectors. If we have a set of vectors $\{\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_k\}$ in a vector space V, the span of these vectors, denoted by $\text{span}(\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_k)$, is defined as:

$$\text{span}(\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_k) = \{c_1\mathbf{v}_1 + c_2\mathbf{v}_2 + \cdots + c_k\mathbf{v}_k \mid c_1, c_2, \dots, c_k \in \mathbb{R}\} .$$

Consider the following example illustrated in Figure 1.1 where we show the span of two vectors $\mathbf{v}_1, \mathbf{v}_2 \in \mathbb{R}^2$. Taking for instance coefficients $c_1 = 0.5$ and $c_2 = 1$ yields

the vector $\begin{pmatrix} 0.5 \\ 4 \end{pmatrix}$ since:

$$\underbrace{\begin{pmatrix} -3 & 2 \\ 2 & 3 \end{pmatrix}}_{\mathbf{V}} \begin{pmatrix} 0.5 \\ 1 \end{pmatrix} = 0.5 \underbrace{\begin{pmatrix} -3 \\ 2 \end{pmatrix}}_{\mathbf{v}_1} + 1 \underbrace{\begin{pmatrix} 2 \\ 3 \end{pmatrix}}_{\mathbf{v}_1} = \begin{pmatrix} 0.5 \\ 4 \end{pmatrix}$$

As another example, consider the span of two dependent vectors $\mathbf{v}_2 = 2\mathbf{v}_1$, then one can check that we can only obtain a multiple of the vector $\mathbf{v}_1$, for instance:

$$\underbrace{\begin{pmatrix} 1 & 2 \\ 2 & 4 \end{pmatrix}}_{\mathbf{V}} \begin{pmatrix} 1 \\ 0.5 \end{pmatrix} = 1 \underbrace{\begin{pmatrix} 1 \\ 2 \end{pmatrix}}_{\mathbf{v}_1} + 0.5 \underbrace{\begin{pmatrix} 2 \\ 4 \end{pmatrix}}_{\mathbf{v}_1} = \begin{pmatrix} 2 \\ 4 \end{pmatrix}$$

If we consider the vectors $\mathbf{v}_1$ and $\mathbf{v}_2$ as columns of a matrix $\mathbf{V}$, then the linear combinations of these vectors constitute the column space, as described in the definition below. A similar definition applies to the row space.

Definition 1. *The column space (also called the range or image) of a matrix* $\mathbf{A}$ *is the span (set of all possible linear combinations) of its column vectors.*

1.1.2 Matrices

Matrices will be denoted by a boldface capital letter, for instance

$$\mathbf{A} = \begin{pmatrix} A_{11} & \dots & A_{1d} \\ & \dots & \\ A_{p1} & \dots & A_{pd} \end{pmatrix}, \quad \mathbf{A}^\top = \begin{pmatrix} A_{11} & \dots & A_{p1} \\ & \dots & \\ A_{1d} & \dots & A_{pd} \end{pmatrix}$$

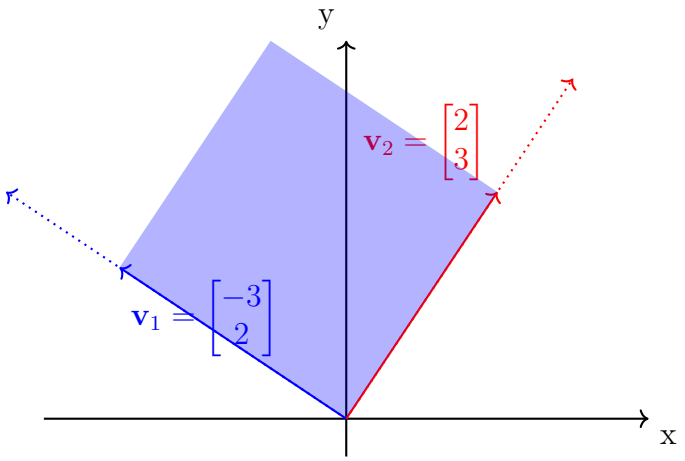


Figure 1.1: Illustration of the span of two vectors $\mathbf{v}_1$ and $\mathbf{v}_2$ in $\mathbb{R}^2$. The blue parallelogram shows the span of these two vectors for coefficients $\mathbf{c} = \begin{pmatrix} c_1 \\ c_2 \end{pmatrix}$ where $c_i \in [0, 1]$.

Recall that the multiplication of two matrices $\mathbf{A} \in \mathbb{R}^{m \times n}, \mathbf{B} \in \mathbb{R}^{n \times d}$ is

$$\mathbf{C} = \mathbf{AB}, \quad C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}.$$

The matrix multiplication operation has the following properties:

- **Distributive property:** Matrix multiplication is distributive over matrix addition. For matrices $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ of compatible dimensions,

$$\mathbf{A}(\mathbf{B} + \mathbf{C}) = \mathbf{AB} + \mathbf{AC} \quad \text{and} \quad (\mathbf{A} + \mathbf{B})\mathbf{C} = \mathbf{AC} + \mathbf{BC}.$$

- **Associative property:** Matrix multiplication is associative. For matrices $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ of compatible dimensions,

$$(\mathbf{AB})\mathbf{C} = \mathbf{A}(\mathbf{BC}).$$

- **Not commutative:** Matrix multiplication is generally not commutative. For matrices $\mathbf{A}$ and $\mathbf{B}$ of compatible dimensions,

$$\mathbf{AB} \neq \mathbf{BA} \quad \text{in general.}$$

The **inverse** of a matrix $\mathbf{A}$ is a matrix, denoted by $\mathbf{A}^{-1}$, such that when it is multiplied by $\mathbf{A}$, it yields the identity matrix. Specifically, for an $n \times n$ matrix $\mathbf{A}$,

$$\mathbf{A}\mathbf{A}^{-1} = \mathbf{A}^{-1}\mathbf{A} = \mathbf{I}_n,$$

where $\mathbf{I}_n$ is the $n \times n$ identity matrix.

1.1.3 Notation

- Bold small letter $\mathbf{x}$ is a column vector
- Bold capital letter $\mathbf{A}$ is a matrix
- $\mathbf{x}^\top, \mathbf{A}^\top$: transpose of a vector (i.e. a row vector) and a matrix respectively
- $\mathbb{R}$: reals
- $a := b$: a is defined by b
- $\frac{\partial f}{\partial x}$: partial derivative of a function f with respect to x
- $\frac{df}{dx}$: total derivative of a function f with respect to x
- ∇: gradient
- $\|\cdot\|$: norm (by default $\|\mathbf{x}\| = \|\mathbf{x}\|_2$).

Math symbols

- $C(X, Y)$ Space of continuous functions $f : X \to Y$.
- $C(\mathbb{R})$ space of continuous functions $f : \mathbb{R} \to \mathbb{R}$, usually endowed with the uniform norm topology
- $C_c(\mathbb{R})$ continuous functions with compact support
- $C([a, b])$ space of all continuous functions that are defined on a closed interval $[a, b]$
- $C_b(\mathbb{R})$ space of continuous bounded functions
- $B(\mathbb{R})$ bounded functions
- $C_0(\mathbb{R})$ continuous functions which vanish at infinity
- $C^r(\mathbb{R})$ continuous functions that have continuous first r derivatives.
- $C^\infty(\mathbb{R})$ smooth functions
- $C_c^\infty(\mathbb{R})$ smooth functions compact support
- $\mathbb{F}$ Field of either real $\mathbb{R}$ or complex numbers $\mathcal{C}$
- ℓ^p is used to indicate a p-summable *discrete* set of values. For example, $\ell^p(\mathbb{Z}^+)$ is the set of complex-valued sequences $\{(a_n)\}$ such that $\sum_{n\in\mathbb{Z}^+} |a_n|^p < \infty$. For example:
  - ℓ^1, the space of sequences whose series is absolutely convergent
  - ℓ^2, the space of square-summable sequences, which is a Hilbert space
  - ℓ^∞, the space of bounded sequences
- L^p is typically used to indicate p-summable functions (with respect to some measure) on a *non-discrete* measure space, such as the usual $L^p(\mathbb{R})$, the set of functions $f : \mathbb{R} \to \mathbb{C}$ such that $\int_{\mathbb{R}} |f(x)|^p \, dx < \infty$.
- The expectation of a random variable X is denoted by $\mathbb{E}[X]$ or $\mathbb{E}_D[X]$ to make the distribution of X, denoted by D, explicit
- Given a symmetric matrix $\mathbf{A}$, $\mathbf{A} \succcurlyeq 0$ means that $\mathbf{A}$ is positive semidefinite, i.e. $\mathbf{x}^\top \mathbf{A}\mathbf{x} \geq 0$ for all $\mathbf{x}$.

1.2 Vector spaces

Vector space Recall that a vector space (also called a linear space) is a collection of objects called vectors, which may be added together and multiplied by scalars. Formally, a vector space over a field F (e.g. the field of real numbers) is a set V together with two operations that satisfy several axioms (associativity, commutativity, identity, inverse).

Subspace A subspace of a vector space V is a subset U of V that is itself a vector space under the same operations of vector addition and scalar multiplication as those defined on V. One simple example is a line through the origin in $\mathbb{R}^2$.

Norm A normed vector space is a space equipped with a norm, i.e. a function from V to $\mathbb{R}$ (we give a formal definition of a norm below). Informally this means that we need to be able to add vectors and scale them with a scalar.

Definition 2. *Given a vector space V over a subfield F of the complex numbers, a norm on V is a nonnegative-valued scalar function $p : V \to [0, +\infty)$ with the following properties:*
For all $a \in F$ and all $u, v \in V$,

- $p(v) \geq 0$ *(non-negativity)*
- *If* $p(v) = 0$ *then* $v = 0$ *(positive definite)*
- $p(av) = |a|p(v)$ *(absolutely homogeneous)*
- $p(u + v) \leq p(u) + p(v)$ *(triangle inequality).*

Example 1: vector norm A vector norm is a function that assigns a non-negative scalar value to a vector in a vector space, which intuitively represents the length or size of the vector. More formally, if $\mathbf{v}$ is a vector in a vector space V, a norm $\|\mathbf{v}\|$ satisfies the following properties:

i) **Non-negativity (or Positivity)**:

$$\|\mathbf{v}\| \geq 0 \quad \text{and} \quad \|\mathbf{v}\| = 0 \iff \mathbf{v} = \mathbf{0}$$

This means the norm of any vector is always non-negative, and it is zero if and only if the vector itself is the zero vector.

ii) **Scalar Multiplication**:

$$\|\alpha\mathbf{v}\| = |\alpha|\|\mathbf{v}\|$$

for any scalar α. This means that scaling a vector by a scalar α scales the norm of the vector by the absolute value of α.

iii) **Triangle Inequality**:

$$\|\mathbf{u}+\mathbf{v}\| \leq \|\mathbf{u}\| + \|\mathbf{v}\|$$

for any vectors $\mathbf{u}$ and $\mathbf{v}$. This means that the norm of the sum of two vectors is less than or equal to the sum of the norms of the two vectors.

Common examples of vector norms include:

- **Euclidean norm (or 2-norm)**:

$$\|\mathbf{v}\|_2 = \left(\sum_{i=1}^{n} |v_i|^2\right)^{1/2}$$

where $\mathbf{v} = (v_1, v_2, \ldots, v_n)$.

- **1-norm (or Manhattan norm)**:

$$\|\mathbf{v}\|_1 = \sum_{i=1}^{n} |v_i|$$

- **Infinity norm (or maximum norm)**:

$$\|\mathbf{v}\|_\infty = \max_{1\leq i\leq n} |v_i|$$

Example 2: matrix norm Suppose a vector norm $\|\cdot\|$ on $\mathbb{R}^m$ is given (e.g. the Euclidean norm). Any $m \times n$ matrix $\mathbf{A}$ induces a linear operator from $\mathbb{R}^n$ to $\mathbb{R}^m$ with respect to the standard basis, and one defines the corresponding operator norm on the space $\mathbb{R}^{m\times n}$ of all $m \times n$ matrices as follows:

$$\begin{aligned}\|\mathbf{A}\| &= \sup\{\|\mathbf{Ax}\| : \mathbf{x} \in \mathbb{R}^n \text{ with } \|\mathbf{x}\| = 1\} \\ &= \sup\left\{\frac{\|\mathbf{Ax}\|}{\|\mathbf{x}\|} : \mathbf{x} \in \mathbb{R}^n \text{ with } \mathbf{x} \neq 0\right\}. \end{aligned} \tag{1.1}$$

In particular, if the p-norm for vectors is used for both spaces $\mathbb{R}^n$ and $\mathbb{R}^m$, the corresponding induced operator norm is:

$$\|\mathbf{A}\|_p = \sup\left\{\frac{\|\mathbf{Ax}\|_p}{\|\mathbf{x}\|_p} : \mathbf{x} \in \mathbb{R}^n \text{ with } \mathbf{x} \neq 0\right\}. \tag{1.2}$$

In practice, one is often interested in the case $p = 2$, for which

$$\begin{aligned}\|\mathbf{A}\|_2 &= \sup\left\{\frac{\|\mathbf{Ax}\|_2}{\|\mathbf{x}\|_2} : \mathbf{x} \in \mathbb{R}^n \text{ with } \mathbf{x} \neq 0\right\} \\ &= \sigma_1(\mathbf{A}), \end{aligned} \tag{1.3}$$

where $\sigma_1(\mathbf{A})$ is the largest singular value of $\mathbf{A}$.

Norm inequalities We introduce the Cauchy-Schwarz inequality, which is a pillar inequality in the field of mathematics and will be a recurrent inequality in many proofs discussed in this course.

Proposition 1. *The Cauchy-Schwarz inequality states that for all vectors* $\mathbf{x}$ *and* $\mathbf{y}$ *of an inner product space, the following holds:*

$$|\langle \mathbf{x}, \mathbf{y} \rangle| \leq \|\mathbf{x}\| \|\mathbf{y}\|. \tag{1.4}$$

Writing $\mathbf{x} = [x_1 \dots x_n]$ *and* $\mathbf{y} = [y_1 \dots y_n]$*, this can also be written as*

$$\left(\sum_{i=1}^{n} x_i y_i \right)^2 \leq \sum_{i=1}^{n} x_i^2 \sum_{i=1}^{n} y_i^2. \tag{1.5}$$

Equality occurs if and only if there exists a constant c *such that* $x_i = c y_i \ \forall i = 1, \dots n$*.*

Proof. We will give two different proofs (there are many more):

Proof I) A very simple proof can be obtained using the definition of an inner product:

$$\mathbf{x}^\top \mathbf{y} = \|\mathbf{x}\| \|\mathbf{y}\| \cos(\theta), \tag{1.6}$$

with the fact that $\cos(\theta) \leq 1$.

Proof II) Another proof is as follows. Define the following non-negative function:

$$\begin{aligned} f(z) &= \sum_{i=1}^{n} (x_i - z y_i)^2 \geq 0 \\ &= \sum_{i=1}^{n} (x_i^2 - 2 x_i y_i z + z^2 y_i^2) \\ &= \left(\sum_{i=1}^{n} y_i^2 \right) z^2 - 2 \left(\sum_{i=1}^{n} x_i y_i \right) z + \sum_{i=1}^{n} x_i^2. \end{aligned}$$

Note that this is a quadratic function in z of the form:

$$f(z) = A z^2 - B z + C,$$

and whose minimum value occurs when $z = \frac{B}{2A}$. Since the whole expression has to be non-negative, we need

$$\Delta = B^2 - 4AC \leq 0 \implies C \geq \frac{B^2}{4A} \implies B^2 \leq 4AC.$$

One can verify that $B^2 \leq 4AC$ implies the inequality we are looking for:

$$\left(\sum_{i=1}^{n} x_i y_i\right)^2 \leq 4\left(\sum_{i=1}^{n} y_i^2\right)\left(\sum_{i=1}^{n} x_i^2\right). \tag{1.7}$$

□

1.3 Functions

We here review the definition of a gradient and then discuss two fundamental properties of functions.

1.3.1 Gradients

Consider a real-valued (univariate) function $f : \mathbb{R} \to \mathbb{R}$. Its derivative is defined by

$$f'(x) := \lim_{\epsilon \to 0} \frac{f(x+\epsilon) - f(x)}{\epsilon}. \tag{1.8}$$

The function f is *differentiable* at x if the limit in Eq. (1.8) exists.

Let's generalize this definition to the case where the function f is multivariate, i.e. $f : \mathbb{R}^d \to \mathbb{R}$. In this case, we have one (partial) derivative for each dimension, i.e. $\frac{\partial f}{\partial x_i}$ defined as

$$\frac{\partial f}{\partial x_i} := \lim_{\epsilon \to 0} \frac{f(x_1, \ldots, x_i + \epsilon, \ldots, x_d) - f(x_1, \ldots, x_i, \ldots, x_d))}{\epsilon}. \tag{1.9}$$

The gradient denoted by $\nabla f(\mathbf{x})$ gives us a way to pack all partial derivatives into one vector. Denoting by $\mathbf{x} = (x_1, x_2, \ldots x_d)$, we then define the gradient as

$$\nabla f(\mathbf{x}) = \begin{pmatrix} \frac{\partial f}{\partial x_1} \\ \vdots \\ \frac{\partial f}{\partial x_d}, \end{pmatrix} \tag{1.10}$$

where we can also define

$$\frac{\partial f}{\partial x_i} := \lim_{\epsilon \to 0} \frac{f(\mathbf{x} + \epsilon \mathbf{e}_i) - f(\mathbf{x})}{\epsilon}, \tag{1.11}$$

with $\mathbf{e}_i$ being a standard basis vector (composed of zeros and a single one at the i-th coordinate). Note that $\frac{\partial f}{\partial x_i}$ is also the directional derivative of f along the direction $\mathbf{e}_i$, and can be expressed as $\nabla f(\mathbf{x}) \cdot \mathbf{e}_i$ (where $\cdot$ is the standard inner product in $\mathbb{R}^d$.

1.3.2 Chain rule

The chain rule is a formula to compute the derivative of a composite function $f(g(x))$. We start by describing the case where the functions involved in the composition are single variable functions, i.e. $f : \mathbb{R} \to \mathbb{R}$ and $g : \mathbb{R} \to \mathbb{R}$. In this case, the chain rule is

$$\frac{d}{dx} f(g(x)) = f'(g(x))g'(x)$$

Alternatively, it can also be written as

$$\frac{d}{dx} f(g(x)) = \frac{df}{dg} \cdot \frac{dg}{dx},$$

where dg is an abbreviation for $dg(x)$.

Proof idea. First note that by definition of the derivative as a limit:

$$(f \circ g)'(a) = \lim_{x \to a} \frac{f(g(x)) - f(g(a))}{x - a}.$$

Assuming that $g(x) \neq g(a)$ any x near a, then the previous expression is equal to the product of two factors:

$$\lim_{x \to a} \frac{f(g(x)) - f(g(a))}{g(x) - g(a)} \cdot \frac{g(x) - g(a)}{x - a} = \lim_{x \to a} \frac{f(g(x)) - f(g(a))}{g(x) - g(a)} \cdot \lim_{x \to a} \frac{g(x) - g(a)}{x - a},$$

where the equality uses the fact that the limit of a product is equal to the product of the limits (limit law).

We have therefore reached the desired expression. The case $g(x) = g(a)$ has to be handled with more care, see e.g. Rudin et al. (1976) for a complete proof.

□

Let's consider the general case where $\mathbf{f} : \mathbb{R}^m \to \mathbb{R}^d$ and $\mathbf{g} : \mathbb{R}^k \to \mathbb{R}^m$. Then the composed function is $f(g(\mathbf{x})) : \mathbb{R}^k \to \mathbb{R}^d$.

In this case, the chain rule is written as

$$\frac{\partial}{\partial \mathbf{x}} \mathbf{f}(\mathbf{g}(\mathbf{x})) = \frac{\partial \mathbf{f}}{\partial \mathbf{g}}(g(\mathbf{x})) \cdot \frac{\partial \mathbf{g}}{\partial \mathbf{x}}(\mathbf{x}) \in \mathbb{R}^{d \times k},$$

where $\frac{\partial \mathbf{f}}{\partial \mathbf{g}}(g(\mathbf{x})) \in \mathbb{R}^{d \times m}$ is the Jacobian matrix of $\mathbf{f}$ with respect to $\mathbf{g}$ and $\frac{\partial \mathbf{g}}{\partial \mathbf{x}}(\mathbf{x}) \in \mathbb{R}^{m \times k}$ is the Jacobian matrix of $\mathbf{g}$ with respect to $\mathbf{x}$.

The Jacobian $\frac{\partial \mathbf{f}}{\partial \mathbf{g}}(g(\mathbf{x})) \in \mathbb{R}^{d \times m}$ contains all first-order partial derivatives of the components of $\mathbf{f}$ w.r.t. the components of $\mathbf{g}$. Specifically, if $\mathbf{f} = [f_1, f_2, \ldots, f_d]^\top$ and $\mathbf{g} = [g_1, g_2, \ldots, g_m]^\top$, then the (i, j)-th entry of this Jacobian matrix is $\frac{\partial f_i}{\partial g_j}$.

1.3.3 Taylor's Theorem

Taylor's theorem gives an approximation of a k-times differentiable function f around a given point.

Scalar functions We first state the scalar version where the function f is defined over $\mathbb{R}$.

Theorem 2 (Taylor's theorem)**.** *Let $k \geq 1$ be an integer and let the function $f : \mathbb{R} \to \mathbb{R}$ be k times differentiable at the point $a \in \mathbb{R}$. Then there exists a function $h : \mathbb{R} \to \mathbb{R}$ such that*

$$f(x) = f(a) + f'(a)(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \cdots + \frac{f^{(k)}(a)}{k!}(x-a)^k + h_k(x)(x-a)^k, \tag{1.12}$$

and

$$\lim_{x \to a} h_k(x) = 0. \tag{1.13}$$

Proof. We prove it for $k = 3$ as the generalization follows easily.

Note that we assumed that the function f is k-times differentiable. We can therefore repeatedly apply the fundamental theorem of calculus as follows:

$$\begin{aligned}
f(x) &= f(a) + (f(x) - f(a)) = f(a) + \int_a^x f'(x_1)dx_1 \\
&= f(a) + \int_a^x f'(a) + (f'(x_1) - f'(a))dx_1 \\
&= f(a) + f'(a)(x-a) + \int_a^x (f'(x_1) - f'(a))dx_1 \\
&= f(a) + f'(a)(x-a) + \int_a^x \int_a^{x_1} f^{(2)}(x_2)dx_2dx_1 \\
&= \dots \\
&= f(a) + f'(a)(x-a) + \int_a^x \int_a^{x_1} f^{(2)}(a) + (f^{(2)}(x_2) - f^{(2)}(a))dx_2dx_1 \\
&= f(a) + f'(a)(x-a) + f^{(2)}(a)(x-a)^2 + \int_a^x \int_a^{x_1} \int_a^{x_2} f^{(3)}(x_3)dx_3dx_2dx_1.
\end{aligned}$$

Recall $f^{(2)}$ is continuous, therefore $|f^{(3)}(x)| \leq M \ \forall x$. The remainder can therefore

be bounded as follows:

$$\begin{aligned}\left|\int_a^x \int_a^{x_1} \int_a^{x_2} f^{(3)}(x_3)\, dx_3 dx_2 dx_1\right| &\leq \int_a^x \int_a^{x_1} \int_a^{x_2} |f^{(3)}(x_3)|\, dx_3 dx_2 dx_1 \\ &\leq M \int_a^x \int_a^{x_1} \int_a^{x_2} 1\, dx_3 dx_2 dx_1 \\ &= M\frac{(x-a)^3}{3!}\end{aligned}$$

□

By choosing $k = 1$ in Theorem 2, we recover the mean value theorem stated below.

Theorem 3 (Mean value theorem (scalar version))**.** *Let $f : [a, b] \to \mathbb{R}$ be a continuous function on the closed interval $[a, b]$, and differentiable on the open interval (a, b), where $a < b$. Then there exists some $c \in (a, b)$ such that*

$$f'(c) = \frac{f(b) - f(a)}{b - a} \tag{1.14}$$

Multivariable functions An extension of the previous result to a multivariate function $f : \mathbb{R}^d \to \mathbb{R}$ can be stated in two ways, either as

$$f(\mathbf{x} + \mathbf{y}) - f(\mathbf{x}) = \nabla f(\mathbf{x} + c\mathbf{y})^\top \mathbf{y}, \text{ for some } c \in (0, 1), \tag{1.15}$$

or in an integral form, as stated in the theorem below.

Theorem 4 (Mean value theorem (multivariable version))**.** *Given a continuously differentiable function $f : \mathbb{R}^d \to \mathbb{R}$ and any vector $\mathbf{y} \in \mathbb{R}^d$, we have that*

$$f(\mathbf{x} + \mathbf{y}) - f(\mathbf{x}) = \int_0^1 \nabla f(\mathbf{x} + t\mathbf{y})^\top \mathbf{y}\, dt. \tag{1.16}$$

Proof. Define $g : [0, 1] \to \mathbb{R}$ as $g(t) = f(\mathbf{x} + t\mathbf{y})$. Then by applying the fundamental theorem of calculus to g, we get

$$f(\mathbf{x} + \mathbf{y}) - f(\mathbf{x}) = g(1) - g(0) = \int_0^1 g'(t) dt.$$

Let $\mathbf{u}(t) = \mathbf{x} + t\mathbf{y}$ with entries $u_i(t)$. By the multivariate chain rule,

$$g'(t) = \sum_{i=1}^d \frac{\partial f(\mathbf{x} + t\mathbf{y})}{\partial u_i} \cdot \frac{\partial u_i}{\partial t} = \sum_{i=1}^d \frac{\partial f(\mathbf{x} + t\mathbf{y})}{\partial u_i} \cdot y_i = \nabla f(\mathbf{x} + t\mathbf{y})^\top \mathbf{y}.$$

Therefore

$$f(\mathbf{x}+\mathbf{y}) - f(\mathbf{x}) = \int_0^1 \nabla f(\mathbf{x}+t\mathbf{y})^\top \mathbf{y}\, dt.$$

□

Exercise Try to prove Eq. (1.15) by modifying the proof of Theorem 4.

High-order variant A similar expression to Eq. (1.16) can be stated for twice differentiable functions:

$$f(\mathbf{x}+\mathbf{y}) = f(\mathbf{x}) + \nabla f(\mathbf{x})^\top \mathbf{y} + \frac{1}{2}\int_0^1 \mathbf{y}^\top \nabla^2 f(\mathbf{x}+t\mathbf{y})\mathbf{y}\, dt. \tag{1.17}$$

Gradient Note that the mean value theorem also applies to the gradient ∇f for functions f that are twice differentiable. It of course also applies to higher-order derivatives if they exist. For the first derivative, we simply get:

$$\nabla f(\mathbf{x}+\mathbf{y}) = \nabla f(\mathbf{x}) + \int_0^1 \nabla^2 f(\mathbf{x}+t\mathbf{y})^\top \mathbf{y}\, dt. \tag{1.18}$$

1.3.4 Calculating gradients

Example: Gradient of a Multivariable Function

Consider the function $f(\mathbf{x}) = \mathbf{A}\mathbf{x} \in \mathbb{R}^p$, where $\mathbf{x} \in \mathbb{R}^d, \mathbf{A} \in \mathbb{R}^{p\times d}$. Our goal is to compute the gradient $\frac{df}{d\mathbf{x}}$.
First, note that the dimension of the gradient $\frac{df}{d\mathbf{x}}$ is $\mathbb{R}^{p\times d}$. Let's compute the partial derivative of f w.r.t. a single x_j. We have

$$f_i(\mathbf{x}) = \sum_{j=1}^{d} A_{ij}x_j \implies \frac{\partial f_i}{\partial x_j} = A_{ij}.$$

Collecting all the partial derivatives in the Jacobian, we obtain the following expression for the gradient:

$$\frac{df}{d\mathbf{x}} = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_d} \\ & \cdots & \\ \frac{\partial f_p}{\partial x_1} & \cdots & \frac{\partial f_p}{\partial x_d} \end{pmatrix} = \begin{pmatrix} A_{11} & \cdots & A_{1d} \\ & \cdots & \\ A_{p1} & \cdots & A_{pd} \end{pmatrix} = \mathbf{A} \in \mathbb{R}^{p\times d}.$$

Composition of functions and chain rule

When considering compositions of functions of vectors or matrices, one often requires the use of the chain rule to calculate gradients. For instance, assume we want

to calculate a loss $L(\mathbf{g}(\mathbf{x})) := \|\mathbf{g}(\mathbf{x})\|_2^2$, where $\mathbf{g} : \mathbb{R}^d \to \mathbb{R}^n$. By the chain rule, we have

$$\frac{\partial L}{\partial \mathbf{x}} = \frac{\partial L}{\partial \mathbf{g}} \cdot \frac{\partial \mathbf{g}}{\partial \mathbf{x}}. \tag{1.19}$$

As an example, let's consider the minimization of the least-square loss (with a linear model), which is defined as

$$\min_{\mathbf{x} \in \mathbb{R}^d} \left[L(\mathbf{A}\mathbf{x}) := \|\mathbf{A}\mathbf{x} - \mathbf{y}\|_2^2 \right], \tag{1.20}$$

where $\mathbf{A} \in \mathbb{R}^{n \times d}$, $\mathbf{x} \in \mathbb{R}^d$ and $\mathbf{y} \in \mathbb{R}^n$ (thus $L : \mathbb{R}^n \to \mathbb{R}$). Sometimes, we will simply write $L(\mathbf{x}) := L(\mathbf{A}\mathbf{x})$.

In this case, we have $\mathbf{g}(\mathbf{x}) = \mathbf{A}\mathbf{x} - \mathbf{y} \in \mathbb{R}^{n \times 1}$ (column vector) and $L(\mathbf{g}) = \|\mathbf{g}\|_2^2$, thus $\frac{\partial L}{\partial \mathbf{g}} = 2\mathbf{g}^\top$. The other derivative is simply $\frac{\partial \mathbf{g}}{\partial \mathbf{x}} = \mathbf{A}$, therefore

$$\frac{\partial L}{\partial \mathbf{x}} = \frac{\partial L}{\partial \mathbf{g}} \cdot \frac{\partial \mathbf{g}}{\partial \mathbf{x}} = 2(\mathbf{A}\mathbf{x} - \mathbf{y})^\top \mathbf{A}. \tag{1.21}$$

Note that with the above notation, $\frac{\partial L}{\partial \mathbf{x}}$ is a row vector since $\frac{\partial L}{\partial \mathbf{x}} \in \mathbb{R}^{1 \times d}$, but by convention, one might want a column vector, in which case we should transpose the result to obtain $2\mathbf{A}^\top(\mathbf{A}\mathbf{x} - \mathbf{y})$.

One alternative to the chain rule for this loss can be obtained by noting that

$$L(\mathbf{x}) := \|\mathbf{A}\mathbf{x} - \mathbf{y}\|_2^2 = (\mathbf{A}\mathbf{x} - \mathbf{y})^\top (\mathbf{A}\mathbf{x} - \mathbf{y}), \tag{1.22}$$

and taking the derivative of the inner product (exercise: try to figure this out yourself).

1.3.5 Lipschitz property

Intuitively, a Lipschitz continuous function is limited in how fast it can change. See the formal definition below.

Definition 3. *The function $f : \mathbb{R}^d \to \mathbb{R}$ is L-Lipschitz continuous if:*

$$|f(\mathbf{x}) - f(\mathbf{y})| \leq L \|\mathbf{x} - \mathbf{y}\| \quad \forall \mathbf{x}, \mathbf{y} \in \mathbb{R}^d. \tag{1.23}$$

The following lemma gives a characterization of Lipschitz continuity using the gradient.

Lemma 5. *A function $f(\mathbf{x})$ is L-Lipschitz continuous if its gradient is bounded by L.*

Proof. By the mean value theorem, we have :

$$f(\mathbf{x}) - f(\mathbf{y}) = \nabla f(\mathbf{y} + c(\mathbf{x} - \mathbf{y}))^\top (\mathbf{x} - \mathbf{y})$$

Using the Cauchy-Schwarz inequality (see Proposition 1), we conclude that

$$|f(\mathbf{x}) - f(\mathbf{y})| \leq \sup_{\mathbf{z} \in \mathbb{R}^d} \|\nabla f(\mathbf{z})\| \|\mathbf{x} - \mathbf{y}\| \leq L \|\mathbf{x} - \mathbf{y}\|.$$

□

1.3.6 Smoothness

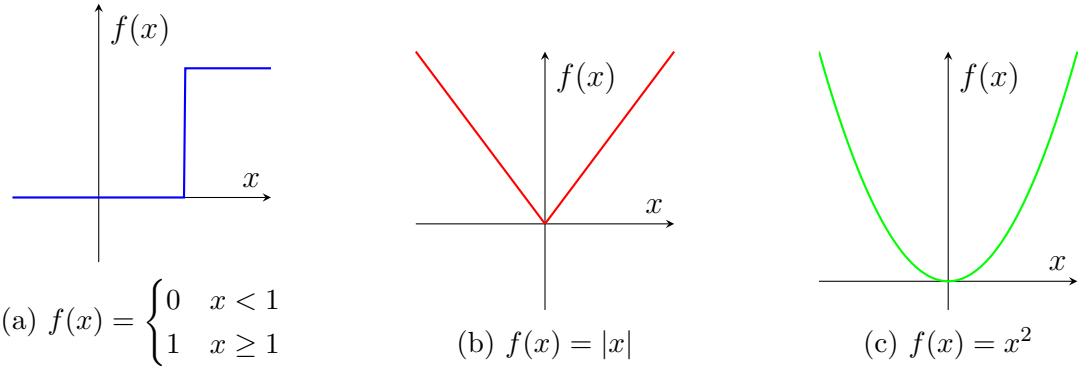
We say that a differentiable function is smooth if its gradient is Lipschitz continuous. We formalize this below.

Definition 4. *The function $f : \mathbb{R}^d \to \mathbb{R}$ is said to be smooth if it is differentiable and it has L-Lipschitz-continuous gradient, i.e. if*

$$\|\nabla f(\mathbf{x}) - \nabla f(\mathbf{y})\| \leq L \|\mathbf{x} - \mathbf{y}\| \, \forall \mathbf{x}, \mathbf{y} \in \mathbb{R}^d. \tag{1.24}$$

Figure 1.2 shows some examples of continuous and differentiable functions. For instance, the function $f(x) = x^2$ is smooth with constant $L = 2$ since:

$$|f'(x) - f'(y)| = 2|x - y|.$$



(a) $f(x) = \begin{cases} 0 & x < 1 \\ 1 & x \geq 1 \end{cases}$ (b) $f(x) = |x|$ (c) $f(x) = x^2$

Continuous: No	Continuous: Yes	Continuous: Yes
Differentiable: No	Differentiable: No	Differentiable: Yes

Figure 1.2: Examples of functions illustrating their continuity and differentiability properties.

1.4 Linear Algebra

1.4.1 Eigenvalues

We recall the definition of eigenvalues and eigenvectors in a real-vector space (this definition can be generalized to the complex domain but we will mostly deal with reals in this lecture). Consider a square matrix $\mathbf{A} \in \mathbb{R}^{d \times d}$ (recall that eigenvalues are not defined for rectangular matrices, for that we need the concept of singular values). Then $\lambda \in \mathbb{R}$ is an eigenvalue of $\mathbf{A}$ and $\mathbf{v} \in \mathbb{R}^d \backslash \{\mathbf{0}\}$ is the corresponding eigenvector of $\mathbf{A}$ if the following holds:

$$\mathbf{A}\mathbf{v} = \lambda \mathbf{v}. \tag{1.25}$$

Note that this relation is also equivalent to $(\mathbf{A} - \lambda \mathbf{I})\mathbf{v} = 0$ or $\det(\mathbf{A} - \lambda \mathbf{I}) = 0$. The polynomial $\det(\mathbf{A} - \lambda \mathbf{I})$ plays an important role in linear algebra. For instance, the number of distinct eigenvalues of $\mathbf{A}$ is equal to the multiplicity of λ as a root of this polynomial.

In the following, we will denote the eigenvalues of $\mathbf{A}$ by λ_i and assume there are sorted as follows:

$$\lambda_1(\mathbf{A}) \geq \ldots \geq \lambda_n(\mathbf{A}).$$

Trace of a matrix The trace of a square matrix $\mathbf{A}$, denoted as $\mathrm{tr}(\mathbf{A})$, is the sum of its diagonal elements. If $\mathbf{A}$ is an $n \times n$ matrix given by:

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}$$

then the trace of $\mathbf{A}$ is defined as:

$$\mathrm{tr}(\mathbf{A}) = \sum_{i=1}^{n} a_{ii}$$

The trace of a matrix is also equal to the sum of its eigenvalues. If $\lambda_1, \lambda_2, \ldots, \lambda_n$ are the eigenvalues of $\mathbf{A}$, then:

$$\mathrm{tr}(\mathbf{A}) = \sum_{i=1}^{n} \lambda_i.$$

Below are some important properties of the trace we will be using later on:

- **Linearity**: For any two $n \times n$ matrices $\mathbf{A}$ and $\mathbf{B}$, and any scalars α and β,

$$\mathrm{tr}(\alpha \mathbf{A} + \beta \mathbf{B}) = \alpha \,\mathrm{tr}(\mathbf{A}) + \beta \,\mathrm{tr}(\mathbf{B})$$

- **Cyclic property**: For any $n \times n$ matrices $\mathbf{A}$ and $\mathbf{B}$,

$$\mathrm{tr}(\mathbf{AB}) = \mathrm{tr}(\mathbf{BA})$$

- **Trace of a transpose**: For any $n \times n$ matrix $\mathbf{A}$,

$$\mathrm{tr}(\mathbf{A}^\top) = \mathrm{tr}(\mathbf{A})$$

Loewner order Recall that a matrix $\mathbf{A}$ is positive semi-definite (PSD) if $\mathbf{u}^\top \mathbf{A}\mathbf{u} \geq 0$ for all $\mathbf{u} \in \mathbb{R}^d$, or equivalently all the eigenvalues λ_i of $\mathbf{A}$ are non-negative, i.e. $\lambda_i(\mathbf{A}) \geq 0 \ \ \forall\, i = 1, \ldots d$. We will use the notation $\mathbf{A} \succcurlyeq 0$ to denote that $\mathbf{A}$ is PSD.

Definition 5 (Loewner order)**.** *Let* $\mathbf{A}$ *and* $\mathbf{B}$ *be two symmetric matrices. We say that* $\mathbf{A} \succcurlyeq \mathbf{B}$ *if* $\mathbf{A} - \mathbf{B}$ *is positive semi-definite.*

Note that $\succcurlyeq$ is only a partial order (instead of a total order).

1.4.2 Singular values

The main object of interest will now be a rectangular $m \times n$ matrix $\mathbf{A}$ with real entries. We denote by $\sigma_i(\mathbf{A})$ the i-th singular value of $\mathbf{A}$ which is equal to

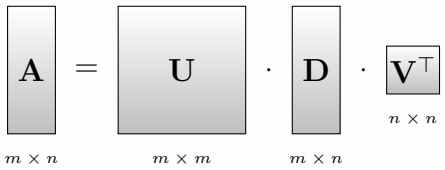
$$\sigma_i(\mathbf{A}) = \sqrt{\lambda_i(\mathbf{A}\mathbf{A}^\top)} = \sqrt{\lambda_i(\mathbf{A}^\top \mathbf{A})}. \tag{1.26}$$

The number of nonzero singular values of $\mathbf{A}$ equals to $\mathrm{rank}(\mathbf{A}) \leq \min(m, n)$, where we recall that the rank of a matrix is the maximum number of linearly independent rows or columns in the matrix.

We also note that given a matrix $\mathbf{A}$, the largest singular value $\sigma_1(\mathbf{A})$ is equal to the operator norm of $\mathbf{A}$.

Singular Value Decomposition (SVD) The Singular Value Decomposition (SVD) is a widely-used technique to decompose a matrix $\mathbf{A}$, and to expose some of its properties.

Theorem 6 (SVD). *Every real (or complex) matrix* $\mathbf{A}$ *can be decomposed into*



where $\mathbf{U}$, $\mathbf{V}$ *orthogonal (or unitary),* $\mathbf{D}$ *diagonal. More precisely:*

- $\mathbf{U}$ *is an* m *by* m *orthogonal matrix,* $\mathbf{U}^\top\mathbf{U} = \mathbf{U}\mathbf{U}^\top = \mathbf{I}$,
- $\mathbf{D}$ *is an* m *by* n *diagonal matrix, padded with* $\max\{m,n\} - \min\{m,n\}$ *zero rows or columns,*
- $\mathbf{V}$ *is an* n *by* n *orthogonal matrix,* $\mathbf{V}^\top\mathbf{V} = \mathbf{V}\mathbf{V}^\top = \mathbf{I}$.

Proof. Omitted. See standard linear algebra textbook, e.g. (Golub and Van Loan, 2013). □

We here recall the definition of orthogonal vectors and matrices for the convenience of the reader.

Definition 6 (Orthogonal vectors). *A set of vectors* $\{\mathbf{v}_1, \ldots, \mathbf{v}_K\}$ *in an inner product space are orthogonal if all pairwise inner products are zero, i.e.*

$$\langle \mathbf{v}_i, \mathbf{v}_j \rangle = 0, \quad (\forall i \neq j).$$

Definition 7 (Orthogonal matrix). *An orthogonal matrix* $\mathbf{A}$ *is a square matrix with real entries whose columns and rows are orthogonal unit vectors (i.e. orthonormal vectors):*

$$\langle \mathbf{a}_i, \mathbf{a}_j \rangle = \delta_{ij} \quad (\forall i, j).$$

Equivalently, $\mathbf{A}\mathbf{A}^\top = \mathbf{A}^\top\mathbf{A} = \mathbf{I}$.

Theorem 7. *The inverse of an orthogonal matrix is its transpose.*

Proof. It follows directly from the above equations as

$$\mathbf{A}^\top\mathbf{A} = \mathbf{I} \quad \Longrightarrow \quad \mathbf{A}^\top = \mathbf{A}^{-1}.$$

□

SVD: Singular Values The elements on the diagonal of $\mathbf{D}$ are the singular values and will be denoted by $\sigma_i := d_{ii}$ $(i = 1, \ldots, \min\{m, n\})$, i.e.

$$\mathbf{D} = \operatorname{diag}\left(\sigma_1, \ldots, \sigma_{\min\{m,n\}}\right).$$

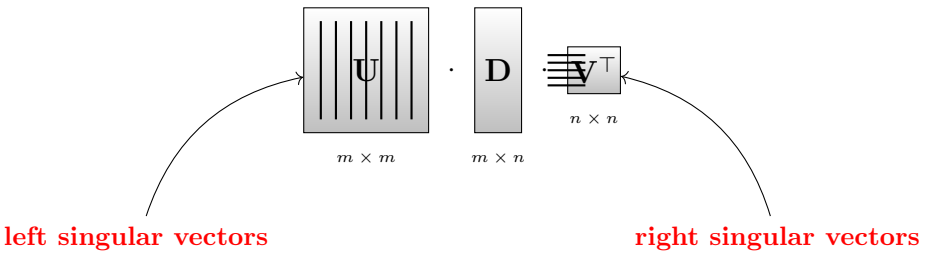
By convention, we typically order the singular values in decreasing order:

$$\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_{\min\{m,n\}} \geq 0$$

Also note the non-negativity of singular values. Finally, an important notion related to singular values is the rank of the matrix $\mathbf{A}$ defined as

$$\sigma_i = d_{ii} = 0, \quad (\forall i > \operatorname{rank}(\mathbf{A})).$$

SVD: Singular Vectors The columns of $\mathbf{U}$ (denoted by $\mathbf{u}_i \in \mathbb{R}^m$) are called the left singular vectors and they form an orthonormal basis for columns space of $\mathbf{A}$. Similarly, the rows of $\mathbf{V}^\top$ = columns of $\mathbf{V}$ (denoted by $\mathbf{v}_i \in \mathbb{R}^n$) are called the right singular vectors and form an orthonormal basis for the row space of $\mathbf{A}$.



The left and right singular vectors $\mathbf{u}_i \in \mathbb{R}^m$ and $\mathbf{v}_i \in \mathbb{R}^n$ are also the eigenvectors of $\mathbf{A}\mathbf{A}^\top$ and $\mathbf{A}^\top\mathbf{A}$ respectively.

We note that the SVD decomposition is only unique up to rotation and degeneracies (i.e. σ_i with two or more linearly independent left (or right) singular vectors).

SVD as sum of rank-1 matrices Let the columns of $\mathbf{U}$ be denoted by $\mathbf{u}_i$ and the columns of $\mathbf{V}$ be denoted by $\mathbf{v}_i$. Then by multiplying the SVD equation by $\mathbf{V}$ one gets

$$\mathbf{A}\mathbf{V} = \mathbf{U}\mathbf{D} \quad \Longleftrightarrow \quad \mathbf{A}\mathbf{v}_i = \sigma_i \mathbf{u}_i \quad (\forall i \leq \min\{m, n\})$$

Similarly

$$\mathbf{A}^\top\mathbf{U} = \mathbf{V}\mathbf{D}^\top \iff \mathbf{A}^\top\mathbf{u}_i = \sigma_i \mathbf{v}_i \quad (\forall i \leq \min\{m, n\})$$

Note that for the special case of $m = n$ and $\mathbf{U} = \mathbf{V}$ (i.e. $\mathbf{A}$ symmetric) we retrieve the eigendecomposition. In this case, the vectors $\mathbf{u}_i = \mathbf{v}_i$ are just the eigenvectors.

Based on the above equations, we observe that we can write the SVD decomposition of a matrix as a sum of rank-1 matrices:

$$\mathbf{A} = \sum_{i=1}^{r} \sigma_i \mathbf{u}_i \mathbf{v}_i^\top, \text{where } r = \text{rank}(\mathbf{A}), \tag{1.27}$$

where $\mathbf{u}_i \mathbf{v}_i^\top$ is an outer product between two vectors and therefore has rank 1 (exercise: try to prove this claim yourself).

Interpretation As shown in Figure 1.3, the SVD gives us a way to decompose a linear map as three subsequent operations: rotation, axis scaling, and another rotation

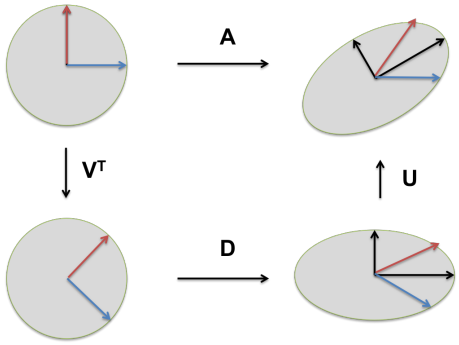


Figure 1.3: Illustration of the SVD transformation where: **Top:** The effect of $\mathbf{A}$ on the unit disc D and the canonical unit vectors $\mathbf{e}_1$ and $\mathbf{e}_2$. **Left:** The effect of $\mathbf{V}^\top$ is a rotation on D, $\mathbf{e}_1$, and $\mathbf{e}_2$. **Bottom:** The effect of $\mathbf{D}$, scaling horizontally by the singular value σ_1 and vertically by σ_2. **Right:** The effect of $\mathbf{U}$, another rotation. Source: Wikipedia.

Relation between singular values and eigenvalues For symmetric matrices, the eigenvalues and singular values are closely related. A nonnegative eigenvalue, $\lambda \geq 0$, is also a singular value, $\sigma = \lambda$. The corresponding vectors are equal to each other, $\mathbf{u} = \mathbf{v}$. A negative eigenvalue, $\lambda < 0$, must reverse its sign to become a singular value, $\sigma = |\lambda|$. One of the corresponding singular vectors is the negative of the other, $\mathbf{u} = -\mathbf{v}$. So in general, if $\mathbf{A}$ is a symmetric matrix then the singular values of $\mathbf{A}$ are the absolute values of the eigenvalues λ_i of $\mathbf{A}$: $\sigma_i(\mathbf{A}) = |\lambda_i(\mathbf{A})|$.

1.5 Probability Theory

We first recall the definition of a probability space.

Definition 8 (Probability space)**.** *A probability space W is a unique triple $W = \{\Omega, \mathcal{F}, \Pr\}$, where Ω is its sample space , $\mathcal{F}$ its σ-algebra of events, and* Pr *its probability measure.*

Sample space The sample space Ω is the set of all possible samples or elementary events ω. Take for example the case where we throw a die once and define the random Variable X (we will later give a formal definition of this concept) as "the score shown on the top face". This random variable X can take the values 1, 2, 3, 4, 5 or 6. Therefore, the sample space is $\Omega := \{1, 2, 3, 4, 5, 6\}$. Let us list a few more examples of sample spaces:

- Toss of a coin (with head H and tail T): $\Omega = \{H, T\}$
- Two tosses of a coin: $\Omega = \{HH, HT, TH, TT\}$
- The positive integers: $\Omega = 1, 2, 3, \ldots$.

σ-algebra A σ-algebra is a collection of subsets of the sample space that satisfies certain properties. The σ-algebra represents the collection of events for which we can assign probabilities. It provides a structure to define and manipulate sets of outcomes or events.

Formally, the σ-algebra $\mathcal{F}$ is the set of all of the *considered* events A, i.e., subsets of Ω: $\mathcal{F} = \{A | A \subseteq \Omega, A \in \mathcal{F}\}$. It also has to satisfy the following properties:

1. $\Omega \in \mathcal{F}$ and $\emptyset \in \mathcal{F}$
2. $A \in \mathcal{F} \Rightarrow \Omega \backslash A \in \mathcal{F}$ (closed under complementation)
3. $A_1, A_2, \cdots \in \mathcal{F} \Rightarrow \cup_n A_n \in \mathcal{F}$ (closed under countable unions)

Below are several examples of σ-algebra:

- **The trivial σ-algebra:**
  The smallest σ-algebra on any set X is the trivial σ-algebra, which contains only the empty set and the set itself.
  $$\mathcal{A} = \{\emptyset, X\}$$
- **The Power set σ-algebra:**
  The largest σ-algebra on a set X is the power set of X, which contains all subsets of X.
  $$\mathcal{A} = \mathcal{P}(X)$$

- **The σ-Algebra generated by a partition:**
  If $X = \{1, 2, 3, 4\}$ and we consider the partition $\{\{1, 2\}, \{3, 4\}\}$, the σ-algebra generated by this partition is:

$$\mathcal{A} = \{\emptyset, \{1, 2\}, \{3, 4\}, \{1, 2, 3, 4\}\}$$

- **The Borel σ-algebra on $\mathbb{R}$:**
  The Borel σ-algebra on the real numbers $\mathbb{R}$, denoted by $\mathcal{B}(\mathbb{R})$, is generated by all open intervals $(a, b) \subset \mathbb{R}$. This σ-algebra contains all Borel sets, which are the sets that can be formed from open intervals using countable unions, countable intersections, and relative complements.

- **σ-Algebra on a finite set:**
  If $X = \{a, b, c\}$, one possible σ-algebra on X could be:

$$\mathcal{A} = \{\emptyset, \{a\}, \{b, c\}, \{a, b, c\}\}$$

- **The σ-Algebra generated by a single set:**
  Let X be a set and $A \subseteq X$. The σ-algebra generated by A is:

$$\mathcal{A} = \{\emptyset, A, A^c, X\}$$

Probability measure A function $\Pr : \mathcal{F} \to [0, 1]$ is called a probability measure if it satisfies the following three properties:

1. Non-negativity: For every $A \in \mathcal{F}$, $\Pr(A) \geq 0$.

2. Normalization: $\Pr(\Omega) = 1$, where Ω is the entire sample space.

3. σ-additivity: For any countable collection $\{A_n\}$ of pairwise disjoint sets in $\mathcal{F}$ (i.e., $A_i \cap A_j = \varnothing$ for $i \neq j$), the probability of the union is equal to the sum of the probabilities of the individual sets:

$$\Pr\left(\bigcup_{n=1}^{\infty} A_n\right) = \sum_{n=1}^{\infty} \Pr(A_n).$$

An illustration of a probability measure is shown in Figure 1.4.

Random variable A random variable is similar to a variable in mathematics, but instead of taking on just one value, it can take on a range of possible values, each with a certain probability. Below, we state a more formal definition of a random variable.

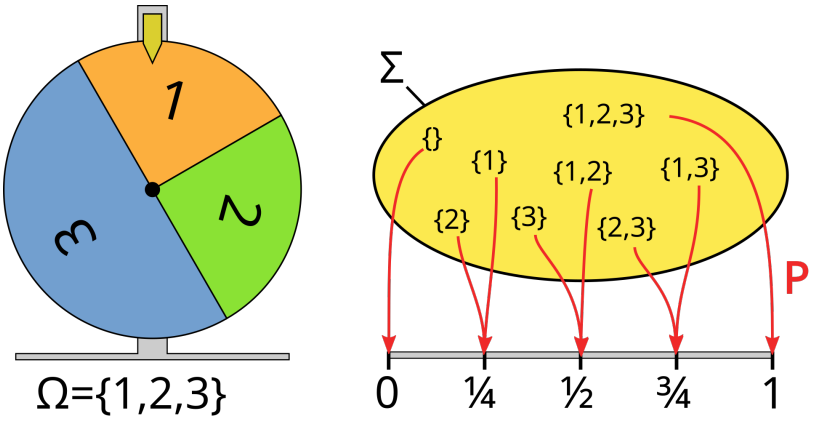


Figure 1.4: A probability measure mapping the σ-algebra for 2^3 events to the unit interval. Source: Wikipedia.

Definition 9 (Random variable)**.** *Given a probability space* $(\Omega, \mathcal{F}, \Pr)$*, a random variable is a function from* Ω *to a measurable space* E [a]*. This function must satisfy the constraint* $\{\omega : X(\omega) \in B\} \in \mathcal{F}$ *for any Borel set* $B \subset E$ [b]*, i.e. it is a valid event* $\forall B \in E$*. Each random variable has a distribution* $F_X(x) = \Pr(X \leq x)$*.*

[a] A measurable space is a set equipped with a collection of subsets that are designated as measurable.

[b] In simple terms, a Borel set is any set that you can create starting from open sets and applying these operations a finite or countable number of times

More formal definition of a Borel set A Borel set is a special type of set that is defined using the concept of open sets in a topological space. The collection of Borel sets includes all open sets and is closed under operations like countable union, countable intersection, and relative complement. For example, consider the real number line. An open interval like $(0, 1)$ is a Borel set because it is an open set. If you take the union of open intervals $(0, 1)$ and $(2, 3)$, you get another Borel set, $(0, 1) \cup (2, 3)$. Even if you take more complex combinations like countable unions or intersections of such intervals, the resulting sets are still Borel sets. So, any set that can be built from open intervals on the real line using operations like union, intersection, and complement will be a Borel set.

Given a set $S \subseteq E$, the probability of a random variable is defined as

$$\Pr(X \in S) = \Pr(\{\omega \in \Omega | X(\omega) \in S\}). \tag{1.28}$$

If X maps onto a finite or countable set, it is *discrete* and has a probability mass function (PMF) where $p_X(x) = \Pr(X = x)$.
If $dF_X(x)/dx$ exists and is finite for all x, then X is continuous and has a density

$f_X(x) = dF_X(x)/dx$.

Example of random variables:
a) Throw two dices and take $\Omega = \{1,2,3,4,5,6\}^2$ and $\mathcal{F} = P(\Omega)$ where P is the power set, i.e. the set of all subsets of Ω (thus $\mathcal{F} = \{(1,1),(1,2),\ldots\}$). Then $X : \Omega \to \mathbb{R}$ defined as $(\omega_1, \omega_2) \to \omega_1 + \omega_2$ (i.e. sum the numbers on each die) is indeed a random variable.
b) Throw two dices and take $\Omega = \{1,2,3,4,5,6\}^2$ and $\mathcal{F} = \{\emptyset, \Omega\}$. Then X as defined in a) is not a random variable. For instance $X^{-1}(\{2\}) = \{(1,1)\} \notin \mathcal{F}$.

Continuous vs discrete probability spaces The following table compares the case where the probability space is continuous and discrete (note that one can also mix them but we won't be discussing this case in these notes).

Characteristic	**Discrete Probability Space**	**Continuous Probability Space**
Definition	Consists of a sample space Ω, a σ-algebra $\mathcal{F}$, and a probability mass function (PMF) $\Pr(X = x)$	Consists of a sample space Ω, a σ-algebra $\mathcal{F}$, and a probability density function (PDF) $f(x)$
Sample Space	Countable and finite or countably infinite, e.g. $\Omega = \{1,2,3,4,5,6\}$ (for a six-sided die)	Uncountably infinite, e.g. $\Omega = [0,1]$
σ-algebra	Typically the power set $\mathcal{F} = P(\Omega)$	Typically the Borel set formed by Ω
Probability Function	PMF $\Pr(X = x)$ for each x in Ω	PDF $f(x)$ such that $\Pr(a \leq X \leq b) = \int_a^b f(x)\,dx$
Probability Assignment	$\Pr(X = x) \geq 0$ for all x and $\sum_{x\in\Omega} \Pr(X = x) = 1$	$f(x) \geq 0$ for all x and $\int_{-\infty}^{\infty} f(x)\,dx = 1$
Probability at a Point	$\Pr(X = x) > 0$ for some x	$\Pr(X = x) = 0$ for all x
Sum/Integral of Probabilities	$\sum_{x\in\Omega} \Pr(X = x) = 1$	$\int_{-\infty}^{\infty} f(x)\,dx = 1$
Random Variable	X takes values in Ω with probabilities assigned by the PMF	X takes values in Ω with probabilities assigned by the PDF
Example	Throwing a six-sided die: $\Pr(X = 1) = \frac{1}{6}, \Pr(X = 2) = \frac{1}{6}, \ldots, \Pr(X = 6) = \frac{1}{6}$	Choosing a point in $[0,1]$: $f(x) = 1$ for $0 \leq x \leq 1$, and $f(x) = 0$ elsewhere

Table 1.1: Differences Between Discrete and Continuous Probability Spaces

Probability density function (PDF) and probability mass function (PMF) A probability density function is most commonly associated with absolutely continuous univariate distributions. A random variable X has density f_X, where f_X is a non-negative Lebesgue-integrable function, if:

$$\Pr[a \leq X \leq b] = \int_a^b f_X(x)\,dx. \tag{1.29}$$

Hence, if F_X is the cumulative distribution function of X, then:

$$F_X(x) = \int_{-\infty}^{x} f_X(u)\,du, \tag{1.30}$$

and (if f_X is continuous at x)

$$f_X(x) = \frac{d}{dx}F_X(x). \tag{1.31}$$

Intuitively, one can think of $f_X(x)\,dx$ as being the probability of X falling within the infinitesimal interval$[x, x + dx]$.

Similar properties hold for discrete probability spaces where the density function $f_X(x)$ is replaced by a discrete function $p : \mathcal{F} \to [0, 1]$ defined by $p_X(x) = \Pr(X = x)$. Since the function p is defined over a discrete set, the integral is replaced by a sum, therefore for a given set A, we have

$$\Pr(A) = \sum_{\omega \in A} p_X(\omega).$$

Expectation If a random variable X has a continuous density $f_X(x)$, then its expectation is given by

$$\mathbb{E}[X] = \int_{-\infty}^{\infty} x f_X(x)\,dx.$$

For a discrete random variable with a finite list $x_1, \ldots, x_k$ of possible outcomes each of which (respectively) has probability $p_1, \ldots, p_k$ of occurring, the expectation can be written as

$$\mathbb{E}[X] = \sum_{i=1}^{k} x_i\, p_i.$$

Finally, the expected value of a non-negative random variable can be written as the integral of its tail distribution. This is because if X is a nonnegative real number, then

$$X = \int_0^{+\infty} [X \geq t]\mathrm{d}t,$$

where $[\cdot]$ is the indicator function.

Then one integrates both sides of the relevant identity with respect to the distribution $\Pr_X$ of X and one uses Fubini's theorem to change the order of the summation/integral and of the expectation:

$$\mathbb{E}[X] = \int_\Omega X \, \mathrm{d}\Pr = \int_\Omega \int_0^{+\infty} [X > t] \mathrm{d}t \, \mathrm{d}\Pr = \int_0^{+\infty} \int_\Omega [X > t] \mathrm{d}\mathbb{P} \, \mathrm{d}t$$

that is,

$$\mathbb{E}[X] = \int_0^{+\infty} \Pr(X \geq t) \, \mathrm{d}t.$$

1.6 Basic Topological Concepts

In this section, we explained some important concepts in topology that are needed for the course. We will only need a basic understanding of these concepts but we refer the interested reader to (Munkres, 2000) for more detailed explanations.

Let S be a subset of a topological space X, i.e. a set whose collection of open subsets satisfies certain conditions. We recall that topological spaces are abstractions of other spaces such as metric spaces, see illustration in Figure 1.5.

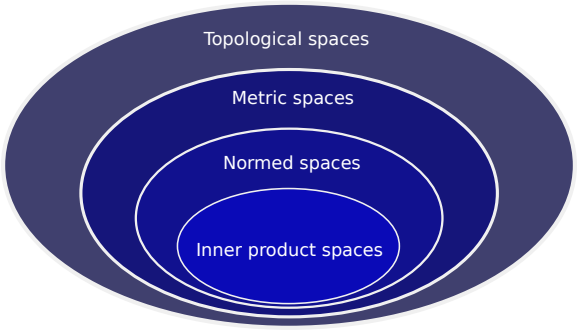


Figure 1.5: Source: Wikimedia Commons

Open set A set is open if it does *not* include any of its boundary points. Formally, the concept of open sets can be formalized with various degrees of generality. For example, in the case of metric spaces, we have the following definition.

Definition 10 (Open set)**.** *A subset U of a metric space (M, d) is called open if, given any point $x \in U$, there exists a real number $\epsilon > 0$ such that, given any point $y \in M$ with $d(x, y) < \epsilon$, y also belongs to U. Equivalently, U is open if every point in U has a neighborhood contained in U.*

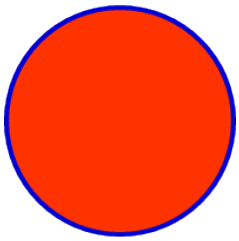


Figure 1.6: Example: The blue circle represents the set of points (x, y) satisfying $x^2 + y^2 = r^2$. The red disk represents the set of points (x, y) satisfying $x^2 + y^2 < r^2$. The red set is an open set, the blue set is its boundary set, and the union of the red and blue sets is a closed set.

Closed sets, Limit points and closure We start with the definition of a closed set.

Definition 11 (Closed set)**.** *A set is closed if its complement is open.*

Note that closed balls are closed sets (proof: show by contradiction that the complement of a closed ball is an open set, i.e. show it violates the triangle inequality).

Definition 12 (Limit point)**.** *We say that $p \in X$ is a limit point of S if every open neighborhood of p contains one point in S (other than p).*

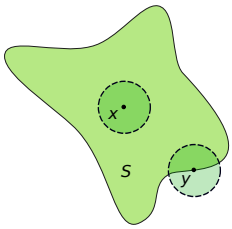


Figure 1.7: Example of limit points: Both x (interior point) and y (boundary point) are limit points of the set S since any neighborhood of these points contain a point in S.

One can give a characterization of a closed set using limit points, see the next proposition.

Proposition 8. *A closed set is a set that contains all of its limit points.*

Proof. Proof by contradiction: Assume a closed set S does not contain all its limit points, i.e. $\exists y \in S^{\complement}$ such that y is a limit point of S. Since $S^{\complement}$ is open and y is an interior point, then there exists a neighborhood of y, denoted by $\mathcal{B}(y, \epsilon)$ s.t. $\mathcal{B}(y, \epsilon) \subseteq S^{\complement}$ but this contradicts the fact that y is a limit point of S.

For the converse, show that any $y \in S^{\complement}$ is not a limit point of S therefore proving $S^{\complement}$ is open. □

The last proposition can also be stated as follows.

Proposition 9. *A set $A \subseteq X$ is closed if and only if for every convergent sequence $(a_n)_{n\in N} \subseteq A$, we have $\lim_{n\to\infty} a_n \in A$.*

Definition 13 (Closure)**.** *The closure of a set A is the union of all its limit points. It is usually denoted by $\bar{A}$ or $cl(A) = A \cup A'$, where A' is the set of all limit points.*

Definition 14 (Dense set)**.** *A subset A of a topological space X is called dense (in X) if every point $x \in X$ either belongs to A or is a limit point of A. Alternatively, A is dense if it has **non-empty intersection** with an arbitrary non-empty open subset $B \subset X$.*

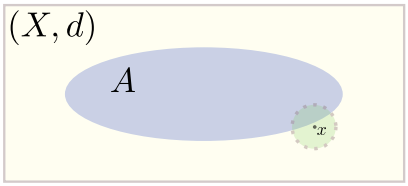


Figure 1.8: The set A shown in blue is dense in X if every $x \in X$ is a limit point of A.

A well-known example is the fact that the rationals are dense in the set of reals, which we formalize in the next theorem.

Theorem 10 (Density theorem, $\mathbb{Q}$ is dense on $\mathbb{R}$)**.**

$$\forall a < b \in \mathbb{R}, \exists x \in \mathbb{Q} \text{ s.t. } x \in (a, b), \text{ i.e. } a < x < b$$

Compact sets. There are typically two characterizations of compact spaces, one in terms of open sets and another one in terms of convergent sequences. We start with the definition in terms of open sets.

Definition 15 (Compact set, definition 1)**.** *A topological space X is called compact if each of its open covers [a] has a finite subcover.*

[a] A cover of a set X is a collection of sets whose union includes X as a subset.

Explicitly, this means that for every arbitrary collection $\{U_\alpha\}_{\alpha\in A}$ of open subsets of X such that $X = \bigcup_{\alpha\in A} U_\alpha$, there is a **finite** subset J of A such that $X = \bigcup_{i\in J} U_i$. See illustration in Figure 1.9.

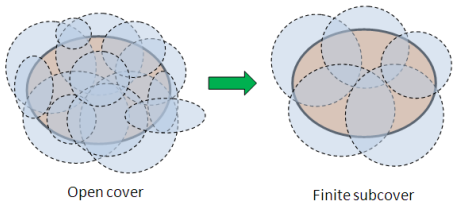


Figure 1.9: Source: `https://mathstrek.blog/`

Definition 16 (Compact set, definition 2)**.** *We call X a compact set if all sequences $(f_n)_{n\geq 1} \subset X$ have a convergent subsequence $(f_{n(k)})$ with limit point in X.*

Finally, we note that the following characterization of compact sets is often used in the literature: a subset of $\mathbb{R}^d$ is compact if it is closed and bounded (Heine-Borel theorem).

1.7 Exercise: Prerequisites

Problem 1 (Operator Norm):
Let $\mathbf{A} \in \mathbb{R}^{m \times n}$ be a rectangular matrix with $m \geq n$.

a) Recall the definition of the operator norm of $\mathbf{A}$:

$$\|\mathbf{A}\|_2 = \sup_{\mathbf{x} \in \mathbb{R}^n \setminus \{0\}} \frac{\|\mathbf{A}\mathbf{x}\|}{\|\mathbf{x}\|}.$$

Write down the definition of a vector norm. Show that the operator norm is indeed a vector norm in the vector space $\mathbb{R}^{m \times n}$ of matrices.

b) By the singular value decomposition (SVD) of $\mathbf{A}$, show that $\|\mathbf{A}\|_2 = s_1(\mathbf{A})$, the largest singular value of $\mathbf{A}$.

c) By SVD of $\mathbf{A}$ again, show that $s_1(\mathbf{A}) = \sqrt{\lambda_1(\mathbf{A}^\top \mathbf{A})}$, where $\lambda_1(\cdot)$ denotes the largest eigenvalue.

Problem 2 (Calculus):

a) Let $\mathbf{A} \in \mathbb{R}^{m \times n}$ be a rectangular matrix, $\mathbf{x} \in \mathbb{R}^n, \mathbf{y} \in \mathbb{R}^m$ be column vectors. Let $f(\mathbf{x}) = \|\mathbf{A}\mathbf{x} - \mathbf{y}\|_2^2$. Compute its gradient $\nabla f \in \mathbb{R}^n$.

b) Let $\mathbf{U} \in \mathbb{R}^{n \times p}, \mathbf{V} \in \mathbb{R}^{p \times m}$, and $\mathbf{R} \in \mathbb{R}^{n \times m}$. The matrix factorization problem tries to find an approximation of $\mathbf{R}$ as the product of two matrices with smaller common dimension p, that is

$$\mathbf{R} \approx \mathbf{U}\mathbf{V}.$$

This problem can be solved for instance by minimizing the loss $L(\mathbf{U}, \mathbf{V}) := \frac{1}{2}\|\mathbf{U}\mathbf{V} - \mathbf{R}\|_F^2$, where $\|\mathbf{A}\|_F := \sqrt{\sum_i \sum_j |A_{ij}|^2}$ is the Frobenius norm.
Compute the derivative of L w.r.t. $\mathbf{U}$, $\frac{\partial L}{\partial \mathbf{U}}$, and w.r.t. $\mathbf{V}$, $\frac{\partial L}{\partial \mathbf{V}}$, respectively.

c) Consider the problem of non-linear least squares regression with some non-linear function $\ell : \mathbb{R} \to \mathbb{R}$ and n data samples $(\mathbf{x}_i, y_i)$,

$$L(\mathbf{w}) = \frac{1}{2n} \sum_{i=1}^{n} \left(y_i - \ell(\mathbf{x}_i^\top \mathbf{w})\right)^2.$$

Compute the gradient $\nabla_{\mathbf{w}} L(\mathbf{w})$ and the Hessian $\nabla_{\mathbf{w}}^2 L(\mathbf{w})$.

Problem 3 (Taylor Expansion):
Let $f : \mathbb{R}^n \to \mathbb{R}$ be a twice differentiable function with a local minimum $\mathbf{x}^*$.

a) Write down the definition of the Hessian $\mathbf{H}(\mathbf{x}) = \nabla^2 f(\mathbf{x}) \in \mathbb{R}^{n\times n}$ of f and the order-2 Taylor expansion of f at $\mathbf{x}^*$.

b) Using Chain rule or otherwise, prove that for any $\mathbf{x}, \mathbf{v} \in \mathbb{R}^n$, the matrix-vector product $\mathbf{H}(\mathbf{x})\mathbf{v} \in \mathbb{R}^n$ is equal to a limit:

$$\mathbf{H}(\mathbf{x})\mathbf{v} = \lim_{t\to 0} \frac{\nabla f(\mathbf{x}+t\mathbf{v}) - \nabla f(\mathbf{x})}{t}.$$

c) Suppose f is a L-smooth function, that is, there is an $L > 0$ such that

$$\|\nabla f(\mathbf{x}_1) - \nabla f(\mathbf{x}_2)\| \le L \|\mathbf{x}_1 - \mathbf{x}_2\|, \forall \mathbf{x}_1, \mathbf{x}_2 \in \mathbb{R}^n.$$

Show that each eigenvalue of $\mathbf{H}(\mathbf{x})$ of any $\mathbf{x} \in \mathbb{R}^n$ is upper bounded by L.

d) Recall that $\mathbf{x}^*$ is a local minimum. Using Problem 1c) and 3c) or otherwise, prove that

$$\|\mathbf{H}(\mathbf{x}^*)\|_2 \le L$$

e) Using Cauchy-Schwartz inequality and Problem 3d) or otherwise, show that

$$f(\mathbf{x}) - f(\mathbf{x}^*) \le \frac{L}{2} \|\mathbf{x} - \mathbf{x}^*\|^2 + o\big(\|\mathbf{x} - \mathbf{x}^*\|^3\big).$$

Problem 4 (Probability Theory):

a) Use the definition of the expectation

$$\mathbb{E}[X] = \int_{-\infty}^{\infty} x f_X(x) dx$$

and variance $\mathrm{Var}(X) := \mathbb{E}\left[(X - \mathbb{E}[X])^2\right]$ to verify that the expectation and variance of a normal distributed random variable $X \sim \mathcal{N}(\mu, \sigma)$ with probability density function

$$f_X(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

is indeed $\mathbb{E}[X] = \mu$ and $\mathrm{Var}(X) = \sigma^2$.

b) Similarly, using

$$\mathbb{E}[\mathbf{X}] = \int_{-\infty}^{\infty} \mathbf{x} f_{\mathbf{X}}(\mathbf{x}) d\mathbf{x}$$

and $\mathrm{Cov}(\mathbf{X}) := \mathbb{E}\left[(\mathbf{X} - \mathbb{E}[\mathbf{X}])(\mathbf{X} - \mathbb{E}[\mathbf{X}])^\top\right]$, verify that for a multivariate normal distributed random variable $\mathbf{X} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$, with $\boldsymbol{\mu} \in \mathbb{R}^k$, $\boldsymbol{\Sigma} \in \mathbb{R}^{k\times k}$ and probability density function

$$f_{\mathbf{X}}(\mathbf{x}) = (2\pi)^{-\frac{k}{2}} \det(\boldsymbol{\Sigma})^{-\frac{1}{2}} \exp\left(-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1}(\mathbf{x} - \boldsymbol{\mu})\right)$$

that $\mathbb{E}[\mathbf{X}] = \mu$ and $\mathrm{Cov}(\mathbf{X}) = \boldsymbol{\Sigma}$.

c) Show the affine transformation rule for Gaussian random variables. That is, let $\mathbf{X}$ be normally distributed with $\mathbf{X} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ with $\boldsymbol{\mu} \in \mathbb{R}^k$, $\boldsymbol{\Sigma} \in \mathbb{R}^{k\times k}$ and define an affine transformed random variable $\mathbf{Y} := \mathbf{A}\mathbf{X} + \mathbf{b}$ with $\mathbf{A} \in \mathbb{R}^{p\times k}, \mathbf{b} \in \mathbb{R}^p$. Show that $\mathbf{Y}$ is normally distributed with $\mathbb{E}[\mathbf{Y}] = \mathbf{A}\boldsymbol{\mu} + \mathbf{b}$ and $\mathrm{Cov}(\mathbf{Y}) = \mathbf{A}\boldsymbol{\Sigma}\mathbf{A}^T$.

Chapter 2

Deep Learning: The Basics

2.1 A Brief Explanation of Deep Learning

We focus our explanation on the task of classification where the goal is to categorize data into predefined classes. Such a task involves training a model on a labeled dataset, where each data point is assigned a class label. The trained model then predicts the class labels of new, unseen data. The aim is to accurately assign the correct class label to new instances.

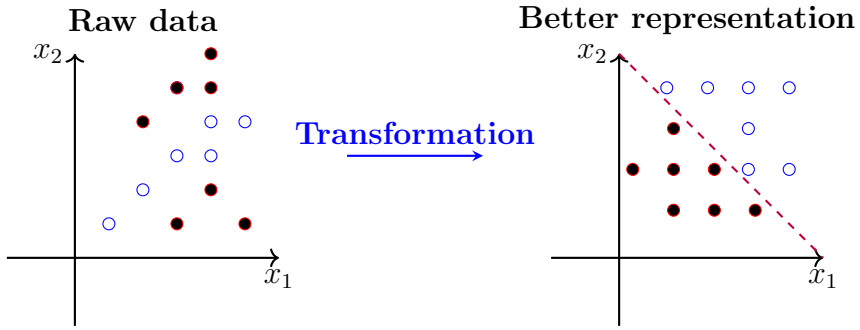


Figure 2.1: The left figure shows some raw data that is not linearly separable. A change of coordinate is applied to transform the data as shown in the right figure. The transformed data is linearly separable as shown by the dotted line that separates the two classes.

Understanding coordinate changes in classification Let's consider a classification setting with a dataset depicted in Figure 2.1 where some given data $\mathcal{X} \in \mathbb{R}^2$ is partitioned into two different classes. The goal is to find a way to effectively classify the data points based on their labels/classes.

Figure 2.1 clearly shows that the raw data is not linearly separable (two sets of points in a vector space are linearly separable if they can be separated by a single linear decision boundary). By applying a series of coordinate changes, we can transform the data into a representation where it becomes easier to distinguish between the different classes. This transformation is crucial in machine learning, especially in deep learning, where each layer of a neural network performs a coordinate change to better represent the data.

The role of neural networks Neural networks are powerful tools for obtaining better data representations through nested transformations. By passing the raw data through a series of layers, each applying a specific coordinate change, neural networks can transform the data into a more useful representation.

This nested representation illustrated in Figure 2.2 is key to the success of neural networks. Each layer in the network builds upon the previous one, refining the data representation step by step, making it easier to classify or make predictions. This process is particularly evident in tasks like digit classification, where the network progressively learns to recognize patterns and features in the input data.

Layer parameterization with weights As shown in Figure 2.3, each layer in a neural network is parameterized by a set of weights that define the transformation applied to the input of that layer. These weights are crucial as they determine how the input data is modified as it passes through the network.

The process of learning involves adjusting these weights so that the transformations applied by each layer lead to better data representations and, ultimately, more accurate classifications or predictions. The weights are learned by minimizing a loss function that measures the network's performance.

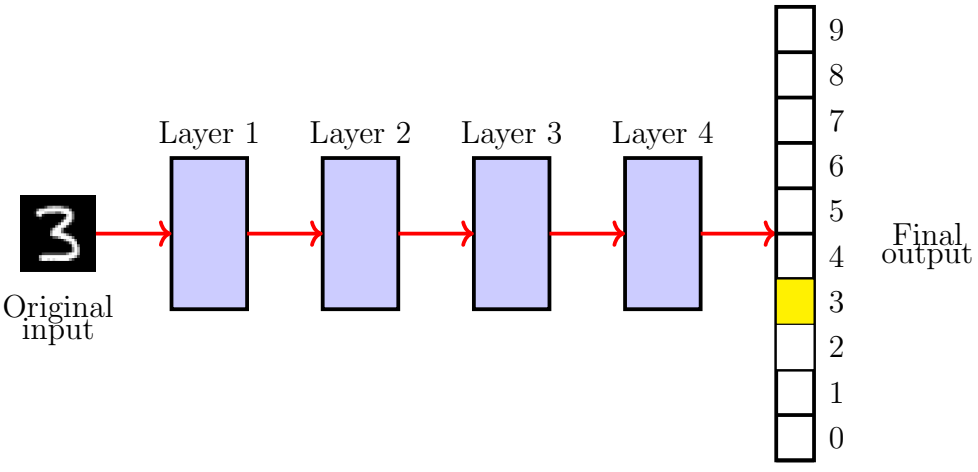


Figure 2.2: Neural network and nested representation.
Figure based on "Deep Learning with Python" by Francois Chollet.

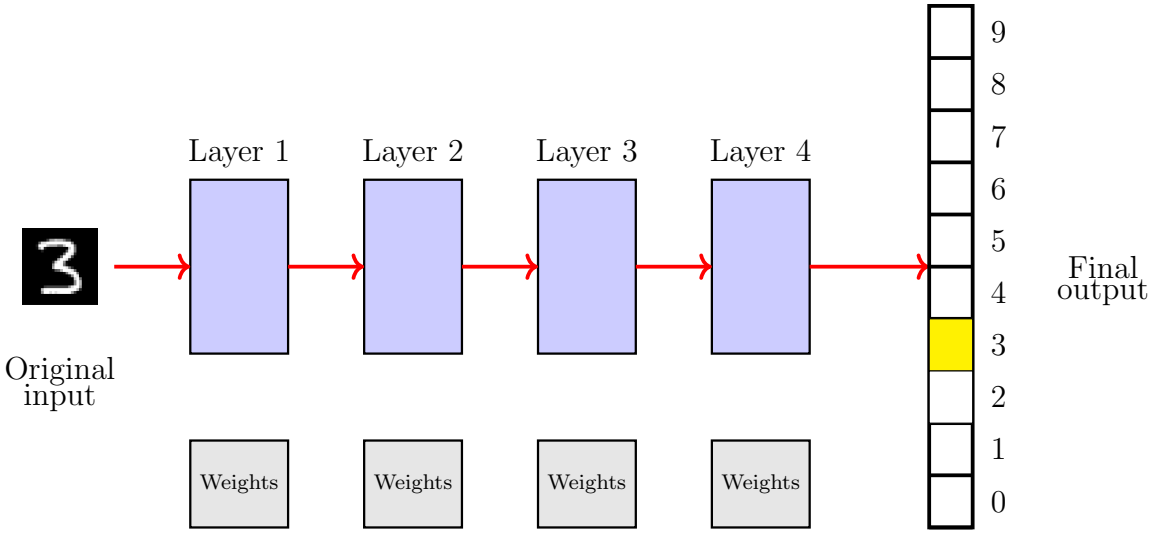


Figure 2.3: Layer weights in a neural network.
Figure based on "Deep Learning with Python" by Francois Chollet.

Learning through loss functions The effectiveness of a neural network in making accurate predictions is quantified by a loss function. This function measures the discrepancy between the predicted outputs y' and the actual labels y of the data.

The goal of training a neural network is to adjust the weights to minimize this loss function. A lower loss indicates a better fit of the network to the data, meaning that the network's predictions are closer to the actual labels.

Optimization with gradient descent Minimizing the loss function is achieved using an optimizer, such as gradient descent (this will be discussed in detail in a chapter about optimization). As shown in Figure 2.4, the optimizer iteratively adjusts the weights to reduce the loss, improving the network's performance. Gradient descent works by computing the gradient of the loss function with respect to the weights and updating the weights in the opposite direction of the gradient. This process continues until the loss function converges to a minimum, ideally resulting in a set of weights that allow the network to make accurate predictions.

2.2 Formalizing Deep Neural Networks

We start our journey by discussing a simple type of neural network known as a deep linear network.

2.2.1 Deep Linear Networks

Basic components: linear units The main idea of deep learning is to define a complex system from its elementary building blocks. In the case of (deep) neural

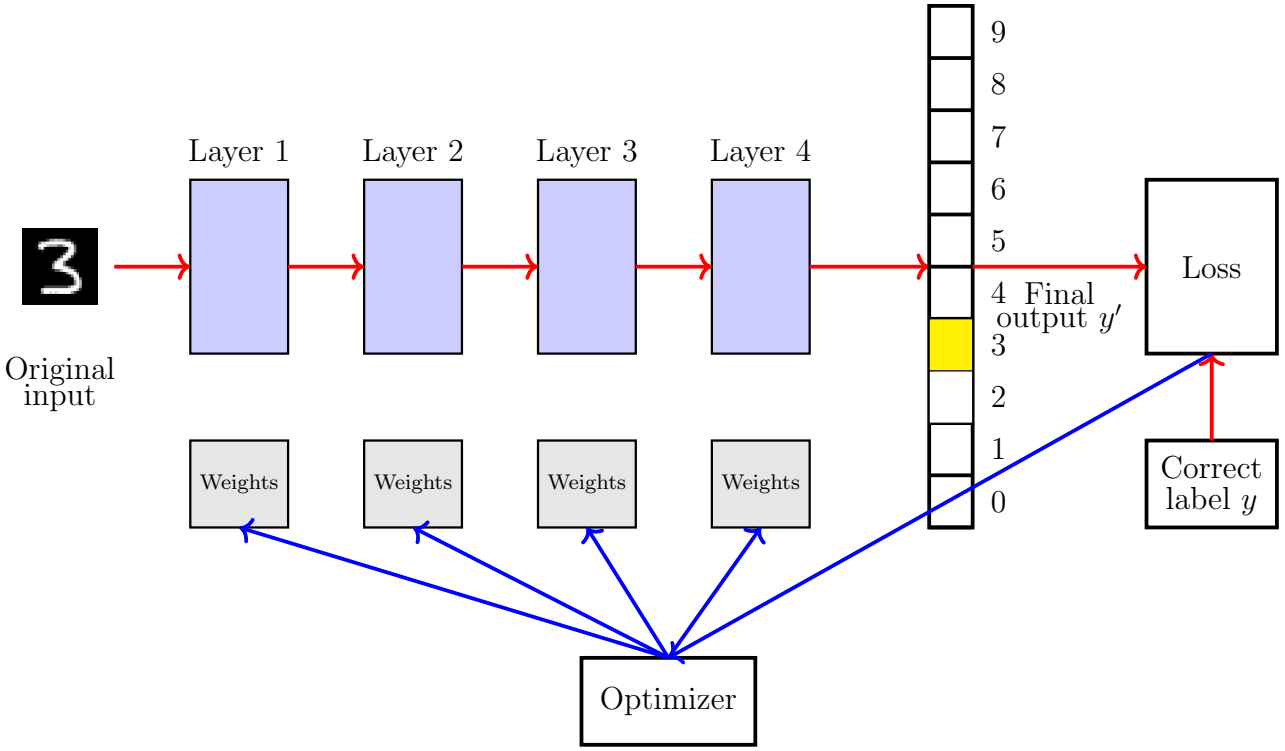


Figure 2.4: Optimizer in neural network training.
Figure based on "Deep Learning with Python" by Francois Chollet.

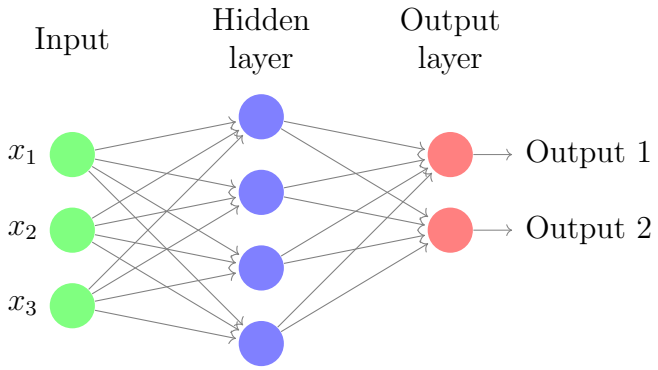


Figure 2.5: A depiction of a neural network showing that the layers are composed of individual units.

networks, these building blocks are **computational units** or neurons which themselves are arranged in layers as illustrated in Figure 2.5. Complicated functions can be obtained through **nested composition** of the layers and therefore of the units.

Mathematically, the units are simply real-valued functions of the form:

$$g : \mathbb{R}^n \times \mathbb{R}^d \to \mathbb{R}, \quad (\mathbf{x}, \boldsymbol{\theta}) \mapsto g(\mathbf{x}; \boldsymbol{\theta}) .$$

Linear functions A **linear unit** is defined as:

$$g(\mathbf{x};\boldsymbol{\theta}) = \mathbf{x}^\top\boldsymbol{\theta}\,, \quad \mathbf{x}^\top\boldsymbol{\theta} \equiv \sum_{i=1}^{n} x_i\theta_i$$

where the output is an additively weighted combination of inputs.

An **affine unit** [1] is obtained by adding an offset b as follows:

$$g(\mathbf{x};\boldsymbol{\theta},b) = \mathbf{x}^\top\boldsymbol{\theta} + b = \begin{pmatrix}\mathbf{x}\\1\end{pmatrix}\cdot\begin{pmatrix}\boldsymbol{\theta}\\b\end{pmatrix}\,.$$

Note that an offset $b \neq 0$ allows affine functions to be such that $f(0) \neq 0$.

The linear unit defines the **direction of change** via $\boldsymbol{\theta}/\|\boldsymbol{\theta}\|$ and the **rate of change** via $\|\boldsymbol{\theta}\|$. To create more expressive functions, multiple units can be arranged in a **linear layer**, resulting in:

$$L : \mathbb{R}^n \to \mathbb{R}^m, \quad L(\mathbf{x};\Theta) = \Theta\mathbf{x}, \Theta = \begin{pmatrix}\boldsymbol{\theta}_1^\top\\\dots\\\boldsymbol{\theta}_m^\top\end{pmatrix}, \quad \Theta \in \mathbb{R}^{m\times n}$$

Note that the layer weight matrix Θ is related to the parameters associated with single units:

$$g_j(\mathbf{x};\boldsymbol{\theta}_j) = \mathbf{x}\cdot\boldsymbol{\theta}_j, \quad \Theta = \begin{pmatrix}\boldsymbol{\theta}_1^\top\\\dots\\\boldsymbol{\theta}_m^\top\end{pmatrix}, \quad \Theta\mathbf{x} = \begin{pmatrix}\mathbf{x}\cdot\boldsymbol{\theta}_1\\\dots\\\mathbf{x}\cdot\boldsymbol{\theta}_m\end{pmatrix}.$$

Increasing the layer width m allows for the extraction of many features simultaneously, with each unit g_j tuning to a different direction $\boldsymbol{\theta}_j$. However, a constant rate of change can be too restrictive and we will soon see how to address this problem.

2.2.2 Linearity: A Deeper Look

Superposition principle A more foundational way to define linearity is through the **superposition principle**. A function $f : \mathbb{R}^n \to \mathbb{R}^m$ is linear if:

$$f(\alpha\mathbf{x}) = \alpha f(\mathbf{x}) \quad \forall\mathbf{x} \in \mathbb{R}^n, \alpha \in \mathbb{R} \qquad \text{(homogeneity)}$$

and

$$f(\mathbf{x}+\mathbf{y}) = f(\mathbf{x}) + f(\mathbf{y}) \quad \forall\mathbf{x},\mathbf{y} \in \mathbb{R}^n. \qquad \text{(additivity)}$$

A function f is linear if and only if:

$$f(\alpha\mathbf{x}+\beta\mathbf{y}) = \alpha f(\mathbf{x}) + \beta f(\mathbf{y}), \quad \forall\mathbf{x},\mathbf{y} \in \mathbb{R}^n, \quad \forall\alpha,\beta \in \mathbb{R}.$$

[1] Recall that a function $f : \mathbb{R}^n \to \mathbb{R}^m$ is affine if $f(\alpha\mathbf{x}+\beta\mathbf{y}) = \alpha f(\mathbf{x})+\beta f(\mathbf{y})+b, \quad \forall\mathbf{x},\mathbf{y};\ \forall\alpha,\beta \in \mathbb{R}$, for a given constant b.

Representation via matrices This abstract definition can be linked back to the parameterized view of linear maps via weight matrices Θ. In the Euclidean basis $\{\mathbf{e}_1, \ldots, \mathbf{e}_n\}$, a vector $\mathbf{x}$ can be written as $\mathbf{x} = \sum_i x_i \mathbf{e_i}$. This leads to the result that f is linear if and only if $f(\mathbf{x}) = \mathbf{A}\mathbf{x}$ for some unique $\mathbf{A} \in \mathbb{R}^{n \times m}$.

2.2.3 Compositionality

Deep compositional networks rely on the ability to increase expressivity with compositional depth. Composing linear functions does not lead to any such gains:

Theorem 11. *Let f and g be matching linear functions. Then $g \circ f$ is linear.*

Proof.

$$(g \circ f)(\alpha \mathbf{x}) = g(f(\alpha \mathbf{x})) = g(\alpha f(\mathbf{x})) = \alpha g(f(\mathbf{x})) = \alpha (g \circ f)(\mathbf{x})$$
$$(g \circ f)(\mathbf{x} + \mathbf{y}) = g(f(\mathbf{x}) + f(\mathbf{y})) = g(f(\mathbf{x})) + g(f(\mathbf{y})) = (g \circ f)(\mathbf{x}) + (g \circ f)(\mathbf{y})$$

□

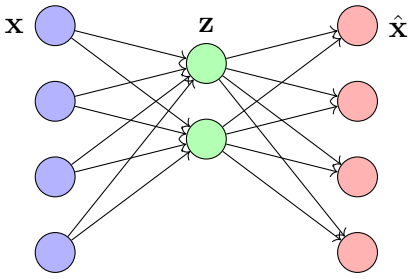
Let $f(\mathbf{x}) = \mathbf{A}\mathbf{x}$, $g(\mathbf{z}) = \mathbf{B}\mathbf{z}$, then $(g \circ f)(\mathbf{x}) = \mathbf{C}\mathbf{x}$, where $\mathbf{C} = \mathbf{B}\mathbf{A}$. Composing linear functions amounts to multiplying matrices, the result of which will be another matrix.

2.3 Linear Autoencoder

An autoencoder is a type of artificial neural network used to learn efficient encodings of unlabeled data. The goal of an autoencoder is to compress the input data into a lower-dimensional representation and then reconstruct the input data from this representation. This is achieved through a bottleneck layer, which forces the network to learn the most important features of the input data.
Mathematically, the process can be described as follows:

$$\mathbf{x} \mapsto \mathbf{z} \mapsto \mathbf{y}, \quad \mathbf{z} = \mathbf{C}\mathbf{x},\ \mathbf{y} = \mathbf{D}\mathbf{z}, \quad \mathbf{C}, \mathbf{D}^\top \in \mathbb{R}^{m \times d},$$

where $\mathbf{x} \in \mathbb{R}^d$ is the input data, $\mathbf{z} \in \mathbb{R}^m$ is the compressed representation (where $m < d$), $\mathbf{y} \in \mathbb{R}^d$ is the reconstructed data, $\mathbf{C}$ is the compression matrix, $\mathbf{D}$ is the decompression matrix.
The objective is to minimize the difference between the input data $\mathbf{x}$ and the reconstructed data $\mathbf{y}$. This is quantified by the loss function which for a single instance $\mathbf{x}$ is defined as



Goal: minimize $\|\mathbf{x} - \hat{\mathbf{x}}\|^2$

$$\ell_{\mathbf{x}}(\theta) = \frac{1}{2}\|\mathbf{x} - \mathbf{DCx}\|^2 \qquad \theta = (\mathbf{C}, \mathbf{D})$$

Optimal weights: Let's now focus on the following question: wow should you choose $\mathbf{C}$ and $\mathbf{D}$ to be "optimal"?

To find the optimal $\mathbf{C}$ and $\mathbf{D}$ matrices, we need to minimize the loss over a given training set. Specifically, let's denote by $(\mathbf{x}_i)_{i=1}^n$ the set of n training points $\mathbf{x}_i$. The overall loss function over the entire training dataset is defined as

$$L(\boldsymbol{\theta}) = \frac{1}{2n}\sum_{i=1}^{n} \ell_{\mathbf{x}_i}(\theta) = \frac{1}{2n}\sum_{i=1}^{n} \|\mathbf{x}_i - \mathbf{DCx}_i\|^2, \tag{Loss}$$

where n is the number of samples in the dataset.

In matrix notation, we can express the dataset as:

$$\mathbf{X} = [\mathbf{x}_1 \dots \mathbf{x}_n] = \begin{pmatrix} | & & | \\ \mathbf{x}_1 & \cdots & \mathbf{x}_n \\ | & & | \end{pmatrix} \in \mathbb{R}^{d \times n},\ \mathbf{Y} = [\mathbf{y}_1 \dots \mathbf{y}_n] = \mathbf{DCX} \in \mathbb{R}^{d \times n},$$

Thus, the loss function can be rewritten using the Frobenius norm as:

$$L(\boldsymbol{\theta}) = \frac{1}{2}\|\mathbf{X} - \mathbf{Y}\|_F^2 \tag{Frobenius}$$

where $\|\cdot\|_F$ denotes the Frobenius norm, which measures the difference between the original data $\mathbf{X}$ and the reconstructed data $\mathbf{Y}$.

By optimizing this loss function, we can find the best matrices $\mathbf{C}$ and $\mathbf{D}$ that minimize the reconstruction error, thus achieving the most efficient compression and reconstruction of the input data. The goal is essentially to learn the identity map, $\mathbf{DC} \approx \mathbf{I}$, but with respect to a specific pattern distribution, represented by a sample, and to find an efficient lossy compression scheme.

The next logical question is how to minimize the loss function $L(\theta)$. Generally, there are two approaches to consider:

1. **Closed-form solution:**
    - When the function has a straightforward structure, a closed-form solution can be derived, which is an explicit expression in terms of the problem variables. This solution can often be obtained by setting the gradients to zero, leading to a first-order optimal solution. While this approach only guarantees to find a local optimum, additional problem structures like convexity can ensure that this solution is also global. We will explore this further in later sections.
2. **No closed-form solution:**
    - In cases where a closed-form solution is not available, especially common in deep learning, the typical approach is to use gradient-based optimizers. These methods iteratively step in the negative direction of the gradient. We will discuss gradient descent and related techniques in detail in a dedicated chapter.

Gradients As mentioned above, calculating gradients is essential to find the optimal weight matrices. Let's go through the calculation in more detail. First, we calculate the (per datapoint) gradients of the loss $\ell_{\mathbf{x}}$ with respect to single entries of the matrices $\mathbf{C}$ and $\mathbf{D}$:

$$\frac{\partial \ell_{\mathbf{x}}(\theta)}{\partial C_{ki}} = \sum_{j=1}^{d} (y_j - x_j) \frac{\partial y_j}{\partial C_{ki}} = x_i \sum_{j=1}^{d} D_{jk}(y_j - x_j),$$

$$\frac{\partial \ell_{\mathbf{x}}(\theta)}{\partial D_{jk}} = \sum_{i=1}^{d} (y_i - x_i) \frac{\partial y_i}{\partial D_{jk}} = (y_j - x_j) \sum_{i=1}^{d} C_{ki} x_i.$$

Once we have calculated the derivative with respect to a single entry, we can write the derivative with respect to the entire matrix as $\partial_{\mathbf{A}} f := \left(\frac{\partial f}{\partial A_{ij}}\right)_{ij}$. Therefore, we obtain the following derivatives:

$$\frac{\partial \ell_{\mathbf{x}}(\theta)}{\partial \mathbf{C}} = \mathbf{D}^\top (\mathbf{y} - \mathbf{x}) \mathbf{x}^\top \in \mathbb{R}^{m \times d},$$

$$\frac{\partial \ell_{\mathbf{x}}(\theta)}{\partial \mathbf{D}} = (\mathbf{y} - \mathbf{x})\mathbf{x}^\top \mathbf{C}^\top \in \mathbb{R}^{d \times m}.$$

Rank constraint Because the autoencoder structure is a product of two matrices, the final solution is constrained as follows:

$$\operatorname{rank}(\mathbf{DC}) \leq \min\{\operatorname{rank}(\mathbf{C}), \operatorname{rank}(\mathbf{D})\} \leq m < d,$$

which implies:

$$\mathbf{Y} = \mathbf{DCX}, \quad \operatorname{rank}(\mathbf{Y}) \leq m.$$

This constraint is a fundamental property of the matrix product, limiting the rank of the resulting matrix $\mathbf{Y}$ to be at most m, which is the smaller dimension of the matrices involved in the factorization. This is crucial in applications such as dimensionality reduction, where the objective is to represent high-dimensional data in a lower-dimensional space without losing significant information.

Note that the Singular Value Decomposition (SVD) of a matrix $\mathbf{X} \in \mathbb{R}^{d \times n}$ is:

$$\mathbf{X} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top, \quad \mathbf{U} \in \mathbb{R}^{d \times d}, \mathbf{V} \in \mathbb{R}^{n \times n}, \text{ orthogonal},$$

$$\mathbf{\Sigma} = \operatorname{diag}(\sigma_1, \ldots, \sigma_{\min\{d,n\}}).$$

One can similarly define the reduced or truncated SVD for $r \leq \min(d, n)$ as:

$$\mathbf{X}_r = \mathbf{U}_r \mathbf{\Sigma}_r \mathbf{V}_r^\top, \ \mathbf{\Sigma} = \operatorname{diag}(\sigma_1, \ldots, \sigma_r) \in \mathbb{R}^{r \times r},$$

$$\mathbf{U}_r = [\mathbf{U}_1 \ldots \mathbf{U}_r], \ \mathbf{V}_r = [\mathbf{v}_1 \ldots \mathbf{v}_r].$$

In SVD, $\mathbf{U}$ and $\mathbf{V}$ are orthogonal matrices, and $\mathbf{\Sigma}$ is a diagonal matrix containing the singular values σ_i, which represent the magnitude of each corresponding dimension. The decomposition separates the matrix into its constituent components, revealing the intrinsic structure of the data.

Theorem 12 (Eckart-Young Theorem)**.** *The rank-r matrix $\mathbf{X}_r$, obtained from the truncated singular value decomposition, is such that:*

$$\|\mathbf{X} - \mathbf{X}_r\|_F = \min_{\operatorname{rank}(\mathbf{Y}) \leq r} \|\mathbf{X} - \mathbf{Y}\|_F.$$

The Eckart-Young Theorem states that the truncated SVD provides the best low-rank approximation of a matrix in terms of the Frobenius norm. This means that $\mathbf{X}_r$ is the closest rank-r approximation to $\mathbf{X}$, minimizing the reconstruction error. This theorem is foundational in various applications, including principal component analysis (PCA), where it ensures that the dimensionality reduction process retains as much information as possible.

Optimal linear autoencoder As mentioned earlier, one can in some cases derive closed-form expressions for some problems, which turn out to be the case for the linear auto-encoder loss (more on the required condition for this to hold will be discussed later on). In our setting, it turns out that we can simply use the Eckart-Young theorem to derive the optimal closed-form expressions for the matrices $\mathbf{C}$ and $\mathbf{D}$.

Theorem 13. *The optimal* $\mathbf{C}$ *and* $\mathbf{D}$ *matrices to minimize the autoencoder loss are:*

$$\mathbf{C} = \mathbf{U}_m^\top, \ \mathbf{D} = \mathbf{U}_m,$$
$$\mathbf{DCX} = \mathbf{X}_m.$$

Proof.

$$\begin{aligned}\mathbf{DCX} &= \mathbf{DCU\Sigma V}^\top \\ &= \mathbf{U}_m(\mathbf{U}_m^\top \mathbf{U})\mathbf{\Sigma V}^\top \\ &= \mathbf{U}_m \mathbf{I}_m \mathbf{\Sigma V}^\top \\ &= \mathbf{U}_m \mathbf{\Sigma}_m \mathbf{V}_m^\top \\ &= \mathbf{X}_m.\end{aligned}$$

□

2.4 Non-linear Activations

2.4.1 Ridge Functions

Ridge functions generalize linear functions to allow for powerful compositionality. The simple idea is to compose a linear (or affine) function with a non-linear (continuous) function ϕ, also called an **activation function**:

$$f = \phi \circ g, \quad f(\mathbf{x}; \boldsymbol{\theta}) = \phi(\mathbf{x}^\top \boldsymbol{\theta}).$$

The term "ridge" comes from the fact that these functions are constant on hyperplanes (ridges) orthogonal to the vector θ:

$$\phi(\mathbf{x}^\top \boldsymbol{\theta}) = \phi((\mathbf{x} + \Delta\mathbf{x}) \cdot \boldsymbol{\theta}) \quad \text{for} \quad \Delta\mathbf{x} \perp \boldsymbol{\theta}.$$

Ridge functions have a variable rate of change, governed by ϕ or rather ϕ' (in case ϕ is differentiable):

$$\|\nabla_{\mathbf{x}} f(\mathbf{x}; \boldsymbol{\theta})\| = |\phi'(\mathbf{x}^\top \boldsymbol{\theta})| \ \|\boldsymbol{\theta}\|.$$

Layers Ridge units can also easily be arranged in layers. To do so, we extend ϕ to be a vector as follows:

$$F(\mathbf{x}; \Theta) = \phi(\Theta \mathbf{x}), \quad \phi(\mathbf{z}) = \begin{pmatrix} \phi(z_1) \\ \dots \\ \phi(z_m) \end{pmatrix}.$$

Parameter gradients Parameter update directions for ridge functions are always in the direction of the input vector. For any loss function ℓ, the chain rule yields:

$$\nabla_{\boldsymbol{\theta}} (\ell \circ \phi)(\mathbf{x}^\top \boldsymbol{\theta}) = (\ell \circ \phi)'(\mathbf{x}^\top \boldsymbol{\theta})\, \mathbf{x}.$$

This property simplifies the gradient computation in optimization algorithms, making ridge functions efficient for training in machine learning models.

2.4.2 Threshold Units

A **linear threshold unit** is a function that uses the Heavyside or sign function (see Figure 2.6) as activation and produces Boolean outputs:

$$f^H(\mathbf{x}; \boldsymbol{\theta}) = H(\mathbf{x}^\top \boldsymbol{\theta}),\ H(z) = \begin{cases} 1, & \text{if } z \geq 0 \\ 0, & \text{else} \end{cases}$$

Equivalently, one can also map to $\{-1, 1\}$:

$$f^{\pm}(\mathbf{x}; \boldsymbol{\theta}) = \text{sign}(\mathbf{x}^\top \boldsymbol{\theta}).$$

Threshold units are a cartoon version of a biological neuron. If activation exceeds a threshold, the unit 'fires'.

A major drawback is that H has a jump at 0, meaning there is no derivative information to guide the adaptation of the parameters, ruling out gradient-based methods.

2.4.3 Sigmoid Units

The sigmoid (or logistic) function is often preferred over the sign function in neural networks due to its differentiability property. The goal is to retain some characteristics of the threshold unit, in particular the limits $f(\mathbf{x}, \boldsymbol{\theta}) \to (0, 1)$ as $\mathbf{x}^\top \boldsymbol{\theta} \to (-\infty, \infty)$, but to define a smooth, monotone function. One such choice is the **sigmoid/logistic unit**:

$$\sigma\left(\mathbf{x}^\top \boldsymbol{\theta}\right) = \frac{1}{1 + \exp[-\mathbf{x}^\top \boldsymbol{\theta}]}.$$

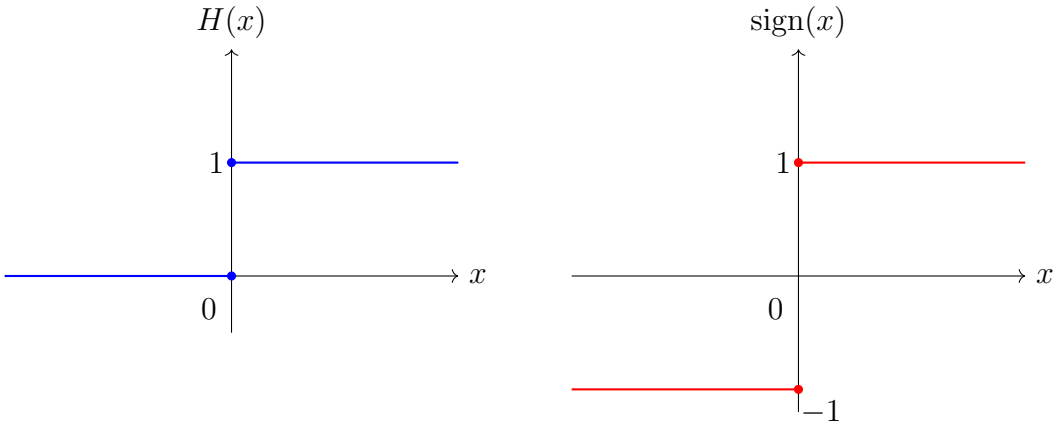


Figure 2.6: The Heaviside step function $H(x)$ on the left (in blue) and of the sign function on the right (in red).

Sigmoid unit: derivatives The sigmoid function has a derivative:

$$\sigma'(z) = \sigma(z)(1 - \sigma(z)) = \sigma(z)\sigma(-z).$$

Proof: With the quotient rule,

$$\begin{aligned}\sigma'(z) &= e^{-z}(1 + e^{-z})^{-2} = \frac{1}{1 + e^{-z}} \frac{e^{-z}}{1 + e^{-z}} \\ &= \sigma(z)(1 - \sigma(z)).\end{aligned}$$

This implies that all higher-order derivatives are polynomials in σ and hence σ is smooth:

$$\sigma \in C^\infty.$$

Hyperbolic tangent Another popular choice of activation function is the hyperbolic tangent, which is a simple transformation of the logistic function:

$$\tanh(z) := \frac{e^z - e^{-z}}{e^z + e^{-z}} = 2\sigma(2z) - 1.$$

The identity follows from:

$$2\sigma(2z) - 1 = \frac{2}{1 + e^{-2z}} - \frac{1 + e^{-2z}}{1 + e^{-2z}} = \frac{1 - e^{-2z}}{1 + e^{-2z}} = \frac{e^z - e^{-z}}{e^z + e^{-z}}.$$

As the range of the tanh function $R = (-1; 1)$ is symmetric around zero, it is sometimes preferred.

Inverse of the logistic function The inverse of the logistic function is derived for later use:

$$\sigma^{-1}(t) = \ln \frac{t}{1-t}.$$

The validity of this formula can be checked via:

$$\sigma\left(\ln \frac{t}{1-t}\right) = \frac{1}{1+\frac{1-t}{t}} = t.$$

2.4.4 Softmax

The softmax function is a function that converts a vector of values (often called logits) into a probability distribution. It is widely used in machine learning, especially in the context of classification problems where the goal is to assign probabilities to different classes. Specifically, it defines class conditional probabilities via:

$$\sigma_i^{\max}(\mathbf{x}; \Theta) = \frac{\exp[\mathbf{x}^\top \boldsymbol{\theta}_i]}{\sum_{j=1}^k \exp[\mathbf{x}^\top \boldsymbol{\theta}_j]}, \quad \Theta = [\boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_k].$$

The softmax function has two important properties that make it ideal to encode probability distributions:

- It is non-negative,
- It is normalized: $\sum_{i=1}^k \sigma_i^{\max}(\mathbf{x}; \Theta) = 1$.

Also note that one can subtract any (constant) vector $\boldsymbol{\theta}_i' = \boldsymbol{\theta}_i + \Delta\boldsymbol{\theta}$ for all i and obtain the same output (left as an exercise to the reader). This can be avoided by setting $\boldsymbol{\theta}_k = \mathbf{0}$ (and not updating it).

Finally, we note that the sigmoid unit can be recovered for $k = 2$:

$$\text{softmax with } \boldsymbol{\theta}_1, \boldsymbol{\theta}_2 \iff \text{sigmoid with } \boldsymbol{\theta} = \boldsymbol{\theta}_1 - \boldsymbol{\theta}_2.$$

$$\begin{aligned}
\sigma_1^{\max}(\mathbf{x}) &= \frac{e^{\mathbf{x}^\top \boldsymbol{\theta}_1}}{e^{\mathbf{x}^\top \boldsymbol{\theta}_1} + e^{\mathbf{x}^\top \boldsymbol{\theta}_2}} \\
&= \frac{e^{\mathbf{x}\cdot(\boldsymbol{\theta}_1 - \boldsymbol{\theta}_1)}}{e^{\mathbf{x}\cdot(\boldsymbol{\theta}_1 - \boldsymbol{\theta}_1)} + e^{\mathbf{x}\cdot(\boldsymbol{\theta}_2 - \boldsymbol{\theta}_1)}} \\
&= \frac{1}{1 + e^{-\mathbf{x}\cdot(\boldsymbol{\theta}_1 - \boldsymbol{\theta}_2)}} = \sigma(\mathbf{x}).
\end{aligned}$$

Softmax + cross-entropy The softmax function is typically coupled with the cross-entropy loss where one encodes class information in a one-hot vector, $\mathbf{y} \in \{\mathbf{e}_1, \ldots, \mathbf{e}_k\}$:

$$\begin{aligned}\ell(\mathbf{x}, \mathbf{y}; \Theta) &= -\mathbf{y} \cdot \ln \sigma^{\max}(\mathbf{x}; \Theta) \\ &= -\sum_{i=1}^{k} y_i \cdot \ln \sigma^{\max}(\mathbf{x}; \Theta)_i.\end{aligned}$$

Alternatively, separating out the normalization:

$$\ell(\mathbf{x}, \mathbf{y}; \Theta) = -\sum_{i=1}^{k} y_i \, \mathbf{x}^\top \boldsymbol{\theta}_i + \ln \left(\sum_{i=1}^{k} \exp(\mathbf{x}^\top \boldsymbol{\theta}_i) \right),$$

leads to the gradient formula:

$$\nabla_{\boldsymbol{\theta}_i} \ell(\mathbf{x}, \mathbf{y}; \Theta) = (\sigma_i^{\max} - y_i)\mathbf{x}.$$

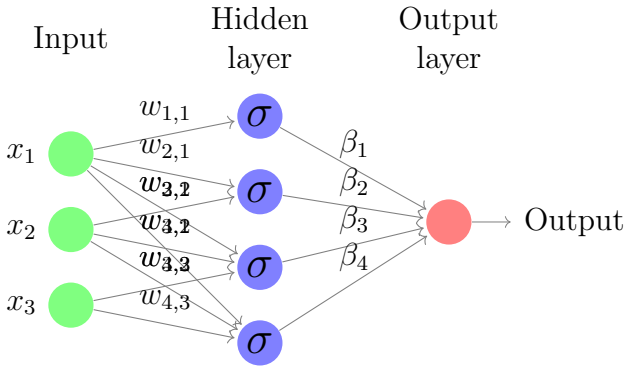
2.5 Multilayer Perceptron

A multilayer perceptron (MLP) is the name for a classical feedforward artificial neural network, consisting of fully connected neurons with a nonlinear sigmoid activation function.

2.5.1 MLP: Architecture

A three-layer reference architecture consists of:

- n inputs.
- One hidden layer of m sigmoid units.
- One real-valued output (could be more).



The function computed by the MLP is:

$$f(\mathbf{x}; \boldsymbol{\beta}, \mathbf{w}) = \sum_{j=1}^{m} \beta_j \, \sigma(\mathbf{w}_j \mathbf{x}), \qquad \sigma(z) = \frac{1}{1 + e^{-z}}. \tag{2.1}$$

Note that in Eq. (2.1) each $\mathbf{w}_j$ is a vector of weights associated with the j-th sigmoid unit in the hidden layer, while each β_j is a scalar weight applied to the output of the j-th sigmoid unit.

Each unit computes a weighted combination of inputs using the weights $\mathbf{w}$ and applies a logistic activation function. These activations (in the range $(0, 1)$) are then summed up with weights $\boldsymbol{\beta}$.

2.5.2 MLP: Learning

In order to learn the optimal set of weights $\mathbf{w}$ and β, we need to define a loss function, which we typically take to be the squared loss:

$$\mathcal{L}(\boldsymbol{\beta}, \mathbf{w}; \mathbf{x}, y) = \frac{1}{2}\big(f(\mathbf{x}) - y\big)^2,$$

where we use the shortcut notation $f(\mathbf{x}) = f(\mathbf{x}; \boldsymbol{\beta}, \mathbf{w})$.

By the chain rule,

$$\frac{\partial \mathcal{L}}{\partial \theta} = \big(f(\mathbf{x}) - y\big)\,\frac{\partial f(\mathbf{x})}{\partial \theta} \quad \text{for any parameter } \theta.$$

Gradient w.r.t. β_j.

$$\frac{\partial f(\mathbf{x})}{\partial \beta_j} = \sigma(\mathbf{w}_j \cdot \mathbf{x}) \quad \Longrightarrow \quad \boxed{\frac{\partial \mathcal{L}}{\partial \beta_j} = \big(f(\mathbf{x}) - y\big)\,\sigma(\mathbf{w}_j \cdot \mathbf{x}) = \frac{f(\mathbf{x}) - y}{1 + e^{-\mathbf{w}_j \cdot \mathbf{x}}}.}$$

Gradient w.r.t. w_{ji}. First,

$$\frac{\partial f(\mathbf{x})}{\partial w_{ji}} = \beta_j\,\sigma'(\mathbf{w}_j \cdot \mathbf{x})\,\frac{\partial (\mathbf{w}_j \cdot \mathbf{x})}{\partial w_{ji}} = \beta_j\,\sigma'(\mathbf{w}_j \cdot \mathbf{x})\,x_i.$$

Using $\sigma'(z) = \sigma(z)\big(1 - \sigma(z)\big)$ and

$$1 - \sigma(z) = \frac{e^{-z}}{1 + e^{-z}} = \frac{1}{1 + e^{z}}, \quad \text{we get} \quad \sigma'(z) = \frac{1}{1 + e^{-z}} \cdot \frac{1}{1 + e^{z}}.$$

Therefore,

$$\frac{\partial f(\mathbf{x})}{\partial w_{ji}} = \beta_j x_i\,\frac{1}{1 + e^{-\mathbf{w}_j \cdot \mathbf{x}}}\,\frac{1}{1 + e^{\mathbf{w}_j \cdot \mathbf{x}}}.$$

Applying the outer chain rule,

$$\boxed{\frac{\partial \mathcal{L}}{\partial w_{ji}} = \big(f(\mathbf{x}) - y\big)\,\frac{1}{1 + e^{-\mathbf{w}_j \cdot \mathbf{x}}} \cdot \frac{\beta_j x_i}{1 + e^{\mathbf{w}_j \cdot \mathbf{x}}}.}$$

Gradient Descent Learning the weights β and w using a gradient-based optimizer involves the following steps:

- Forward pass: computes hidden activities and output.
- Re-combination with a few more operations to get derivatives.
- Updates via stochastic gradient descent (SGD):

  $$\theta \leftarrow \theta - \eta \frac{\mathcal{L}(\boldsymbol{\beta}, \mathbf{w}; \mathbf{x}, y)}{\partial \theta}, \quad \theta \in \{\beta_j, w_{ji}\},$$

  with some step size $\eta > 0$.

2.6 Backpropagation

Backpropagation is a cornerstone of supervised learning algorithms used for training neural networks. The technique involves calculating the gradient of the loss function with respect to each weight by utilizing the chain rule and iterating backward from the output to the input layer. The gradient calculated by backpropagation can then be used to minimize the loss function by adjusting the weights of the neural networks accordingly.

To illustrate the process, consider a simple neural network structure illustrated in Figure 2.7. This network consists of an input layer, one hidden layer, and an output layer. The nodes in each layer represent the neurons, and the edges between them represent the weights.

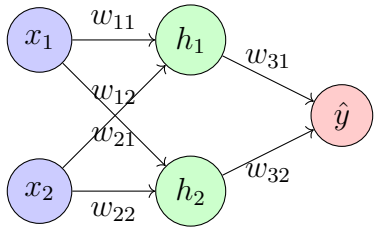


Figure 2.7: A simple neural network with one input layer (consisting of two nodes), one hidden layer (also consisting of two nodes), and one output layer.

The learning process begins with a forward pass through the network. During this pass, we calculate the input to the hidden layer neurons as follows:

$$\text{net}_{h_1} = w_{11}x_1 + w_{21}x_2$$

$$\text{net}_{h_2} = w_{12}x_1 + w_{22}x_2$$

Next, we apply an activation function σ to determine the output of the hidden layer neurons:

$$h_1 = \sigma(\text{net}_{h_1})$$
$$h_2 = \sigma(\text{net}_{h_2})$$

These outputs are then used to compute the input to the output neuron:

$$\text{net}_{o1} = w_{31}h_1 + w_{32}h_2$$

Finally, we apply the activation function again to obtain the predicted output:

$$\hat{y} = \sigma(\text{net}_{o1})$$

After the forward pass, we calculate the loss (error) using a loss function such as Mean Squared Error (MSE):

$$L(y, \hat{y}) = \frac{1}{2}(y - \hat{y})^2,$$

where y represents the true label, and $\hat{y}$ is the predicted output.

To reduce this loss, we perform a backward pass. We start by computing the gradient of the loss with respect to the output:

$$\frac{\partial L}{\partial \hat{y}} = \hat{y} - y.$$

Then, we compute the gradient of the loss with respect to the weights connecting the hidden layer to the output layer.

$$\frac{\partial L}{\partial w_{31}} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial \text{net}_{o1}} \cdot \frac{\partial \text{net}_{o1}}{\partial w_{31}} = (\hat{y} - y) \cdot \sigma'(\text{net}_{o1}) \cdot h_1$$

$$\frac{\partial L}{\partial w_{32}} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial \text{net}_{o1}} \cdot \frac{\partial \text{net}_{o1}}{\partial w_{32}} = (\hat{y} - y) \cdot \sigma'(\text{net}_{o1}) \cdot h_2$$

The next step is to compute the gradient of the loss with respect to the hidden layer neurons:

$$\frac{\partial L}{\partial h_1} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial \text{net}_{o1}} \cdot \frac{\partial \text{net}_{o1}}{\partial h_1} = (\hat{y} - y) \cdot \sigma'(\text{net}_{o1}) \cdot w_{31}$$

$$\frac{\partial L}{\partial h_2} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial \text{net}_{o1}} \cdot \frac{\partial \text{net}_{o1}}{\partial h_2} = (\hat{y} - y) \cdot \sigma'(\text{net}_{o1}) \cdot w_{32}$$

We then compute the gradient of the loss with respect to the weights connecting the input layer to the hidden layer. This also involves the derivative of the activation function σ applied to net_{h_1} and net_{h_2}:

$$\frac{\partial L}{\partial w_{11}} = \frac{\partial L}{\partial h_1} \cdot \frac{\partial h_1}{\partial \text{net}_{h_1}} \cdot \frac{\partial \text{net}_{h_1}}{\partial w_{11}} = (\hat{y} - y) \cdot \sigma'(\text{net}_{o1}) \cdot w_{31} \cdot \sigma'(\text{net}_{h_1}) \cdot x_1$$

$$\frac{\partial L}{\partial w_{21}} = \frac{\partial L}{\partial h_1} \cdot \frac{\partial h_1}{\partial \text{net}_{h_1}} \cdot \frac{\partial \text{net}_{h_1}}{\partial w_{21}} = (\hat{y} - y) \cdot \sigma'(\text{net}_{o1}) \cdot w_{31} \cdot \sigma'(\text{net}_{h_1}) \cdot x_2.$$

The backward pass involves propagating these gradients through the network to adjust the weights. The illustration below highlights the backpropagation process, with the red gradients representing the backpropagated gradients.

Finally, we update the weights using gradient descent:

$$w_{ij} := w_{ij} - \eta \cdot \frac{\partial L}{\partial w_{ij}}$$

where η is the learning rate. This process is repeated through multiple iterations (epochs) until the loss converges, thereby refining the network's predictions and improving its performance.

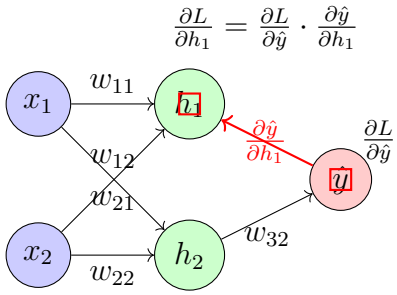


Figure 2.8: Illustration of the backward pass in the neural network. The red gradients are backpropagated gradients.

2.7 MLP Python Implementation from Scratch

In the following, we will see how to implement a Multi-Layer Perceptron (MLP) entirely from scratch in Python, i.e. without relying on any external libraries. We will apply our MLP to a binary classification problem using a toy dataset, where data points for each class are sampled from two distinct Gaussian distributions, and we will train the model by optimizing the cross-entropy loss function using gradient descent (this algorithm will be discussed in more detail later).

Cross-Entropy (CE) Loss for Multiple Classes

For discrete classes, the Cross-Entropy (CE) loss of a single training example is defined as:

$$L(z) = -\sum_{i=1}^{C} y_i \log \sigma(z_i),$$

where the softmax function σ is given by:

$$\sigma(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}.$$

Binary Cross-Entropy (BCE) Loss

When $C = 2$, we typically write $y_2 = 1 - y_1$. The binary cross-entropy loss is:

$$L(\hat{y}) = -\Big[\, y \log(\hat{y}) + (1 - y) \log\,(1 - \hat{y})\Big],$$

where

$$\hat{y} = \sigma(z) = \frac{1}{1 + e^{-z}}.$$

Derivative of the Loss

First note that by the chain rule:

$$\frac{dL}{dz} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{d\hat{y}}{dz}.$$

The sigmoid function derivative is

$$\frac{d\hat{y}}{dz} = \hat{y}(1-\hat{y}),$$

while the derivative of the loss L with respect to $\hat{y}$ is:

$$\frac{\partial L}{\partial \hat{y}} = -\left[\frac{y}{\hat{y}} - \frac{1-y}{1-\hat{y}}\right].$$

Therefore, the derivative of the loss is:

$$\frac{dL}{dz} = \hat{y} - y.$$

2.7.1 BCE and ReLU Classes

```python
#!/usr/bin/env python
# coding: utf-8

import numpy as np

class BCE:
    def __init__(self):
        pass

    def sigmoid(self, z):
        return 1.0 / (1.0 + np.exp(-z))

    # Compute the output of the function (forward pass)
    def forward(self, z, y):
        """
        z shape: (N, 1) where N is the number of datapoints
        y shape: (N, 1)
        Returns:
            loss (scalar), prediction y_hat (N, 1)
        """
        y_hat = self.sigmoid(z)
        # Clip to avoid log(0)
        y_hat = np.clip(y_hat, 1e-12, 1 - 1e-12)

        # Compute Binary Cross-Entropy loss
        N = len(y)
        loss = - (1.0 / N) * np.sum(y * np.log(y_hat) + (1 - y) * np
    .log(1 - y_hat))
        return loss, y_hat

    # Compute the gradient dL/dZ (backward pass)
    def backward(self, y_hat, y):
        """
```

```
        dL/dZ = (y_hat - y)
        Returns:
            dZ shape: (N, 1)
        """
        N = len(y)
        return (y_hat - y) / N


class ReLU:
    def __init__(self):
        self.requires_grad = False  # ReLU activation does not have any parameter

    def forward(self, Z):
        """
        Z shape: (N, d_out) or any shape
        """
        self.Z = Z  # save for backward
        return np.maximum(0, Z)

    def backward(self, dA):
        """
        dA shape: same as Z
        dZ = dA * (Z > 0)
        """
        dZ = dA * (self.Z > 0).astype(float)
        return dZ
```

2.7.2 LinearLayer Class

```
class LinearLayer:
    def __init__(self, d_in, d_out):
        self.requires_grad = True  # layer has trainable parameters
        # Initialize weights and biases
        self.W = np.random.randn(d_in, d_out) / np.sqrt((d_in + d_out) / 2.0)  # Example: He or Xavier-like init
        self.b = np.zeros(d_out)

    def forward(self, X):
        """
        X shape: (N, d_in)
        W shape: (d_in, d_out)
        b shape: (d_out,)
        output shape: (N, d_out)
        """
        self.X = X  # Save input for backward
        out = X @ self.W + self.b
        return out

```

```
    def backward(self, dZ):
        """
        dZ shape: (N, d_out)
        dW shape: (d_in, d_out)
        db shape: (d_out,)
        dX shape: (N, d_in)
        """
        N = self.X.shape[0]
        self.dW = (1.0 / N) * (self.X.T @ dZ)
        self.db = (1.0 / N) * np.sum(dZ, axis=0)
        dX = dZ @ self.W.T
        return dX
```

2.7.3 MLP Class

```
class MLP:
    """
    layer_dims: list of ints representing layer sizes.
    e.g., [d_in, 10, 5, d_out] => 3 hidden layers.
    act: activation function class (e.g., ReLU).
    """
    def __init__(self, layer_dims, act):
        num_layers = len(layer_dims) - 1
        self.layers = []

        for i in range(num_layers):
            # Add a linear layer
            self.layers.append(LinearLayer(layer_dims[i], layer_dims
    [i+1]))

            # Add the activation layer except for the last layer
            if i < num_layers - 1:
                self.layers.append(act())
        # Note: The final activation (e.g. sigmoid) may be included
        #       in the loss function (e.g. BCE) instead of here.

    def forward(self, X):
        for layer in self.layers:
            X = layer.forward(X)
        return X

    def backward(self, dlogits):
        """
        dlogits: typically y_pred - y from the BCE loss,
                 or the gradient from the previous layer.
        """
        dZ = dlogits
        for layer in reversed(self.layers):
            dZ = layer.backward(dZ)
```

2.7.4 Gradient Descent (GD) Class

```
class GD():
    def __init__(self, lr=1e-3, weight_decay=0.0):
        self.lr = lr
        self.weight_decay = weight_decay

    def step(self, model):
        for layer in model.layers:
            if layer.requires_grad: # make sure layer has trainable parameters
                layer.W = layer.W - (self.lr * layer.dW + self.weight_decay * layer.W)
                layer.b = layer.b - self.lr * layer.db
```

2.7.5 Creating Artificial 2D Dataset

```
import numpy as np

# Set random seed for reproducibility
np.random.seed(0)

# Generate dataset
num_samples = 1000
# Class 0: Gaussian centered at (2, 2)
mean0 = [2, 2]
cov0 = [[2, 1], [1, 2]]
data0 = np.random.multivariate_normal(mean0, cov0, num_samples)
labels0 = np.zeros(num_samples)

# Class 1: Gaussian centered at (-1, -1)
mean1 = [-1, -1]
cov1 = [[2, -1], [-1, 2]]
data1 = np.random.multivariate_normal(mean1, cov1, num_samples)
labels1 = np.ones(num_samples)

# Combine data
X = np.vstack((data0, data1))
y = np.hstack((labels0, labels1))

y = y.reshape(-1, 1)
```

2.7.6 Plotting the Data

```
import matplotlib.pyplot as plt

x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
```

```
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
                     np.arange(y_min, y_max, 0.1))

plt.figure(figsize=(10, 5))
plt.scatter(X[y.ravel() == 0, 0], X[y.ravel() == 0, 1], label='Class 0', alpha=0.5)
plt.scatter(X[y.ravel() == 1, 0], X[y.ravel() == 1, 1], label='Class 1', alpha=0.5)
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.title('Data')
plt.legend()
plt.grid(True)
plt.show()
```

Figure 2.9: Data

2.7.7 Training Loop

```
# Construct the model
model = MLP(layer_dims=[2, 200, 200, 1], act=ReLU)
loss_fn = BCE()
optim = GD(lr=5e-1)
losses = []
accs = []
num_iter = 500

# Start training
for i in range(num_iter):
    Z = model.forward(X)
    loss, y_hat = loss_fn.forward(Z, y)
    losses.append(loss)
    accs.append(np.mean((y_hat > 0.5).astype(int) == y))
    dZ = loss_fn.backward(y_hat, y)
    model.backward(dZ)
    optim.step(model)
```

2.7.8 Plotting Training Loss and Accuracy

```
import matplotlib.pyplot as plt
plt.figure(figsize=(8, 5))
plt.plot(losses, marker='o')
plt.xlabel('Iteration')
plt.ylabel('Loss')
plt.title('Training Loss')
plt.grid(True)
plt.show()

plt.figure(figsize=(8, 5))
plt.plot(accs, marker='o')
plt.xlabel('Iteration')
plt.ylabel('Accuracy')
plt.title('Training Accuracy')
plt.grid(True)
plt.show()
```

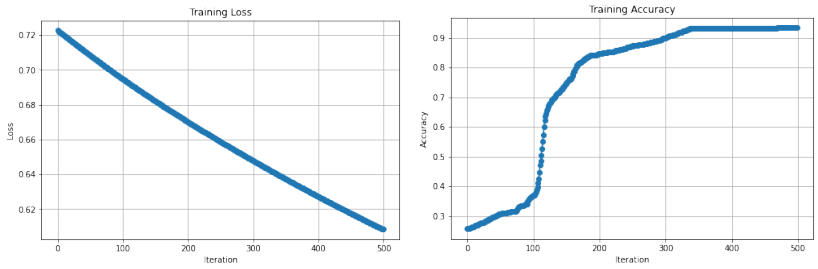


Figure 2.10: Loss and accuracy

2.7.9 Plotting Neural Network Predictions

```
Z = model.forward(X)
loss, y_hat = loss_fn.forward(Z, y)
plt.scatter(X[:, 0], X[:, 1], c=(y_hat > 0.5).astype(int))
plt.show()
```

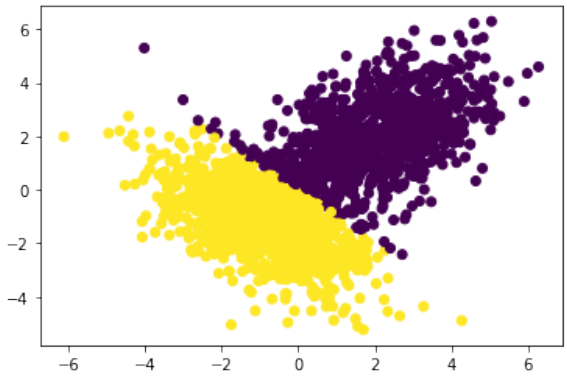


Figure 2.11: Predictions

Chapter 3

Approximation Theory

Disclaimer 2: *The results discussed in Section 3.2 are from Telgarsky (2021).*

In this chapter, we explore a key result in deep learning: the universal approximation theorem. This theorem provides guarantees about how well certain deep neural networks can approximate a class of functions. Before delving into the theorem, we must first address the following question: How do we measure the quality of an approximation?

3.1 Quality of approximation

In order to take a formal approach, we will consider a normed vector space over the space of functions. The typical candidate is an L^p space which may be defined as a space of functions $f : S \to \mathbb{R}$, for which the p-th power of the absolute value is Lebesgue integrable, i.e.

$$\|f\|_p = \begin{cases} \left(\int_S |f|^p \, \mathrm{d}\mu\right)^{1/p} & 1 \leq p < \infty \\ \sup_{x \in S} |f(x)| & p = \infty \end{cases} \qquad < \infty.$$

Using the above ℓ^p norm, we can define the quality of approximation between two functions $f : S \to \mathbb{R}$, $g : S \to \mathbb{R}$ as

$$\text{approx-err}(f, g) := \|f - g\|_p.$$

We can then easily extend this notion of approximation to entire function classes $\mathcal{G}$ as follows:

$$\text{approx-err}(f, \mathcal{G}) := \inf \{g \in \mathcal{G} : \text{approx-err}(f, g)\} \ .$$

Approximability Let's consider a family of models denoted by $\mathcal{G}_m$, $m = 1, 2, \ldots$; $\mathcal{G} = \bigcup_m \mathcal{G}_m$. This could for instance be MLPs with m hidden units. We will use the notation

$$f \simeq \mathcal{G} : \quad \text{approx-err}(f, \mathcal{G}) = 0.$$

Clearly,

$$f \in \mathcal{G} \quad \Longrightarrow \quad f \simeq \mathcal{G}\,.$$

But note that we also have a similar definition for converging sequences:

$$\mathcal{G} \ni g_m \xrightarrow{\text{unif.}} f \Longrightarrow f \simeq \mathcal{G},$$

where uniform convergence is defined as

$$(g_m) \xrightarrow{\text{unif.}} f \iff \forall \epsilon > 0 : \ \exists m \geq 1 : \ \|g_m - f\|_\infty < \epsilon\,.$$

We can generalize the concept of approximability to function classes using the definition of denseness as follows:

$$\mathcal{G} \subseteq \mathcal{F} \text{ is dense in } \mathcal{F} \iff \mathcal{F} \simeq \mathcal{G} \iff \forall f \in \mathcal{F} : f \simeq \mathcal{G}.$$

The largest class of approximated functions is the closure $\text{cl}(\mathcal{G})$.

$$\text{cl}(\mathcal{G}) \simeq \mathcal{G} \quad \text{and} \quad \mathcal{F} \simeq \mathcal{G} \Longrightarrow \mathcal{F} \subseteq \text{cl}(\mathcal{G}).$$

Using the concept introduced so far, we arrive to the following formal definition of a universal approximator for a continuous target function $C(\mathbb{R}^n)$:

$$\boxed{\mathcal{G} \text{ is a universal approximator} \iff C(S) \simeq \mathcal{G}(S) \text{ for any compact } S \subset \mathbb{R}^n} \tag{3.1}$$

We will also use the shortcut notation $\mathcal{G}_S$ to denote the restriction of functions in $\mathcal{G}$ to S.

3.2 Elementary Folklore Construction

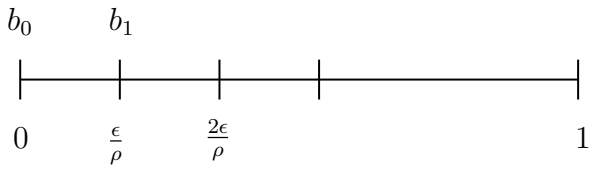
We will start with some constructive results, where we explicitly choose the form of the function used to approximate a given target function. We will for instance rely on some simple step piecewize functions for which we can easily derive some results. We first discuss the unitary case where $f : [0, 1] \to \mathbb{R}$ and then discuss the extension to the multivariate case where $f : \mathbb{R}^d \to \mathbb{R}$ for an arbitrary input dimension $d \geq 1$.

Univariate case The following theorem shows that a ρ-Lipschitz [1] target function can be approximated by a function f that uses threshold activations of the form $\mathbf{1}[x \geq b] = \begin{cases} 1 \text{ if } x \geq b \\ 0 \text{ else.} \end{cases}$

Theorem 14 (Telgarsky (2021))**.** *Let $\epsilon > 0$ and assume $g : [0,1] \to \mathbb{R}$ is ρ-Lipschitz and $f : [0,1] \to \mathbb{R}$ is a 2-layer neural network with $\lceil \frac{\rho}{\epsilon} \rceil$ threshold nodes. Then we have*

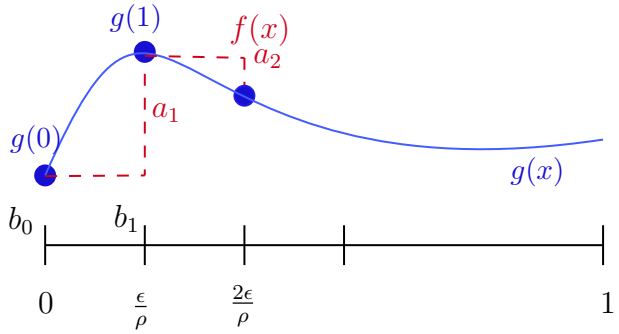
$$\sup_{x \in [0,1]} |f(x) - g(x)| \leq \epsilon. \tag{3.2}$$

Proof. Let $m := \lceil \frac{\rho}{\epsilon} \rceil$ (where ρ is the Lipschitz constant of g) and define a grid of points over the interval $[0,1]$ as $b_i = i \cdot \frac{\epsilon}{\rho}$ for $i \in \{0, \ldots, m-1\}$, as illustrated below.



We will also define a sequence a_i where $a_0 = g(0)$ and $a_i = g(b_i) - g(b_{i-1})$ for $i > 1$.

Next we define the neural network function $f(x) := \sum_{i=0}^{m-1} a_i \mathbf{1}[x \geq b_i]$. Then for any $x \in [0,1]$, letting k be the largest index so that $b_k \leq x$, then f is constant on the interval $[b_k, x]$ (since $\mathbf{1}[x \geq b_j] = 0$ for all $j > k$).



[1] Recall a function $g : \mathbb{R} \to \mathbb{R}$ is ρ-Lipschitz if $|g(x) - g(y)| \leq \rho |x - y|$.

The distance $|g(x) - f(x)|$ is upper bounded as follows:

$$\begin{aligned}
|g(x) - f(x)| &\overset{(i)}{\leq} |g(x) - g(b_k)| + |g(b_k) - f(b_k)| + |f(b_k) - f(x)| \\
&\leq \rho|x - b_k| + \left| g(b_k) - \sum_{i=0}^{k} a_i \right| + 0 \\
&\leq \rho\frac{\epsilon}{\rho} + \left| g(b_k) - g(b_0) - \sum_{i=1}^{k} g(b_i) - g(b_{i-1}) \right| \\
&= \epsilon,
\end{aligned}$$

where (i) uses the triangle inequality and the last inequality holds because $\sum_{i=1}^{k} g(b_i) - g(b_{i-1}) = g(b_k) - g(b_0)$ (telescoping sum).

□

Note that the number of nodes required in Theorem 14 to approximate the target function f scales with the Lipschitz constant ρ. Essentially, the number of nodes increases for functions that can vary more (i.e. functions with larger ρ).

Multivariate case Next, let us consider an extension to the multivariate case. We first present an auxiliary lemma that allows us to approximate continuous functions with a piecewize constant function h.

Lemma 15. *Let g, δ, ϵ be given as in Theorem 14. Assume we are given $U \subseteq \mathbb{R}^d$ and a partition $\mathcal{P}$ of the set U into rectangles of side lengths at most δ, i.e. $\mathcal{P} = (R_1, \ldots, R_n)$. Then $\exists(\alpha_1, \ldots \alpha_n)$ such that $\sup_{\mathbf{x}\in U} |g(\mathbf{x}) - h(\mathbf{x})| \leq \epsilon$ where $h(\mathbf{x}) = \sum_{i=1}^{n} \alpha_i \mathbf{1}_{R_i}(\mathbf{x})$ and $\mathbf{1}_{R_i}(\mathbf{x}) = \begin{cases} 1 \text{ if } \mathbf{x} \in R_i \\ 0 \text{ else.} \end{cases}$*

Proof. For each rectangle R_i of the partition $\mathcal{P}$, choose $\mathbf{x}_i \in R_i$ (for instance the center) and set $\alpha_i := g(\mathbf{x}_i)$. We then have

$$\begin{aligned}
\sup_{\mathbf{x}\in U} |g(\mathbf{x}) - h(\mathbf{x})| &= \sup_{i\in[1,n]} \sup_{\mathbf{x}\in R_i} |g(\mathbf{x}) \pm g(\mathbf{x}_i) - h(\mathbf{x})| \\
&\overset{(i)}{\leq} \sup_{i\in[1,n]} \sup_{\mathbf{x}\in R_i} |g(\mathbf{x}) - g(\mathbf{x}_i)| + |g(\mathbf{x}_i) - h(\mathbf{x})| \\
&\leq \sup_{i\in[1,n]} \sup_{\mathbf{x}\in R_i} \rho\|\mathbf{x} - \mathbf{x}_i\| + |g(\mathbf{x}_i) - h(\mathbf{x})| \\
&\overset{(ii)}{\leq} \sup_{i\in[1,n]} \sup_{\mathbf{x}\in R_i} \epsilon + |\alpha_i - \alpha_i| = \epsilon,
\end{aligned}$$

where (i) uses the triangle inequality and (ii) simply uses the conditions required on g, δ, ϵ.

□

Using Lemma 15, we will see that one can approximate a continuous function with a 3-layer neural network.

Theorem 16 (Telgarsky (2021))**.** *Consider a continuous function $g : \mathbb{R}^d \to \mathbb{R}$. Assume that, given $\epsilon > 0$, there exists $\delta > 0$ such that for $\mathbf{x}, \mathbf{x}' \in \mathbb{R}^d$ with $\|\mathbf{x}-\mathbf{x}'\| \leq \delta$, then $|g(\mathbf{x}) - g(\mathbf{x}')| \leq \epsilon$. Then there exists a 3-layer neural network $f : \mathbb{R}^d \to \mathbb{R}$ with $\Omega(\delta^{-d})$ ReLU activation functions such that*

$$\int_{[0,1]^d} |f(\mathbf{x}) - g(\mathbf{x})| d\mathbf{x} \leq 2\epsilon. \tag{3.3}$$

Proof. Before we start with the proof, we recall the definition of the ℓ_1 norm for function spaces defined in a space $M \subseteq \mathbb{R}^d$: $\|f\|_1 = \int_M |f(\mathbf{x})| d\mathbf{x}$. We will also denote the j-th coordinate of $\mathbf{x}$ as $x^{(j)}$, i.e. $\mathbf{x} = [x^{(1)}, \ldots, x^{(d)}] \in \mathbb{R}^d$.

Let $\mathcal{P}$ be a partition of $M = [0, 2)^d$ (we will later restrict this to the set $[0, 1)^d$) into rectangles $\times_{j=1}^d [a_j, b_j)$ with $b_j - a_j \leq \delta$. By Lemma 15, we already know that g can be approximated by a piecewize constant function h such that $\|g - h\|_1 \leq \epsilon$. Our goal is to show that $\|f - g\|_1 \leq 2\epsilon$. Based on the triangle inequality, we have

$$\|f - g\|_1 \leq \|f - h\|_1 + \|h - g\|_1,$$

which implies that we now need to prove that $\|f - h\|_1 \leq \epsilon$.

To do so, we will choose the network function f to be $f = \sum_i \alpha_i g_i(\mathbf{x})$ where $g_i(\mathbf{x})$ is a ReLU network with two hidden layers and $\mathcal{O}(d)$ nodes. With such f, we can bound the term as follows:

$$\|f - h\|_1 \leq \sum_i |\alpha_i| \|\mathbf{1}_{R_i} - g_i\|_1,$$

where we simply used the definition of h as $h(\mathbf{x}) = \sum_{i=1}^n \alpha_i \mathbf{1}_{R_i}(\mathbf{x})$.

In order to complete the proof, we need to construct a function g and a set of rectangles R_i such that $\|\mathbf{1}_{R_i} - g_i\|_1 \leq \frac{\epsilon}{\sum_i |\alpha_i|}$.

Let's fix the index i. We will choose the rectangle R_i to be of the form $R_i := \times_{j=1}^d [a_j, b_j]$. The function g is chosen to be a network with threshold activation functions that outputs the following:

$$g_{i,\gamma}(\mathbf{x}) = \begin{cases} 1, & \mathbf{x} \in R_i \\ 0, & \mathbf{x} \notin \times_j [a_j - \gamma, b_j + \gamma] \\ [0, 1] & \text{else,} \end{cases} \tag{3.4}$$

where $\gamma > 0$ is a constant to be chosen later.

For instance, we can choose a function $g_{i,\gamma}$ using a set of threshold activations σ as follows:

$$g_{i,\gamma}(\mathbf{x}) := \sigma\left(\sum_{j=1}^{d} g_{i,\gamma}^{(j)}(x^{(j)}) - (d-1)\right),$$

where each $g_{i,\gamma}^{(j)}(z)$ is a function for the j-th coordinate defined as

$$\begin{aligned} g_{i,\gamma}^{(j)}(z) &:= \sigma\left(\frac{z-(a_j-\gamma)}{\gamma}\right) - \sigma\left(\frac{z-a_j}{\gamma}\right) - \sigma\left(\frac{z-b_j}{\gamma}\right) + \sigma\left(\frac{z-(b_j+\gamma)}{\gamma}\right) \\ &\in \begin{cases} 1 & z \in [a_j, b_j], \\ 0 & z \notin [a_j-\gamma, b_j+\gamma], \\ [0,1] & \text{else.} \end{cases} \end{aligned}$$

Note that the function $g_{i,\gamma}(\mathbf{x})$ is a two-layer neural network that approximates the $\mathbf{1}_{R_i}$ function. In fact, the approximation quality depends on the γ parameter, as shown next:

$$\begin{aligned} &\|g_{i,\gamma} - \mathbf{1}_{R_i}\|_1 \\ &= \int_{R_i} |g_{i,\gamma}(\mathbf{x}) - \mathbf{1}_{R_i}(\mathbf{x})| d\mathbf{x} + \int_{\times_j [a_j-\gamma, b_j+\gamma] \setminus R_i} |g_{i,\gamma}(\mathbf{x}) - \mathbf{1}_{R_i}(\mathbf{x})| d\mathbf{x} + \int_{[0,2)^d \setminus \times_j [a_j-\gamma, b_j+\gamma]} |g_{i,\gamma}(\mathbf{x}) - \mathbf{1}_{R_i}(\mathbf{x}) \\ &\leq 0 + \left(\prod_{j=1}^{d}(b_j - a_j + 2\gamma) - \prod_{j=1}^{d}(b_j - a_j)\right) + 0 \\ &\leq \mathcal{O}(\gamma). \end{aligned}$$

Finally, we conclude by choosing γ to ensure that $\|\mathbf{1}_{R_i} - g_i\|_1 \leq \frac{\epsilon}{\sum_i |\alpha_i|}$. We have thus constructed a three-layer network where two layers are used to approximate the $\mathbf{1}_{R_i}$ function and the third layer applies an appropriate scaling α_i.

□

Remark 1 (Curse of Dimensionality)**.** *The results presented in Theorem 14 exhibit the well-known curse of dimensionality. In order to obtain a reasonable approximation, the number of units scales exponentially with the dimension d.*

3.3 Weierstrass Theorem

In the last section, we discussed a constructive proof where we designed two- and three-layer neural networks to approximate a continuous function. In this section, we will instead see that one can *approximate continuous functions with a single layer*

under some appropriate conditions. Before we proceed with the theorem, we introduce the concept of Bernstein basis polynomial that will serve as a key tool in the proof of the well-known Weierstrass Theorem.

Bernstein basis poynomials As the name indicates, these polynomials will serve as a basis to construct polynomials as linear combinations of the polynomials in the basis. We fix a degree $m > 0$, then the $m + 1$ Bernstein basis polynomials of degree m are defined as

$$\boxed{b_k^m(x) = \binom{m}{k} x^k (1-x)^{m-k}.} \tag{3.5}$$

These polynomials form a partition of unity, i.e. a convex combination for every x.

$$\sum_{k=0}^{m} b_k^m(x) = 1 \quad \forall x \in [0;1] \tag{3.6}$$

Figure 3.1 shows an example of these polynomials for the case where $m = 3$. Note that these polynomials sum up to 1. Additionally, $b_k^m(x)$ has a small value when x is far away from $\frac{k}{m}$.

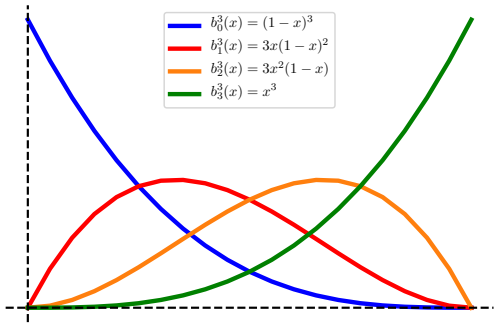


Figure 3.1: Example of Bernstein basis for degree $m = 3$. Source: Wikipedia.

The following theorem is a classical result in the field of neural networks and more broadly in functional analysis. The original version of this result was established by Karl Weierstrass in 1885. It has also been generalized by Stone in 1937, under the name of Stone-Weierstrass theorem.
We will use the notation $C([a, b])$ to denote the space of all continuous functions that are defined on a closed interval $[a, b]$.

Theorem 17 (Weierstrass Theorem)**.** *Polynomials $\mathcal{P}$ are dense in $C(I)$, where $I = [a; b]$ for any $a < b$.*

Before we give a proof sketch, we highlight the main ideas of the proof. We start with a proposition that reduces the proof to the interval $I = [0; 1]$.

Proposition 18. *W.l.o.g. we can focus on* $I = [0; 1]$ *as*

$$C([0;1]) \simeq \mathcal{P}([0;1]) \implies C([a;b]) \simeq \mathcal{P}([a;b]),\ \forall a < b.$$

Proof sketch. We follow the following steps:

1. Define $\phi : C([0;1]) \to C([a;b])$ as $\phi(f) = f \circ \phi,\ \phi = (1-t)a + tb$.
2. if $g_m \xrightarrow{\text{unif.}} f \in C([0;1)]$, then $g_m \circ \phi \xrightarrow{\text{unif.}} f \circ \phi \in C([a;b])$
3. The result follows from observing that $g_m \circ \phi$ is a polynomial, if g_m is.

□

We use the basis polynomials to convexly combine function values on a lattice with spacing $1/m$ and define Bernstein polynomials

$$\boxed{q_m(x) = \sum_{k=0}^{m} f\left(\frac{k}{m}\right) b_k^m(x), \quad b_k^m(x) = \binom{m}{k} x^k (1-x)^{m-k}.} \tag{3.7}$$

which is a polynomial of degree m. We then show that $q_m \xrightarrow{\text{unif.}} f$ by looking (independently) at the residuals

$$|f(x) - q_m(x)| = \left|\sum_{k=0}^{m} r_k^m(x)\right|, \quad r_k^m(x) := \left[f(x) - f\left(\frac{k}{m}\right)\right] b_k^m(x) \tag{3.8}$$

Informally, the law of large numbers implies that the probability mass of the binomial distribution will concentrate around $x = k/m$ as the numbers of trials $m \to \infty$. This basic observation is at the core of the proof.

Proof sketch of Weierstrass theorem. The proof proceeds in the following steps:

1. Each residual is the product of two factors. It will vanish, if either one vanishes (while the counterpart remains bounded).
2. Chose δ such that $|x - y| \leq \delta$ implies $|f(x) - f(y)| \leq \epsilon/2$. This is possible as f is continuous. Consider lattice points $\mathcal{I} := \{k : |x - \frac{k}{m}| \leq \delta\}$, then we can split the contributions from the points in $\mathcal{I}$ and their complement as follows:

$$|f(x) - q_m(x)| \leq \left|\sum_{k \in \mathcal{I}} r_k^m(x)\right| + \left|\sum_{k \notin \mathcal{I}} r_k^m(x)\right|.$$

3. For the points in the lattice, we have

$$\sum_{k\in\mathcal{I}} r_k^m(x) \leq \frac{\epsilon}{2}\sum_{k\in\mathcal{I}} b_k^m(x) \leq \frac{\epsilon}{2}\sum_{k=1}^{m} b_k^m(x) \overset{(3.6)}{=} \frac{\epsilon}{2}$$

4. Now, let us consider the points in $\mathcal{I}^c$. By continuity, there is a $R > 0$ such that

$$\left|\sum_{k\notin\mathcal{I}} r_k^m(x)\right| \leq \sum_{k\notin\mathcal{I}} |r_k^m(x)| \leq R\sum_{k\notin\mathcal{I}} b_k^m(x) \overset{!}{<} \frac{\epsilon}{2}.$$

To prove the last inequality, one may introduce $(x-k/m)^2 > \delta^2$ and show that

$$\sum_{k=1}^{m}\left(x-\frac{k}{m}\right)^2 b_k^m(x) = \frac{x(1-x)}{m}$$

which is a variance formula (we will not prove this here). Then, for m large enough

$$R\sum_{k\notin\mathcal{I}} b_k^m(x) \leq \frac{Rx(1-x)}{m\delta^2} \leq \frac{R}{4m\delta^2} \overset{!}{<} \frac{\epsilon}{2}, \quad m > \frac{R}{2\epsilon\delta^2}.$$

□

A detailed proof will be discussed in the exercise sheet.

Generalization: Stone-Weierstrass The Stone-Weierstrass theorem is a generalization of Weierstrass theorem that states that any family of functions satisfying some of the same properties as polynomials will also be a universal approximator. Further details regarding this theorem will be provided in the exercise sheet.

3.4 Approximation with Smooth Functions

Univariate case Let us now consider an arbitrary smooth function $\sigma \in C^\infty(\mathbb{R})$ and the span of its composition with affine functions

$$\mathcal{G}_\sigma^1 = \{g : g(x) = \sigma(ax+b) \text{ for some } a, b \in \mathbb{R}\}$$
$$\mathcal{H}_\sigma^1 = \text{span}(\mathcal{G}_\sigma^1)$$

In a seminal paper, Hornik et al. (1989) showed that $\mathcal{H}_\sigma^1$ can be a universal approximator whenever the activation function is continuous, bounded and nonconstant.

This result was later generalized by Leshno et al. (1993) who proved the following result.

Theorem 19 (Leshno et al. (1993)). *For any $\sigma \in C^\infty(\mathbb{R})$ not a polynomial, $\mathcal{H}^1_\sigma$ is a universal approximator.*

Proof sketch. We proceed backward for didactic reasons.

1. It is sufficient to show that polynomials can be approximated

$$\mathcal{P} = \left\{ \sum_{k=0}^{r} \alpha_k x^k \mid \alpha_k \in \mathbb{R},\ r \geq 0 \right\} \subseteq \mathrm{cl}(\mathcal{H}^1_\sigma),$$

   We can then evoke the Weierstrass theorem along with the subadditivity of the sup-norm (triangle inequality) to prove the claim.

2. Since $\mathcal{H}^1_\sigma$ is a linear span, it is sufficient to show that monomials can be approximated

$$\{x^k : k \geq 0\} \subset \mathrm{cl}(\mathcal{H}^1_\sigma).$$

   This is because if $h_k \in \mathcal{H}^1_\sigma$ are ϵ_k-approximations of x^k then for $h(x) := \sum_{k=0}^r \alpha_k h_k(x) \in \mathcal{H}^1_\sigma$ one obtains by virtue of subadditivity

$$\left\| \sum_{k=0}^{r} \alpha_k x^k - h(x) \right\|_\infty \leq \sum_{k=0}^{r} |\alpha_k| \|x^k - h_k(x)\|_\infty < \sum_{k=0}^{r} |\alpha_k| \epsilon_k := \epsilon\,.$$

3. As $\sigma \in C^\infty$ all its k-th order derivatives exist (denoted by $\sigma^{(k)}$), namely for $z = ax + b$,

$$\frac{d^k \sigma(z)}{da^k} = x^k \sigma^{(k)}(z) \overset{a=0}{=} x^k \sigma^{(k)}(b)$$

   Note that if σ was a k-th degree polynomial, we would get $\sigma^{(k)} = 0$. It turns out the opposite is also true. If $\sigma \notin \mathcal{P}$, then $\sigma^{(k)} \neq 0$ (see Lemma 20 below). We can thus chose b_k such that $\sigma^{(k)}(b_k) \neq 0$.

4. Let us focus on $k = 1$. The derivative can be uniformly approximated on any compact set by a finite difference

$$\frac{\sigma((a+h)x + b) - \sigma(ax + b)}{h} \xrightarrow{\text{unif.}} \frac{d\sigma(ax + b)}{da} \quad \text{as } h \to 0,$$

   where $\xrightarrow{\text{unif.}}$ means uniform convergence. The left-hand side is indeed a linear combination of σ-ridge functions. Combined with step 3., this means that we can approximate any one-degree monomial. Similar expressions exist for higher-order derivatives (skipped for the sake of brevity) and provide uniform approximations of monomials $x^k \simeq \mathcal{H}^1_\sigma$.

5. Since we can approximate all monomials, we can simply use the Weierstrass theorem to conclude.

□

Remark 2. *The condition that σ is not a polynomial is intuitive. Indeed, think about what types of functions can be approximated with a linear combination of polynomials of degree k? Can you for instance approximate a polynomial of degree $k+1$ or greater?*

The missing step above follows from:

Lemma 20 (Donoghue, 1969; Pinkus 1999)**.** *If σ is C^∞ on $(a;b)$ and it is not a polynomial thereon, then there exists a point $x_0 \in (a;b)$ such that $\sigma^{(k)}(x_0) \neq 0$ for $k = 0, 1, 2, \ldots$.*

The above result can be further generalized and in fact, it can be shown that the smoothness assumption is not necessary and $\sigma \in C(\mathbb{R}) - \mathcal{P}$ is sufficient. See, for instance, Proposition 3.7 of Pinkus (1999).

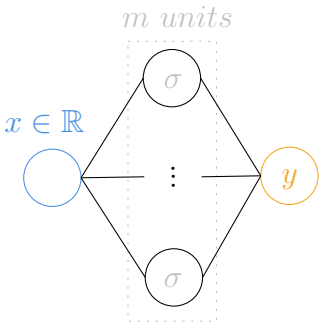


Figure 3.2: For $n = 1$ (one-dimensional inputs), an MLP with smooth activation function σ is a universal approximator, as long as the number of units m is large enough.

What we have shown now is that for the case of $n = 1$ (one-dimensional inputs), an MLP with smooth activation function σ is a universal approximator, unless σ is a polynomial. This is because the linear output layer is picking an element in the span of the hidden units, each one of which computes a function in $\mathcal{G}_\sigma^1$. So as far as universal function approximation is concerned, there is nothing special about the logistic function or the hyperbolic tangent as choices of activation functions.

Generalization to multiple dimensions ($\mathbf{x} \in \mathbb{R}^n$) One basic tool for lifting results from one dimension to higher dimensions are ridge functions. They are universal approximators as shown in the next theorem. First, let us define a function class based on ridge functions:

$$\mathcal{G}_\sigma^n = \{g : g(\mathbf{x}) = \sigma(\boldsymbol{\theta} \cdot \mathbf{x}),\ \boldsymbol{\theta} \in \mathbb{R}^n\}, \quad \mathcal{H}_\sigma^n = \text{span}(\mathcal{G}_\sigma^n)$$

$$\mathcal{G}^n = \bigcup_{\sigma \in C(\mathbb{R})} \mathcal{G}_\sigma^n, \quad \mathcal{H}^n = \text{span}(\mathcal{G}^n)$$

Theorem 21 (Ridge Function Theorem (Vostrecov and Kreines, 1961))**.** *$\mathcal{H}^n$ is a universal function approximator.*

Proof. Omitted. □

The theorem states that neural networks with adaptive activation functions "could be" universal approximators. However, this in itself is not practical due to the union over all possible continuous functions $\sigma \in C(\mathbb{R})$. How would you implement such a neural network? Instead, we will introduce the concept of dimension lifting to fix this problem.

Dimension Lifting The following beautiful theorem will provide the missing link that connects the above result with the universality result we have for $n = 1$.

Recall we showed earlier that for any $\sigma \in C^\infty(\mathbb{R})$ *not* a polynomial, $\mathcal{H}_\sigma^1$ is a universal approximator.

Theorem 22 (Pinkus (1999))**.** *For a fixed $\sigma \in C^\infty$,*

$$\mathcal{H}_\sigma^1 \text{ universal for } C(\mathbb{R}) \implies \mathcal{H}_\sigma^n \text{ universal for } C(\mathbb{R}^n) \text{ for any } n \geq 1$$

Proof sketch. The proof proceeds in the following steps:

1. Fix f and compact $K \subset \mathbb{R}^n$. By Theorem 21, we can find ridge functions g_k s.t. for some $m > 0$, we have

$$\left| f(\mathbf{x}) - \sum_{k=1}^{m} g_k(\boldsymbol{\theta}^k \cdot \mathbf{x}) \right| < \frac{\epsilon}{2} \quad (\forall \mathbf{x} \in K).$$

2. Since K is compact, $\boldsymbol{\theta}^k \cdot \mathbf{x} \in [\alpha_k, \beta_k]$ for $\mathbf{x} \in K$.

3. Because $\mathcal{H}_\sigma^1$ is dense in each $C([\alpha_k, \beta_k])$, Theorem 19 implies that we can find constants s.t.

$$\left| g_k(z) - \sum_{j=1}^{m_k} c_{kj}\sigma(\alpha_{kj}z + \beta_{kj}) \right| \leq \frac{\epsilon}{2m} \quad (\forall k = 1, \ldots, m)$$

4. Plugging things together yields the result.

□

Note that the previous theorem does not specify the exact number m of units required to ensure universal approximation. This topic will be explored in detail in the next chapter.

Summary To summarize the argument presented in this section:

1. For $n = 1$, an MLP with any continuous, non-polynomial activation function is a universal approximator.

2. Spans of ridge functions are universal approximators for $C(\mathbb{R}^n)$.

3. The non-linear part of any/each ridge function can be approximated according to (1).

4. Hence MLPs are universal function approximators.

3.5 Exercise: Approximation Theory

Problem 1 (Weierstrass theorem):

In this exercise, we seek to derive a formal proof of the Weierstrass theorem discussed in the lecture. Recall that the theorem can be stated as follows:

Theorem 23 (Weierstrass). *Given a continuous function $f(x)$ on $a \leq x \leq b$ and an arbitrary positive constant $\epsilon > 0$, it is possible to construct an approximating polynomial $P(x)$ such that*

$$|f(x) - P(x)| \leq \epsilon, \ \forall a \leq x \leq b \tag{3.9}$$

Without loss of generality, we assume $a = 0, \ b = 1$ and extend f on the whole $\mathbb{R}$ by setting $f(x) = 0, \ \forall x \notin (0, 1]$, such that f is continuous on $(-\infty, 1]$. (Why can we do this?)

Let $P_n(x)$ be a polynomial of degree $2n$ such that

$$P_n(x) = \frac{1}{J_n} \int_0^1 f(t)[1 - (t - x)^2]^n dt, \tag{3.10}$$

where $J_n := \int_{-1}^1 (1 - u^2)^n du$ is a constant .

a) Show that

$$f(x) = \frac{1}{J_n} \int_{-1}^1 f(x)(1 - u^2)^n du \tag{3.11}$$

b) Show that

$$P_n(x) - f(x) = \frac{1}{J_n} \int_{-1}^1 [f(x + u) - f(x)](1 - u^2)^n du \tag{3.12}$$

where $x \in [0, 1]$. The problem is now to show that this expression approaches zero as $n \to \infty$.

c) Let $\epsilon > 0$. Use the following facts freely:

- since $f(x)$ is continuous on $[-1, 1]$, there exists a $\delta > 0$ such that $|f(x+u) - f(x)| \leq \frac{\epsilon}{2}$ for each x, u with $|u| < \delta$ and $x, x + u \in [-1, 1]$;
- there exist a positive constant $M > 0$, such that $|f(x)| \leq M, \ \forall x \in [-1, 1]$.

Show that

$$|f(x + u) - f(x)| \leq \frac{\epsilon}{2} + 2M \frac{u^2}{\delta^2}, \ \forall x \in [-1, 1]. \tag{3.13}$$

Hint: Think of the case distinction where $|u| \geq \delta$, i.e. $1 \leq \frac{u^2}{\delta^2}$; and where $|u| < \delta$.

d) Using integration by part, show that

$$J'_n := \int_{-1}^{1} u^2(1-u^2)^n du = \frac{J_{n+1}}{2(n+1)}$$

and also show that

$$J_n > J_{n+1},\ \forall n \in \mathbb{N}.$$

e) Finally, re-using the answers to the previous sub-problems, prove that

$$|f(x) - P_n(x)| \leq \epsilon,\ \forall x \in [0,1] \tag{3.14}$$

for sufficiently large n.

Problem 2 (Stone-Weierstrass theorem):
The following is a generalization of Weierstrass theorem.

Theorem 24 (Stone-Weierstrass, see Theorem 2.2. in Telgarsky (2021))**.** *Let function class $\mathcal{F}$ of real-valued functions defined on $[0,1]^d$ be given as follows:*

i) Each $f \in \mathcal{F}$ is continuous.

ii) For every $\mathbf{x} \in [0,1]^d$, there exists $f \in \mathcal{F}$ with $f(\mathbf{x}) \neq 0$.

iii) For every $\mathbf{x} \neq \mathbf{x}' \in [0,1]^d$, there exists $f \in \mathcal{F}$ with $f(\mathbf{x}) \neq f(\mathbf{x}')$ ($\mathcal{F}$ separates points).

iv) $\mathcal{F}$ is closed under multiplication and vector space operations, i.e. $\mathcal{F}$ is an algebra.

Then $\mathcal{F}$ is a universal approximator: for every continuous $g : [0,1]^d \to \mathbb{R}$ and $\epsilon > 0$, there exists $f \in \mathcal{F}$ with

$$|f(\mathbf{x}) - g(\mathbf{x})| \leq \epsilon, \ \forall \mathbf{x} \in [0,1]^d.$$

a) Consider unbounded width networks with one hidden layer:

$$\mathcal{F}_{\sigma,d,m} := \mathcal{F}_{d,m} := \left\{ x \mapsto a^\top \sigma(Wx + b) : a \in \mathbb{R}^m, W \in \mathbb{R}^{m \times d}, b \in \mathbb{R}^m \right\}. \tag{3.15}$$

$$\mathcal{F}_{\sigma,d} := \mathcal{F}_d := \bigcup_{m \geq 0} \mathcal{F}_{\sigma,d,m}. \tag{3.16}$$

Using Theorem 24, show that $\mathcal{F}_{\cos,d}$ is universal, where $\cos : \mathbb{R} \to \mathbb{R}$ is the cosine function.

b) Show that Theorem 24 does not hold if

i) condition (i) in Theorem 24 does not hold;

ii) condition (ii) in Theorem 24 does not hold;

iii) condition (iii) in Theorem 24 does not hold;

iv) condition (iv) in Theorem 24 does not hold.

Chapter 4

Complexity Theory

Disclaimer 2: *Part of these lecture notes are based on the notes from Telgarsky (2021).*

In the last lecture, we have seen that the set of function classes represented by neural networks is rich. We first saw that a rather elementary constructive proof gives a rough estimate of the number of units required to obtain a desired accuracy to approximate continuous functions. Notably, the approximation rate scaled exponentially with the input dimension. We then turned to universal approximation results that show that a single-hidden layer network can approximate any function, but those results do not give any indication of the number of units required to obtain a desired accuracy. In this lecture, we will focus on two related topics. First, we will see that for functions that exhibit enough regularity, there exists a one-hidden layer neural network that can evade the curse of dimensionality. Second, we will ask whether there is any advantage brought by compositionality (i.e. using a network with multiple layers instead of a single one).

4.1 Escaping the Curse of Dimensionality

We start with a well-known result by (Barron, 1993) that shows that a one-hidden layer neural network can evade the curse of dimensionality. This approach uses Fourier transforms which we will briefly review next.

Review Fourier Transform For any absolutely integrable f, i.e. $\int_{\mathbb{R}^d} |f(\mathbf{x})| d\mathbf{x} < \infty$, we define the Fourier transform of f as

$$\boxed{\widehat{f}(\boldsymbol{\omega}) = \mathcal{F}f(\mathbf{x}) := \int_{\mathbb{R}^d} e^{-2\pi i \boldsymbol{\omega}^\top \mathbf{x}} f(\mathbf{x}) d\mathbf{x}, \quad \widehat{f} : \mathbb{R}^d \to \mathbb{C}.}$$

Let's decipher the different terms in the above equation:

- $f(\mathbf{x})$: This is the original function in the spatial domain.
- $\widehat{f}(\boldsymbol{\omega})$: this is the result of the Fourier transform. It represents the function in the frequency domain. In simple terms, it tells us how much of each spatial frequency (represented by $\boldsymbol{\omega}$) is present in the original function.
- $e^{-2\pi i\boldsymbol{\omega}^\top \mathbf{x}}$: This part represents complex sinusoidal waves at different spatial frequencies ($\boldsymbol{\omega}$). Think of it as a set of patterns or spatial oscillations, each characterized by its spatial frequency. The term $e^{-2\pi i\boldsymbol{\omega}^\top \mathbf{x}}$ is a complex exponential, and it oscillates in space $\mathbf{x}$ at the spatial frequency $\boldsymbol{\omega}$. This part essentially tests how well each spatial frequency component fits with the original function at each spatial position.

A well-known property of the Fourier transform is that it relates to the convolution operation as described in the following statement.

Theorem 25 (Convolution theorem)**.** *Let* $r(\mathbf{x}) = \{g*h\}(\mathbf{x}) \triangleq \int_{-\infty}^{\infty} g(\boldsymbol{\tau})h(\mathbf{x}-\boldsymbol{\tau})\,d\boldsymbol{\tau}$. *Then*

$$\widehat{r}(\boldsymbol{\omega}) = \widehat{g}(\boldsymbol{\omega})\widehat{h}(\boldsymbol{\omega}) \quad \text{and} \quad r(\mathbf{x}) = \mathcal{F}^{-1}(\widehat{g}(\boldsymbol{\omega})\widehat{h}(\boldsymbol{\omega})),$$

where $\mathcal{F}^{-1}\widehat{f}(\boldsymbol{\omega}) = \int \exp(2\pi i\boldsymbol{\omega}^\top \mathbf{x})\widehat{f}(\boldsymbol{\omega})d\boldsymbol{\omega}$. *is the inverse Fourier transform.*

Proof. For the interested reader, we give the proof in the Appendix. □

Regularity Class We will soon see that some assumptions have to be imposed on the type of target functions that can be efficiently approximated by a neural network. The main assumption we will make is a regularity condition on the Fourier transform $\widehat{f}$ of a target function f, namely:

$$\boxed{C_f := \int \|\boldsymbol{\omega}\|\,|\widehat{f}(\boldsymbol{\omega})|d\boldsymbol{\omega} < \infty}$$

Note that if f is differentiable, the Fourier transform of ∇f is given by [1]

$$\widehat{\nabla f}(\boldsymbol{\omega}) = 2\pi i\boldsymbol{\omega}\widehat{f}(\boldsymbol{\omega})\,.$$

The condition $C_f < \infty$ therefore implies that the Fourier transform of the gradient function has to be *absolutely integrable*.

[1] This can be proved using integration by parts.

Barron's construction for infinite-width The main idea is simple. It starts from the inverse Fourier transform

$$f(\mathbf{x}) = \mathcal{F}^{-1}\widehat{f}(\boldsymbol{\omega}) = \int \exp(2\pi i \boldsymbol{\omega}^\top \mathbf{x})\widehat{f}(\boldsymbol{\omega})d\boldsymbol{\omega}. \tag{4.1}$$

Recall that a simple 1-hidden layer network can be written as $f(\mathbf{x}) = \sum_{i=1}^m a_i \phi(\mathbf{x})$ where $\mathbf{a}_i \in \mathbb{R}$ and ϕ is a chosen activation function. Note that in Eq. (4.1) we have written $f(\mathbf{x})$ as an infinite integral which we can interpret as an infinite-width neural network with a rather strange complex activation function. Barron's construction consists into converting these activations into threshold activation functions. The main result is stated below (formulation borrowed from Telgarsky (2021)).

Theorem 26 (Infinite-width representation)**.** *Assume* $\int \|\widehat{\nabla f}(\boldsymbol{\omega})\|d\boldsymbol{\omega} < \infty$, $f \in L^1, \widehat{f} \in L^1$. *Then, for bounded* $\|\boldsymbol{\omega}\|$ *and* $\|\mathbf{x}\| \leq 1$, *we have the following infinite representation of* f *with threshold nodes:*

$$\begin{aligned} f(\mathbf{x}) - f(0) &= \int \frac{\cos(2\pi\boldsymbol{\omega}^\top\mathbf{x} + 2\pi\theta(\boldsymbol{\omega})) - \cos(2\pi\theta(\boldsymbol{\omega}))}{2\pi\|\boldsymbol{\omega}\|}\|\nabla\widehat{f}(\boldsymbol{\omega})\|d\boldsymbol{\omega} \\ &= -2\pi\int\int_0^{\|\boldsymbol{\omega}\|} \mathbf{1}[\boldsymbol{\omega}^\top\mathbf{x} - b \geq 0]\sin(2\pi b + 2\pi\theta(\boldsymbol{\omega}))|\widehat{f}(\boldsymbol{\omega})|dbd\boldsymbol{\omega} \\ &+ 2\pi\int\int_{-\|\boldsymbol{\omega}\|}^0 \mathbf{1}[-\boldsymbol{\omega}^\top\mathbf{x} + b \geq 0]\sin(2\pi b + 2\pi\theta(\boldsymbol{\omega}))|\widehat{f}(\boldsymbol{\omega})|dbd\boldsymbol{\omega}. \end{aligned}$$

Proof. We follow the analysis presented in Telgarsky (2021).

i) We will use the Fourier transform of f and split it into magnitude and radial parts: $\widehat{f}(\boldsymbol{\omega}) = |\widehat{f}(\boldsymbol{\omega})|\exp(2\pi i\theta(\boldsymbol{\omega}))$. We use the notation $\mathbb{R}(a+bi) = a$ to denote the real part of a complex number. Since f is real-valued, we have

$$\begin{aligned} f(\mathbf{x}) &= \mathbb{R}\int \exp(2\pi i\boldsymbol{\omega}^\top\mathbf{x})\widehat{f}(\boldsymbol{\omega})d\boldsymbol{\omega} \\ &= \int \mathbb{R}\exp(2\pi i\boldsymbol{\omega}^\top\mathbf{x})|\widehat{f}(\boldsymbol{\omega})|\exp(2\pi i\theta(\boldsymbol{\omega}))d\boldsymbol{\omega} \\ &= \int \mathbb{R}\exp(2\pi i\boldsymbol{\omega}^\top\mathbf{x} + 2\pi i\theta(\boldsymbol{\omega}))|\widehat{f}(\boldsymbol{\omega})|d\boldsymbol{\omega} \\ &= \int \cos(2\pi\boldsymbol{\omega}^\top\mathbf{x} + 2\pi\theta(\boldsymbol{\omega}))|\widehat{f}(\boldsymbol{\omega})|d\boldsymbol{\omega}, \end{aligned}$$

where we simply used Euler's formula in the last equality: $e^{iz} = \cos(z) + i\sin(z)$. Note how we have already obtained an infinite-size representation of our function. The catch is that the cosine function is not compactly supported. Here comes the second trick to address this problem.

ii) Turning cosines into bumps: By subtracting $f(0)$ to $f(\mathbf{x})$ and scaling by $\|\boldsymbol{\omega}\|$, we obtain

$$\begin{aligned} f(\mathbf{x}) - f(0) &= \int \left[\cos(2\pi\boldsymbol{\omega}^\top\mathbf{x} + 2\pi\theta(\boldsymbol{\omega})) - \cos(2\pi\theta(\boldsymbol{\omega}))\right] |\widehat{f}(\boldsymbol{\omega})| d\boldsymbol{\omega} \\ &= \int \frac{\cos(2\pi\boldsymbol{\omega}^\top\mathbf{x} + 2\pi\theta(\boldsymbol{\omega})) - \cos(2\pi\theta(\boldsymbol{\omega}))}{\|\boldsymbol{\omega}\|} \|\boldsymbol{\omega}\| \cdot |\widehat{f}(\boldsymbol{\omega})| d\boldsymbol{\omega}. \end{aligned} \tag{4.2}$$

Note that cosine is a 1-Lipschitz function, therefore

$$\left|\frac{\cos(2\pi\boldsymbol{\omega}^\top\mathbf{x} + 2\pi\theta(\boldsymbol{\omega})) - \cos(2\pi\theta(\boldsymbol{\omega}))}{\|\boldsymbol{\omega}\|}\right| \leq \frac{2\pi\boldsymbol{\omega}^\top\mathbf{x} + 2\pi\theta(\boldsymbol{\omega}) - 2\pi\theta(\boldsymbol{\omega})}{\|\boldsymbol{\omega}\|} \leq \frac{2\pi|\boldsymbol{\omega}^\top\mathbf{x}|}{\|\boldsymbol{\omega}\|} \leq 2\pi\|\mathbf{x}\|. \tag{4.3}$$

We conclude that $f(\mathbf{x}) - f(0)$ is bounded under the conditions that both $\|\mathbf{x}\|$ and $\|\boldsymbol{\omega}\| \cdot |\widehat{f}(\boldsymbol{\omega})|$ are bounded.

iii) What have we accomplished so far? We have derived an infinite-sized representation for a neural network f based on cosine functions. Next, we will see that we can manipulate our expression to obtain a representation that depends on threshold activation functions. By the fundamental theorem of calculus:

$$\begin{aligned} \cos(2\pi\boldsymbol{\omega}^\top\mathbf{x} + 2\pi\theta(\boldsymbol{\omega})) - \cos(2\pi\theta(\boldsymbol{\omega})) &= \int_0^{\boldsymbol{\omega}^\top\mathbf{x}} -2\pi \sin(2\pi b + 2\pi\theta(\boldsymbol{\omega})) db \\ &= -2\pi \int_0^{\|\boldsymbol{\omega}\|} \mathbf{1}[\boldsymbol{\omega}^\top\mathbf{x} - b \geq 0] \sin(2\pi b + 2\pi\theta(\boldsymbol{\omega})) db \\ &\quad + 2\pi \int_{-\|\boldsymbol{\omega}\|}^{0} \mathbf{1}[-\boldsymbol{\omega}^\top\mathbf{x} + b \geq 0] \sin(2\pi b + 2\pi\theta(\boldsymbol{\omega})) db, \end{aligned}$$

where we used $\|\mathbf{x}\| \leq 1$ to define the domain of integration in the second equality. We have obtained the second expression in the statement of the theorem.

In order to obtain the first expression, we can simply use the expression $\|\widehat{\nabla f}(\boldsymbol{\omega})\| = 2\pi\|\boldsymbol{\omega}\| \cdot |\widehat{f}(\boldsymbol{\omega})|$ (since $\widehat{\nabla f}(\boldsymbol{\omega}) = 2\pi i\boldsymbol{\omega}\widehat{f}(\boldsymbol{\omega})$) in Eq. (4.2), from which we obtain:

$$f(\mathbf{x}) - f(0) = \int \frac{\cos(2\pi\boldsymbol{\omega}^\top\mathbf{x} + 2\pi\theta(\boldsymbol{\omega})) - \cos(2\pi\theta(\boldsymbol{\omega}))}{2\pi\|\boldsymbol{\omega}\|} \|\nabla\widehat{f}(\boldsymbol{\omega})\| d\boldsymbol{\omega}. \tag{4.4}$$

□

In some of the theorems presented in this class so far, we measured the complexity of a neural network by counting the number of parameters required to approximate a given class of functions. Theorem 26, however, does not include any direct mention of the number of parameters. Instead, the complexity is measured through the scaling factor in front of the weight measure, namely $\|\nabla\widehat{f}(\boldsymbol{\omega})\| d\boldsymbol{\omega}$. Therefore, "simpler" networks will be characterized by smaller values of $\int \|\nabla\widehat{f}(\boldsymbol{\omega})\| d\boldsymbol{\omega}$. We will see some examples in the exercise sheet.

Barron's Theorem for finite-size neural network representation We are now ready to state the first main theorem in this lecture that proves that a single hidden-layer network with a finite number of units can approximate a large class of (regular) functions. We will require the following condition to hold on the activation function σ.

Assumption 1 (Condition on σ)**.** *σ is a bounded (measurable) and monotonic function such that $\sigma(t) \overset{t\to\infty}{\longrightarrow} 1$ and $\sigma(t) \overset{t\to-\infty}{\longrightarrow} 0$.*

Theorem 27 (Barron's Theorem (1993))**.** *Under Assumption 1, for every $g : \mathbb{R}^d \to \mathbb{R}$ with finite C_g and any $r > 0$, there is a sequence of MLP functions $g_k(\mathbf{x})$ of the form*

$$g_k(\mathbf{x}) = \sum_{j=1}^{k} \beta_j \sigma(\boldsymbol{\theta}_j \cdot \mathbf{x} + b_j) + b_0$$

such that

$$\int_{\mathbb{B}_r} (g(\mathbf{x}) - g_k(\mathbf{x}))^2 \mu(d\mathbf{x}) \leq \mathcal{O}\left(\frac{1}{k}\right)$$

where $\mathbb{B}_r = \{\mathbf{x} \in \mathbb{R}^d : \|\mathbf{x}\| \leq r\}$ and μ is any probability measure on $\mathbb{B}_r$.

Theorem 27 asserts that we can approximate the function $g(\mathbf{x})$ using a sampled representation $g_k(\mathbf{x})$, with the key insight that the quality of the approximation error decreases as the number of units k increases.

Before we proceed with the proof, let's first discuss the related problem of approximating the mean of a set of random variables by sampling in a Hilbert space H (i.e. a vector space equipped with an inner product). Suppose $X = \mathbb{E}[V]$, where the random variable V is supported on a set $S \subset H$. How can we compute an estimate of the mean? We could simply sample a set of random variables $\{V_1, \ldots, V_k\}$ and compute the empirical mean $\hat{X} = \frac{1}{k}\sum_{i=1}^{k} V_i$. Our hope is to show that $\hat{X}$ gets "closer" to X as we increase the number of samples k (in a Hilbert space, closeness can simply be measured by the norm). This is precisely what the following theorem will show. We note that this result is due to Maurey but it was published in Pisier (1981).

Lemma 28 (Maurey Pisier (1981))**.** *Let $X = \mathbb{E}V$ be given, with V supported on $S \subset H$, and let $V_1, \dots V_k$ be i.i.d. draws from the same distribution. Then*

$$\mathbb{E}_{V_1,\dots V_k} \left\| X - \frac{1}{k}\sum_{i=1}^{k} V_i \right\|^2 \leq \frac{\mathbb{E}\|V\|^2}{k} \leq \frac{\sup_{U\in S}\|U\|^2}{k}.$$

Moreover there exist $(U_1, \dots U_k)$ in S so that

$$\left\| X - \frac{1}{k}\sum_{i=1}^{k} U_i \right\|^2 \leq \mathbb{E}_V \left\| X - \frac{1}{k}\sum_{i=1}^{k} V_i \right\|^2.$$

Proof. See exercise sheet. □

The above theorem gives us a way to approximate the mean of a set random variables defined in a Hilbert space. We are however dealing with functions of random variables. Can we extend our sampling lemma to functions of random variables? First of all, we need to define a valid Hilbert space. An obvious candidate is the L^2 space for which the inner product is defined as follows: $\forall f, g \in \mathcal{F}, \langle f, g\rangle = \int f(x)g(x)dP(x)$ for some probability measure P on x. The corresponding norm is therefore $\|f\|^2_{L^2(P)} = \int f(x)^2 dP(x)$. Equipped with this function space, we adapt our sampling lemma as follows.

Lemma 29 (Maurey for signed measure Pisier (1981))**.** *Let μ denote a nonzero signed measure supported on $S \subseteq \mathbb{R}^p$, and $g(\mathbf{x}) = \int g(\mathbf{x}, \boldsymbol{\omega})d\mu(\boldsymbol{\omega})$. Let $\tilde{\boldsymbol{\omega}}_1, \dots, \tilde{\boldsymbol{\omega}}_k$ be i.i.d. draws from a scaled version $\tilde{\mu}$ of μ and let P be a probability measure on x. Define $\tilde{g}$ such that $g = \mathbb{E}_{\tilde{\mu}}\tilde{g}$. Then*

$$\begin{aligned}\mathbb{E}_{\tilde{\boldsymbol{\omega}}_1,\dots\tilde{\boldsymbol{\omega}}_k} \left\| g(\cdot) - \frac{1}{k}\sum_{i=1}^{k} \tilde{g}(\cdot, \tilde{\boldsymbol{\omega}}_i) \right\|^2_{L^2(P)} &\leq \frac{\mathbb{E}\|\tilde{g}(\cdot, \tilde{\boldsymbol{\omega}})\|^2_{L^2(P)}}{k} \\ &\leq \frac{\|\mu\|_1^2 \sup_{\boldsymbol{\omega}\in S}\|g(\cdot, \boldsymbol{\omega})\|^2_{L^2(P)}}{k}.\end{aligned}$$

Moreover there exist $(\boldsymbol{\omega}_1, \dots \boldsymbol{\omega}_k)$ and $s \in \{\pm 1\}^k$ in S so that

$$\left\| g(\cdot) - \frac{1}{k}\sum_{i=1}^{k} \tilde{g}(\cdot, \boldsymbol{\omega}_i, s_i) \right\|^2_{L^2(P)} \leq \mathbb{E}_{\tilde{\boldsymbol{\omega}}_1,\dots\tilde{\boldsymbol{\omega}}_k} \left\| g(\cdot) - \frac{1}{k}\sum_{i=1}^{k} \tilde{g}(\cdot, \tilde{\boldsymbol{\omega}}_i) \right\|^2_{L^2(P)}$$

Proof sketch. The proof applies the regular Maurey's lemma to $\widetilde{\mu}$ and Hilbert space $L^2(P)$ where $V := \tilde{g}$ and $g = \mathbb{E}[V]$. □

Next, we will leverage the result of Lemma 29 to prove Theorem 27.

Proof sketch of Theorem 27. The general idea is to convert the infinite-size construction introduced in Theorem 26 to a finite-size one. To do so, we sample from the integral $\int \sigma(\boldsymbol{\omega}^\top \mathbf{x}) p(\boldsymbol{\omega}) d\boldsymbol{\omega}$ by using a finite estimate $\sum_{j=1}^m s_j \tilde{\sigma}(\boldsymbol{\omega}_j^\top \mathbf{x})$ with $\tilde{\sigma}(z) = \sigma(z) \int |p(\boldsymbol{\omega})| d\boldsymbol{\omega}$ and where we sample $\boldsymbol{\omega}_j \sim \frac{|p(\boldsymbol{\omega})|}{\int |p(\boldsymbol{\omega})| d\boldsymbol{\omega}}$ and take $s_j = \text{sign}(g(\boldsymbol{\omega}_j))$. One can check that the expectation of this estimate is equal to the original function.

We will give a proof sketch for the case where σ is a threshold node, i.e. $z \mapsto \mathbf{1}[z \geq 0]$ and where $\|\mathbf{x}\| \leq 1$.

In order to apply the result of Lemma 29, we first have to deal with the fact that the distribution of the weights that we used is not a valid probability measure. Concretely, consider the function $h(x) = \sin(2\pi x)$ for $x \in [0, 1]$. By the fundamental theorem of calculus, we have

$$\begin{aligned} h(x) = h(0) + \int_0^x h'(b) db &= 0 + \int_0^1 \mathbf{1}[x \geq b] h'(b) db \\ &= \int_0^1 \mathbf{1}[x \geq b] 2\pi \cos(2\pi b) db. \end{aligned}$$

There are two problems that show that the above expression is not a valid probability measure: 1) $\cos(2\pi b)$ takes both positive and negative values and 2) $\int_0^1 \mathbf{1}[x \geq b] 2\pi \cos(2\pi b) db \neq 1$.

Let's consider a generalized form of a shallow network as $g(\mathbf{x}) = \int g(\mathbf{x}, \boldsymbol{\omega}) d\mu(\boldsymbol{\omega})$ where μ is a nonzero signed measure over some abstract parameter space in $\mathbb{R}^d$. As a concrete example, consider $\boldsymbol{\omega} = (a, \mathbf{b}, \mathbf{v})$ and $g(\mathbf{x}, \boldsymbol{\omega}) = a\sigma(\mathbf{v}^\top \mathbf{x} + \mathbf{b})$ where $\mathbf{a} \in \mathbb{R}$, $\mathbf{b}, \mathbf{v} \in \mathbb{R}^d$. In order to solve the two problems mentioned above, we will modify the weight measure as follows:

- Jordan decomposition: we decompose $\mu = \mu_+ - \mu_-$ into non-negative measures $\mu_\pm$ with disjoint support.
- For nonnegative measures, define total mass $\|\mu_\pm\|_1 = \mu_\pm(\mathbb{R}^d)$. Since the measures are disjoint, we have $\|\mu\|_1 = \|\mu_+\|_1 + \|\mu_-\|_1$.
- Define $\widetilde{\mu}$ to sample $s \in \{\pm 1\}$ with $\Pr[s = \pm 1] = \frac{\|\mu_\pm\|_1}{\|\mu\|_1}$ and then sample $g \sim \frac{\mu_s}{\|\mu_s\|_1} =: \tilde{\mu}_s$, and output $\tilde{g}(\cdot, \boldsymbol{\omega}, s) = s\|\mu\|_1 g(\cdot, \boldsymbol{\omega})$

The above sampling procedure yields:

$$\begin{aligned}\int g(\mathbf{x},\boldsymbol{\omega})d\mu(\boldsymbol{\omega}) &= \int g(\mathbf{x},\boldsymbol{\omega})d\mu_+(\boldsymbol{\omega}) - \int g(\mathbf{x},\boldsymbol{\omega})d\mu_-(\boldsymbol{\omega}) \\ &= \|\mu_+\|_1 \mathbb{E}_{\tilde{\mu}_+} g(\mathbf{x},\boldsymbol{\omega}) + \|\mu_-\|_1 \mathbb{E}_{\tilde{\mu}_-} g(\mathbf{x},\boldsymbol{\omega}) \\ &= \|\mu\|_1 \left[\Pr_{\tilde{\mu}}(s=+1)\mathbb{E}_{\tilde{\mu}_+} g(\mathbf{x},\boldsymbol{\omega}) - \Pr_{\tilde{\mu}}(s=-1)\mathbb{E}_{\tilde{\mu}_-} g(\mathbf{x},\boldsymbol{\omega}) \right] \\ &= \mathbb{E}_{\tilde{\mu}_s} \tilde{g}(\mathbf{x},\boldsymbol{\omega},s).\end{aligned}$$

We see that the neural network function can be written as an expectation over the measure $\tilde{\mu}$.

Finally, by Maurey's lemma, there exist $(\boldsymbol{\omega}_i, b_i, s_i)$ such that, for any probability measure P with support on $\|\mathbf{x}\| \leq 1$,

$$\left\| g(\cdot) - \frac{1}{k}\sum_{i=1}^{k} \tilde{g}(\cdot,\boldsymbol{\omega}_i,s_i) \right\|^2_{L^2(P)} \leq \mathcal{O}\left(\frac{1}{k}\right).$$

□

Remark 3. *Let's emphasize a few important points raised by Theorem 27:*

1. *We have complete freedom in the choice of measure (data distribution)*
2. *There is no dependency on d, thus MLPs* do not suffer from the curse of dimensionality *when approximating regular functions. This means that we can beat the curse of dimensionality by assuming that the function is smooth enough.*
3. *The approximation error rate $\propto 1/m$ is rather remarkable.*

Regarding the last two points, it is worth noting that a linear combination of m basis functions has a much worse approximation error bound proportional to $(1/m)^{2/d}$ (see Theorem 6 in Barron (1993)). The adaptivity of the feature extraction is key to avoiding the curse of dimensionality.

Remark 4 (Approximation of piecewise functions)**.** *We note that the results from Barron require some assumptions on σ, which are for instance satisfied by the sigmoid and ReLU functions. However, all functions in the Barron class are C^1, thus piecewise affine functions are not in the Barron class. Nevertheless, they can still be approximated by a neural network with ReLU activations.*

Readers interested in recent developments in this area are invited to read Bach (2017); Weinan et al. (2019).

4.2 Benefits of Depth

Modern neural networks use multiple layers and consistent empirical evidence has in fact shown that deeper networks yield better approximations. Do deep neural networks really offer representational benefits? At this stage, there is no comprehensive theory of why deeper models are preferred and when. Yet, there are several clues and we will next see two interesting pieces of the puzzle. First, we will discuss a result by Telgarsky (2016) that shows a separation between constant-width deep networks and subexponential width shallow networks. Next, we will see a constructed example of a function that is much easier to approximate with two hidden layers instead of a single layer.

4.2.1 Separation between Shallow and Deep Networks

Theorem 30 (Telgarsky (2016))**.** *Let any integer $L \geq 1$ be given. There exists a ReLU neural network $f : \mathbb{R} \to \mathbb{R}$ with $3L^2 + 6$ nodes and $2L^2 + 4$ layers that can not be approximated by any ReLU network g with $\leq 2^L$ nodes and $\leq L$ layers such that*

$$\int_{[0,1]} |f(x) - g(x)| \, dx \geq \frac{1}{32}.$$

In order to prove this theorem, we will define a notion of complexity that depends on the number of oscillations in the function implemented by the neural network and show that this complexity measure grows polynomially in width, but exponentially in depth.

To measure the oscillations in a function, we simply count the number of affine pieces. Formally, let $\mathcal{F}$ be the set of piecewise univariate linear mappings on $\mathbb{R}$. Given a function $f \in \mathcal{F}$, we denote by $\delta_A(f)$ the number of affine pieces of f.

In order to study the number of oscillations in a neural network function, we will need the following properties of $\delta_A(f)$.

Proposition 31. *1. For all $f \in \mathcal{F}$, and $\sigma(x) = \max(0, x)$, $\delta_A(\sigma(f)) \leq 2\delta_A(f)$,*

2. For all $f_1, \ldots f_m \in \mathcal{F}$, $\delta_A\left(\sum_{i=1}^m f_i\right) \leq \sum_{i=1}^m \delta_A(f_i)$.

Proof. i) Any linear piece of f that does not cross the 0-axis either stays a linear piece (if above the 0-axis) or is mapped to zero. Only pieces that cross the 0-axis are separated into two pieces. See illustration in Figure 4.1.
ii) When summing two piecewize functions $f(x) + g(x)$, there can only be a change in the slope at x if one of the functions f or g also had a change of slope at x. See illustration in Figure 4.2. □

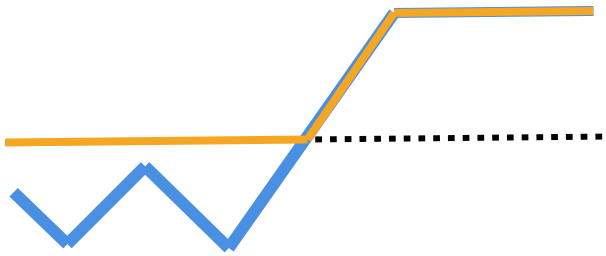


Figure 4.1: Illustration showing the effect of applying a ReLU function on a piecewize linear function. The dotted black line is the 0-axis. The piecewize linear function is shown in blue and the output of the ReLU function is shown in orange.

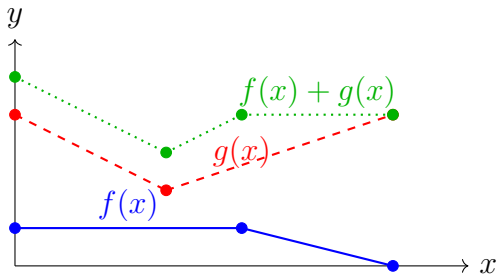


Figure 4.2: Illustration showing the effect of adding piecewise linear functions.

Lemma 32. *Let $f : \mathbb{R} \to \mathbb{R}$ be a ReLU network with L layers of widths $(m_1, \ldots, m_L)$ such that $m = \sum_{i=1}^{L} m_i$. Let $g : \mathbb{R} \to \mathbb{R}$ denote the output of some node in layer i as a function of the input. Then the number of affine pieces $\delta_A(g)$ satisfies*

$$\delta_A(g) \leq 2^i \prod_{j<i} m_j.$$

The number of affine pieces in f satisfies $\delta_A(f) \leq \left(\frac{2m}{L}\right)^L$.

Proof. See the exercise session. □

We are now ready to prove Theorem 30. We will create a highly oscillatory function f which we will approximate with a function g with few oscillations. In order to create the function f, we will compose a function with itself such that the result of the composition increases its complexity (measured by the number of pieces as discussed earlier).

Let $\sigma_{\mathrm{r}}(x) = \max(0, x)$ and consider the following Δ function:

$$\Delta(x) = 2\sigma_{\mathrm{r}}(x) - 4\sigma_{\mathrm{r}}(x - 1/2) + 2\sigma_{\mathrm{r}}(x - 1) = \begin{cases} 2x & x \in [0, 1/2), \\ 2 - 2x & x \in [1/2, 1), \\ 0 & \text{otherwise.} \end{cases}$$

This function is shown in Figure 4.3 (left side) below.

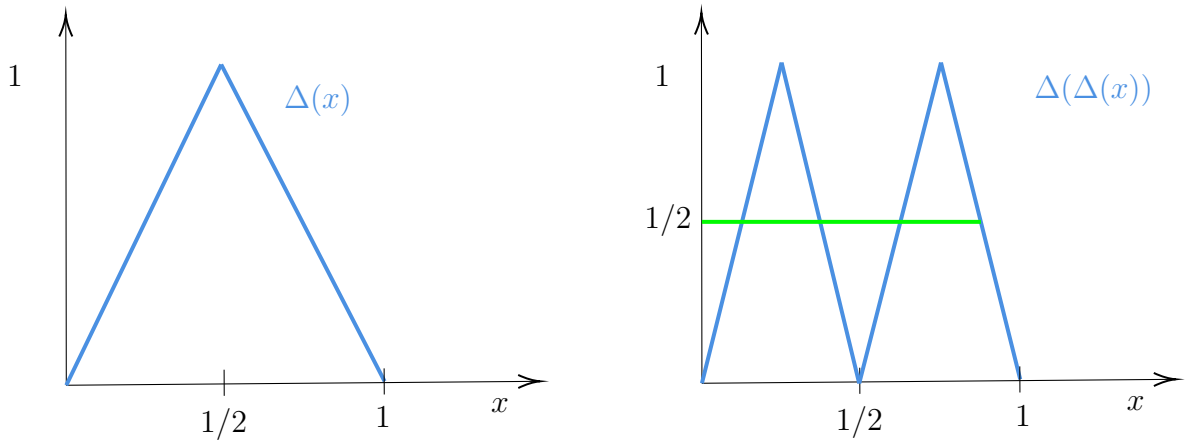


Figure 4.3: Illustration of the $\Delta(x)$ function as well as $\Delta(\Delta(x))$.

Try to compose the function with itself, $\Delta \circ \Delta$, what does the resulting function look like? If you repeat this composition, you will see that Δ^L has 2^{L-1} copies of itself (see Figure 4.3, right side). We will take our oscillatory function f to be $f(x) = \Delta^{L^2+2}(x)$.

Proof of Theorem 30. Consider the highly oscillatory blue function $f(x) = \Delta^{L^2+2}(x)$ shown in Figure 4.4. There are 2^{L^2+1} copies of Δ. Draw the line $x \mapsto \frac{1}{2}$, then half-triangles are formed by intersecting this line with the function f, which translates to $2^{L^2+2} - 1$ half-triangles since we get two half-triangles for each Δ but one is lost on the boundary of $[0, 1]$. The length of the base of each triangle is equal to $\frac{1}{2^{L^2+2}}$ and therefore the area of each triangle is equal to $\frac{1}{4} \cdot \frac{1}{2^{L^2+2}} = 2^{-L^2-4}$.

We want to approximate it with a function $g \in \mathcal{F}$ which has few oscillations (see the red function in Figure 4.4). We will keep track of when g passes above and below the line $x = \frac{1}{2}$; when it is above, we will count the triangles below; when it is above, we will count the triangles below. The function $g \in \mathcal{F}$ crosses the axis $x = \frac{1}{2}$ at most $\delta_A(g)$ times. Therefore, the number of half-triangles on one side of the $x = \frac{1}{2}$ axis is larger than $2^{L^2+2} - 1 - \delta_A(g)$. By Lemma 32, we have $\delta_A(g) \leq (2 \cdot 2^L/L)^L \leq 2^{L^2}$. We obtain the following bound

$$\begin{aligned}
\int_{[0,1]} |f(x) - g(x)| dx &\geq [\text{number surviving triangles}] \cdot [\text{area of triangle}] \\
&\geq \frac{1}{2}\left[2^{L^2+2} - 1 - 2 \cdot 2^{L^2}\right] \cdot \left[2^{-L^2-4}\right] \\
&= \frac{1}{2}\left[2^{L^2+1} - 1\right] \cdot \left[2^{-L^2-4}\right] \\
&\geq \frac{1}{32}.
\end{aligned}$$

□

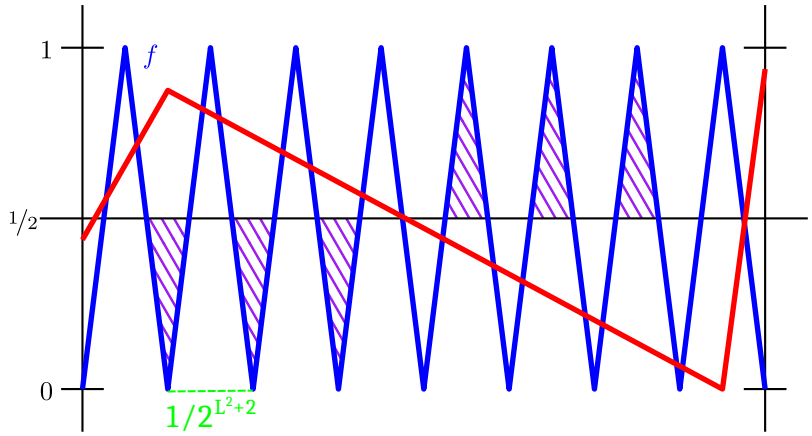


Figure 4.4: The target function $f(x) = \Delta^{L^2+2}(x)$ is shown in blue, while the approximate function $g(\mathbf{x})$ is shown in red. We approximate the error between f and g by counting the number of half-triangles (shown with purple lines) that are created by intersecting the line $x = \frac{1}{2}$ with f. *Source: Telgarsky (2021) (slightly modified).*

4.2.2 Constructed Example: when two Layers are Better than One

The key idea is very simple and can be summarized as follows:

- Define a target radial function $g : \mathbb{R}^n \to \mathbb{R}$ as $g(\mathbf{x}) = \psi(\|\mathbf{x}\|)$ (i.e. a function that only depends on the input norm)

- ... that can be naturally approximated by first approximating the norm (via the span of the first hidden layer) and then approximating ψ (via the span of the second hidden layer).

- ... assuming that the norm can be approximated in an n-efficient manner and that this is not true for g.

The following main result is from Eldan and Shamir (2016). It states that a neural network with two hidden layers and a polynomial number of units in the dimension n can not be well approximated by a neural network with a single hidden layer with less than an exponential number of units.

Theorem 33 (Eldan and Shamir (2016)). *For $n \geq C$ there exists a probability measure μ with density ϕ^2 and a function $g : \mathbb{R}^n \to \mathbb{R}$ with the following properties:*

1. *g is bounded in $[-2; 2]$ supported on $\{\mathbf{x} : \|\mathbf{x}\| \leq C\sqrt{n}\}$ and expressible by a 2 hidden layer network with width $Ccn^{19/4}$.*
2. *Every function $f : \mathbb{R}^n \to \mathbb{R}$ implemented by a one-hidden layer network with width $m \leq ce^{cn}$ satisfies*

$$\mathbb{E}_{\mathbf{x}\sim\mu} \left(f(\mathbf{x}) - g(\mathbf{x})\right)^2 \geq c.$$

Proof idea. The proof of this result is complex. We will therefore focus on the main ingredients and omit the technical details that make the proof rigorous.

Step I: move to Fourier space

We are interested in the L_2 loss between f and a target g with regard to density ϕ^2 (i.e. $\int \phi^2(\mathbf{x})d\mathbf{x} = 1$). Let $\widehat{h}$ denote the (generalized) Fourier transforms of h. Then, we have

$$\begin{aligned}
\ell^\phi(f, g) :=& \int (f(\mathbf{x}) - g(\mathbf{x}))^2 \phi^2(\mathbf{x}) d\mathbf{x} \\
=& \int (f(\mathbf{x})\phi(\mathbf{x}) - g(\mathbf{x})\phi(\mathbf{x}))^2 d\mathbf{x} \\
=& \|f\phi - g\phi\|_{L^2}^2 \\
\overset{(i)}{=}& \|\widehat{f\phi} - \widehat{g\phi}\|_{L^2}^2 \\
\overset{(ii)}{=}& \|\widehat{f} \star \widehat{\phi} - \widehat{g} \star \widehat{\phi}\|_{L^2}^2
\end{aligned}$$

where step (i) is due to Parseval identity and step (ii) is due to the convolution theorem (Theorem 25).

Our next goal will be to choose ϕ and g to separate 1 vs. 2-hidden layer MLPs.

Step II: Spotting the Weakness

We are interested in the support of $\widehat{f} \star \widehat{\phi}$ as it gives us an idea of the type of functions we can approximate (in the sense that one can only approximate a function with another function if their support overlaps). We will choose ϕ such that $\widehat{\phi} = \mathbf{1}[\mathbb{B}^n]$ (i.e. indicator on the unit ball $\mathbb{B}^n$). The resulting function ϕ is shown in Figure 4.5. As a consequence of our choice of $\widehat{\phi}$, ϕ is isotropic and band-limited.

Let's consider what happens to the support of a neural network function f when convolved with $\widehat{\phi}$:

- For a single ridge function: $\sigma(\mathbf{x}) = \sigma(\mathbf{x} \cdot \boldsymbol{\theta})$

$$\Rightarrow \boxed{\text{supp}\left(\widehat{\sigma}\right) = \text{span}\{\boldsymbol{\theta}\}}$$

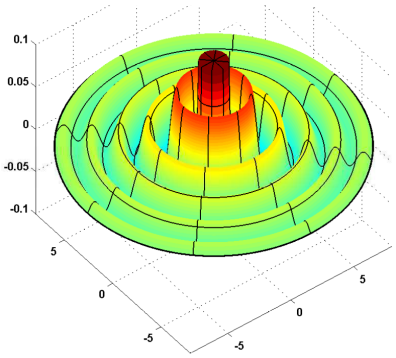


Figure 4.5: Illustration of the $\phi(\mathbf{x})$ function, which is a rescaled version of $\text{sinc}(\|\mathbf{x}\|) = \frac{\sin(\|\mathbf{x}\|)}{\|\mathbf{x}\|}$.

- Convolved ridge function:

$$\Rightarrow \boxed{\text{supp}\left(\widehat{\sigma} \star \widehat{\phi}\right) = \text{span}\{\boldsymbol{\theta}\} + \mathbb{B}^n}$$

The convolution increases the support as shown in Figure 4.6.

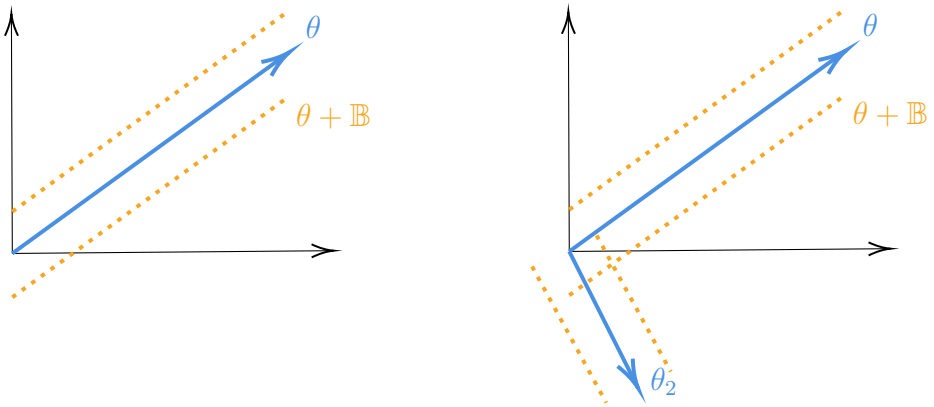


Figure 4.6: Left: Illustration of $\text{supp}\left(\widehat{\sigma} \star \widehat{\phi}\right)$ as the space between the blue arrow and the dotter oranges lines; Right: by adding a second vector θ_2, we increase the support.

- MLP with one hidden layer of width m implements a function

$$f \in \text{span}\{\sigma_j(\mathbf{x}) := \sigma(\boldsymbol{\theta}_j \cdot \mathbf{x}),\ 1 \leq j \leq m\} \subset \text{span}(\mathcal{G}_\sigma^n)$$

Linear combination of convolved ridge functions:

$$\Rightarrow \boxed{\text{supp}(\widehat{f} \star \widehat{\phi}) = \bigcup_{j=1}^{m} (\text{span}\{\boldsymbol{\theta}_j\} + \mathbb{B}^n)}$$

Step III: Covering the space and Curse of Dimensionality

As illustrated in Figure 4.6, the frequency components of $\widehat{f} \star \widehat{\phi}$ for $f \in \text{span}(\mathcal{G}_\sigma^n)$ have a peculiar structure: they are a union of unit width tubes.

We need to ensure that we have full frequency support, i.e. that the width m of f is large enough such that $\text{supp}(\widehat{f} \star \widehat{\phi}) \supseteq r\mathbb{B}$ as r grows. Else, if $\text{supp}(\widehat{f} \star \widehat{\phi}) \not\supseteq r\mathbb{B}$ $\implies \exists\, \boldsymbol{\omega} \in r\mathbb{B}$ representing oscillations that $\widehat{f} \star \widehat{\phi}$ cannot capture.

In fact, one can show the following volume ratio formula as $n \to \infty$

$$\frac{\mathbb{V}\left(\text{supp}(\widehat{f} \star \widehat{\phi}) \cap r\mathbb{B}\right)}{\mathbb{V}(r\mathbb{B})} \lessapprox me^{-n}$$

Note the exponential of n on the RHS, this is the so-called curse of dimensionality, i.e. it becomes more difficult to cover the space as the dimension increases.

Step IV: Designing the Target

As a target function g, we chose a radial function $g = \psi \circ \|\cdot\|$ as random sign indicator functions of thin shells. To construct this function, we assume that $\|\mathbf{x}\| \leq R$ and chose an N-partition $\{\Delta_i\}$ of $[0; R]$. Then we define

$$\psi(z) = \sum_{i=1}^{N} \epsilon_i \psi_i(z), \quad \psi_i(z) = \mathbf{1}\{\Delta_i\}, \quad \epsilon_i \in \{-1, 1\} \tag{4.5}$$

This target function is illustrated in Figure 4.7, it is highly oscillatory which again makes it difficult to approximate.

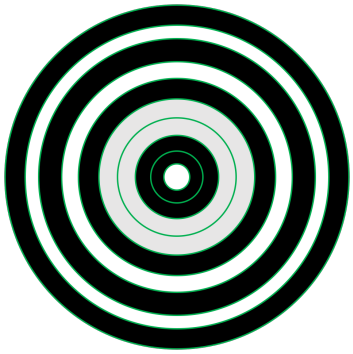


Figure 4.7: The target function is a highly oscillatory function with flipping signs.

□

4.3 Appendix: Proof Convolution Theorem

To prove the Convolution Theorem, we need to show two main results:

1. $\widehat{r}(\boldsymbol{\omega}) = \widehat{g}(\boldsymbol{\omega})\widehat{h}(\boldsymbol{\omega})$

2. $r(\mathbf{x}) = \mathcal{F}^{-1}(\widehat{g}(\boldsymbol{\omega})\widehat{h}(\boldsymbol{\omega}))$

Proof of $\widehat{r}(\boldsymbol{\omega}) = \widehat{g}(\boldsymbol{\omega})\widehat{h}(\boldsymbol{\omega})$:
The Fourier transform of $r(\mathbf{x}) = (g * h)(\mathbf{x})$ is defined as:

$$\widehat{r}(\boldsymbol{\omega}) = \mathcal{F}\{r(\mathbf{x})\} = \int_{-\infty}^{\infty} r(\mathbf{x}) e^{-2\pi i \boldsymbol{\omega}^\top \mathbf{x}} \, d\mathbf{x}$$

Substitute $r(\mathbf{x}) = \int_{-\infty}^{\infty} g(\boldsymbol{\tau}) h(\mathbf{x} - \boldsymbol{\tau}) \, d\boldsymbol{\tau}$ into the above equation:

$$\widehat{r}(\boldsymbol{\omega}) = \int_{-\infty}^{\infty} \left(\int_{-\infty}^{\infty} g(\boldsymbol{\tau}) h(\mathbf{x} - \boldsymbol{\tau}) \, d\boldsymbol{\tau} \right) e^{-2\pi i \boldsymbol{\omega}^\top \mathbf{x}} \, d\mathbf{x}$$

We can change the order of integration (Fubini's theorem):

$$\widehat{r}(\boldsymbol{\omega}) = \int_{-\infty}^{\infty} g(\boldsymbol{\tau}) \left(\int_{-\infty}^{\infty} h(\mathbf{x} - \boldsymbol{\tau}) e^{-2\pi i \boldsymbol{\omega}^\top \mathbf{x}} \, d\mathbf{x} \right) d\boldsymbol{\tau}$$

Let $\mathbf{u} = \mathbf{x} - \boldsymbol{\tau}$. Then, $d\mathbf{x} = d\mathbf{u}$, and we have:

$$\int_{-\infty}^{\infty} h(\mathbf{x} - \boldsymbol{\tau}) e^{-2\pi i \boldsymbol{\omega}^\top \mathbf{x}} \, d\mathbf{x} = \int_{-\infty}^{\infty} h(\mathbf{u}) e^{-2\pi i \boldsymbol{\omega}^\top (\mathbf{u} + \boldsymbol{\tau})} \, d\mathbf{u}$$

This simplifies to:

$$\int_{-\infty}^{\infty} h(\mathbf{u}) e^{-2\pi i \boldsymbol{\omega}^\top \mathbf{u}} e^{-2\pi i \boldsymbol{\omega}^\top \boldsymbol{\tau}} \, d\mathbf{u} = e^{-2\pi i \boldsymbol{\omega}^\top \boldsymbol{\tau}} \int_{-\infty}^{\infty} h(\mathbf{u}) e^{-2\pi i \boldsymbol{\omega}^\top \mathbf{u}} \, d\mathbf{u}$$

Recognizing the integral as the Fourier transform of $h(\mathbf{u})$:

$$\int_{-\infty}^{\infty} h(\mathbf{u}) e^{-2\pi i \boldsymbol{\omega}^\top \mathbf{u}} \, d\mathbf{u} = \widehat{h}(\boldsymbol{\omega})$$

Thus:

$$\int_{-\infty}^{\infty} h(\mathbf{x} - \boldsymbol{\tau}) e^{-2\pi i \boldsymbol{\omega}^\top \mathbf{x}} \, d\mathbf{x} = e^{-2\pi i \boldsymbol{\omega}^\top \boldsymbol{\tau}} \widehat{h}(\boldsymbol{\omega})$$

Substitute back into the integral for $\widehat{r}(\boldsymbol{\omega})$:

$$\widehat{r}(\boldsymbol{\omega}) = \int_{-\infty}^{\infty} g(\boldsymbol{\tau}) e^{-2\pi i \boldsymbol{\omega}^\top \boldsymbol{\tau}} \widehat{h}(\boldsymbol{\omega}) \, d\boldsymbol{\tau}$$

Recognizing the integral as the Fourier transform of $g(\boldsymbol{\tau})$:

$$\int_{-\infty}^{\infty} g(\boldsymbol{\tau}) e^{-2\pi i \boldsymbol{\omega}^\top \boldsymbol{\tau}} \, d\boldsymbol{\tau} = \widehat{g}(\boldsymbol{\omega})$$

Thus:

$$\widehat{r}(\boldsymbol{\omega}) = \widehat{g}(\boldsymbol{\omega})\widehat{h}(\boldsymbol{\omega})$$

Proof of $r(\mathbf{x}) = \mathcal{F}^{-1}(\widehat{g}(\boldsymbol{\omega})\widehat{h}(\boldsymbol{\omega}))$:
The inverse Fourier transform of $\widehat{r}(\boldsymbol{\omega})$ is defined as:

$$r(\mathbf{x}) = \mathcal{F}^{-1}\{\widehat{r}(\boldsymbol{\omega})\} = \int_{-\infty}^{\infty} \widehat{r}(\boldsymbol{\omega}) e^{2\pi i \boldsymbol{\omega}^\top \mathbf{x}} \, d\boldsymbol{\omega}$$

Since $\widehat{r}(\boldsymbol{\omega}) = \widehat{g}(\boldsymbol{\omega})\widehat{h}(\boldsymbol{\omega})$, we substitute this into the inverse Fourier transform:

$$r(\mathbf{x}) = \int_{-\infty}^{\infty} \widehat{g}(\boldsymbol{\omega})\widehat{h}(\boldsymbol{\omega}) e^{2\pi i \boldsymbol{\omega}^\top \mathbf{x}} \, d\boldsymbol{\omega}$$

We know that the product of two Fourier transforms corresponds to the convolution of their respective functions in the spatial domain. Thus:

$$r(\mathbf{x}) = \mathcal{F}^{-1}\{\widehat{g}(\boldsymbol{\omega})\widehat{h}(\boldsymbol{\omega})\} = g(\mathbf{x}) * h(\mathbf{x})$$

This completes the proof, showing that $r(\mathbf{x}) = g(\mathbf{x}) * h(\mathbf{x})$ and $\widehat{r}(\boldsymbol{\omega}) = \widehat{g}(\boldsymbol{\omega})\widehat{h}(\boldsymbol{\omega})$.

4.4 Exercise: Complexity Theory

Problem 1 (Barron norm of Gaussian measure):

Consider the density function of a Gaussian random variable, $f(x) = (2\pi\sigma^2)^{d/2}\exp(-\frac{\|x\|^2}{2\sigma^2})$. We will calculate the complexity measure $\int \left\|\widehat{\nabla f}(\boldsymbol{\omega})\right\| d\boldsymbol{\omega}$ that appears in Barron's theorem discussed in class.

a) Use the following facts: (Why are they true?)

- For f defined as above, we have $|\hat{f}(\boldsymbol{\omega})| = \exp(-2\pi^2\sigma^2\|\boldsymbol{\omega}\|^2)$;
- (Cauchy-Schwartz inequality in expected value) $\mathbb{E}[\|X\|] \leq \left(\mathbb{E}\left[\|X\|^2\right]\right)^{\frac{1}{2}}$ where X is a random vector.

Show that $\int \|\boldsymbol{\omega}\||\hat{f}(\boldsymbol{\omega})|d\boldsymbol{\omega} \leq Z\left(\int \|\boldsymbol{\omega}\|^2 Z^{-1}\hat{f}(\boldsymbol{\omega})d\boldsymbol{\omega}\right)^{1/2}$ where $Z = (2\pi\sigma^2)^{-d/2}$.

b) Using the fact that $\int g(\boldsymbol{\omega})Z^{-1}\hat{f}(\boldsymbol{\omega})d\boldsymbol{\omega}$ is the expectation of the function $g(\boldsymbol{\omega})$ with respect to the density $\mathcal{N}\left(0, \frac{1}{4\pi^2\sigma^2}\right)$, show that $\int \|\boldsymbol{\omega}\||\hat{f}(\boldsymbol{\omega})|d\boldsymbol{\omega} \leq \sqrt{\frac{d}{4\pi^2\sigma^2}} \cdot (2\pi\sigma^2)^{-d/2}$.

c) Since $\int \left\|\widehat{\nabla f}(\boldsymbol{\omega})\right\| d\boldsymbol{\omega} = 2\pi \int \|\boldsymbol{\omega}\| \cdot |\hat{f}(\boldsymbol{\omega})|d\boldsymbol{\omega}$, what do you conclude about the complexity measure $\int \left\|\widehat{\nabla f}(\boldsymbol{\omega})\right\| d\boldsymbol{\omega}$ when d is very large?

Problem 2 (Maurey's lemma):

Let $X = \mathbb{E}V$ be given, with V a random vector supported on a subset S of the event space, and let $V_1, \ldots V_k$ be i.i.d. realization of V.

a) Show that

$$\mathbb{E}_{V_i}\left\|X - \frac{1}{k}\sum_{i=1}^{k} V_i\right\|^2 = \frac{1}{k}\mathbb{E}_V\|V - X\|^2.$$

b) Show that

$$\frac{1}{k}\mathbb{E}_V\|V - X\|^2 \leq \frac{\sup_{U\in S}\|U\|^2}{k}.$$

Problem 3 (Number of affine pieces in a ReLU network):

We will prove the lemma that bounds the number of affine pieces in a ReLU network stated in the lecture notes. Denote by $\delta_A(f)$ be the (minimum) number of affine pieces of a piece-wisely linear continuous function $f : \mathbb{R} \to \mathbb{R}$. Show that

a) $\delta_A(\sum_k a_k f_k + b_k) \leq \sum_k \delta_A(f_k)$, for any finite sequence of piece-wisely linear continuous functions f_k, and for any real sequences a_k and b_k.

b) Let $g_i : \mathbb{R} \to \mathbb{R}$ denote the output of some node in layer i in a ReLU network with L layers of widths $(m_1, \ldots, m_L)$ as a function of the input. Using induction on i, show that the number of affine pieces $\delta_A(g_i)$ satisfies

$$\delta_A(g_i) \leq 2^i \prod_{j<i} m_j.$$

c) Recall that $g_{L+1} : \mathbb{R} \to \mathbb{R}$ is a ReLU network with L layers of widths $(m_1, \ldots, m_L)$ such that $m = \sum_{i=1}^{L} m_i$. The number of affine pieces in g_{L+1} satisfies

$$\delta_A(g_{L+1}) \leq \left(\frac{2m}{L}\right)^L.$$

d) What value of L maximizes the upper bound $\left(\frac{2m}{L}\right)^L$? Although taking this value of L gives a large value of $\delta_A(g_{L+1})$, give a reason why in practice we usually pick a value much smaller than this.

Chapter 5

Optimization

In the last two lectures, we have seen that neural networks are universal approximators, i.e. they can learn arbitrarily complex functions under some assumptions. However, we so far ignore the computational aspect required to learn such functions. This is where optimization comes into play. It allows us to train the parameters of a deep neural network by minimizing a loss function that is typically chosen by a practitioner. Given a target/ground-truth y and a prediction ν (for instance the output a neural network), some commonly encountered loss functions are:

- Zero-one loss

$$\ell_y(\nu) = \begin{cases} 0, & \text{if } \nu = y \\ 1, & \text{else} \end{cases}$$

  This loss is however not smooth or differentiable everywhere. Other surrogate losses are typically used instead.

- Squared loss:

$$\ell_{\mathbf{y}}(\boldsymbol{\nu}) = \frac{1}{2}\|\mathbf{y} - \boldsymbol{\nu}\|^2$$

- Log-loss (multiclass)

$$\ell_y(\nu) = -\log \nu_y$$

- Soft target cross-entropy ($\mathbf{y} \in [0;1]^m$)

$$\ell_{\mathbf{y}}(\boldsymbol{\nu}) = -\sum_{j=1}^{m} y_j \log \nu_j \geq -\sum_{j=1}^{m} y_j \log y_j =: H(\mathbf{y})$$

The loss functions discussed above are defined on single instances. In machine learning, our goal is to learn a model from a large number of datapoints. To do so, we take expectations over loss functions, which results in a so-called *risk function*. We will differentiate between two types of risk functions.

Given a set of $(\xi, \mathbf{y}) \in \mathbb{P}$, drawn from an unknown data generating distribution $\mathbb{P}$, we first define the expected risk as

$$\mathcal{R}(\mathbf{x}) = \mathbb{E}_{\mathbb{P}}[\ell(F(\xi; \mathbf{x})],$$

where F is an $\mathbf{x}$-parameterized model (e.g. a deep neural network). Ideally, we would like to find $\mathbf{x}^* = \text{argmin}_{\mathbf{x}} \mathcal{R}(\mathbf{x})$.

In practice, we only have access to a finite sample $\mathcal{S} = \{(\xi^i, \mathbf{y}^i)\}_{i=1}^n$, which gives rise to the following *empirical risk*:

$$\mathcal{R}_{\mathcal{S}}(\mathbf{x}) = \frac{1}{n} \sum_{i=1}^{n} \ell_{\mathbf{y}^i}(F(\xi^i; \mathbf{x})).$$

We minimize the empirical risk as a proxy for the expected risk, this is the so-called *empirical risk minimization*.

5.1 Optimality

Our goal will be to find $\mathbf{x}^*$ that minimizes the function $\mathcal{R}_{\mathcal{S}}(\mathbf{x})$. We will more generally be interested in optimizing a function $f : \mathbb{R}^d \to \mathbb{R}$ that exhibits certain properties. For instance, the class C^0 consists of all continuous functions. The class C^1 consists of all differentiable functions whose (first) derivative is continuous; such functions are called continuously differentiable. Another important class of functions are convex functions. We recall the definition of convexity below.

Definition 17 (Convex set)**.** *A set S is convex if $\forall \mathbf{x}, \mathbf{y} \in S$, $\lambda \in [0, 1]$,*

$$\lambda \mathbf{x} + (1 - \lambda)\mathbf{y} \in S. \tag{5.1}$$

Definition 18 (Convex function)**.** *A function $f : S \to \mathbb{R}$ is convex if its domain S is a convex set and if for any two points $\mathbf{x}, \mathbf{y} \in S$, the following property holds:*

$$f(\alpha \mathbf{x} + (1 - \alpha)\mathbf{y}) \leq \alpha f(\mathbf{x}) + (1 - \alpha) f(\mathbf{y}), \quad \forall \alpha \in [0, 1]. \tag{5.2}$$

Local vs Global minimum One typically distinguishes between global and local minimum. Of course, a global minimum is also a local minimum. For convex functions, the two notions are exactly the same (see exercise sheet). However, this is

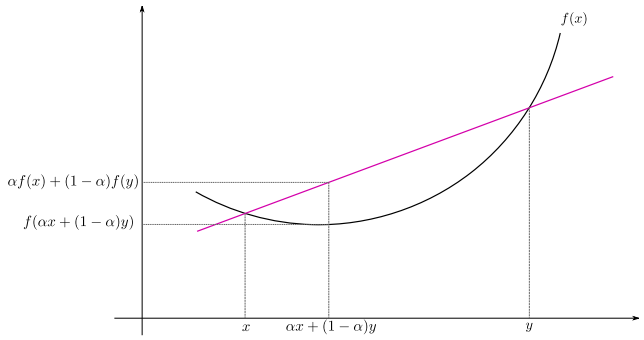


Figure 5.1: Convex function $f : \mathbb{R} \to \mathbb{R}$.

not the case in general for functions that are non-convex. As we will later see, the objective functions optimized when optimizing the parameters of deep networks are in general non-convex. This is due to the mapping function implemented by neural networks that typically involve non-linear activation functions. So even if we train with a convex loss such as least-square, the objective function is not convex with respect to the parameters. We will discuss this in more detail later on.

Descent direction Assume $f \in C^1, \mathbf{x} \in \mathbb{R}^d$ and $\mathbf{s} \in \mathbb{R}^d$ such that $\mathbf{s} \neq \mathbf{0}$. Then $\mathbf{s}$ is a descent direction if

$$\nabla f(\mathbf{x})^\top \mathbf{s} < 0 \implies f(\mathbf{x} + \alpha \mathbf{s}) < f(\mathbf{x}) \quad \forall \alpha \text{ sufficiently small} \tag{5.3}$$

The condition $\nabla f(\mathbf{x})^\top \mathbf{s} < 0$ also implies:

$$\cos\langle -\nabla f(\mathbf{x}), \mathbf{s} \rangle = \frac{(-\nabla f(\mathbf{x}))^\top \mathbf{s}}{\|\nabla f(\mathbf{x})\| \cdot \|\mathbf{s}\|} > 0. \tag{5.4}$$

One can easily verify that $\mathbf{s} = -\nabla f(\mathbf{x})$ is a descent direction and so is any $\mathbf{s}$ that does not deviate too much from $-\nabla f(\mathbf{x})$, i.e. $\langle -\nabla f(\mathbf{x}), \mathbf{s} \rangle \in [0, \pi/2)$.

First-order necessary conditions Assume $f \in C^1(\mathbb{R}^d)$. Then $\nabla f(\mathbf{x}) = 0 \iff \mathbf{x}$ stationary point of f. We also have that if $\mathbf{x}^*$ is a local minimizer of f, then $\nabla f(\mathbf{x}^*) = 0$.

Second-order necessary conditions Assume $f \in C^2(\mathbb{R}^d)$. Then $\mathbf{x}^*$ is a local minimizer of f implies $\nabla^2 f(\mathbf{x}^*)$ is positive semi-definite, i.e. $\mathbf{s}^\top \nabla^2 f(\mathbf{x}^*) \mathbf{s} \geq 0$ for all $\mathbf{s} \in \mathbb{R}^d$.

Example: $f(x) = x^3$. $x^* = 0$ is not a local minimizer but $f'(0) = f''(0) = 0$.

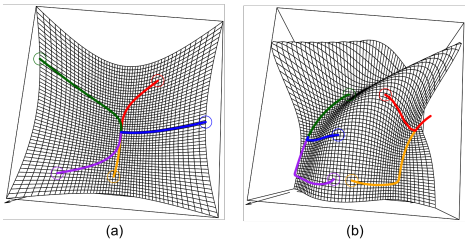


Figure 5.2: Trajectories of gradient flow initialized from different points for (a) a convex function, and (b) a non-convex function. We observe that all trajectories for (a) converge to the same point, while this is not the case for (b). Source: Lecture from Ryan Tibshirani.

Second-order sufficient conditions Assume $f \in C^2(\mathbb{R}^d)$. Then $\nabla f(\mathbf{x}^*) = 0$ and $\nabla^2 f(\mathbf{x}^*)$ is positive definite (i.e. $\mathbf{s}^\top \nabla^2 f(\mathbf{x}^*)\mathbf{s} > 0$ for all $\mathbf{s} \in \mathbb{R}^d$) implies that $\mathbf{x}^*$ is a local minimizer of f.

Note that the condition on the Hessian $\nabla^2 f$ is now more restrictive as it requires positive definiteness instead of positive semi-definiteness. The following example shows why this is not a necessary condition:
Example: $f(x) = x^4$. $x^* = 0$ is a local minimizer but $f''(0) = 0$.

5.2 Gradient Descent

Our goal is to minimize a differentiable function $f : \mathbb{R}^d \to \mathbb{R}$. A workhorse algorithm to optimize such functions is gradient descent that starts from some initial $\mathbf{x}_0$ and then iteratively updates the iterate $\mathbf{x}_k \in \mathbb{R}^d$ as follows,

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \eta \nabla f(\mathbf{x}_k),$$

where $\eta > 0$ is a chosen fixed step size. Larger step sizes are usually preferred from the standpoint of computational complexity and performance, but there is a limit to how large the step size can be. We will discuss this later on.

Gradient descent can be seen as the numerical integration of a continuous time dynamics described by the ordinary differential equation (ODE)

$$\dot{\mathbf{x}} = -\nabla f(\mathbf{x})\,.$$

This is referred to as gradient flow in the literature. Figure 5.2 shows the trajectory of gradient flow initialized from different points for (a) a convex function and (b) a non-convex function. In the convex case, all trajectories end up at a unique minimizer. This is however not the case for non-convex functions.

The trajectory of gradient flow can be approximated by gradient descent under appropriate conditions that depend on the step size η. We won't discuss this further in this lecture and will instead take it for granted that gradient flow can be used as a surrogate to study gradient descent.

5.3 Quadratic Model

Let's consider a local approximation of f formed by taking the second-order Taylor expansion of f as follows,

$$f(\mathbf{x} + \Delta\mathbf{x}) \approx f(\mathbf{x}) + \nabla f(\mathbf{x}) \cdot \Delta\mathbf{x} + \frac{1}{2}\Delta\mathbf{x}^\top \nabla^2 f(\mathbf{x}) \Delta\mathbf{x}$$

The minimizer of the right-hand side over $\Delta\mathbf{x}$ is:

$$\Delta\mathbf{x} = -[\nabla^2 f(\mathbf{x})]^{-1} \nabla f(\mathbf{x})$$

We recover the update of Newton's method. An obvious drawback is that this update requires computing and inverting the Hessian, which is expensive in high dimensions. Alternatively, we would like to choose a different matrix $\mathbf{B}$ that is close to the Hessian (i.e. $\|\nabla^2 f(\mathbf{x}) - \mathbf{B}\|_2 \leq \epsilon$) but with a lower computational complexity. What could it be? In some cases a diagonal approximation $\mathbf{B} = \text{Diag}(\nabla^2 f(\mathbf{x}))$ might work and would be cheaper to compute. Yet another alternative would be a block-diagonal approximation.

Recovering Gradient Descent: Note that if $\nabla^2 f(\mathbf{x}) = \mathbf{I}$, then

$$f(\mathbf{x} + \Delta\mathbf{x}) \approx f(\mathbf{x}) + \nabla f(\mathbf{x}) \cdot \Delta\mathbf{x} + \frac{1}{2\eta}\|\Delta\mathbf{x}\|^2$$

The minimizer is just $\Delta\mathbf{x} = -\eta \nabla f(\mathbf{x})$, which recovers the update of gradient descent. This gives us a different interpretation of gradient descent as optimizing a surrogate quadratic model. This is illustrated in Figure 5.3.

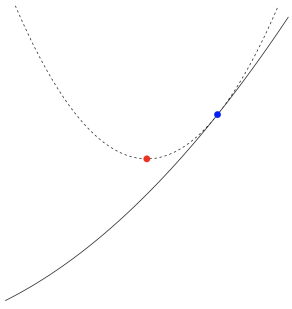


Figure 5.3: A function f is drawn as a plain line, along with its quadratic approximation as a dotted line computed at the blue point. The red point corresponds to the minimizer of the quadratic approximation.

5.4 Convergence for Quadratic Functions

We will start by analysing the gradient dynamics and its convergence properties on a simple convex quadratic objective $f : \mathbb{R}^d \to \mathbb{R}$ defined as

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\top \mathbf{Q}\,\mathbf{x} - \mathbf{q}^\top \mathbf{x}, \quad \mathbf{Q} \in \mathbb{R}^{d\times d} \text{ is positive definite,}$$

To simplify the problem, we diagonalize $\mathbf{Q} = \mathbf{U}\mathbf{\Lambda}\mathbf{U}^\top$ where $\mathbf{U}$ is an orthogonal matrix. We then perform a change of basis $\mathbf{x} \leftarrow \mathbf{U}^\top\mathbf{x}$, $\mathbf{q} \leftarrow \mathbf{U}^\top\mathbf{q}$, which yields an objective function that is separable over the coordinates of $\mathbf{x} = [x^{(1)}, \ldots, x^{(d)}]$:

$$f(\mathbf{x}) = \sum_{i=1}^{n} g_i(x^{(i)}), \quad g_i(z) = \frac{\lambda_i}{2}z^2 - q_i z, \quad \lambda_i > 0\,.$$

Since the problem is separable, we can consider a generic function $g := g_i : \mathbb{R} \to \mathbb{R}$, $g(z) = \frac{\lambda}{2}z^2 - qz$. The derivative is

$$g'(z) = \lambda z - q$$

From first-order optimality, we obtain the following minimizer:

$$z^* = q/\lambda, \quad g(z^*) = \frac{q^2}{2\lambda} - \frac{q^2}{\lambda} = -\frac{q^2}{2\lambda}$$

To simplify the analysis, we shift g as follows:

$$g(z) \leftarrow g(z) - g(z^*) = \frac{\lambda}{2}\left(z^2 - \frac{2q}{\lambda}z + \frac{q^2}{\lambda^2}\right) = \frac{1}{2\lambda}\left(\lambda z - q\right)^2.$$

A gradient step results in

$$\begin{aligned} g(z - \eta(\lambda z - q)) &= \frac{1}{2\lambda}\left((1-\lambda\eta)(\lambda z - q)\right)^2 \\ &= (1-\lambda\eta)^2 g(z). \end{aligned}$$

We see that we need $\eta < \frac{1}{\lambda}$ for the objective to decrease by a constant factor < 1 in every step. This yields exponentially fast convergence to the minimum. I.e. for K steps of gradient descent, we have

$$g(z^K) = (1-\lambda\eta)^{2k} g(z^0).$$

Let's now come back to the multi-dimensional quadratic. The step size condition becomes

$$\eta \leq \frac{1}{\lambda_{max}(\mathbf{Q})}.$$

The optimal step size:

$$\begin{aligned}\eta^* &= \min_{\eta} \max_{i} (1 - \eta\lambda_i)^2 \\ &= \min_{\eta} \max\{\eta\lambda_{\max} - 1, 1 - \eta\lambda_{min}\}\end{aligned}$$

is attained at

$$\eta\lambda_{max} - 1 \stackrel{!}{=} 1 - \eta\lambda_{min}$$

$$\iff \eta^* = \frac{2}{\lambda_{max} + \lambda_{min}}.$$

Optimal Rates The slowest rate is in direction of the eigenvector with smallest eigenvalue, for which

$$\rho = (1 - \lambda_{min}\eta^*)^2 = \left(\frac{\lambda_{max} - \lambda_{min}}{\lambda_{max} + \lambda_{min}}\right)^2$$

$$\leq (1 - \kappa)^2, \quad \kappa = \frac{\lambda_{max}}{\lambda_{min}}.$$

The constant κ is known as the condition number of $\mathbf{Q}$. It shows that gradient descent can be very slow for functions where $\lambda_{max} \gg \lambda_{min}$. We show an example in Figure 5.4, as well as the effect a preconditioner in Figure 5.5.

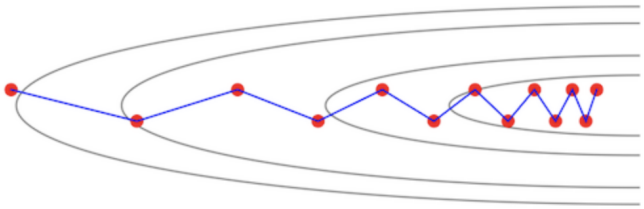


Figure 5.4: Example of a two-dimensional ill-conditioned function where $\lambda_{max} \gg \lambda_{min}$.

5.5 Convergence for Smooth and Strongly-convex Functions

We start with a few definitions.

Definition 19 (L-smooth function)**.** *A function f is L-smooth (alternatively it is said to have L-Lipschitz-continuous gradients) if:*

$$\|\nabla f(\mathbf{x}) - \nabla f(\mathbf{x} + \Delta\mathbf{x})\| \leq L\|\Delta\mathbf{x}\|.$$

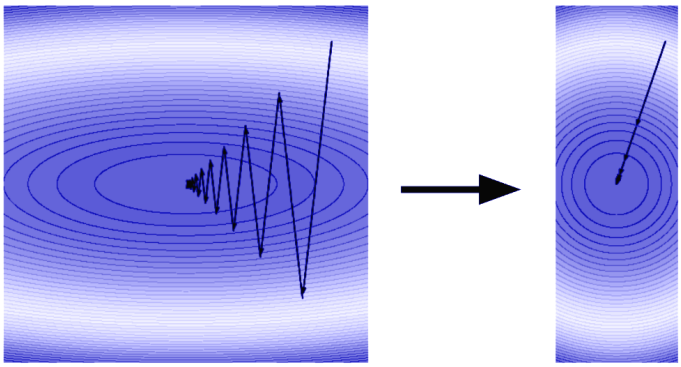


Figure 5.5: Desired effect of a preconditioner. It rescales the different dimension to make the level sets isotropic. This implies that the condition number $\kappa = 1$.

Definition 20 (μ-strongly convex function)**.** *A function f is μ-strongly convex if*

$$f(\mathbf{x}+\Delta\mathbf{x}) \geq f(\mathbf{x}) + \nabla f(\mathbf{x})\cdot\Delta\mathbf{x} + \frac{\mu}{2}\|\Delta\mathbf{x}\|^2.$$

The function is convex if $\mu = 0$.

If f is twice differentiable then smoothness can be restated as

$$\boxed{f(\mathbf{x}+\Delta\mathbf{x}) \leq f(\mathbf{x}) + \nabla f(\mathbf{x})\cdot\Delta\mathbf{x} + \frac{L}{2}\|\Delta\mathbf{x}\|^2}$$

The Hessian of a (twice differentiable) smooth and strongly convex function is thus sandwiched

$$\boxed{\mu\mathbf{I} \preceq \nabla^2 f \preceq L\mathbf{I}}$$

This connects back to the quadratic case, where $\lambda_{\min} = \mu$ and $\lambda_{max} = L$.

Theorem 34. *For a μ-strongly convex, L-smooth function f, the gradient descent iterates $\mathbf{x}_k$ with step size $0 < \eta \leq 1/L$ converge to the unique minimizer $\mathbf{x}^*$ at rate*

$$\|\mathbf{x}_k - \mathbf{x}^*\|^2 \leq (1-\eta\mu)^k \|\mathbf{x}_0 - \mathbf{x}^*\|^2$$

The convergence rate in Theorem 34 is typically called *linear rate*. It is a faster than the rate of convergence obtained for convex (but not strongly-convex) functions as we shall see shortly.

Corollary 35. *Using L-smoothness, we also have*

$$f(\mathbf{x}_k) - f(\mathbf{x}^*) \leq \frac{L}{2}\|\mathbf{x}_k - \mathbf{x}^*\|^2 \leq \frac{L}{2}(1-\eta\mu)^k \|\mathbf{x}_0 - \mathbf{x}^*\|^2$$

The proof makes use of the following lemma:

Lemma 36. *If f is differentiable and L-smooth, then*

$$f(\mathbf{x}) - f(\mathbf{x}^*) \geq \frac{1}{2L}\|\nabla f(\mathbf{x})\|^2.$$

Proof.

$$\begin{aligned} f(\mathbf{x}^*) - f(\mathbf{x}) &\leq f\left(\mathbf{x} - \tfrac{1}{L}\nabla f(\mathbf{x})\right) - f(\mathbf{x}) \\ &\leq f(\mathbf{x}) - \tfrac{1}{L}\|\nabla f(\mathbf{x})\|^2 + \frac{L}{2}\|\tfrac{1}{L}\nabla f(\mathbf{x})\|^2 - f(\mathbf{x}) \\ &= -\frac{1}{2L}\|\nabla f(\mathbf{x})\|^2. \end{aligned}$$

□

Proof Theorem 34. From strong-convexity, we have

$$f(\mathbf{x}^*) - f(\mathbf{x}) \geq \nabla f(\mathbf{x}) \cdot (\mathbf{x}^* - \mathbf{x}) + \frac{\mu}{2}\|\mathbf{x}^* - \mathbf{x}\|^2.$$

By induction using

$$\begin{aligned} &\|\mathbf{x}_{k+1} - \mathbf{x}^*\|^2 \\ &= \|\mathbf{x}_k - \eta\nabla f(\mathbf{x}_k) - \mathbf{x}^*\|^2 \\ &= \|\mathbf{x}_k - \mathbf{x}^*\|^2 - 2\eta\nabla f(\mathbf{x}_k) \cdot (\mathbf{x}_k - \mathbf{x}^*) + \eta^2\|\nabla f(\mathbf{x}_k)\|^2 \\ &\overset{(\mu)}{\leq} (1 - \eta\mu)\|\mathbf{x}_k - \mathbf{x}^*\|^2 - 2\eta(f(\mathbf{x}_k) - f(\mathbf{x}^*)) + \eta^2\|\nabla f(\mathbf{x}_k)\|^2 \\ &\overset{(L)}{\leq} (1 - \eta\mu)\|\mathbf{x}_k - \mathbf{x}^*\|^2 - 2\eta(f(\mathbf{x}_k) - f(\mathbf{x}^*)) + 2L\eta^2(f(\mathbf{x}_k) - f(\mathbf{x}^*)) \\ &= (1 - \eta\mu)\|\mathbf{x}_k - \mathbf{x}^*\|^2 - 2\eta(1 - \eta L)(f(\mathbf{x}_k) - f(\mathbf{x}^*)) \\ &\leq (1 - \eta\mu)\|\mathbf{x}_k - \mathbf{x}^*\|^2, \end{aligned}$$

where (μ) uses strong-convexity and (L) uses Lemma 36. □

The convexity condition alone (with smoothness) can only guarantee a slower sublinear convergence of the order $\mathcal{O}\left(\frac{1}{k}\right)$.

Theorem 37. *Let f be convex, differentiable and L-smooth. Then with step size $\eta \leq \frac{1}{L}$, the suboptimality along the gradient descent iterates decays like*

$$f(\mathbf{x}_k) - f(\mathbf{x}^*) \leq \frac{1}{2\eta k}\|\mathbf{x}_0 - \mathbf{x}^*\|^2$$

where $\mathbf{x}^*$ is a minimizer of f.

5.6 Convergence without Convexity

In the non-convex case, local convergence is typically measured in terms of a vanishing gradient norm (i.e. first-order criticality).

$$\boxed{\|\nabla f(\mathbf{x})\|^2 \leq \epsilon} \qquad (\epsilon\text{-stationarity})$$

Theorem 38. *Assume f is differentiable, L-smooth, but not necessarily convex and with minimum f^*. The gradient descent iterates with step size $\eta \leq 1/L$ satisfy*

$$\min_{i=0}^{k} \|\nabla f(\mathbf{x}_i)\|^2 \leq \frac{2(f(\mathbf{x}_0) - f^*)}{\eta(k+1)}$$

Note that in the convex case, the decay of the function value $f(\mathbf{x}_k) - f(\mathbf{x}^*)$ (called suboptimality) also implies that the squared gradient norm vanishes at the same rate. Here the gradient norm vanishes at rate $1/k$ but without making any statement about the suboptimality of the iterates.

A fundamental lemma to prove convergence is the descent lemma, which we state next.

Lemma 39 (Descent lemma)**.** *Consider a gradient descent step $\mathbf{x}_{k+1} = \mathbf{x}_k - \eta\nabla f(\mathbf{x}_k)$ on a L-smooth function f. For $\eta \leq 1/L$ this yields the following function decrease:*

$$f(\mathbf{x}_{k+1}) - f(\mathbf{x}_k) \leq -\frac{\eta}{2}\|\nabla f(\mathbf{x}_k)\|^2. \tag{5.5}$$

Proof. By Taylor expanding f and using smoothness

$$\begin{aligned} f(\mathbf{x}_{k+1}) &\leq f(\mathbf{x}_k) + \nabla f(\mathbf{x}_k)^\top(\mathbf{x}_{k+1} - \mathbf{x}_k) + \frac{L}{2}\|\mathbf{x}_{k+1} - \mathbf{x}_k\|^2 \\ &= f(\mathbf{x}_k) - \eta\|\nabla f(\mathbf{x}_k)\|^2 + \frac{\eta^2 L}{2}\|\nabla f(\mathbf{x}_k)\|^2 \\ &\leq f(\mathbf{x}_k) - \frac{\eta}{2}\|\nabla f(\mathbf{x}_k)\|^2, \end{aligned}$$

where we used the condition $\eta \leq \frac{1}{L}$ in the last inequality. □

Proof of Theorem 38. 1. For any $\eta \leq 1/L$ by the descent lemma,

$$\frac{\eta}{2}\|\nabla f(\mathbf{x}_k)\|^2 \leq f(\mathbf{x}_k) - f(\mathbf{x}_{k+1}).$$

2. Summing over iterates we get a telescoping sum

$$\frac{\eta}{2}\sum_{i=0}^{k}\|\nabla f(\mathbf{x}_i)\|^2 \leq f(\mathbf{x}_0) - f(\mathbf{x}_{k+1}) \leq f(\mathbf{x}_0) - f(\mathbf{x}^*)$$

3. We can lower bound the sum by the smallest summand

$$\sum_{i=0}^{k}\|\nabla f(\mathbf{x}_i)\|^2 \geq (k+1)\min_{i=0}^{k}\|\nabla f(\mathbf{x}_i)\|^2$$

We conclude the proof by combining steps 2 and 3. □

Polyak-Łojasiewicz (PL) Condition The PL condition is a weaker assumption than convexity that still allows gradient descent to obtain a fast (i.e. linear) rate of convergence. It is defined as follows.

Definition 21. *For $\mu > 0$, a function f is μ-PL if*

$$\frac{1}{2}\|\nabla f(\mathbf{x})\|^2 \geq \mu(f(\mathbf{x}) - f^*),\ \forall \mathbf{x},\ f^* = \min_{\mathbf{x}} f(\mathbf{x})$$

Two examples of PL functions are shown in Figure 5.6.

Figure 5.6: Examples of PL functions: 2-dimensional case on the left and 3-dimensional case on the right.

Theorem 40 (Convergence Theorem with PL Condition)**.** *Let f be differentiable and L-smooth, not necessarily convex with minimum f^* and fulfilling the PL condition with $\mu > 0$. The gradient descent iterates with step size $\eta = \frac{1}{2L}$ satisfy*

$$f(\mathbf{x}_k) - f^* \leq \left(1 - \frac{\mu}{L}\right)^k (f(\mathbf{x}_0) - f^*)$$

Proof sketch. 1. By the descent lemma for L-smooth functions

$$f(\mathbf{x}_{k+1}) - f(\mathbf{x}_k) \leq -\frac{1}{2L}\|\nabla f(\mathbf{x}_k)\|^2.$$

2. By the PL condition,

$$-\frac{1}{2L}\|\nabla f(\mathbf{x}_k)\|^2 \leq -\frac{\mu}{L}(f(\mathbf{x}_k) - f^*).$$

3. Subtracting f^* on both sides

$$f(\mathbf{x}_{k+1}) - f^* \leq \left(1 - \frac{\mu}{L}\right)(f(\mathbf{x}_k) - f^*).$$

The claim follows by induction.

□

Challenges: Local Minima Modern deep neural networks architecture are almost exclusively trained with first-order gradient-based methods. This might appear somehow surprising at first as one would suspect that the landscape of the risk function of a neural network to be complex as it could include many local minima or saddle points. If this were to be the case, gradient descent would perform poorly. Yet, this is not typically what is seen in practice. Here are some common observations:

1. Are local minima a practical issue? Not always: Gori & Tesi, 1992
2. Do local minima even exist? Sometimes not (auto-encoder): Baldi & Hornik, 1989
3. Are local minima typically worse? Often not (large networks): e.g. Choromanska et al, 2015
4. Which local minima generalize well (wide vs. sharp.isolated): ongoing discussion
5. Can we understand the learning dynamics? Deep linear case has similarities with non-linear case, e.g. Saxe et al., 2013

5.7 Stochastic Gradient Descent

The computational complexity of gradient descent is *linear* in the number of samples, which becomes prohibitive for large data sets. How can we reduce the complexity of this approach? A simple idea would be to approximate the empirical average over all training instances by an empirical average on a smaller set. In order to retain statistical efficiency, this will require resampling at every update. This is the main principle behind *stochastic* gradient descent.

Let's formalize this idea. We consider a general empirical risk function of the form

$$f(\mathbf{x}) = \frac{1}{n}\sum_{i=1}^{n} f_i(\mathbf{x}), \quad \text{s.t. } \nabla f(\mathbf{x}) = \frac{1}{n}\sum_{i=1}^{n} \nabla f_i(\mathbf{x}).$$

Stochastic Gradient Descent (SGD) starts from an iterate $\mathbf{x}_0$ and performs the following iterative update:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \eta_k \nabla f_{I(k)}(\mathbf{x}_k), \quad I(k) \sim \text{Uniform}\{1, \dots, n\},$$

where $\eta_k > 0$ is a step size schedule. Note that unlike gradient descent, the step size η_k depends on the iteration number k. A popular choice is to decrease the step size with k.

Bias and Variance The update direction of SGD is *unbiased*, i.e.

$$\mathbb{E}[\nabla f_I(\mathbf{x})] = \sum_{i=1}^{n} \frac{1}{n} \nabla f_i(\mathbf{x}) = \nabla f(\mathbf{x}).$$

Note that the expectation $\mathbb{E}$ in these notes is always taken w.r.t. the data. The stochastic effect can be quantified by the variance function

$$\text{var}(\mathbf{x}) = \frac{1}{n} \sum_{i=1}^{n} \|\nabla f(\mathbf{x}) - \nabla f_i(\mathbf{x})\|^2 .$$

A key quantity that we will discuss later is the variance around a global (or local) minimum $\mathbf{x}^*$ where $\nabla f(\mathbf{x}^*) = 0$. We will see it has an important effect on the convergence of SGD.

Minibatch SGD The term minibatching refers to sampling more than one function f_i (i.e. $|I(k)| = r > 1$). Minibatch SGD gives an unbiased update and reduces the variance. When training deep neural networks, selecting the batch size is not trivial as the theory only gives some rough guidelines. We here give a few recommendations that can be followed in practice:

- size r or the minibatch can range from $r = 1$ (single data point) to r in the 100s or 1000s.
- smaller r introduce more noise, yet usually give better results in the non-convex case.
- choice of r: performance and hardware (e.g. GPU memory)

General idea to prove convergence The general idea can be seen from the following decomposition:

$$\begin{aligned}
&\mathbb{E}\|\mathbf{x}_{k+1} - \mathbf{x}^*\|^2 \\
&\quad = \mathbb{E}\|\mathbf{x}_k - \eta_k \nabla f_i(\mathbf{x}_k) - \mathbf{x}^*\|^2 \\
&\quad = \mathbb{E}\|\mathbf{x}_k - \mathbf{x}^*\|^2 - \underbrace{2\eta_k \mathbb{E}(\mathbf{x}_k - \mathbf{x}^*)^\top \nabla f_i(\mathbf{x}_k)}_{\text{Progress term}} + \underbrace{\eta_k^2 \mathbb{E}\|\nabla f_i(\mathbf{x}_k)\|^2}_{\text{Variance term}} .
\end{aligned}$$

The progress term contributes to decreasing the distance $\mathbb{E}\left\|\mathbf{x}_{k+1}-\mathbf{x}^*\right\|^2$. If f is strongly-convex, this can be seen by using the inequality $f(\mathbf{x}^*)-f(\mathbf{x}) \geq \nabla f(\mathbf{x})\cdot(\mathbf{x}^*-\mathbf{x})+\frac{\mu}{2}\|\mathbf{x}^*-\mathbf{x}\|^2$. On the contrary, the variance term has the opposite effect. One therefore needs to balance these two terms carefully. How? There are several options such as:

1. Averaging iterates: although this is common for analysis purposes it is less often used in practice. One type of averaging is called Polyak-Rupert averages:
$$\bar{\mathbf{x}}_{k+1}=\frac{k}{k+1}\bar{\mathbf{x}}_k+\frac{1}{k+1}\mathbf{x}_{k+1}.$$
2. Use an appropriate decreasing step-size.
3. Use an explicit mechanism to reduce the variance.

5.7.1 Convergence for smooth function and bounded gradients

In the non-convex case, local convergence is typically measured in terms of a vanishing gradient norm (i.e. first-order criticality). In the stochastic case, we naturally extend this notion by taking an expectation over the randomness of the datapoint sampling, i.e.

$$\boxed{\mathbb{E}\|\nabla f(\mathbf{x})\|^2 \leq \epsilon} \qquad (\epsilon\text{-stationarity})$$

The following result derived in Ghadimi and Lan (2013); Reddi et al. (2016) demonstrates convergence in terms of the expected norm of the gradient, i.e. we reach a first-order critical point in expectation.

Theorem 41. *Assume that the gradient of function f is L-Lipschitz and the stochastic gradients are bounded by a constant σ. Let $\eta_k=\frac{c}{\sqrt{K}}$ with $c=\sqrt{\frac{2(f(\mathbf{x}_0)-f^*)}{L\sigma^2}}$, then the iterates of SGD satisfy*

$$\min_{s\in[0,K-1]}\mathbb{E}\left\|\nabla f(\mathbf{x}_s)\right\|^2 \leq \sqrt{\frac{2(f(\mathbf{x}_0)-f^*)L}{K}}\sigma \tag{5.6}$$

Proof. Since ∇f is L-Lipschitz, we get

$$\begin{aligned}
\mathbb{E}f(\mathbf{x}_{k+1}) &\leq \mathbb{E}[f(\mathbf{x}_k)+\langle\nabla f(\mathbf{x}_k),\mathbf{x}_{k+1}-\mathbf{x}_k\rangle+\frac{L}{2}\left\|\mathbf{x}_{k+1}-\mathbf{x}_k\right\|^2] \\
&\leq \mathbb{E}f(\mathbf{x}_k)-\eta\mathbb{E}\|\nabla f(\mathbf{x}_k)\|^2+\frac{L\eta^2}{2}\mathbb{E}\left\|\nabla f_i(\mathbf{x}_k)\right\|^2 \\
&\leq \mathbb{E}f(\mathbf{x}_k)-\eta\mathbb{E}\left\|\nabla f(\mathbf{x}_k)\right\|^2+\frac{L\eta^2}{2}\sigma^2,
\end{aligned} \tag{5.7}$$

where the second inequality follows from the unbiasness of the stochastic gradients, i.e. $\mathbb{E}_i \nabla f_i(\mathbf{x}) = \nabla f(\mathbf{x})$.

By rearranging the terms in the equation above, we get

$$\mathbb{E} \left\| \nabla f(\mathbf{x}_k) \right\|^2 \leq \frac{1}{\eta} \mathbb{E}[f(\mathbf{x}_k) - f(\mathbf{x}_{k+1})] + \frac{L\eta}{2} \sigma^2 \tag{5.8}$$

By summing from $k = 0$ to $K - 1$,

$$\begin{aligned} \min_{s \in [0, K-1]} \mathbb{E} \left\| \nabla f(\mathbf{x}_s) \right\|^2 &\leq \frac{1}{K} \sum_{k=0}^{K-1} \mathbb{E} \left\| \nabla f(\mathbf{x}_k) \right\|^2 \\ &\leq \frac{1}{\eta K} \mathbb{E}[f(\mathbf{x}_0) - f(\mathbf{x}_{K-1})] + \frac{L\eta}{2} \sigma^2 \\ &\leq \frac{1}{\eta K} [f(\mathbf{x}_0) - f(\mathbf{x}^*)] + \frac{L\eta}{2} \sigma^2 \\ &\leq \frac{1}{\sqrt{K}} \left(\frac{1}{c} [f(\mathbf{x}_0) - f(\mathbf{x}^*)] + \frac{Lc}{2} \sigma^2 \right). \end{aligned} \tag{5.9}$$

We conclude by inserting our specific choice of c in the last inequality.

□

Convergence of the last iterate Why do we have a $\min_{s \in [0, K-1]}$ instead of $\mathbb{E} \left\| \nabla f(\mathbf{x}_{K-1}) \right\|^2$? This is because the process is stochastic and the norm can in fact increase due to the variance captured by σ^2. The process oscillates around the critical point. The only way to exactly converge would be to ensure that the variance decreases to zero (by sampling the full gradient, or by other means).

Alternatively, one could select an iterate $\mathbf{x}_I$ of SGD uniformly at random from the sequence $(\mathbf{x}_0, \ldots, \mathbf{x}_{K-1})$. By taking the expectation with respect to this randomization and the noise in the stochastic gradients we have that

$$\mathbb{E} \left\| \nabla f(\mathbf{x}_I) \right\|^2 = \frac{1}{K} \sum_{k=0}^{K-1} \mathbb{E} \left\| \nabla f(\mathbf{x}_k) \right\|^2.$$

Iteration complexity Finally, one can also talk about the iteration complexity of the algorithm by asking how many iterations K are required to achieve $\mathbb{E} \| \nabla f(\mathbf{x}) \|^2 \leq \epsilon$? A simple calculation shows we need $K = \mathcal{O}\left(\frac{1}{\epsilon^2}\right)$. In comparison, the iteration complexity of GD is $K = \mathcal{O}\left(\frac{n}{\epsilon}\right)$ where n is the per-iteration complexity to compute the full gradient (which depends on the number of training datapoints in machine learning).

5.8 Exercise: Optimization

Problem 1 (Characterizations of convex functions):
In class, you have seen that a function $\ell : \mathbb{R}^d \to \mathbb{R}$ is convex if

$$\forall \mathbf{x}, \mathbf{x}' \in \mathbb{R}^d,\ \forall \lambda \in [0,1] : \ell(\lambda \mathbf{x} + (1-\lambda)\mathbf{x}') \leq \lambda \ell(\mathbf{x}) + (1-\lambda)\ell(\mathbf{x}'). \tag{5.10}$$

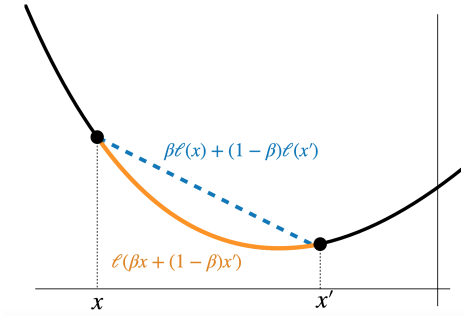
a) Show that, if ℓ is differentiable, this condition implies that

$$\forall \mathbf{x}, \mathbf{x}' \in \mathbb{R}^d : \ell(\mathbf{x}) \geq \ell(\mathbf{x}') + \nabla \ell(\mathbf{x}')^\top (\mathbf{x} - \mathbf{x}'). \tag{5.11}$$

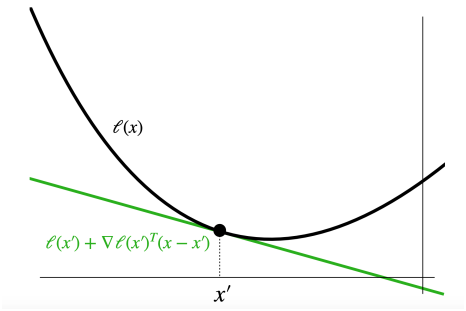
b) Show that, if ℓ is differentiable, conditions (5.10) and (5.11) are actually equivalent.

c) Now we assume that ℓ is also Lipschitz-smooth, i.e. there exists $L > 0$ such that $\|\nabla \ell(x) - \nabla \ell(\mathbf{x}')\| \leq L\|\mathbf{x} - \mathbf{x}'\|,\ \forall \mathbf{x}, \mathbf{x}' \in \mathbb{R}^d$. Using the fundamental theorem of calculus as well as Cauchy-Schwarz inequality, show that

$$\ell(\mathbf{x}) \leq \ell(\mathbf{x}') + \nabla \ell(\mathbf{x}')^\top (\mathbf{x} - \mathbf{x}') + \frac{L}{2}\|\mathbf{x} - \mathbf{x}'\|^2 \tag{5.12}$$



Definition in the lecture



This exercise

Problem 2 (Gradient Descent for Least-square Problem):
Consider the problem of solving the linear system

$$\mathbf{X}\mathbf{w} = \mathbf{y}$$

where $\mathbf{X} = (\mathbf{x}_i^\top)_{i=1}^n \in \mathbb{R}^{n\times d}$ is the data matrix, $\mathbf{w} \in \mathbb{R}^d$ the parameter vector of the model, and $\mathbf{y} \in \mathbb{R}^n$ the target. We assume that there exists a unique solution provided by the mean-squared loss

$$\mathbf{w}_* = \arg\min_{\mathbf{w}\in\mathbb{R}^d} \left(\frac{1}{2}\|\mathbf{X}\mathbf{w} - \mathbf{y}\|_2^2 =: \ell(\mathbf{w}) \right).$$

We consider the gradient descent method

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \nabla \ell(\mathbf{w}_t) \tag{5.13}$$

with a fixed step size η.

a) Calculate the gradient $\nabla\ell(\mathbf{w})$ and use it to re-write the gradient descent iterates of Eq. (5.13).

b) Calculate the Hessian $\nabla^2\ell(\mathbf{w})$ and argue whether or not $\ell(\mathbf{w})$ is convex.

c) Use the following facts.

- Remember that $\mathbf{y} = \mathbf{X}\mathbf{w}_*$;
- Recall the definition of the operator norm of a matrix:

$$\|\mathbf{A}\|_{\text{op}} = \sup\left\{ \frac{\|\mathbf{A}\mathbf{x}\|_2}{\|\mathbf{x}\|_2} : \mathbf{x} \in \mathbb{R}^d \text{ with } \mathbf{x} \neq 0 \right\} = \sqrt{\lambda_{\max}(\mathbf{A}^T\mathbf{A})},$$

  where $\lambda_{\max}(\mathbf{A}^T\mathbf{A})$ denotes the largest eigenvalue of $\mathbf{A}^T\mathbf{A}$.
- for any matrix $\mathbf{A} \in \mathbb{R}^{d\times d}$ and vector $\mathbf{v} \in \mathbb{R}^d$, we have $\|\mathbf{A}\mathbf{v}\|_2 \leq \|\mathbf{A}\|_{\text{op}} \cdot \|\mathbf{v}\|_2$, where $\|\mathbf{A}\|_{\text{op}}$ denotes the operator norm of the matrix $\mathbf{A}$;

Show that the following bound on the distance to the optimizer holds

$$\|\mathbf{w}_{t+1} - \mathbf{w}_*\|_2 \leq \|\mathbf{I} - \eta\mathbf{X}^\top\mathbf{X}\|_{\text{op}}^t \cdot \|\mathbf{w}_1 - \mathbf{w}_*\|_2 \tag{5.14}$$

d) Show that, if $\eta < \frac{1}{\|\mathbf{X}^\top\mathbf{X}\|_{\text{op}}}$, i.e. if $\|\eta\mathbf{X}^\top\mathbf{X}\|_{\text{op}} < 1$, then the Gradient Descent converge, i.e. $\|\mathbf{w}_{t+1} - \mathbf{w}_*\|_2 \to 0$ as $t \to \infty$.

e) One, obviously cheaper, alternative is to only compute the update (indexed by k) based on the gradient of one specific datapoint $\mathbf{x}_i$. This is the updated of *stochastic* gradient descent (SGD), which is arguably the most widely used optimizer in ML:

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \eta\nabla\ell_i(\mathbf{w}_k), \quad i \in \{1, \ldots, n\};\ \eta > 0.$$

where we denote $\ell_i(\mathbf{w}) := \frac{n}{2}(\mathbf{x}_i^\top \mathbf{w} - y_i)^2$ for each $i = 1, ..., n$. In each iteration k, a datapoint $\mathbf{x}_i$ is chosen uniformly at random such that $\mathbb{E}[\nabla \ell_i(\mathbf{w}_k)] = \nabla \ell(\mathbf{w}_k)$.

Show that, given a constant positive step size $\eta \leq \frac{1}{2\|\mathbf{X}^\top \mathbf{X}\|_{\text{op}}}$, we have

$$\mathbb{E}\left[\|\mathbf{w}_{k+1} - \mathbf{w}_*\|_2^2\right] \geq \eta^2 \mathbb{E}\left[\|\nabla \ell_i(\mathbf{w}_k)\|_2^2\right] > 0$$

In other words, SGD does not converge to the critical point $\mathbf{w}_*$ on average.

Chapter 6

Optimization Landscape of Neural Networks

We will discuss the properties of the loss surface of deep neural networks and will see they highly depend on the type of architecture, including parameters such as width, depth, activation functions, etc.

We start with a few definitions that will be used throughout these notes.

Definition 22 (Global minimum)**.** *Given a function $f(\mathbf{w}) : \mathbb{R}^d \to \mathbb{R}$, a point $\mathbf{w}^*$ is called a global minimum of f if for every $\mathbf{w}$, we have $f(\mathbf{w}^*) \leq f(\mathbf{w})$.*

Definition 23 (Local minimum)**.** *Given a function $f(\mathbf{w}) : \mathbb{R}^d \to \mathbb{R}$, a point $\mathbf{w}^*$ is called a local minimum of f if there exists a neighborhood of size ϵ, i.e. $\{\mathbf{w} \in \mathbb{R}^d \mid \|\mathbf{w} - \mathbf{w}^*\| \leq \epsilon\}$ such that $f(\mathbf{w}^*) \leq f(\mathbf{w})$.*

Definition 24 (Saddle point)**.** *Given a function $f(\mathbf{w}) : \mathbb{R}^d \to \mathbb{R}$, a point $\mathbf{w}_s$ is called a saddle point of f if $\nabla f(\mathbf{w}_s) = 0$ and the Hessian matrix $\nabla^2 f(\mathbf{w}_s)$ is indefinite, meaning that it has both positive and negative eigenvalues.*

Existing studies are typically restricted to some specific settings, including:

- **Linear networks**: every local minima is global and every critical point that is not a global minimum is a saddle point (Kawaguchi, 2016).
- **Small-size two-layer ReLU networks:** these networks have spurious local minima and there is a high probability of reaching them Safran and Shamir (2018).
- **Heavily over-parametrized networks:** every local minima is global, see Du et al. (2018) for two-layers and Allen-Zhu et al. (2018) for arbitrary depth.

- **Large (practical) networks:** the landscape of large networks is complicated and one can find arbitrary low-dimensional patterns (Czarnecki et al., 2019).

In this lecture, we focus on linear networks. We will later discuss the case of over-parametrized networks in the "Neural Tangent Kernel" lecture.

6.1 Loss Landscape of Deep Linear Networks

We focus on the case of a linear network and will see that, for a simple architecture with two layers and a squared loss function, there are no bad local minima (Baldi and Hornik, 1989; Kawaguchi, 2016).

6.1.1 Gradients of Deep Linear Network

In cases where closed-form updates are not available, one has to rely on gradient descent to find the optimal weights. The calculation of the gradients is therefore essential. In this section, we will see how to derive the gradients of the parameter matrices of a linear network, which will first require reviewing differentiation of expressions that involve matrices.

Matrix differentiation

We start by covering differentiation of a number of expressions that involve matrices. As an exercise to the reader, we suggest trying to derive these formula yourselves. To do so, start by deriving the formula with respect to a single entry. Take as an example the gradient of $\nabla_{\mathbf{X}}\|\mathbf{X}\|_F^2$ where $\mathbf{X} \in \mathbb{R}^{m\times n}$ is a matrix with entries x_{ij}. We have

$$\nabla_{x_{ij}}\|\mathbf{X}\|_F^2 = \nabla_{x_{ij}}\left(\sum_{i=1}^{m}\sum_{j=1}^{n} x_{ij}^2\right) = 2x_{ij}.$$

Gradient of the matrix Frobenius norm For a matrix $\mathbf{X} \in \mathbb{R}^{m\times n}$:

$$\nabla_{\mathbf{X}}\|\mathbf{X}\|_F^2 = 2\mathbf{X} \tag{6.1}$$

Gradient of the trace of a matrix product For matrices $\mathbf{X} \in \mathbb{R}^{m\times n}$ and $\mathbf{A} \in \mathbb{R}^{n\times m}$:

$$\begin{aligned}\nabla_{\mathbf{X}}\mathrm{tr}(\mathbf{AX}) &= \mathbf{A}^\top \\ \nabla_{\mathbf{X}}\mathrm{tr}(\mathbf{AX}^\top) &= \mathbf{A}\end{aligned} \tag{6.2}$$

Gradient of the quadratic form (symmetric A) For a symmetric matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$ and vector $\mathbf{x} \in \mathbb{R}^n$:

$$\nabla_{\mathbf{x}}(\mathbf{x}^\top \mathbf{A}\mathbf{x}) = 2\mathbf{A}\mathbf{x} \tag{6.3}$$

Gradient of the quadratic form (non-symmetric A) For a non-symmetric matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$ and vector $\mathbf{x} \in \mathbb{R}^n$:

$$\nabla_{\mathbf{x}}(\mathbf{x}^\top \mathbf{A}\mathbf{x}) = (\mathbf{A} + \mathbf{A}^\top)\mathbf{x} \tag{6.4}$$

Gradient of the quadratic form with respect to A For a matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$ and vector $\mathbf{x} \in \mathbb{R}^n$:

$$\nabla_{\mathbf{A}}(\mathbf{x}^\top \mathbf{A}\mathbf{x}) = \mathbf{x}\mathbf{x}^\top \tag{6.5}$$

Gradient of $\mathbf{x}^\top \mathbf{A}^\top \mathbf{A}\mathbf{x}$ For a matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$ and vector $\mathbf{x} \in \mathbb{R}^n$:

$$\nabla_{\mathbf{x}}(\mathbf{x}^\top \mathbf{A}^\top \mathbf{A}\mathbf{x}) = 2\mathbf{A}^\top \mathbf{A}\mathbf{x} \tag{6.6}$$

Gradient of a more complex quadratic form with respect to A For a matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$ and vectors $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$:

$$\frac{\partial}{\partial \mathbf{A}}(\mathbf{y} - \mathbf{A}\mathbf{x})^\top (\mathbf{y} - \mathbf{A}\mathbf{x}) = -2(\mathbf{y} - \mathbf{A}\mathbf{x})\mathbf{x}^\top. \tag{6.7}$$

We give the full proof below.

Proof. Step 1: Expand the objective function Expand the quadratic form:

$$f(\mathbf{A}) = (\mathbf{y} - \mathbf{A}\mathbf{x})^\top (\mathbf{y} - \mathbf{A}\mathbf{x}) = \mathbf{y}^\top \mathbf{y} - \mathbf{y}^\top \mathbf{A}\mathbf{x} - (\mathbf{A}\mathbf{x})^\top \mathbf{y} + (\mathbf{A}\mathbf{x})^\top (\mathbf{A}\mathbf{x}).$$

Since $(\mathbf{A}\mathbf{x})^\top = \mathbf{x}^\top \mathbf{A}^\top$, we have:

$$f(\mathbf{A}) = \mathbf{y}^\top \mathbf{y} - \mathbf{y}^\top \mathbf{A}\mathbf{x} - \mathbf{x}^\top \mathbf{A}^\top \mathbf{y} + \mathbf{x}^\top \mathbf{A}^\top \mathbf{A}\mathbf{x}.$$

Step 2: Compute the gradient with respect to $\mathbf{A}$ To compute $\frac{\partial f}{\partial \mathbf{A}}$, we take the gradient of each term separately.

- The term $\mathbf{y}^\top \mathbf{y}$ is constant with respect to $\mathbf{A}$, so its gradient is zero.
- For the term $\mathbf{y}^\top \mathbf{A}\mathbf{x}$:

$$\frac{\partial}{\partial \mathbf{A}}(\mathbf{y}^\top \mathbf{A}\mathbf{x}) = \frac{\partial}{\partial \mathbf{A}}\mathrm{tr}(\mathbf{y}^\top \mathbf{A}\mathbf{x}) = \mathbf{y}\mathbf{x}^\top.$$

- For the term $\mathbf{x}^\top \mathbf{A}^\top \mathbf{y}$:

$$\frac{\partial}{\partial \mathbf{A}}(\mathbf{x}^\top \mathbf{A}^\top \mathbf{y}) = \frac{\partial}{\partial \mathbf{A}}\mathrm{tr}(\mathbf{x}^\top \mathbf{A}^\top \mathbf{y}) = \mathbf{y}\mathbf{x}^\top.$$

- For the term $\mathbf{x}^\top \mathbf{A}^\top \mathbf{A}\mathbf{x}$:

$$\frac{\partial}{\partial \mathbf{A}}(\mathbf{x}^\top \mathbf{A}^\top \mathbf{A}\mathbf{x}) \overset{(6.6)}{=} 2\mathbf{A}\mathbf{x}\mathbf{x}^\top.$$

Putting it all together:

$$\frac{\partial f}{\partial \mathbf{A}} = -\mathbf{y}\mathbf{x}^\top - \mathbf{y}\mathbf{x}^\top + 2\mathbf{A}\mathbf{x}\mathbf{x}^\top = -2\mathbf{y}\mathbf{x}^\top + 2\mathbf{A}\mathbf{x}\mathbf{x}^\top.$$

Step 3: Simplify the gradient Notice that:

$$-2\mathbf{y}\mathbf{x}^\top + 2\mathbf{A}\mathbf{x}\mathbf{x}^\top = 2(-\mathbf{y}\mathbf{x}^\top + \mathbf{A}\mathbf{x}\mathbf{x}^\top) = 2(-(\mathbf{y} - \mathbf{A}\mathbf{x})\mathbf{x}^\top).$$

So, the gradient is:

$$\frac{\partial f}{\partial \mathbf{A}} = -2(\mathbf{y} - \mathbf{A}\mathbf{x})\mathbf{x}^\top.$$

This completes the proof. □

Application to gradients of neural networks

In this section, we demonstrate how to calculate gradients of a two-hidden layer linear neural network whose input-output map is defined as

$$\mathbf{y} = \mathbf{A}\mathbf{B}\mathbf{x}, \tag{6.8}$$

where $\mathbf{x} \in \mathbb{R}^d, \mathbf{y} \in \mathbb{R}^k, \mathbf{B} \in \mathbb{R}^{m \times d}, \mathbf{A} \in \mathbb{R}^{k \times m}$.

Given a set of n training examples $(\mathbf{x}_i, \mathbf{y}_i)_{i=1}^n$, we train the parameters of the network by optimizing a squared loss

$$L(\mathbf{A}, \mathbf{B}) = \frac{1}{2} \sum_{i=1}^n \|\mathbf{y}_i - \mathbf{A}\mathbf{B}\mathbf{x}_i\|^2. \tag{6.9}$$

In order to calculate the gradients, let's first expand the above expression:

$$\begin{aligned}
L(\mathbf{A}, \mathbf{B}) &= \frac{1}{2} \sum_{i=1}^n \|\mathbf{y}_i - \mathbf{A}\mathbf{B}\mathbf{x}_i\|^2 \\
&= \frac{1}{2} \sum_{i=1}^n (\mathbf{y}_i - \mathbf{A}\mathbf{B}\mathbf{x}_i)^\top (\mathbf{y}_i - \mathbf{A}\mathbf{B}\mathbf{x}_i) \\
&= \frac{1}{2} \sum_{i=1}^n \mathbf{y}_i^\top \mathbf{y}_i - \mathbf{y}_i^\top \mathbf{A}\mathbf{B}\mathbf{x}_i - (\mathbf{A}\mathbf{B}\mathbf{x}_i)^\top \mathbf{y}_i + (\mathbf{A}\mathbf{B}\mathbf{x}_i)^\top (\mathbf{A}\mathbf{B}\mathbf{x}_i) \\
&= \frac{1}{2} \sum_{i=1}^n \mathbf{y}_i^\top \mathbf{y}_i - \mathbf{y}_i^\top \mathbf{A}\mathbf{B}\mathbf{x}_i - \mathbf{x}_i^\top \mathbf{B}^\top \mathbf{A}^\top \mathbf{y}_i + \mathbf{x}_i^\top \mathbf{B}^\top \mathbf{A}^\top \mathbf{A}\mathbf{B}\mathbf{x}_i.
\end{aligned} \tag{6.10}$$

Recall the following equalities:

$$\begin{aligned}\frac{\partial \mathbf{a}^\top \mathbf{X}\mathbf{b}}{\partial \mathbf{X}} &= \mathbf{a}\mathbf{b}^\top \\ \frac{\partial \mathbf{a}^\top \mathbf{X}^\top \mathbf{D}\mathbf{X}\mathbf{b}}{\partial \mathbf{X}} &= \mathbf{D}^\top \mathbf{X}\mathbf{a}\mathbf{b}^\top + \mathbf{D}\mathbf{X}\mathbf{b}\mathbf{a}^\top,\end{aligned} \tag{6.11}$$

therefore we have

$$\begin{aligned}\frac{\partial \mathbf{y}_i^\top \mathbf{A}\mathbf{B}\mathbf{x}_i}{\partial \mathbf{B}} &= \mathbf{A}^\top \mathbf{y}_i \mathbf{x}_i^\top \\ \frac{\partial \mathbf{x}_i^\top \mathbf{B}^\top \mathbf{A}^\top \mathbf{y}_i}{\partial \mathbf{B}} &= \mathbf{x}_i \mathbf{y}_i^\top \mathbf{A} \\ \frac{\partial \mathbf{x}_i^\top \mathbf{B}^\top \mathbf{A}^\top \mathbf{A}\mathbf{B}\mathbf{x}_i}{\partial \mathbf{B}} &= \mathbf{A}\mathbf{A}^\top \mathbf{B}\mathbf{x}_i \mathbf{x}_i^\top + \mathbf{A}^\top \mathbf{A}\mathbf{B}\mathbf{x}_i \mathbf{x}_i^\top \\ &= 2\mathbf{A}\mathbf{A}^\top \mathbf{B}\mathbf{x}_i \mathbf{x}_i^\top.\end{aligned} \tag{6.12}$$

Note that the first two derivatives are equal (since $\mathbf{y}_i^\top \mathbf{A}\mathbf{B}\mathbf{x}_i = \mathbf{x}_i^\top \mathbf{B}^\top \mathbf{A}^\top \mathbf{y}_i$), thus

$$\begin{aligned}\frac{\partial L(\mathbf{A}, \mathbf{B})}{\partial \mathbf{B}} &= -\sum_{i=1}^{n} \mathbf{A}^\top \mathbf{y}_i \mathbf{x}_i^\top + \sum_{i=1}^{n} \mathbf{A}^\top \mathbf{A}\mathbf{B}\mathbf{x}_i \mathbf{x}_i^\top \\ &= -\mathbf{A}^\top (\Sigma_{xy} - \mathbf{A}\mathbf{B}\Sigma_{xx}),\end{aligned} \tag{6.13}$$

where $\Sigma_{xx} = \sum_{i=1}^{n} \mathbf{x}_i \mathbf{x}_i^\top$ is the input correlation matrix and $\Sigma_{xy} = \sum_{i=1}^{n} \mathbf{y}_i \mathbf{x}_i^\top$ is the input-output correlation matrix.

Proceeding similarly for the gradient of the loss w.r.t $\mathbf{A}$, we obtain

$$\begin{aligned}\frac{\partial L}{\partial \mathbf{A}} &= \sum_{i=1}^{n} -(\mathbf{y}_i - \mathbf{A}\mathbf{B}\mathbf{x}_i)\mathbf{x}_i^\top \mathbf{B}^\top \\ &= -(\sum_{i=1}^{n} \mathbf{y}_i \mathbf{x}_i^\top - \mathbf{A}\mathbf{B}\sum_{i=1}^{n} \mathbf{x}_i \mathbf{x}_i^\top)\mathbf{B}^\top \\ &= -(\Sigma_{xy} - \mathbf{A}\mathbf{B}\Sigma_{xx})\mathbf{B}^\top.\end{aligned} \tag{6.14}$$

6.1.2 Landscape Properties

Two-layer network (Baldi and Hornik, 1989) Consider a linear network with two layers whose weights are defined by matrices $\mathbf{A} \in \mathbb{R}^{d \times p}$ and $\mathbf{B} \in \mathbb{R}^{p \times d}$. Assume that both $\mathbf{x}_i$ and $\mathbf{y}_i$ are d-dimensional vectors and that the network consists of one input layer with d inputs, one hidden layer with p ($p \leq d$) units, and one output layer with d units.

Let $\mathbf{W} = \mathbf{AB}$, and define the following loss function,

$$\mathcal{L}(\mathbf{W}) = \sum_{i=1}^{n} \|\mathbf{y}_i - \mathbf{ABx}_i\|^2. \tag{6.15}$$

Define the following covariance matrices, $\Sigma_{\mathbf{xx}} = \sum_i \mathbf{x}_i\mathbf{x}_i^\top$, $\Sigma_{\mathbf{xy}} = \sum_i \mathbf{x}_i\mathbf{y}_i^\top$ and $\Sigma_{\mathbf{yy}} = \sum_i \mathbf{y}_i\mathbf{y}_i^\top$.

We can also write the objective in matrix form. To do so, let us introduce the input data matrix $\mathbf{X} = [\mathbf{x}_1 \dots \mathbf{x}_n]$ and the output data matrix $\mathbf{Y} = [\mathbf{y}_1 \dots \mathbf{y}_n]$. Then we can rewrite the objective as

$$\mathcal{L}(\mathbf{W}) = \|\mathbf{Y} - \mathbf{ABX}\|_F^2 = \mathrm{tr}[(\mathbf{Y} - \mathbf{ABX})(\mathbf{Y} - \mathbf{ABX})^\top]. \tag{6.16}$$

Theorem 42. *For any fixed $d \times p$ matrix $\mathbf{A}$ the function $\mathcal{L}(\mathbf{A}, \mathbf{B})$ is convex in the coefficients of $\mathbf{B}$ and attains its minimum for any $\mathbf{B}$ satisfying the equation*

$$\mathbf{A}^\top\mathbf{AB}\Sigma_{\mathbf{xx}} = \mathbf{A}^\top\Sigma_{\mathbf{yx}}. \tag{6.17}$$

Proof. We use the matrix form of the objective, i.e.

$$\begin{aligned}
\mathcal{L}(\mathbf{W}) &= \mathrm{tr}[(\mathbf{Y} - \mathbf{ABX})(\mathbf{Y} - \mathbf{ABX})^\top] \\
&= \mathrm{tr}[\mathbf{YY}^\top - \mathbf{Y}(\mathbf{ABX})^\top - \mathbf{ABXY}^\top + \mathbf{ABX}(\mathbf{ABX})^\top] && (6.18)\\
&= \mathrm{tr}[\mathbf{YY}^\top - \mathbf{YX}^\top\mathbf{B}^\top\mathbf{A}^\top - \mathbf{ABXY}^\top + \mathbf{ABXX}^\top\mathbf{B}^\top\mathbf{A}^\top]. && (6.19)
\end{aligned}$$

We then set the derivative of $\mathcal{L}(\mathbf{W})$ to zero:

$$\begin{aligned}
\frac{\partial\mathcal{L}(\mathbf{W})}{\partial\mathbf{B}} &= -2\mathbf{A}^\top\mathbf{YX}^\top + 2\mathbf{A}^\top\mathbf{ABXX}^\top \stackrel{!}{=} 0 \\
&\implies \mathbf{A}^\top\mathbf{YX}^\top = \mathbf{A}^\top\mathbf{ABXX}^\top \\
&\implies \mathbf{A}^\top\Sigma_{\mathbf{yx}} = \mathbf{A}^\top\mathbf{AB}\Sigma_{\mathbf{xx}} && (6.20)
\end{aligned}$$

where we used the following expressions for arbitrary matrices $\mathbf{Z}, \mathbf{C}$:

$$\begin{aligned}
\frac{\partial}{\partial\mathbf{B}}\mathrm{tr}[\mathbf{ABC}] &= \mathbf{A}^\top\mathbf{C}^\top \\
\frac{\partial}{\partial\mathbf{B}}\mathrm{tr}[\mathbf{CB}^\top\mathbf{A}^\top] &= \mathbf{A}^\top\mathbf{C} && (6.21)\\
\frac{\partial}{\partial\mathbf{B}}\mathrm{tr}[\mathbf{ABCB}^\top\mathbf{A}^\top] &= \mathbf{A}^\top\mathbf{ABC}^\top + \mathbf{A}^\top\mathbf{ABC}. && (6.22)
\end{aligned}$$

□

Theorem 43. *For any fixed $p \times d$ matrix $\mathbf{B}$ the function $\mathcal{L}(\mathbf{A}, \mathbf{B})$ is convex in the coefficients of $\mathbf{A}$ and attains its minimum for any $\mathbf{A}$ satisfying the equation*

$$\mathbf{AB}\Sigma_{\mathbf{xx}}\mathbf{B}^\top = \Sigma_{\mathbf{yx}}\mathbf{B}^\top. \tag{6.23}$$

Proof. Left as an exercise. □

Theorem 44. *Assume that $\Sigma_{\mathbf{xx}}$ is invertible. If two matrices $\mathbf{A}$ and $\mathbf{B}$ define a critical point of $\mathcal{L}$ (i.e., a point where $\frac{\partial \mathcal{L}}{\partial a_{ij}} = \frac{\partial \mathcal{L}}{\partial b_{ij}} = 0$) then the global map $\mathbf{W} = \mathbf{AB}$ is of the form*

$$\mathbf{W} = P_{\mathbf{A}}\Sigma_{\mathbf{xy}}\Sigma_{\mathbf{xx}}^{-1}, \tag{6.24}$$

where $P_{\mathbf{A}}$ is the matrix of the orthogonal projection onto the subspace spanned by the columns of $\mathbf{A}$.

Proof. If $\Sigma_{\mathbf{xx}}$ is invertible and $\mathbf{A}$ has full rank p, then $\mathcal{L}$ is strictly convex w.r.t. $\mathbf{B}$ and therefore it has a unique minimum (note that this can be shown for instance by showing that $\frac{\partial^2 \mathcal{L}(\mathbf{W})}{\partial \mathbf{B}^2} \succ 0$). By Theorem 42, the unique minimum is reached at

$$\mathbf{B}(\mathbf{A}) = (\mathbf{A}^\top\mathbf{A})^{-1}\mathbf{A}^\top\Sigma_{\mathbf{yx}}\Sigma_{\mathbf{xx}}^{-1}.$$

Therefore, the global map is

$$\mathbf{W} = \mathbf{AB}(\mathbf{A}) = \underbrace{\mathbf{A}(\mathbf{A}^\top\mathbf{A})^{-1}\mathbf{A}^\top}_{=P_{\mathbf{A}}}\Sigma_{\mathbf{yx}}\Sigma_{\mathbf{xx}}^{-1}.$$

□

Note that $\Sigma_{\mathbf{xy}}\Sigma_{\mathbf{xx}}^{-1}$ is the slope matrix for the ordinary least squares regression of $\mathbf{Y}$ on $\mathbf{X}$. Finally, we conclude with a result about the landscape of the objective function.

Theorem 45. *If $\Sigma_{\mathbf{xx}}$ and $\Sigma_{\mathbf{xy}}$ are full rank and $\Sigma = \Sigma_{\mathbf{yx}}\Sigma_{\mathbf{xx}^{-1}}\Sigma_{\mathbf{xy}}$ is full rank, then any local minimum is global and other critical points are saddle points.*

Proof. Proof omitted. □

We further illustrate this result in Figure 6.1 taken from the original paper of Baldi and Hornik (1989).

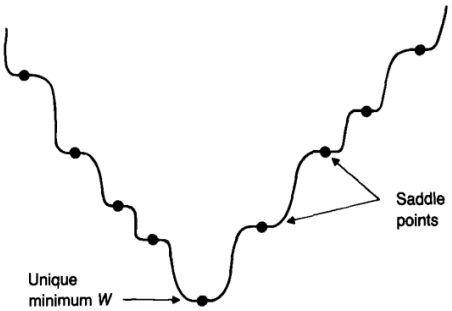


Figure 6.1: The landscape of the objective function (figure taken from Baldi and Hornik (1989).

Deep linear network A more recent result by (Kawaguchi, 2016) extend the results of Baldi and Hornik (1989) to a network with multiple layers. Consider the model $\hat{\mathbf{Y}} = \mathbf{W}_{H+1}\mathbf{W}_H \dots \mathbf{W}_1\mathbf{X}$ and the loss $\mathcal{L}(\mathbf{W}) = \frac{1}{2}\|\hat{\mathbf{Y}}(\mathbf{W}, \mathbf{X}) - \mathbf{Y}\|_F^2$.

First (Kawaguchi, 2016) shows that if $\mathbf{W}$ is a critical point of the loss $\mathcal{L}(\mathbf{W})$, then

$$\mathbf{W}_{H+1}\mathbf{W}_H \dots \mathbf{W}_1 = \mathbf{C}(\mathbf{C}^\top\mathbf{C})^{-1}\mathbf{C}^\top\mathbf{Y}\mathbf{X}^\top(\mathbf{X}\mathbf{X}^\top)^{-1}, \tag{6.25}$$

where $\mathbf{C} = \mathbf{W}_{H+1}\mathbf{W}_H \dots \mathbf{W}_2$.

One can then prove the following theorem regarding the existence of saddle points and local minima.

Theorem 46. *For any depth $H \geq 1$ and for any layer widths and any input-output dimensions, the loss surface has the following properties:*

1. *It is non-convex and non-concave*
2. *Every local minimum is a global minimum*
3. *Every critical point that is not a global minimum is a saddle point*
4. *Let p denote the smallest width in the hidden layers. If rank$(\mathbf{W}_H\mathbf{W}_{H-1} \dots \mathbf{W}_2) \geq p$, the Hessian at any saddle point has at least one negative eigenvalue.*

Proof. Proof omitted. □

To conclude this section, we have seen that the landscape of deep linear networks has a relatively nice structure. In fact, stochastic gradient descent is known to escape saddle points (Ge et al., 2015), which implies it will eventually find a global minimum in deep linear networks. As mentioned earlier, the landscape of non-linear networks does not have similar properties. However, over-parametrizing does again make the landscape easier to optimize. We will discuss this in more detail in the next chapter on NTK (Neural Tangent Kernel).

6.2 Vanishing and Exploding Gradients

In the second part of this lecture, we will discuss the problem of vanishing and exploding gradient. In brief, as we increase the depth of a neural network, the norm of the gradient (computed to optimize the weights of the network) can become increasingly large or small.

6.2.1 Prerequisites

We first review how to compute the gradient of vector-valued and matrix-valued functions. To do so, we consider two examples that are discussed in the book by Deisenroth et al. (2020).

Gradient of a Vector-Valued Function Consider the function $f(\mathbf{x}) = \mathbf{A}\mathbf{x} \in \mathbb{R}^p$, where $\mathbf{x} \in \mathbb{R}^d, \mathbf{A} \in \mathbb{R}^{p \times d}$. Our goal is to compute the gradient $\frac{df}{d\mathbf{x}}$. First, note that the dimension of the gradient $\frac{df}{d\mathbf{x}}$ is $\mathbb{R}^{p \times d}$. Let's compute the partial derivative of f w.r.t. a single x_j. We have

$$f_i(\mathbf{x}) = \sum_{j=1}^{d} A_{ij} x_j \implies \frac{\partial f_i}{\partial x_j} = A_{ij}. \tag{6.26}$$

Collecting all the partial derivatives in the Jacobian, we obtain the following expression for the gradient:

$$\frac{df}{d\mathbf{x}} = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \dots & \frac{\partial f_1}{\partial x_d} \\ & \dots & \\ \frac{\partial f_p}{\partial x_1} & \dots & \frac{\partial f_p}{\partial x_d} \end{pmatrix} = \begin{pmatrix} A_{11} & \dots & A_{1d} \\ & \dots & \\ A_{p1} & \dots & A_{pd} \end{pmatrix} = \mathbf{A} \in \mathbb{R}^{p \times d}. \tag{6.27}$$

Gradient of a Matrix-Valued Function Consider the function $f(\mathbf{x}) = \mathbf{A}\mathbf{x} \in \mathbb{R}^p$, where $\mathbf{x} \in \mathbb{R}^d, \mathbf{A} \in \mathbb{R}^{p \times d}$. Our goal is to compute the gradient $\frac{df}{d\mathbf{A}}$. First, note that the dimension of the gradient $\nabla f := \frac{df}{d\mathbf{A}}$ is $\mathbb{R}^{p \times (p \times d)}$. Let's compute the partial derivative of f w.r.t. a single x_j. We have

$$f_i(\mathbf{x}) = \sum_{j=1}^{d} A_{ij} x_j \implies \frac{\partial f_i}{\partial A_{iq}} = x_q. \tag{6.28}$$

Collecting all the partial derivatives, we can compute partial derivative of f_i

w.r.t. the i-th row of $\mathbf{A}$:

$$\frac{\partial f_i}{\partial A_{i,:}} = \mathbf{x}^\top \in \mathbb{R}^{1\times 1\times d}$$
$$\frac{\partial f_i}{\partial A_{k\neq i,:}} = \mathbf{0}^\top \in \mathbb{R}^{1\times 1\times d}. \tag{6.29}$$

Stacking the partial derivatives, we obtain the gradient of f_i w.r.t. $\mathbf{A}$:

$$\frac{\partial f_i}{\partial \mathbf{A}} = \begin{pmatrix} \mathbf{0}^\top \\ \dots \\ \mathbf{0}^\top \\ \mathbf{x}^\top \\ \mathbf{0}^\top \\ \dots \\ \mathbf{0}^\top \end{pmatrix} \in \mathbb{R}^{1\times(p\times d)}. \tag{6.30}$$

One can use the Kronecker product notation $\otimes$ to write the total derivative of f w.r.t. $\mathbf{A}$ as

$$\frac{\partial f}{\partial \mathbf{A}} = \mathbf{x}^\top \otimes \mathbf{I}. \tag{6.31}$$

Auxiliary proposition We will need the following proposition.

Proposition 47. *Let* $\mathbf{x}, \mathbf{y} \in \mathbb{R}^d$. *Then*

$$\|\mathbf{x}+\mathbf{y}\|_2^2 \leq 2\|\mathbf{x}\|_2^2 + 2\|\mathbf{y}\|_2^2. \tag{6.32}$$

Analogously, for $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{d\times m}$, *we have*

$$\|\mathbf{A}+\mathbf{B}\|_F^2 \leq 2\|\mathbf{A}\|_F^2 + 2\|\mathbf{B}\|_F^2. \tag{6.33}$$

Proof. First, we need a simple inequality, for $a, b \in \mathbb{R}$,

$$a^2 + b^2 \geq 2ab.$$

This can be derived from the fact that $(a-b)^2 \geq 0 \implies a^2 + b^2 - 2ab \geq 0$. We can generalize this inequality to d-dimensional vectors $\mathbf{x}, \mathbf{y} \in \mathbb{R}^d$:

$$\|\mathbf{x}\|^2 + \|\mathbf{y}\|^2 = \sum_{i=1}^d x_i^2 + y_i^2 \geq 2\sum_{i=1}^d x_i y_i = 2\mathbf{x}^\top \mathbf{y}$$

Using this inequality, we get

$$\begin{aligned}\|\mathbf{x}+\mathbf{y}\|^2 &= \|\mathbf{x}\|^2 + \|\mathbf{y}\|^2 + 2\mathbf{x}^\top\mathbf{y} \\ &\leq 2(\|\mathbf{x}\|^2 + \|\mathbf{y}\|^2).\end{aligned}$$

The proof of the second equality $\|\mathbf{A}+\mathbf{B}\|_F^2 \leq 2\|\mathbf{A}\|_F^2 + 2\|\mathbf{B}\|_F^2$ is left to the reader. □

6.2.2 Setting

To make things simpler, we consider a regression problem with a single datapoint $\mathbf{x} \in \mathbb{R}^d$ and a corresponding target $\mathbf{y} \in \mathbb{R}^d$.

A deep linear network with L layers represents a linear function as a product of weight matrices, i.e.

$$\hat{\mathbf{y}} := F(\mathbf{x}) = \mathbf{W}^{L:1}\mathbf{x}, \quad \mathbf{W}^{L:1} = \mathbf{W}^L \cdots \mathbf{W}^1, \quad \mathbf{W}^k \in \mathbb{R}^{d\times d} \tag{6.34}$$

where for simplicity we consider a fixed layer width d. We are interested in the case of random weight matrices, e.g. at initialization, in particular we want to calculate or bound the expected magnitude of derivatives. We will focus on the squared loss, where given a single target $\mathbf{y}$,

$$\ell_{\mathbf{x},\mathbf{y}}(\hat{\mathbf{y}}) = \frac{1}{2}\|\mathbf{y}-\hat{\mathbf{y}}\|^2, \quad \frac{\partial \ell_{\mathbf{x},\mathbf{y}}}{\partial \hat{\mathbf{y}}} = \hat{\mathbf{y}} - \mathbf{y} = \mathbf{W}^{L:1}\mathbf{x} - \mathbf{y} =: \boldsymbol{\delta} \tag{6.35}$$

and where we get the gradient map

$$\frac{\partial \ell}{\partial \mathbf{W}^k} = \underbrace{[\mathbf{W}^{k+1:L}\boldsymbol{\delta}]}_{\text{backward}} \cdot \underbrace{[\mathbf{W}^{k-1:1}\mathbf{x}]^\top}_{\text{forward}} = \mathbf{W}^{k+1:L}[\mathbf{W}^{L:1}\mathbf{x}\mathbf{x}^\top - \mathbf{y}\mathbf{x}^\top]\mathbf{W}^{1:k-1}, \tag{6.36}$$

with $\mathbf{W}^{k+1:L} := \left(\mathbf{W}^{k+1}\right)^\top \cdots \left(\mathbf{W}^L\right)^\top$.

Detailed calculation of the gradient $\frac{\partial \ell}{\partial \mathbf{W}^k}$

Step 1: the chain rule Using the chain rule, we can express the gradient as:

$$\frac{\partial \ell}{\partial \mathbf{W}^k} = \frac{\partial \ell}{\partial \hat{\mathbf{y}}}\frac{\partial \hat{\mathbf{y}}}{\partial \mathbf{W}^k}, \qquad \text{where} \quad \frac{\partial \ell}{\partial \hat{\mathbf{y}}} = \boldsymbol{\delta}.$$

Step 2: Calculate $\frac{\partial \hat{\mathbf{y}}}{\partial \mathbf{W}^k}$

Recall the network function:

$$\hat{\mathbf{y}} = \mathbf{W}^{L:1}\mathbf{x} = \mathbf{W}^L\mathbf{W}^{L-1}\cdots\mathbf{W}^1\mathbf{x}.$$

Notice that the weight matrix $\mathbf{W}^k$ appears in the middle of the product:

$$\hat{\mathbf{y}} = \mathbf{W}^L \cdots \mathbf{W}^{k+1}\mathbf{W}^k\mathbf{W}^{k-1} \cdots \mathbf{W}^1\mathbf{x}$$

Now take the derivative of $\hat{\mathbf{y}}$ with respect to $\mathbf{W}^k$:

$$\frac{\partial \hat{\mathbf{y}}}{\partial \mathbf{W}^k} = \mathbf{W}^L \cdots \mathbf{W}^{k+1} \cdot \frac{\partial(\mathbf{W}^k\mathbf{z})}{\partial \mathbf{W}^k},$$

where $\mathbf{z} = \mathbf{W}^{k-1} \cdots \mathbf{W}^1\mathbf{x}$.
The derivative of the matrix-vector product $\mathbf{W}^k\mathbf{z}$ with respect to $\mathbf{W}^k$ is simply $\mathbf{z}^\top \otimes \mathbf{I}$, leading to:

$$\frac{\partial \hat{\mathbf{y}}}{\partial \mathbf{W}^k} = \mathbf{W}^{L:k+1}\mathbf{z}^\top \otimes \mathbf{I},$$

where $\mathbf{W}^{k+1:L} = (\mathbf{W}^{k+1})^\top \cdots (\mathbf{W}^L)^\top$ and $\mathbf{W}^{k-1:1} = \mathbf{W}^{k-1} \cdots \mathbf{W}^1$.
Step 3: Combine the Results
Now, combining the results from Steps 2 and 3, we get:

$$\frac{\partial \ell}{\partial \mathbf{W}^k} = \boldsymbol{\delta} \cdot \frac{\partial \hat{\mathbf{y}}}{\partial \mathbf{W}^k}.$$

Substituting $\boldsymbol{\delta} = \mathbf{W}^{L:1}\mathbf{x} - \mathbf{y}$ (and using properties of the Kronecker product), we can express this as:

$$\frac{\partial \ell}{\partial \mathbf{W}^k} = \mathbf{W}^{k+1:L}[\mathbf{W}^{L:1}\mathbf{x}\mathbf{z}^\top - \mathbf{y}\mathbf{x}^\top]\mathbf{W}^{1:k-1}.$$

6.2.3 Gradient Norm

Our goal is to examine the Frobenius norm of the gradient, $\left\|\frac{\partial \ell}{\partial \mathbf{W}^k}\right\|_F^2$, at initialization to determine the optimal strategy for initializing the weight matrices. Using Proposition 47, we can bound the norm of the gradient as follows,

$$\begin{aligned}\left\|\frac{\partial \ell}{\partial \mathbf{W}^k}\right\|_F^2 &= \|\mathbf{W}^{k+1:L}\mathbf{W}^{L:1}\mathbf{x}\mathbf{x}^\top\mathbf{W}^{1:k-1} - \mathbf{W}^{k+1:L}\mathbf{y}\mathbf{x}^\top\mathbf{W}^{1:k-1}\|_F^2 \\ &\leq 2\|\mathbf{W}^{k+1:L}\mathbf{W}^{L:1}\mathbf{x}\mathbf{x}^\top\mathbf{W}^{1:k-1}\|_F^2 + 2\|\mathbf{W}^{k+1:L}\mathbf{y}\mathbf{x}^\top\mathbf{W}^{1:k-1}\|_F^2.\end{aligned}$$

We will then bound each of the two terms that appear in the above upper bound. To do so, we will make use of the following simple lemma.

Lemma 48. *Let* $\mathbf{W}$ *be an i.i.d. random matrix with zero mean entries that have variance* σ^2 *and* $\mathbf{A}, \mathbf{B}$ *are arbitrary matrices, then*

$$\mathbb{E}\|\mathbf{AWB}\|_F^2 = \sigma^2 \|\mathbf{A}\|_F^2 \|\mathbf{B}\|_F^2.$$

Moreover, if $\mathbf{A}, \mathbf{B}$ *are stochastic but* $\{\mathbf{A}, \mathbf{B}, \mathbf{W}\}$ *uncorrelated, then*

$$\mathbb{E}\|\mathbf{AWB}\|_F^2 = \sigma^2 \cdot \mathbb{E}\|\mathbf{A}\|_F^2 \cdot \mathbb{E}\|\mathbf{B}\|_F^2.$$

Proof. Using the definition of the Frobenius norm $\|\mathbf{A}\|_F^2 = \operatorname{tr}(\mathbf{A}^\top \mathbf{A})$, we get

$$\begin{aligned}
\mathbb{E}\|\mathbf{AWB}\|_F^2 &= \mathbb{E}\operatorname{tr}(\mathbf{AWBB}^\top \mathbf{W}^\top \mathbf{A}^\top) \\
&= \sum_{i,j,k,l,m,n} a_{ij} b_{kl} b_{ml} a_{in} \mathbb{E}[w_{jk} w_{nm}] \\
&\overset{(i)}{=} \sum_{i,j,k,l} a_{ij}^2 b_{kl}^2 \mathbb{E}[w_{jk}^2] \\
&= \sigma^2 \sum_{i,j,k,l} a_{ij}^2 b_{kl}^2,
\end{aligned}$$

where (i) is implied by the fact that $\mathbb{E}[w_{ij} w_{kl}] = 0$ for $(i,j) \neq (k,l)$.

The statement in the stochastic case follows from the law of the unconscious statistician. □

Corollary 49. *Under the same conditions as in Lemma 48,*

$$\mathbb{E}\|\mathbf{W}^{t:1}\mathbf{x}\|^2 = (d\sigma^2)^t \|\mathbf{x}\|^2.$$

Proof. By applying the above lemma t times, noting that

$$\mathbb{E}\|\mathbf{W}^{t:1}\mathbf{x}\|_2^2 = \mathbb{E}\|\mathbf{I}\,\mathbf{W}^t \mathbf{W}^{t-1:1}\mathbf{x}\|_F^2 = \sigma^2 \|\mathbf{I}\|_F^2 \mathbb{E}\|\mathbf{W}^{t-1}\mathbf{x}\|_2^2 = d\sigma^2 \mathbb{E}\|\mathbf{W}^{t-1}\mathbf{x}\|_2^2 . \quad (6.37)$$

□

Lemma 50 (Statistics after multiplication with a random matrix)**.** *Let* $\mathbf{W}$ *be an iid random matrix, with zero mean entries that have variance* σ^2 *and kurtosis* κ*. Let* $\boldsymbol{\xi} \in \mathbb{R}^d$ *be an arbitrary random vector (not necessarily symmetric or with uncorrelated squared entries). Then*

$$\mathbb{E}\|\mathbf{W}\boldsymbol{\xi}\|_2^4 = d(d+2)\sigma^4 \mathbb{E}\|\boldsymbol{\xi}\|_2^4 + (\kappa - 3) d\sigma^4 \mathbb{E}\|\boldsymbol{\xi}\|_4^4.$$

Proof.

$$\|\mathbf{W}\boldsymbol{\xi}\|_2^4 = \Big(\sum_i \Big(\sum_r w_{ir}\xi_r\Big)^2\Big)^2 \tag{6.38}$$

$$= \sum_{i,j}\sum_r w_{ir}\xi_r \sum_s w_{is}\xi_s \sum_u w_{ju}\xi_u \sum_v w_{jv}\xi_v. \tag{6.39}$$

Taking expectations, yields

$$\frac{\mathbb{E}\|\mathbf{W}\boldsymbol{\xi}\|_2^4}{\sigma^4} = \underbrace{\sum_{i\neq j}}_{d(d-1)} \underbrace{\mathbb{E}\left[\Big(\sum_{r=s}\xi_r^2\Big)\Big(\sum_{u=v}\xi_u^2\Big)\right]}_{\mathbb{E}\|\boldsymbol{\xi}\|_2^4} + 3\underbrace{\sum_{i=j}}_{3d}\underbrace{\sum_{r\neq s}\mathbb{E}\left[\xi_r^2\xi_s^2\right]}_{\mathbb{E}\|\boldsymbol{\xi}\|_2^4-\mathbb{E}\|\boldsymbol{\xi}\|_4^4} + \kappa\underbrace{\sum_{i=j}}_{\kappa d}\underbrace{\sum_{r=s}\mathbb{E}[\xi_r^4]}_{\mathbb{E}\|\boldsymbol{\xi}\|_4^4} \tag{6.40}$$

$$= d(d+2)\mathbb{E}\|\boldsymbol{\xi}\|_2^4 + (\kappa-3)d\ \mathbb{E}\|\boldsymbol{\xi}\|_4^4.$$

The factor 3 appears because, if 4 indices are paired in groups of two, we have a total of 3 disjoint choices: $\{i=j, u=v\}$, $\{i=u, j=v\}$, $\{i=v, j=u\}$. □

We derive an intermediary lemma before obtaining our main result.

Lemma 51. *Let $\mathbf{W}^k$ be random matrices with iid entries such that $\mathbb{E}[w_{ij}]=0$ and $\mathbb{E}[w_{ij}^2]=\sigma^2$. For a fixed input/output pair $(\mathbf{x},\mathbf{y})$ one has*

$$\mathbb{E}\left\|\frac{\partial \ell}{\partial \mathbf{W}^k}\right\|_F^2 \leq 2\sigma^2\,\mathbb{E}\|\mathbf{W}^{k-1:1}\mathbf{x}\|_2^4\,\mathbb{E}\|\mathbf{W}^{k+1:L}\mathbf{W}^{L:k+1}\|_F^2 + 2(d\sigma^2)^{L-1}\|\mathbf{x}\|^2\|\mathbf{y}\|^2.$$

Proof. Recall the expression of the gradient derived earlier:

$$\frac{\partial \ell}{\partial \mathbf{W}^k} = \mathbf{W}^{k+1:L}[\mathbf{W}^{L:1}\mathbf{x}\mathbf{x}^\top - \mathbf{y}\mathbf{x}^\top]\mathbf{W}^{1:k-1}. \tag{6.41}$$

Using Proposition 47 [1], we can bound the norm of the gradient as follows,

$$\begin{aligned}\left\|\frac{\partial \ell}{\partial \mathbf{W}^k}\right\|_F^2 &= \|\mathbf{W}^{k+1:L}\mathbf{W}^{L:1}\mathbf{x}\mathbf{x}^\top\mathbf{W}^{1:k-1} - \mathbf{W}^{k+1:L}\mathbf{y}\mathbf{x}^\top\mathbf{W}^{1:k-1}\|_F^2 \\ &\leq 2\|\mathbf{W}^{k+1:L}\mathbf{W}^{L:1}\mathbf{x}\mathbf{x}^\top\mathbf{W}^{1:k-1}\|_F^2 + 2\|\mathbf{W}^{k+1:L}\mathbf{y}\mathbf{x}^\top\mathbf{W}^{1:k-1}\|_F^2.\end{aligned}$$

Let us consider the term involving the target $\mathbf{y}$. We can exploit independence and the fact that $\|\mathbf{u}\mathbf{v}^\top\|_F^2 = \|\mathbf{u}\|^2\|\mathbf{v}\|^2$ to get

$$\mathbb{E}\|\mathbf{W}^{k+1:L}\mathbf{y}\mathbf{x}^\top\mathbf{W}^{1:k-1}\|_F^2 = \mathbb{E}\|\mathbf{W}^{k+1:L}\mathbf{y}\|^2 \cdot \mathbb{E}\|\mathbf{W}^{k-1:1}\mathbf{x}\|^2 \tag{6.42}$$

[1] Alternatively, one could use the sub-additivity of the Frobenius norm, i.e. $\|\mathbf{A}+\mathbf{B}\|_F \leq \|\mathbf{A}\|_F + \|\mathbf{B}\|_F$

By applying Corollary 49 this reduces to the second summand in the claim:

$$\mathbb{E}\|\mathbf{W}^{k+1:L}\mathbf{y}\mathbf{x}^\top\mathbf{W}^{1:k-1}\|_F^2 \leq (d\sigma^2)^{L-k}\|\mathbf{x}\|^2(d\sigma^2)^{k-1}\|\mathbf{y}\|^2. \tag{6.43}$$

For the $\mathbf{x}\mathbf{x}^\top$ term, we can apply Lemma 48 to get

$$\begin{aligned}\mathbb{E}\|\mathbf{W}^{k+1:L}\mathbf{W}^{L:k+1}\underbrace{\mathbf{W}^k}_{*}\mathbf{W}^{k-1:1}\mathbf{x}\mathbf{x}^\top\mathbf{W}^{1:k-1}\|_F^2 \\ = \sigma^2\,\mathbb{E}\|\mathbf{W}^{k+1:L}\mathbf{W}^{L:k+1}\|_F^2\,\mathbb{E}\|\mathbf{W}^{k-1:1}\mathbf{x}\mathbf{x}^\top\mathbf{W}^{1:k-1}\|_F^2\,.\end{aligned} \tag{6.44}$$

Finally note that $\|\mathbf{u}\mathbf{u}^\top\|_F^2 = \|\mathbf{u}\|_2^4$ and with $\mathbf{u} = \mathbf{W}^{k-1}\mathbf{x}$ this gives

$$\mathbb{E}\|\mathbf{W}^{k-1:1}\mathbf{x}\mathbf{x}^\top\mathbf{W}^{1:k-1}\|_F^2 = \mathbb{E}\|\mathbf{W}^{k-1:1}\mathbf{x}\|_2^4 \tag{6.45}$$

Collecting all terms yields the claimed result. □

Main result about asymptotic convergence We are now ready to state our main result that bound the expected square norm of the gradient.

Theorem 52. *Let $\mathbf{W}^k$ be Gaussian matrices with iid entries such that $\mathbb{E}[w_{ij}] = 0$ and $\mathbb{E}[w_{ij}^2] = \sigma^2$. Define the following sufficient statistics with regard to the data distribution, $\rho = \|\mathbf{x}\|^4$, $\gamma = \|\mathbf{x}\|^2\|\mathbf{y}\|^2$. Then:*

$$\mathbb{E}\left\|\frac{\partial \ell}{\partial \mathbf{W}^k}\right\|_F^2 \leq 2\frac{\rho}{3\sigma^2}[\sigma^4 d(d+2)]^L + 2\gamma(\sigma^2 d)^{L-1}.$$

Proof. From the previous lemma, we have

$$\mathbb{E}\left\|\frac{\partial \ell}{\partial \mathbf{W}^k}\right\|_F^2 \leq 2\sigma^2\,\mathbb{E}\|\mathbf{W}^{k-1:1}\mathbf{x}\|_2^4\,\mathbb{E}\|\mathbf{W}^{k+1:L}\mathbf{W}^{L:k+1}\|_F^2 + 2(d\sigma^2)^{L-1}\|\mathbf{x}\|^2\|\mathbf{y}\|^2.$$

We need to bound the first term. First, using Lemma 48, we have

$$\mathbb{E}\|\mathbf{W}^{1:m}\mathbf{W}^{m:1}\|_F^2 \leq \frac{1}{3}\sigma^{4m}(d(d+2))^{m+1}.$$

Using Lemma 50 (where $\kappa = 3$ in the Gaussian case), we also have

$$\mathbb{E}\|\mathbf{W}^{m:1}\mathbf{x}\|_2^4 = \sigma^{4m}d^m(d+2)^m\|\mathbf{x}\|^4. \tag{6.46}$$

Combining the two above inequalities yields the desired result.

□

Note that the exponents of the two contributions differ. For small weight initializations, the gradient is dominated by the term involving $\mathbf{y}$, while for large $\sigma^4 > d(d+2)$ the term involving $\hat{\mathbf{y}}$ dominates (assuming $\rho \approx \gamma$).

Let's consider the setting where d is very large, then the dominating term in the theorem is of the form $(\sigma^4 d^2)^L$. This means that in order to stabilize the gradient for large depths L, we require $\sigma = \frac{1}{\sqrt{d}}$. This is known as Xavier initialization, initially proposed by Glorot and Bengio (2010).

6.3 Additional Material: the Case of the Cross-entropy Loss

Setting In this section, we consider a linear neural network with L layers, which are parametrized by weight matrices $\mathbf{W}^l \in \mathbb{R}^{d_l \times d_{l-1}}$ (where $d_L = k$ and $d_0 = d$) trained using the cross-entropy loss. For an input $\mathbf{x} \in \mathbb{R}^d$ the forward propagation is defined as:

$$\begin{aligned}
&\mathbf{a}^0 = \mathbf{x} \in \mathbb{R}^d, \\
&\mathbf{a}^l = \mathbf{W}^l \, \mathbf{a}^{l-1} \quad \text{for } l = 1, \dots, L-1, \\
&\mathbf{z} \equiv \mathbf{a}^L = \mathbf{W}^L \, \mathbf{a}^{L-1} = \underbrace{\mathbf{W}^L}_{d_L \times d_{L-1}} \underbrace{\mathbf{W}^{L-1}}_{d_{L-1} \times d_{L-2}} \cdots \underbrace{\mathbf{W}^1}_{d_1 \times d} \mathbf{x} \in \mathbb{R}^k,
\end{aligned}$$

where k is the number of classes.

The output of the network $\mathbf{z} \in \mathbb{R}^k$ is then passed through the softmax:

$$\mathbf{p} = \mathrm{softmax}(\mathbf{z}) \in \mathbb{R}^k.$$

We recall that the softmax function is defined componentwise for each $i = 1, \dots, k$ as follows:

$$p_i = \frac{e^{z_i}}{\sum_{j=1}^k e^{z_j}},$$

so that the output vector $\mathbf{p} \in \mathbb{R}^k$ satisfies $\sum_{i=1}^k p_i = 1$ and $p_i > 0$ for all i. This function maps the raw scores (logits) $\mathbf{z}$ into a probability distribution over the k classes.

Finally, the cross-entropy loss for a one-hot target vector $\mathbf{y} \in \mathbb{R}^k$ is given by:

$$\ell(\mathbf{z}) = -\sum_{i=1}^k y_i \log p_i.$$

Output Layer For the output layer, we have seen earlier that the gradient of the loss is equal to $\frac{\partial \ell}{\partial \mathbf{z}} = \mathbf{p} - \mathbf{y}$. For simplicity, we will use the shortcut notation

$$\delta^L \triangleq \frac{\partial \ell}{\partial \mathbf{z}} = \mathbf{p} - \mathbf{y}.$$

Propagation of the error backward For a hidden layer l (with $1 \leq l < L$), the activation is given by:

$$\mathbf{a}^{l+1} = \mathbf{W}^{l+1} \, \mathbf{a}^l.$$

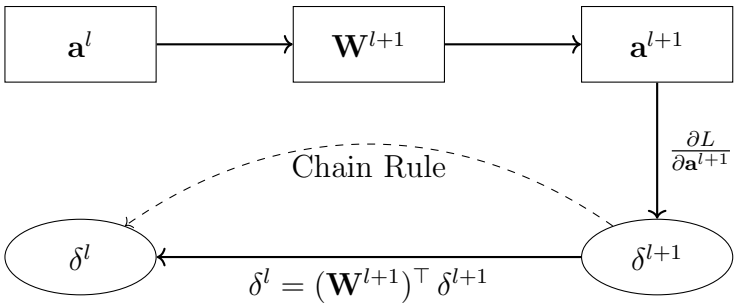


Figure 6.2: Illustration of the forward and backward pass.

As illustrated in Figure 6.2, we can apply the chain rule to calculate the derivative of the loss (also called error signal) at layer l as:

$$\delta^l \triangleq \frac{\partial \ell}{\partial \mathbf{a}^l} = \frac{\partial \ell}{\partial \mathbf{a}^{l+1}} \frac{\partial \mathbf{a}^{l+1}}{\partial \mathbf{a}^l}.$$

Since

$$\frac{\partial \mathbf{a}^{l+1}}{\partial \mathbf{a}^l} = \mathbf{W}^{l+1},$$

we have:

$$\delta^l = \left(\mathbf{W}^{l+1}\right)^\top \delta^{l+1}.$$

Recursively, starting from the output layer we obtain:

$$\delta^l = \left(\prod_{k=l+1}^{L} (\mathbf{W}^k)^\top \right) (\mathbf{p} - \mathbf{y}),$$

where the product indicates the ordered multiplication:

$$\mathbf{W}^{l+1:L} \triangleq \underbrace{(\mathbf{W}^{l+1})^\top}_{d_l \times d_{l+1}} \underbrace{(\mathbf{W}^{l+2})^\top}_{d_{l+1} \times d_{l+2}} \cdots \underbrace{(\mathbf{W}^{L})^\top}_{d_{L-1} \times d_L} \in \mathbb{R}^{d_l \times d_L}$$

Gradient with respect to $\mathbf{W}^l$ Since the activation at layer l is:

$$\mathbf{a}^l = \mathbf{W}^l \, \mathbf{a}^{l-1},$$

the gradient with respect to $\mathbf{W}^l$ is given by:

$$\frac{\partial L}{\partial \mathbf{W}^l} = \delta^l \, (\mathbf{a}^{l-1})^\top.$$

Thus, substituting the expression for δ^l, we have:

$$\begin{aligned}\frac{\partial L}{\partial \mathbf{W}^l} &= \left(\prod_{k=l+1}^{L} (\mathbf{W}^k)^\top\right)(\mathbf{p}-\mathbf{y})\,(\mathbf{a}^{l-1})^\top \\ &= \left(\prod_{k=l+1}^{L} (\mathbf{W}^k)^\top\right)(\mathbf{p}-\mathbf{y})\left(\prod_{k=1}^{l-1} \mathbf{W}^k\,\mathbf{x}\right)^\top \\ &= \underbrace{\mathbf{W}^{l+1:L}}_{d_l\times k}\underbrace{(\mathbf{p}-\mathbf{y})\,\mathbf{x}^\top}_{k\times d}\underbrace{\mathbf{W}^{1:l-1}}_{d\times d_{l-1}} \in \mathbb{R}^{d_l\times d_{l-1}}.\end{aligned}$$

One can then further proceed to bound the norm of the gradient using Corollary 49, which would also yield to conclude that $\sigma = \frac{1}{\sqrt{d}}$ (assuming $d_k = d$ for all layers k) is required to stabilize the gradients.

6.4 Exercise: Optimization Landscape of Neural Networks

Problem 1 (Matrix Completion):

We consider the problem of matrix sensing where we have a model parametrized by $\mathbf{W} \in \mathbb{R}^{d\times d}$. We observe a set of linear measurements of the form $\langle \mathbf{W}, \mathbf{A}_n \rangle_F$ where $[\mathbf{A}_n]_{ij} \sim \mathcal{N}(0,1)$. We will further assume that the data is labeled by a matrix sensing model parameterized by $\mathbf{W}^* \in \mathbb{R}^{d\times d}$ (this is sometimes called "planted model" in the literature).
We will study the dynamics of this model trained with gradient flow on a squared loss. As we will soon see, this setting is related to the problem of training a deep linear network. In order to simulate depth, we will consider the matrix $\mathbf{W}$ as the product of a set of square matrices $\mathbf{W}_i \in \mathbb{R}^{d\times d}$, i.e. $\mathbf{W} = \mathbf{W}_L \dots \mathbf{W}_1$.
The objective function is given by

$$\mathcal{L}(\mathbf{W}) = \frac{1}{2L}\mathbb{E}_{\mathbf{A}}\langle \mathbf{W} - \mathbf{W}^*, \mathbf{A}\rangle_F^2 = \frac{1}{2L}\|\mathbf{W}_L \dots \mathbf{W}_1 - \mathbf{W}^*\|_F^2.$$

We will use gradient flow to optimize the parameters and denote by $\mathbf{W}_k(t)$ the matrices $\mathbf{W}_k$ at time t.

a) Denote $\mathbf{W}_{j:k}^\top = \prod_{i=j}^k \mathbf{W}_i^\top = \mathbf{W}_j^\top \mathbf{W}_{j+1}^\top \cdots \mathbf{W}_k^\top$. The partial gradient of $\mathcal{L}$ with respect to $\mathbf{W}_k$ (where $k = 1, \dots, L$) is

$$\frac{\partial \mathcal{L}(\mathbf{W})}{\partial \mathbf{W}_k} = \frac{1}{L}\mathbf{W}_{k+1:L}^\top (\mathbf{W} - \mathbf{W}^*)\mathbf{W}_{1:k-1}^\top.$$

What is the gradient flow equation for the matrix $\mathbf{W}_k$?

b) Prove that for all $t \geq 0$ and $k = 1, ..., L-1$:

$$\mathbf{W}_{k+1}^\top(t)\dot{\mathbf{W}}_{k+1}(t) = \dot{\mathbf{W}}_k(t)\mathbf{W}_k^\top(t). \tag{6.47}$$

And hence

$$\mathbf{W}_{k+1}^\top(t)\mathbf{W}_{k+1}(t) = \mathbf{W}_k(t)\mathbf{W}_k^\top(t). \tag{6.48}$$

c) For any $t \geq 0$ and $k = 1, ..., L$, assume the singular values of $\mathbf{W}_k$ are all distinct and indexed in strictly decreasing order: $\sigma_1 > ... > \sigma_d > 0$, and let $\mathbf{W}_k = \mathbf{U}_k \mathbf{\Sigma}_k \mathbf{V}_k^\top$ be its (unique) singular value decomposition (SVD). Show that

$$\mathbf{\Sigma}_{k+1} = \mathbf{\Sigma}_k, \qquad \mathbf{U}_k = \mathbf{V}_{k+1},$$

for any $t \geq 0$ and $k = 1, ..., L$.

d) Denote $\mathbf{\Sigma} = \mathbf{\Sigma}_1 = \cdots = \mathbf{\Sigma}_L \in \mathbb{R}^{d\times d}$. Now prove that the gradient flow equation can be written as:

$$\dot{\mathbf{W}}_k = \frac{1}{L}\mathbf{V}_{k+1}\mathbf{\Sigma}^{L-k}\mathbf{U}_L^\top \left(\mathbf{W}^* - \mathbf{U}_L\mathbf{\Sigma}^L\mathbf{V}_1^\top\right)\mathbf{V}_1\mathbf{\Sigma}^{k-1}\mathbf{V}_{k-1}^\top,$$

for all $k = 1, .., L$.

e) By the product rule $\dot{\mathbf{W}} = \sum_{k=1}^L \mathbf{W}_{L:k+1}\dot{\mathbf{W}}_k\mathbf{W}_{k-1:1}$, show that

$$\dot{\mathbf{W}} = \frac{1}{L}\sum_{k=1}^{L}\mathbf{U}_L\mathbf{\Sigma}^{2L-2k}\mathbf{U}_L^\top \left(\mathbf{W}^* - \mathbf{U}_L\mathbf{\Sigma}^L\mathbf{V}_1^\top\right)\mathbf{V}_1\mathbf{\Sigma}^{2k-2}\mathbf{V}_1^\top.$$

f) Alternatively, show that the gradient flow can be expressed solely in terms of $\mathbf{W}$:

$$\dot{\mathbf{W}} = \frac{1}{L}\sum_{k=1}^{L}\left[\mathbf{W}\mathbf{W}^\top\right]^{\frac{L-k}{L}}\left(\mathbf{W}^* - \mathbf{W}\right)\left[\mathbf{W}^\top\mathbf{W}\right]^{\frac{k-1}{L}}.$$

Problem 2 (Network near initialization):
Given an input vector $\mathbf{x} \in \mathbb{R}^d$, consider a shallow neural network defined by

$$f(\mathbf{W}) := \sum_{j=1}^{m} a_j \sigma(\mathbf{w}_j^\top \mathbf{x}), \qquad \mathbf{W} := \begin{bmatrix} \leftarrow \mathbf{w}_1^\top \rightarrow \\ \vdots \\ \leftarrow \mathbf{w}_m^\top \rightarrow \end{bmatrix} \in \mathbb{R}^{m \times d},$$

where σ is an activation function, $\mathbf{a} \in \mathbb{R}^m$ is a fixed (not trainable) vector of weights initialized such that $a_j \overset{\text{i.i.d.}}{\sim} \text{Unif}\left(\pm \frac{1}{\sqrt{m}}\right)$ [2], and $\mathbf{W} \in \mathbb{R}^{m \times d}$ are trainable weights.

We will consider the linearization of the function f around some initial weights $\mathbf{W}_0$ defined by

$$f_0(\mathbf{W}) = f(\mathbf{W}_0) + \langle \nabla f(\mathbf{W}_0), \mathbf{W} - \mathbf{W}_0 \rangle_F \,, \tag{6.49}$$

where $\langle \mathbf{A}, \mathbf{B} \rangle_F = \text{tr}(\mathbf{A}^\top \mathbf{B})$ is the Frobenius inner product.

a) Show that

$$\nabla f(\mathbf{W}) = \mathbf{D}\mathbf{a}\mathbf{x}^\top, \text{ where } \mathbf{D} = \text{diag}(\sigma'(\mathbf{w}_i^\top \mathbf{x})) = \begin{pmatrix} \sigma'(\mathbf{w}_1^\top \mathbf{x}) & & & \\ & \sigma'(\mathbf{w}_2^\top \mathbf{x}) & & \\ & & \cdots & \\ & & & \sigma'(\mathbf{w}_m^\top \mathbf{x}) \end{pmatrix}.$$

b) Show that

$$\langle \nabla f(\mathbf{W}_0), \mathbf{W} - \mathbf{W}_0 \rangle_F = \sum_{j=1}^{m} a_j \sigma'(\mathbf{w}_{0,j}^\top \mathbf{x})(\mathbf{w}_j - \mathbf{w}_{0,j})^\top \mathbf{x},$$

where $\mathbf{W}_0 = \begin{bmatrix} \leftarrow \mathbf{w}_{0,1}^\top \rightarrow \\ \vdots \\ \leftarrow \mathbf{w}_{0,m}^\top \rightarrow \end{bmatrix}$ denotes the weight at initialization.

c) Now assume the activation function σ is β-smooth, which satisfies:

$$|\sigma(r) - \sigma(s) - \sigma'(s)(r - s)| \leq \frac{\beta (r - s)^2}{2}$$

for all $r, s \in \mathbb{R}$. Show that for any $\mathbf{W} \in \mathbb{R}^{m \times d}$,

$$|f(\mathbf{W}) - f_0(\mathbf{W})| \leq \frac{\beta}{2\sqrt{m}} \|\mathbf{W} - \mathbf{W}_0\|_F^2 \, \|\mathbf{x}\|^2.$$

d) What do you conclude about the role of over-parametrization?

[2] Equivalently, one can define $f(\mathbf{W}) := \frac{1}{\sqrt{m}} \sum_{j=1}^{m} a_j \sigma(\mathbf{w}_j^\top \mathbf{x})$ with $a_j \overset{\text{i.i.d.}}{\sim} \text{Unif}(\pm 1)$.

Chapter 7

Random Feature Regression and Neural Tangent Kernel

Disclaimer 2: Some of the results discussed in these notes are from Nica (2021). Part of these notes were also written with Gregor Bachmann at ETH Zürich.

7.1 Kernels: a brief introduction

Our journey starts with the introduction of a pivotal idea - the concept of a kernel. Let's consider the typical setting in machine learning where we have access to some training data $(\mathbf{x}_i, y_i)_{i=1}^n$ where $\mathbf{x}_i \in \mathcal{X} \subseteq \mathbb{R}^d$ are some features, and y_i are corresponding labels, e.g. $y_i \in \mathbb{R}$ for regression or $y_i \in \{-1, +1\}$ for classification.

The simplest model we can use to classify/regress the data is a linear model where we weight the importance of each feature with a learned parameter vector $\theta \in \mathbb{R}^d$, for instance $f(\theta, \mathbf{x}) = \theta^\top \mathbf{x}$. This model is obviously fine when the data is linearly separable, but what should we do if that's not the case? One solution offered by kernel methods is to use a *feature map* $\phi : \mathbb{R}^d \mapsto \mathcal{H}$ to map the data into a (typically high-dimensional) vector space $\mathcal{H}$ where linear relations exist among the data [1]. This is illustrated in Figure 7.1.

The new feature space given by $\phi(\mathbf{x})$ might be very large (potentially infinite) making it difficult to learn an optimal set of parameters. Instead of directly finding a decision function f in this new space, we will use a similarity metric between the datapoints. This is where *kernels* come into play...

[1] To be more precise (although we will not need this level of detail for this lecture), the space $\mathcal{H}$ is a Hilbert space, i.e. an inner product space that is complete and separable with respect to the norm defined by the inner product associated to the space.

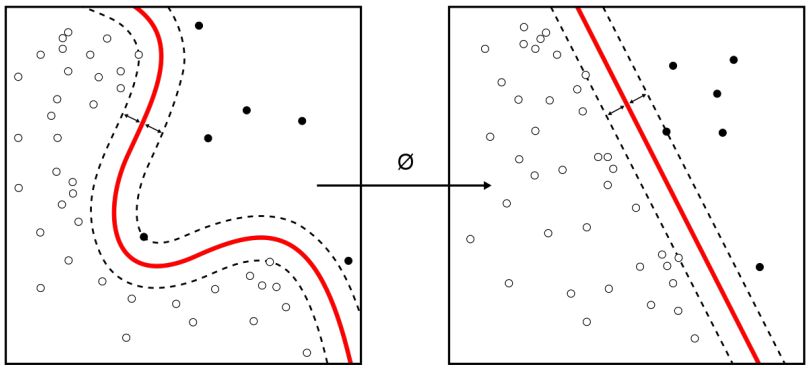


Figure 7.1: (2-D illustration) The white and black circles are datapoints belonging to two different classes. The red lines represent the max-margin decision boundary between the two classes of points. $\phi : \mathbb{R}^d \to \mathcal{H}$ maps features from a finite-dimensional space $\mathbb{R}^d$ to a vector space $\mathcal{H}$ that is typically of much higher dimension. The desired effect is that the datapoints embedded in the new space $\mathcal{H}$ become linearly separable (i.e. the decision boundary in red becomes a straight line). Source: Wikipedia.

Definition 25 (Kernel)**.** *Given two datapoints* $\mathbf{x}, \mathbf{y} \in \mathcal{X}$ *and a map* $\phi : \mathcal{X} \to \mathbb{R}^p$*, a kernel function* K *is defined as:*

$$K(\mathbf{x}, \mathbf{y}) = \langle \phi(\mathbf{x}), \phi(\mathbf{y}) \rangle.$$

Two well-known examples of kernels are polynomial kernels and RBF kernels. For instance, a polynomial kernel can be created in two dimensions using the feature map $\phi : \mathbf{x} = (x_1, x_2) \mapsto \phi(\mathbf{x}) = (x_1^2, x_2^2, \sqrt{2}x_1x_2)$.

7.1.1 Learning with Kernels

Given a feature map $\phi : \mathbb{R}^d \to \mathbb{R}^p$, we study functions of the form

$$f_\theta(\mathbf{x}) = \theta^\top \phi(\mathbf{x})$$

where $\theta \in \mathbb{R}^p$ are the learnable parameters of the model.

We aim to choose $\theta \in \mathbb{R}^p$ such that we minimize the least-squares loss, i.e.

$$L(\theta) = \frac{1}{2} \sum_{i=1}^{n} \left(f_\theta(\mathbf{x}_i) - y_i \right)^2 . \tag{7.1}$$

Representer Theorem The Representer theorem gives a characterization of the optimal solution f_{θ^*} where $\theta^* = \operatorname{argmin}_{\theta \in \mathbb{R}^p} L(\theta)$.

Theorem 53 (Representer theorem)**.** *The Representer theorem states that any function that minimizes a regularized risk functional over a Reproducing Kernel Hilbert Space (RKHS) can be expressed as a linear combination of the kernel functions evaluated at the training points. Specifically, for the loss function $L(\theta)$ defined in Eq.* (7.1)*, the optimal function f_{θ^*} can be expressed as:*

$$f_{\theta^*}(\mathbf{x}) = \sum_{i=1}^{n} \alpha_i K(\mathbf{x}, \mathbf{x}_i),$$

where α_i's are coefficients to be determined.

Closed-Form Solution Next, we will derive a closed-form solution to find $\theta^* = \operatorname{argmin}_{\theta \in \mathbb{R}^p} L(\theta)$. To do so, we define the following notations:

- $\boldsymbol{\phi} \in \mathbb{R}^{n \times p}$ with $\phi_i = \phi(\mathbf{x}_i)$,
- $\mathbf{K}_\mathbf{x} \in \mathbb{R}^n$ with $(\mathbf{K}_\mathbf{x})_i = K(\mathbf{x}, \mathbf{x}_i)$,
- $\mathbf{K} \in \mathbb{R}^{n \times n}$ with $K_{ij} = K(\mathbf{x}_i, \mathbf{x}_j)$.

Note that, using the above notation, the loss function can be written as:

$$L(\theta) = \frac{1}{2} \|\boldsymbol{\phi}\theta - \mathbf{y}\|_2^2,$$

where $\mathbf{y} = (y_i)_{i=1}^n \in \mathbb{R}^n$ is the vector of target values.

By first-order optimality (setting $\nabla_\theta L(\theta) = \boldsymbol{\phi}^\top(\boldsymbol{\phi}\theta - \mathbf{y})$ to zero), we can check that the closed-form solution for the parameters θ^* that minimize the loss is:

$$f_{\theta^*}(\mathbf{x}) = \mathbf{K}_\mathbf{x}^\top \mathbf{K}^{-1} \mathbf{y},$$

where we assume for convenience that $\boldsymbol{K}$ is invertible.

Note that we can show this solution satisfies the Representer theorem as follows. Since $\mathbf{K}_\mathbf{x}$ is the vector of kernel evaluations $K(\mathbf{x}, \mathbf{x}_i)$ and $\mathbf{K}$ is the kernel matrix, we can write:

$$f_{\theta^*}(\mathbf{x}) = \sum_{i=1}^{n} \left(\mathbf{K}^{-1}\mathbf{y}\right)_i K(\mathbf{x}, \mathbf{x}_i).$$

Let $\alpha_i = (\mathbf{K}^{-1}\mathbf{y})_i$. Therefore, we have:

$$f_{\theta^*}(\mathbf{x}) = \sum_{i=1}^{n} \alpha_i K(\mathbf{x}, \mathbf{x}_i).$$

Remark 5 (Efficiency of calculating the solution)**.** *Note that we wrote f_{θ^*} only in terms of $\boldsymbol{K}$, and not directly in terms of $\phi(\mathbf{x})$. This means that we do not need to compute the high-dimensional representation $\phi(\mathbf{x}) \in \mathbb{R}^p$ (as mentioned earlier, p is typically very large, potentially infinite). This is known as the Kernel trick.*

7.1.2 Training Dynamics:

Let's now turn to the goal of learning the parameters θ that minimize the loss defined in Eq. 7.1. To keep things simple, we will consider gradient flow defined as

$$\frac{d}{dt}\theta_t = -\nabla_\theta L(\theta)\big|_{\theta=\theta_t}$$

In essence, this is gradient descent with an infinitesimal learning rate.

We can write down the evolution of the function f_{θ_t} at any timestep $t \geq 0$ (solving above ODE):

$$f_{\theta_t}(\mathbf{x}) = \boldsymbol{K}_{\mathbf{x}}\boldsymbol{K}^{-1}\left(\mathbf{1}_n - e^{-\boldsymbol{K}t}\right)\mathbf{y}$$

where $e^{\mathbf{A}}$ is the matrix exponential. Notice that $f_{\theta_\infty} = f_{\theta^*}$.

Remark 6 (Function-space vs weight-space perspective)**.** *In the following, we will mostly focus on the function-space perspective, i.e. considering the evolution of the function f_θ instead of the evolution of the weights. Doing so will allow us to derive closed-form expressions that can not be so easily derived when considering the weights θ.*

7.2 Random Feature Regression

We first introduce some notations that we will use in these notes.

Notation	Space	Definition
n	$\mathbb{N}$	Number of training datapoints
$\mathbf{x}$	$\mathbb{R}^d$	Datapoint
θ	$\mathbb{R}^p$	Parameters
$\phi(\mathbf{x})$	$\mathbb{R}^d \to \mathbb{R}^p$	Feature map
$\mathbf{X}$	$\mathbb{R}^{n\times d}$	Data matrix
$\mathbf{y}$	$\mathbb{R}^n$	Vector of labels
$L(\theta)$	$\mathbb{R}^p \to \mathbb{R}$	Loss function
K	$\mathbb{R}^p \times \mathbb{R}^p \to \mathbb{R}$	Kernel

Random feature model: We now study a specific feature map given by $\phi_i(\mathbf{x}) = \varphi(\mathbf{w}_i \cdot \mathbf{x})$ where

- $\mathbf{w}_i \in \mathbb{R}^d$, $i = 1, \dots, p$ are independent random variables
- $\varphi : \mathbb{R} \to \mathbb{R}$ is some non-linearity function

An example of such a model is a one-hidden layer neural network where we only train the weights of the top layer.

We then combine the features linearly to construct a model:

$$\boxed{f(\mathbf{x}, \theta) = \frac{1}{\sqrt{p}} \theta^\top \phi(\mathbf{x}),} \tag{model}$$

where $\theta \in \mathbb{R}^p$ are all the model parameters. Note that the $\frac{1}{\sqrt{p}}$ scaling is used to make sure that the sum does not explode when we take the limit $p \to \infty$. In the literature, this is sometimes called the NTK parametrization.

Next, we will see that the kernel $K = \frac{1}{p}\langle \phi(\mathbf{x}), \phi(\mathbf{x}') \rangle$ does two things:

1. It determines the evolution of $f(\mathbf{x}, \theta_t)$ (Theorem 54)
2. Under Gaussian initialization, it determines the distribution of $f(\mathbf{x}, \theta_0)$ (Theorem 55).

7.2.1 Evolution of f

Let $f := f^k$ be the output a neural network with k layers. The following theorem shows that the evolution of the function f over time is controlled by the kernel K.

Theorem 54 (Evolution of f)**.** *For $\mathbf{x} \in \mathbb{R}^d$:*

$$\frac{d}{dt} f(\mathbf{x}, \theta_t) = K(\mathbf{x}, \mathbf{X})(\mathbf{y} - f(\mathbf{X}, \theta_t)),$$

where $K(\mathbf{x}, \mathbf{X}) \in \mathbb{R}^{1 \times n}$ and $(\mathbf{y} - f(\mathbf{X}, \theta_t)) \in \mathbb{R}^{n \times 1}$.

The solution of this differential equation is

$$f(\mathbf{x}, \theta_\infty) - f(\mathbf{x}, \theta_0) = K(\mathbf{x}, \mathbf{X}) K(\mathbf{X}, \mathbf{X})^{-1} (\mathbf{y} - f(\mathbf{X}, \theta_0)),$$

where $K(\mathbf{X}, \mathbf{X}) = [K(\mathbf{x}_i, \mathbf{x}_j)]_{i,j} \in \mathbb{R}^{n \times n}$.

Proof. Part 1: Differential equation From the gradient flow rule, and the chain rule, and using

$$\frac{1}{2} \|f(\mathbf{X}, \theta) - \mathbf{y}\|^2 = \frac{1}{2} \sum_{i=1}^{n} (f(\mathbf{x}_i, \theta) - y_i)^2,$$

we get (using the chain rule),

$$\begin{aligned}\frac{d}{dt}\theta_t &= -\nabla_\theta L(\theta_t) \\ &= -\nabla_\theta \left(\frac{1}{2}\|f(\mathbf{X},\theta_t)-\mathbf{y}\|^2\right) \\ &= -\underbrace{\nabla_\theta f(\mathbf{X},\theta_t)}_{p\times n}\underbrace{(f(\mathbf{X},\theta_t)-\mathbf{y})}_{n\times 1},\end{aligned} \tag{7.2}$$

where $\nabla_\theta f(\mathbf{X},\theta_t) \in \mathbb{R}^{p\times n}$ is a Jacobian matrix.

Then,

$$\begin{aligned}\frac{d}{dt}f(\mathbf{x},\theta_t) &= \nabla_\theta f(\mathbf{x},\theta_t)^\top \frac{d}{dt}\theta_t \\ &= -\nabla_\theta f(\mathbf{x},\theta_t)^\top \nabla_\theta f(\mathbf{X},\theta_t)(f(\mathbf{X},\theta_t)-\mathbf{y}),\end{aligned} \tag{7.3}$$

where $\nabla_\theta f(\mathbf{x},\theta_t)^\top \in \mathbb{R}^{1\times p}$, $\nabla_\theta f(\mathbf{X},\theta_t) \in \mathbb{R}^{p\times n}$, and $(f(\mathbf{X},\theta_t)-\mathbf{y}) \in \mathbb{R}^{n\times 1}$.

Finally, since the model is linear, i.e. $f(\mathbf{x},\theta) = \frac{1}{\sqrt{p}}\theta^\top\phi(\mathbf{x})$, then $\nabla_\theta f(\mathbf{x},\theta_t) = \frac{1}{\sqrt{p}}\phi(\mathbf{x})$. Therefore, we have

$$K(\mathbf{x},\mathbf{x}') = \frac{1}{p}\langle\phi(\mathbf{x}),\phi(\mathbf{x}')\rangle = \nabla_\theta f(\mathbf{x},\theta_t)^\top \nabla_\theta f(\mathbf{x}',\theta_t), \tag{7.4}$$

for all θ_t. Combining the last equation with Eq. (7.3), we conclude that

$$\frac{d}{dt}f(\mathbf{x},\theta_t) = -K(\mathbf{x},\mathbf{X})(f(\mathbf{X},\theta_t)-\mathbf{y}), \tag{7.5}$$

which proves the first statement of the theorem.

Part 2: Solving the differential equation for training points
Plugin $\mathbf{x}=\mathbf{x}^{(1)}, \mathbf{x}=\mathbf{x}^{(2)}, \dots$ in $f(\mathbf{x},\theta)$. We have that for all $\mathbf{x}^{(i)}$,

$$\frac{d}{dt}f(\mathbf{x}^{(i)},\theta_t) = -K(\mathbf{x}^{(i)},\mathbf{X})(f(\mathbf{X},\theta_t)-\mathbf{y}). \tag{7.6}$$

Stacking the equations for all $\mathbf{x}^{(i)}$, we get the following differential equation:

$$\frac{d}{dt}f(\mathbf{X},\theta_t) = -K(\mathbf{X},\mathbf{X})(f(\mathbf{X},\theta_t)-\mathbf{y}). \tag{7.7}$$

This is a differential equation of the form $y'(t) = c(y(t)-a)$, whose solution can be found by shifting: $z'(t) = cz(t)$, from which we get $z(t) = e^{ct}z(0)$.

Let $\mathbf{v}_t = (f(\mathbf{X}, \theta_t) - \mathbf{y}) \in \mathbb{R}^{n \times 1}$, then

$$\frac{d}{dt}\mathbf{v}_t = \frac{d}{dt} f(\mathbf{X}, \theta_t) = -K(\mathbf{X}, \mathbf{X})\mathbf{v}_t. \tag{7.8}$$

The solution is $\mathbf{v}_t = e^{-K(\mathbf{X},\mathbf{X})t}\mathbf{v}_0$, i.e.

$$f(\mathbf{X}, \theta_t) = \mathbf{y} + e^{-K(\mathbf{X},\mathbf{X})t}(f(\mathbf{X}, \theta_0) - \mathbf{y}). \tag{7.9}$$

Part 3: Solving the differential equation for test points
For a test point $\mathbf{x}$, we have

$$\begin{aligned}\frac{d}{dt} f(\mathbf{x}, \theta_t) &= -K(\mathbf{x}, \mathbf{X})(f(\mathbf{X}, \theta_t) - \mathbf{y}) \\ &\overset{(7.9)}{=} -K(\mathbf{x}, \mathbf{X})(e^{-K(\mathbf{X},\mathbf{X})t}(f(\mathbf{X}, \theta_0) - \mathbf{y})).\end{aligned} \tag{7.10}$$

We have a differential equation of the form $y'(t) = e^{ct}$ whose solution is $y(t) - y(0) = \int_0^t e^{ct} dt = \frac{1}{c}(e^{ct} - 1)$. Therefore the solution for a test point $\mathbf{x}$ is

$$f(\mathbf{x}, \theta_t) - f(\mathbf{x}, \theta_0) = -K(\mathbf{x}, \mathbf{X}) \left(\int_0^t e^{-K(\mathbf{x},\mathbf{X})t} dt \right) (f(\mathbf{X}, \theta_0) - \mathbf{y}), \tag{7.11}$$

where $e^{-K(\mathbf{x},\mathbf{X})t} \in \mathbb{R}^{n \times n}$ is a matrix exponential. Thus,

$$\begin{aligned}f(\mathbf{x}, \theta_t) - f(\mathbf{x}, \theta_0) &= -K(\mathbf{x}, \mathbf{X})(-K(\mathbf{X}, \mathbf{X}))^{-1}(e^{-K(\mathbf{x},\mathbf{X})t} - \mathbf{I})(f(\mathbf{X}, \theta_0) - \mathbf{y}) \\ &= K(\mathbf{x}, \mathbf{X})(K(\mathbf{X}, \mathbf{X}))^{-1}(e^{-K(\mathbf{x},\mathbf{X})t} - \mathbf{I})(f(\mathbf{X}, \theta_0) - \mathbf{y}).\end{aligned} \tag{7.12}$$

Taking the limit $t \to \infty$, we obtain

$$f(\mathbf{x}, \theta_\infty) - f(\mathbf{x}, \theta_0) = -K(\mathbf{x}, \mathbf{X})(K(\mathbf{X}, \mathbf{X}))^{-1}(f(\mathbf{X}, \theta_0) - \mathbf{y}). \tag{7.13}$$

□

7.2.2 Initial and Final Distribution

We will now see that, under Gaussian initialization, the distribution of the output $f(\mathbf{x}, \theta_0)$ has a specific form, it follows a Gaussian Process (GP). We first give a mathematical definition of such a process and then provide some intuition (assuming $d = 1$).

Definition 26 (Gaussian process)**.** *We call $f \sim \mathcal{GP}(\mu, \Sigma)$ a Gaussian process with mean function $\mu : \mathbb{R} \to \mathbb{R}$ and covariance function $\Sigma : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ if it holds that*

$$\forall n \in \mathbb{N}, \forall t_1, \dots t_n \in \mathbb{R}, (f(t_1), \dots, f(t_n)) \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\sigma}),$$

where $\mu_i = \mu(t_i)$ and $\Sigma_{ij} = \Sigma(t_i, t_j)$.

We will typically write $f \sim \mathrm{GP}(\mu, \Sigma)$ to say that f is a GP with mean μ and covariance Σ.

Intuitively, a GP extends the Gaussian distribution from vectors to functions by describing all the finite marginal distributions, i.e. for any choice of inputs $t_1, \ldots, t_n \subset \mathbb{R}$, we know that the joint distribution $(f(t_1), \ldots, f(t_n))$ follows a Gaussian law, governed by the mean μ and covariance function Σ. Notice that a Gaussian process is fully characterized by its mean and covariance function, no additional information is needed. The covariance function determines the smoothness of the resulting random functions. Recall that a covariance function has to be a symmetric function and positive semi-definite in order to produce valid covariance matrices. There is thus a natural bijection between covariance and kernel functions. We will hence use the two terms interchangeably in the following. To give some intuition, consider the one-dimensional RBF kernel $K_{\mathrm{RBF}}(x, x) = e^{-\frac{1}{2}(x-x')^2}$. Take two inputs $x, x' \in \mathbb{R}$ that are very close, we then have that $K_{\mathrm{RBF}}(x, x') \approx 1$. If we consider $f \sim \mathcal{GP}(0, K_{\mathrm{RBF}})$, we can write down the approximate joint distribution as

$$\begin{pmatrix} f(x) \\ f(x') \end{pmatrix} \sim \mathcal{N}\left(\begin{pmatrix} 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix} \right)$$

This means that $f(x)$ and $f(x')$ are almost perfectly correlated, forcing the "random" function to assign them very similar values with a high likelihood. This is how random objects can still look very smooth as the kernel controls the degree of correlation. On the other hand, for inputs x, x' that are far away, the correlation decays rapidly and $f(x)$ and $f(x')$ can take completely different values. In Fig. 7.2 we show several such random functions drawn from the RBF Gaussian process $\mathcal{GP}(0, K_{\mathrm{RBF}})$. Observe how all the functions are indeed smooth but how far apart inputs (in the x-axis) are allowed to take very different values (in the y-axis). If you are interested in delving deeper into Gaussian processes, we recommend that you consult (Rasmussen, 2003) for further information.

Initial value of f Assume θ_0 is chosen so that all parameters are independent Gaussians with mean 0 and variance 1. This random initialization of θ induces a distribution over $f(\mathbf{x}; \theta)$ whose statistics we will analyze, both at initialization and throughout training (gradient descent on a specified dataset). We will see that:

- At initialization, an ensemble of wide neural networks is a zero-mean Gaussian process
- During training (gradient descent on mean-square error), the ensemble evolves according to the kernel

In other words, one can think of each draw of θ_0 as a different neural network: each network in the ensemble represents a different realization of the parameter perturbations. This is illustrated in Figure 7.2.2.

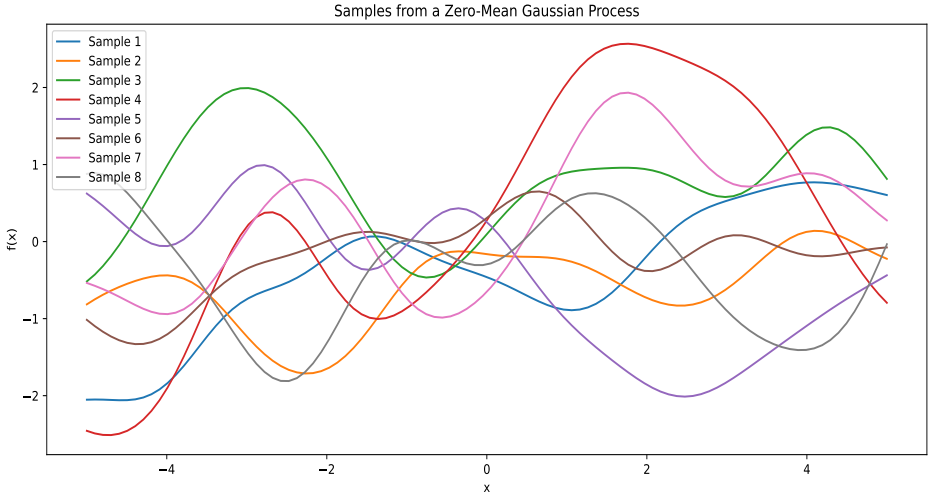


Figure 7.2: Samples drawn from a Gaussian process prior $\mathcal{GP}(0, K_{\text{RBF}})$.

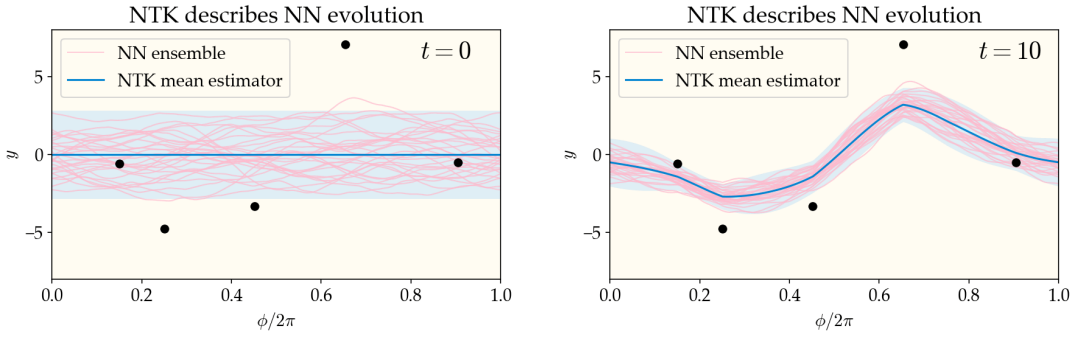


Figure 7.3: We train the NTK to fit the black points. The purple lines are different realizations of the neural networks (using different draws of θ_0 while the blue line shows the mean $\mathbb{E}_\theta f(\mathbf{x}, \theta)$. Source: `https://en.wikipedia.org/wiki/Neural_tangent_kernel`

We start by studying the value of the random feature model at initialization. The random initialization of θ_0 induces a distribution over $f(\mathbf{x};\theta_0)$ whose statistics we will analyze. The next theorem states that it is in fact a Gaussian process with zero mean and covariance defined by the kernel $K(\mathbf{x},\mathbf{x})$, i.e. $f(\mathbf{x},\theta_0) \sim \mathrm{GP}(0, K(\mathbf{x},\mathbf{x}))$.

Theorem 55 (Initial value)**.** *Assume θ_0 is chosen so that all parameters are independent Gaussians with mean 0 and variance 1. Then $f(\mathbf{x},\theta_0) = \theta_0^\top \phi(\mathbf{x})$ is also Gaussian with*

$$\mathbb{E}_{\theta_0}[f(\mathbf{x},\theta_0)] = 0, \quad \mathrm{var}[f(\mathbf{x},\theta_0)] = K(\mathbf{x},\mathbf{x}).$$

Also, $\mathrm{Cov}[f(\mathbf{x},\theta_0), f(\mathbf{x}',\theta_0)] = K(\mathbf{x},\mathbf{x}')$.

Proof idea. Since $f(\mathbf{x},\theta_0) = \theta_0^\top \phi(\mathbf{x})$, it's a linear combination of Gaussians, which is Gaussian. Indeed if $\mathbf{x} \sim \mathcal{N}(\mu_\mathbf{x}, \Sigma_\mathbf{x})$, and $\mathbf{y} = a + \mathbf{B}\mathbf{x}$, where $\mathbf{B}$ is a matrix, then $\mathbf{y} \sim \mathcal{N}(a + \mathbf{B}\mu_x, \mathbf{B}\Sigma_\mathbf{x}\mathbf{B}^\top)$. □

Value at $t \to \infty$ Finally, we look at what happens at time $t \to \infty$. Recall that $K(\mathbf{x},\mathbf{X}) \in \mathbb{R}^{1\times n}, K(\mathbf{X},\mathbf{X}) \in \mathbb{R}^{n\times n}, \mathbf{y} \in \mathbb{R}^{n\times 1}$. We will see that: $f(\mathbf{x},\theta_\infty) \sim \mathrm{GP}(m(\mathbf{x}), \Sigma(\mathbf{x},\mathbf{X}))$ where the expressions for the mean $m(\mathbf{x})$ and covariance $\Sigma(\mathbf{x},\mathbf{X})$ are given below.

Theorem 56 (Value at $t \to \infty$)**.** *Assuming Gaussian initialization, at $t \to \infty$, we get*

$$\mathbb{E}[f(\mathbf{x},\theta_\infty)] = K(\mathbf{x},\mathbf{X})K(\mathbf{X},\mathbf{X})^{-1}\mathbf{y}$$

and

$$\mathrm{var}[f(\mathbf{x},\theta_\infty)] = K(\mathbf{x},\mathbf{x}) - K(\mathbf{x},\mathbf{X})K(\mathbf{X},\mathbf{X})^{-1}K(\mathbf{x},\mathbf{X})$$

Proof. **i)** From Theorem 55, we have $\begin{pmatrix} f(\mathbf{x},\theta_0) \\ f(\mathbf{X},\theta_0) \end{pmatrix}$ is a GP with mean 0 and covariance

$$\begin{pmatrix} K(\mathbf{x},\mathbf{x}) & K(\mathbf{x},\mathbf{X}) \\ K(\mathbf{X},\mathbf{x}) & K(\mathbf{X},\mathbf{X}) \end{pmatrix}$$

(note the above is a block matrix, where $K(\mathbf{x},\mathbf{X})$ is a $1\times n$ matrix, $K(\mathbf{X},\mathbf{x})$ is $n\times 1$, $K(\mathbf{X},\mathbf{X})$ is $n\times n$.)

ii) From Theorem 54, we also have

$$\begin{pmatrix} f(\mathbf{x},\theta_\infty) \\ f(\mathbf{X},\theta_\infty) \end{pmatrix} = \begin{pmatrix} 1_{1\times 1} & -K(\mathbf{x},\mathbf{X})K^{-1}(\mathbf{X},\mathbf{X}) \\ 0_{n\times 1} & 0_{n\times n} \end{pmatrix} \begin{pmatrix} f(\mathbf{x},\theta_0) \\ f(\mathbf{X},\theta_0) \end{pmatrix} + \begin{pmatrix} K(\mathbf{x},\mathbf{X})K^{-1}(\mathbf{X},\mathbf{X})\mathbf{y} \\ 0_{n\times 1} \end{pmatrix}$$

The result is a Gaussian that is multiplied by a matrix and adding a shift $\implies$ This is a Gaussian.

□

7.3 Wide neural networks

Next, we will see how the analysis of the random feature model we discussed so far can be adapted to study a wide neural network. Formally, the type of network we study can be defined in a recursive fashion as follows. Starting with the first layer, we define

$$f^1(\mathbf{x}, \theta) = \frac{\sigma_w}{\sqrt{n_1}} \mathbf{W}^1 \mathbf{x} + \sigma_b \mathbf{b}^1,$$

where $\mathbf{W}^1 \in \mathbb{R}^{n_1 \times d}$ is the weight matrix and $\mathbf{b}^1 \in \mathbb{R}^{n_1}$ the bias of the first layer. The parameters (σ_w, σ_b) are the standard deviation of the weights and biases.

As in the case of the random feature model, the division by $\frac{1}{\sqrt{n_1}}$ is to ensure that we get proper convergence when we take the limit of infinitely wide networks $n_1 \to \infty$.

The post-activation of the first layer is defined as $h^1(\mathbf{x}, \theta) = \phi(f^1(\mathbf{x}, \theta))$ where ϕ is some entrywize non-linear activation function. We can then define a network by recursively applying the same function used in the first layer. For the l-th layer, we obtain

$$f^l(\mathbf{x}, \theta) = \frac{\sigma_w}{\sqrt{n_l}} \mathbf{W}^l h^{l-1}(\mathbf{x}, \theta) + \sigma_b \mathbf{b}^l,$$

where $\mathbf{W}^l \in \mathbb{R}^{n_l \times n_{l-1}}$ is the weight matrix and $\mathbf{b}^l \in \mathbb{R}^{n_l}$ the bias of the l-th layer.

Finally, the output of a network with L layers is $f(\mathbf{x}, \theta) = f^L(\mathbf{x}, \theta)$.

The parameter θ contains all the weight matrices $\mathbf{W}^l$ and biases $\mathbf{b}^l$. Importantly, we assume that at iteration 0, θ_0 is random.

General idea We will see that very wide neural networks are "close" (in terms of Taylor series approximation) to a random feature regression model.

We can make another interesting connection between training networks and infinite width by considering a linearization of the network. We again start with a neural network $f(\mathbf{x}, \theta)$ and we consider its Taylor expansion around its initialization θ_0 (i.e. we Taylor-expand in the parameters, not the input $\mathbf{x}$):

$$f^{\text{lin}}(\mathbf{x}, \theta) = f(\mathbf{x}, \theta_0) + (\theta - \theta_0)^\top \nabla_\theta f(\mathbf{x}, \theta)\big|_{\theta=\theta_0}.$$

It is important to realize that $\nabla_\theta f(\mathbf{x}, \theta)\big|_{\theta=\theta_0}$ does not depend on θ anymore as we explicitly evaluate the gradient at θ_0. As the name suggests, $f^{\text{lin}}(\mathbf{x}, \theta)$ is hence a linear model in θ (but not linear in $\mathbf{x}$). When the initialization θ_0 is chosen randomly, i.e. $\theta_0 \sim \mathcal{N}(\mathbf{0}, \mathbf{1}_{P \times P})$, $f^{lin}(\mathbf{x}, \theta)$ is a feature regression model where the features are given by $\phi(\mathbf{x}) = \nabla_\theta f(\mathbf{x}, \theta_0)$.

Let us again train such a model using gradient flow. What is the empirical neural tangent kernel of this model? We can easily calculate

$$\begin{aligned}\hat{\Theta}_t^{\text{lin}}(\mathbf{x}, \mathbf{x}') &= \left(\nabla_\theta f^{\text{lin}}(\mathbf{x}, \theta)\right)^\top \nabla_\theta f^{\text{lin}}(\mathbf{x}', \theta)\big|_{\theta=\theta_t} \\ &= \left(\nabla_\theta f(\mathbf{x}, \theta)\big|_{\theta=\theta_0}\right)^\top \nabla_\theta f(\mathbf{x}, \theta)\big|_{\theta=\theta_0} \\ &= \hat{\Theta}_0(\mathbf{x}, \mathbf{x}')\end{aligned}$$

The empirical NTK of $f^{\text{lin}}(\mathbf{x}, \theta)$ does thus not depend on time t even for finite width as it coincides with the NTK at initialization of the original model $f(\mathbf{x}, \theta)$. But we know that $\hat{\Theta}_t(\mathbf{x}, \mathbf{x}')$ in the infinite-width limit also becomes time-independent and thus coincides with the NTK at initialization. This means that for very large widths, a neural network starts behaving like its linearization and in the limit becomes indistinguishable from it! We refer the reader to (Lee et al., 2019) for a more precise, non-asymptotic statement, showing that

$$\left|f^{\text{lin}}(\mathbf{x}, \theta_t) - f(\mathbf{x}, \tilde{\theta}_t)\right| = \mathcal{O}\left(\frac{1}{\sqrt{P}}\right),$$

where the total number of parameters P is assumed to be large enough, and where we use $\tilde{\theta}_t$ to denote the trajectory of the non-linearized model.

Kernels: Feature regression vs wide network In the following, we will study the dynamics of infinitely-wide neural networks both at initialization and during training, similarly to the analysis we performed for the random feature model. We will see that both stages are controlled by a kernel, but that unlike the feature regression model, each stage is controlled by a different kernel. This is illustrated in Table 7.1. We will see that the formulas for Σ and Θ are defined recursively (details later). The NTK also depends on the NNGP kernel.

	Feature regression	**Wide neural network**
Initialization	$K(\mathbf{x}, \mathbf{x}')$	NNGP $\Sigma^{L+1}(\mathbf{x}, \mathbf{x}')$
Training	$K(\mathbf{x}, \mathbf{x}') = \frac{1}{p}\phi(\mathbf{x})^\top \phi(\mathbf{x})$	NTK $\Theta^{L+1}(\mathbf{x}, \mathbf{x}')$

Table 7.1: Comparison between feature regression vs wide neural networks.

At initialization

We first consider the distribution of the network output at initialization. We will again see that it is a Gaussian process with zero mean and a covariance given by the NNGP kernel that is defined by a recursive formula stated in the next theorem.

Theorem 57 (Distribution at initialization)**.** *For any point* $\mathbf{x}$*,* $f^l(\mathbf{x}, \theta_0)$ *is Gaussian with*

$$\mathbb{E}_{\theta_0}[f^l(\mathbf{x}, \theta_0)] = 0 \qquad \text{var}[f^l(\mathbf{x}, \theta_0)] = \Sigma^l(\mathbf{x}, \mathbf{x}),$$

where $\Sigma^l(\mathbf{x}, \mathbf{x}')$ *is defined recursively as*

$$\Sigma^1(\mathbf{x}, \mathbf{x}') = \sigma_w^2 \langle \mathbf{x}, \mathbf{x}' \rangle + \sigma_b^2, \quad \Sigma^l(\mathbf{x}, \mathbf{x}') = \sigma_w^2 \mathbb{E}_{\mathbf{z}, \mathbf{z}'}[\phi(\mathbf{z})^\top \phi(\mathbf{z}')] + \sigma_b^2,$$

with

$$\begin{pmatrix} \mathbf{z} \\ \mathbf{z}' \end{pmatrix} \sim \mathcal{N}\left(0, \begin{pmatrix} \Sigma^{l-1}(\mathbf{x}, \mathbf{x}) & \Sigma^{l-1}(\mathbf{x}, \mathbf{x}') \\ \Sigma^{l-1}(\mathbf{x}', \mathbf{x}) & \Sigma^{l-1}(\mathbf{x}', \mathbf{x}') \end{pmatrix}\right).$$

Proof idea. By induction: show that if one layer is a Gaussian process, then the next layer is also a Gaussian process. □

Definition NTK Kernel While the NNGP kernel controlled the network at initialization, the training dynamics depends on the NTK kernel $\Theta^l(\mathbf{x}, \mathbf{x}')$ which is defined as follows:

$$\Theta^l(\mathbf{x}, \mathbf{x}') = \Theta^{l-1}(\mathbf{x}, \mathbf{x}')\dot{\Sigma}(\mathbf{x}, \mathbf{x}') + \Sigma(\mathbf{x}, \mathbf{x}'),$$

where

$$\Sigma(\mathbf{x}, \mathbf{x}') = \sigma_w^2 \mathbb{E}[\phi(\mathbf{z})\phi(\mathbf{z}')] + \sigma_b^2$$

$$\dot{\Sigma}(\mathbf{x}, \mathbf{x}') = \sigma_w^2 \mathbb{E}[\dot{\phi}(\mathbf{z})\dot{\phi}(\mathbf{z}')].$$

Training dynamics

We previously showed that the training dynamics of a random feature model is controlled by the kernel K. Can we draw a similar conclusion for the dynamics of a wide network? The answer is yes as the next theorem states that the NTK kernel controls the dynamics of an infinitely wide neural network.

Theorem 58 (NTK Kernel)**.** *There is a kernel* $\Theta^l(\mathbf{x}, \mathbf{x}')$ *such that, for the layer widths* $n_1, \dots n_L \to \infty$*,*

$$\mathbb{E}[\nabla_\theta f^l(\mathbf{x}, \theta_0)^\top \nabla_\theta f^l(\mathbf{x}', \theta_0)] \to \Theta^l(\mathbf{x}, \mathbf{x}')$$

and

$$\frac{d}{dt} f(\mathbf{x}, \theta_t) = -\Theta^{L+1}(\mathbf{x}, \mathbf{X})(f(\mathbf{X}, \theta_t) - \mathbf{y})$$

Proof idea. We omit the proof due to its technical nature but the general idea is essentially the same as the feature regression model. □

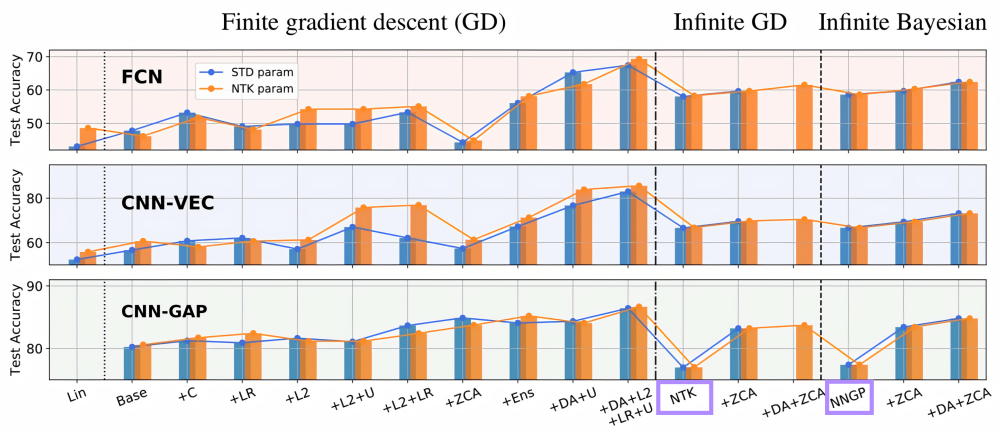


Figure 7.4: Performance of different architectures on *CIFAR10*, both finite and infinite width. We observe that the optimal performance is always achieved in the finite width regime. Source: (Lee et al., 2020).

Limitation of kernels The infinite width regime allows us to draw some interesting clear conclusions about the behavior of neural networks. However, it also has certain drawbacks. Arguably the most important one is the loss of expressivity of the model, namely that the kernel governing the dynamics of the model remains frozen to its value at initialization and can thus not adapt to the data. Exactly as in kernel learning, we pick a feature representation ϕ "by hand" (implicitly via the kernel) and this choice is made without seeing any data. We then have the freedom of specifying a linear separator in the space induced by ϕ but we cannot adapt the representation ϕ itself. It seems obvious that being able to learn a representation through data is advantageous compared to having to specify it beforehand, it is very doubtful for instance that a fixed representation will work well for both images of dogs and word embeddings. Finite-width neural networks on the other hand can learn data-specific representations as their kernel $\hat{\Theta}_t$ can indeed change with time (i.e. as training progresses). This disadvantage becomes very clear when comparing the two models empirically on standard benchmark tasks. We show such a comparison in Fig. 7.4, due to (Lee et al., 2020). Several architectures are compared here, their details are not important here but the main take-away is that the optimal performance is always achieved in the finite width regime. Moreover, the NTK theory is lagging behind and many state-of-the-art networks (some of which we will encounter later in this class) do not have their corresponding tangent kernel developed yet. We can hence not compare them against their corresponding kernel but networks such as DenseNet (Huang et al., 2017) achieve accuracies around 96% on *CIFAR10* (a popular benchmark dataset in the field of computer vision), a number that we are currently very far away from in the infinite width regime.

7.4 Exercise: Neural Tangent Kernel

Problem 1 (NTK for two-layer ReLU network):
Consider a two-layer neural network with the second layer fixed,

$$f(\mathbf{W}, \mathbf{x}) = \frac{1}{\sqrt{m}} \sum_{r=1}^{m} \sigma(\mathbf{w}_r^\top \mathbf{x})$$

where $\mathbf{x} \in \mathbb{R}^d$ is the input, $\mathbf{W} = (\mathbf{w}_r)_{r=1}^m \in \mathbb{R}^{m \times d}$ is a matrix containing the weight vectors $\mathbf{w}_r \in \mathbb{R}^d$ of the first layer, and σ is the ReLU activation function. Throughout this problem, only the first layer (with weight vectors $\mathbf{w}_r,\ r = 1, ..., m$) is trained. We are given a training dataset $(\mathbf{x}_i, y_i)$ where each $\mathbf{x}_i \in \mathbb{R}^d$ is a feature vector such that $\|\mathbf{x}_i\| = 1$, and $y_i \in \mathbb{R}$ is the corresponding target label. We consider the following square loss:

$$\ell(\mathbf{W}) = \frac{1}{2} \sum_{i=1}^{n} \left(f(\mathbf{W}, \mathbf{x}_i) - y_i \right)^2 .$$

We optimize over each $\mathbf{w}_r$ using gradient flow:

$$\frac{d\mathbf{w}_r(t)}{dt} = -\frac{\partial \ell(\mathbf{W}(t))}{\partial \mathbf{w}_r(t)} \in \mathbb{R}^d$$

for $r = 1, ..., m$.

a) We denote $u_i(t) = f(\mathbf{W}(t), \mathbf{x}_i)$ the prediction on input $\mathbf{x}_i$ at time t. Show that, at any time t, we have

$$\frac{\partial \ell(\mathbf{W}(t))}{\partial \mathbf{w}_r(t)} = \sum_{i=1}^{n} \left(u_i(t) - y_i \right) \frac{\partial f(\mathbf{W}(t), \mathbf{x}_i)}{\partial \mathbf{w}_r(t)} \in \mathbb{R}^d$$

b) Let $\mathbf{u}(t) = (u_1(t), \ldots, u_n(t)) \in \mathbb{R}^n$ be the prediction vector at time t. Show that, using chain rule on $\mathbf{u}(t)$, the dynamics of the predictions can be written as

$$\frac{d}{dt}\mathbf{u}(t) = \mathbf{H}(t)(\mathbf{y} - \mathbf{u}(t)),$$

where $\mathbf{H}(t)$ is an $n \times n$ matrix with (i, j)-th entry

$$\mathbf{H}_{ij}(t) = \sum_{r=1}^{m} \left\langle \frac{\partial f(\mathbf{W}(t), \mathbf{x}_i)}{\partial \mathbf{w}_r(t)}, \frac{\partial f(\mathbf{W}(t), \mathbf{x}_j)}{\partial \mathbf{w}_r(t)} \right\rangle$$

Note that the so-called Gram matrix $\mathbf{H}$ defined above is essentially the neural tangent kernel on the training data.

c) Show that, at any time t, we have the following expression of the entries in the Gram matrix $\mathbf{H}$:

$$\mathbf{H}_{ij}(t) = \frac{1}{m}\sum_{r=1}^{m} \mathbf{x}_i^\top \mathbf{x}_j \mathbb{I}\left\{\mathbf{w}_r(t)^\top \mathbf{x}_i \geq 0, \mathbf{w}_r(t)^\top \mathbf{x}_j \geq 0\right\}$$

where the indicator $\mathbb{I}\{A\}$ is 1 when the the constraint A holds; 0 otherwise.
Hint: The derivative $\sigma'(z)$ of the ReLU function is the step function $\text{step}(z) = \begin{cases} 1, & x > 0 \\ 0, & x < 0 \end{cases}$.

d) If we assume the weight vectors $\mathbf{w}_r$ are initialized as independent standard Gaussian vectors, i.e. $\mathbf{w}_r(0) \sim \mathcal{N}(0, \mathbf{I})$ for $r = 1, ..., m$, and we take the the limit of the layer width $m \to \infty$, we have the so-called neural tangent kernel (NTK) matrix $\mathbf{H}^\infty$ with entries:

$$\mathbf{H}_{ij}^\infty = \lim_{m\to\infty} \mathbf{H}_{ij}(0) = \mathbb{E}_{\mathbf{w}\sim\mathcal{N}(0,\mathbf{I})}\left[\mathbf{x}_i^\top \mathbf{x}_j \mathbb{I}\left\{\mathbf{w}^\top \mathbf{x}_i \geq 0, \mathbf{w}^\top \mathbf{x}_j \geq 0\right\}\right]$$

Show that we have the following closed-form for the entries of NTK $\mathbf{H}^\infty$:

$$\mathbf{H}_{ij}^\infty = \frac{\mathbf{x}_i^\top \mathbf{x}_j \left(\pi - \arccos(\mathbf{x}_i^\top \mathbf{x}_j)\right)}{2\pi}, \quad \forall i, j = 1, ..., n. \tag{7.14}$$

Problem 2 (Upper bound of classification error):

Assume we have a dataset with n data points, $\{(\mathbf{x}_i, y_i)\}_{i=1}^n$ where each input $\mathbf{x}_i \in \mathbb{R}^d$ has norm 1 and $y_i = \beta^\top \mathbf{x}_i$ for some $\beta \in \mathbb{R}^d$. Write $\mathbf{X} = (\mathbf{x}_i)_{i=1}^n \in \mathbb{R}^{n\times d}$.

We consider the same two-layer ReLU network used in the previous problem. An upper bound on the classification error of the NTK is given by

$$\frac{\sqrt{2\mathbf{y}^\top (\mathbf{H}^\infty)^{-1}\mathbf{y} \cdot \mathrm{tr}(\mathbf{H}^\infty)}}{n},$$

where the NTK matrix $\mathbf{H}^\infty$ is defined in Eq. (7.14).

This complexity measure can be derived using *Rademacher complexity* bounds, which will be the subject of a later course. Our goal will be to derive a bound on this complexity measure that decreases as n increases. This will imply that the NTK can achieve zero classification error given a sufficient large number of datapoints n.

a) First, show that $\mathbf{H}^\infty$ admits the following expression of entries:

$$\mathbf{H}^\infty_{ij} = \frac{\mathbf{x}_i^\top \mathbf{x}_j}{4} + \frac{1}{2\pi}\sum_{l=0}^{\infty} \frac{(2l)!}{2^{2l}(l!)^2}\frac{(\mathbf{x}_i^\top \mathbf{x}_j)^{2l+2}}{2l+1}.$$

Hint: Use the following Taylor approximation of $\arccos(z)$:

$$\arccos(z) = \frac{\pi}{2} - \sum_{l=0}^{\infty} \frac{(2l)!}{2^{2l}(l!)^2}\frac{z^{2l+1}}{2l+1}.$$

b) Using the following facts: (You do not need to prove them)

- $4\mathbf{K}^{-1} - (\mathbf{H}^\infty)^{-1}$ is a positive semi-definite matrix; (why?)
- $\mathrm{tr}(\mathbf{H}^\infty) \le n$; (why?)
- the operator norm of the matrix $\mathbf{X}(\mathbf{X}\mathbf{X}^\top)^{-1}\mathbf{X}^\top$ is less than or equal to 1. (why?)

and prove the following upper bound on the classification error of the NTK :

$$\frac{\sqrt{2\mathbf{y}^\top (\mathbf{H}^\infty)^{-1}\mathbf{y} \cdot \mathrm{tr}(\mathbf{H}^\infty)}}{n} \le \frac{2\sqrt{2}\|\beta\|_2}{\sqrt{n}}.$$

Chapter 8

Generalization I

Generalization in machine learning refers to the ability of a model to make accurate predictions on new, unseen data. Generalization is crucial for machine learning models because in real-world applications, the model will be exposed to new data that it has not seen before. If a model does not generalize well, it will not be able to make accurate predictions in these situations, and its performance will be limited. In this lecture, we will discuss generalization bounds which are important in deep learning because they provide a theoretical framework for understanding and quantifying the expected error of a model on new, unseen data. These bounds give us insight into how well a model will perform in practice.

Let's introduce the problem formally. We consider a hypothesis space as a set $\mathcal{F}$ of functions from $\mathcal{X} \subseteq \mathbb{R}^d$ to $\mathcal{Y} \subseteq \mathbb{R}$. Given a data distribution $\mathcal{D}$ over $(\mathcal{X}, \mathcal{Y})$, we define the true risk of a function $f \in \mathcal{F}$ as

$$R(f) = \mathbb{E}_{(\mathbf{x},y)\sim\mathcal{D}}[l((\mathbf{x}, y), f)].$$

We will mostly discuss a classification setting where $l((\mathbf{x}, y), f) = \mathbf{1}_{f(\mathbf{x})=y}$.

Since we typically do not have access to the exact data distribution but only a sample set $S = \{(\mathbf{x}_i, y_i)\}_{i=1}^n \sim D$, we also define the empirical risk as

$$R_S(f) = \frac{1}{n} \sum_{(x,y)\in S} [l((\mathbf{x}, y), f)].$$

The typical form of a generalization bound we are interested in is:

$$\underbrace{R(f)}_{\text{True risk}} \leq \underbrace{R_S(f)}_{\text{Empirical risk}} + \underbrace{m(\text{complexity of class of functions}, n)}_{\text{(a function that approaches 0 as n approaches infinity)}},$$

where n is the number of training datapoints and m is a function that measures the complexity of a class of functions, such as the VC dimension, or the Rademacher

complexity, which we will discuss in this lecture.

8.1 Basic Concentration Bound

We can obtain a basic bound by applying Hoeffding's inequality to a given function class and then using the union bound over the set of function classes.

More precisely, let $Z_i = (x_i, y_i)$ be i.i.d. random variables with $f(Z) \in [a, b]$. The quantity we are interested in is

$$R(f) - R_S(f) = \mathbb{E}[f(Z)] - \frac{1}{n}\sum_{i=1}^{n} f(Z_i).$$

Using Hoeffding's inequality, as well as the independence of the random variables $f(Z_i)$ [1] we get:

$$P\left[\left|\mathbb{E}[f(Z)] - \frac{1}{n}\sum_{i=1}^{n} f(Z_i)\right| > \epsilon\right] \leq 2\exp\left(-\frac{2n\epsilon^2}{(b-a)^2}\right).$$

Setting the right hand side to be δ, we obtain:

$$\begin{aligned}
\delta &= 2\exp\left(-\frac{2n\epsilon^2}{(b-a)^2}\right) \\
\implies \log\delta &= \log 2 - \left(\frac{2n\epsilon^2}{(b-a)^2}\right) \\
\implies \log\frac{1}{\delta} &= -\log 2 + \left(\frac{2n\epsilon^2}{(b-a)^2}\right) \\
\implies \epsilon^2 &= \frac{(b-a)^2}{2n}\left(\log 2 + \log\frac{1}{\delta}\right) \\
\implies \epsilon &= (b-a)\sqrt{\frac{\log\frac{2}{\delta}}{2n}}. \qquad (8.1)
\end{aligned}$$

Therefore, with probability at least $1-\delta$, for a given $f \in \mathcal{F}$,

$$|R_S(f) - R(f)| \leq (b-a)\sqrt{\frac{\log\frac{2}{\delta}}{2n}}.$$

The result obtained so far is for one given element $f \in \mathcal{F}$. In order to generalize this result to all functions in $\mathcal{F}$, we rely on the union bound which states that for

[1] Recall that if $(Z_i)_i$ are i.i.d. variables, then $(f(Z_i))_i$ are also i.i.d. as long as f is a Borel-measurable function, which includes all functions of practical interest in machine learning.

a family of N functions, the probability of the union of a set of events C_i is upper bounded as:

$$P(C_1 \cup \cdots \cup C_N) \leq \sum_{n=1}^{N} P(C_n) \leq N\delta. \tag{8.2}$$

Using the union bound for all set of function classes $\mathcal{F} = \{f_1, \dots f_N\}$, we get that with probability at least $1 - \delta$,

$$\forall f \in \mathcal{F}, R(f) \leq R_S(f) + (b-a)\sqrt{\frac{\log N + \log \frac{2}{\delta}}{2n}}.$$

Indeed, the derivation is similar to what we did above:

$$\begin{aligned}
\delta &= 2N \exp\left(-\frac{2n\epsilon^2}{(b-a)^2}\right) \\
\implies \log \delta &= \log 2N - \left(\frac{2n\epsilon^2}{(b-a)^2}\right) \\
\implies \log \frac{1}{\delta} &= -\log 2N + \left(\frac{2n\epsilon^2}{(b-a)^2}\right) \\
\implies \epsilon^2 &= \frac{(b-a)^2}{2n}\left(\log 2N + \log \frac{1}{\delta}\right) \\
\implies \epsilon &= (b-a)\sqrt{\frac{\log \frac{2N}{\delta}}{2n}}
\end{aligned} \tag{8.3}$$

Let $f_S = \operatorname{argmin}_{f \in \mathcal{F}} R_S(f)$. A bound on the estimation error can then be derived as follows:

$$\begin{aligned}
R(f_S) &= R(f_S) - R(f^*) + R(f^*) \\
&\leq R_S(f^*) - R_S(f_S) + R(f_S) - R(f^*) + R(f^*) \\
&\leq 2 \sup_{f \in \mathcal{F}} |R(f) - R_S(f)| + R(f^*)
\end{aligned} \tag{8.4}$$

The first inequality uses the fact that $R_S(f^*) - R_S(f_S) \geq 0$.

We conclude that, with probability at least $1 - \delta$,

$$\forall f \in \mathcal{F}, R(f_S) \leq R(f^*) + 2(b-a)\sqrt{\frac{\log N + \log \frac{2}{\delta}}{2n}}. \tag{8.5}$$

Remark 7 (Uncountable case)**.** *When the set $\mathcal{F}$ is uncountable, the previous approach does not directly work. There are several measures of capacity or size of classes of function: the VC dimension and growth function both distribution independent, and the VC entropy which depends on the distribution.*

8.2 Uniform Bounds

Although the above result seems nice (since it applies to any class of bounded functions), it is actually severely limited. Indeed, what it essentially says is that for each (fixed) function $f \in \mathcal{F}$, there is a collection of sample sets S for which the bound holds, and this collection of sample sets has measure $\mathbb{P}[S] \geq 1 - \delta$. However, these sets S may be different for different functions f. In other words, for the observed sample, only some of the functions in $\mathcal{F}$ will satisfy the bound. This means that f cannot change with different draws of S for the bound to hold.
Note that for simplicity, we will consider the case where $(b - a) = 1$.

Before seeing the data, we do not know which function f the algorithm will choose. The idea is to consider uniform bounds, i.e. bounds that hold uniformly over all functions in the class, hence also for the supremum (i.e. the worst-case),

$$\sup_{f \in \mathcal{F}} R(f) - R_S(f). \tag{8.6}$$

Theorem 59. *Given a set S of size n and $\mathcal{F} = \{f_1, \dots f_N\}$, let $f^* = \operatorname{argmin}_{f \in \mathcal{F}} R(f)$ and $f_S = \operatorname{argmin}_{f \in \mathcal{F}} R_S(f)$. Then*

$$R(f_S) \leq R(f^*) + 2\sqrt{\frac{\log N + \log \frac{2}{\delta}}{2n}}. \tag{8.7}$$

Proof. For each function f_i, we define a set of n-tuples:

$$C_i = \{\{(\mathbf{x}_1, y_1), \dots (\mathbf{x}_n, y_n)\} \mid R(f_i) - R_S(f_i) > \epsilon\} \quad i = 1, 2, \dots, N. \tag{8.8}$$

Note that the set C_i contains all the "bad" samples, i.e. those for which the bound fails.

For each function, we can use Hoeffding's inequality to predict that

$$P(C_i) \leq \delta \tag{8.9}$$

and therefore

$$P(C_1 \cup C_2) \leq P(C_1) + P(C_2) \leq 2\delta \tag{8.10}$$

which can be generalized to a family of N functions:

$$P(C_1 \cup \dots \cup C_N) \leq \sum_{n=1}^{N} P(C_n) \leq N\delta. \tag{8.11}$$

We thus have that for any family of N functions,

$$P(\sup_{f\in\mathcal{F}} R(f) - R_S(f) \geq \epsilon) \leq Ne^{-2n\epsilon^2}, \tag{8.12}$$

or equivalently that if $\mathcal{F} = \{f_1, \dots f_N\}$, with probability $1-\delta$,

$$\forall f \in \mathcal{F}, R(f) - R_S(f) + \sqrt{\frac{\log N + \log\frac{2}{\delta}}{2n}}. \tag{8.13}$$

Now, we need to derive a bound for the minimum of the empirical risk f_S, i.e. $R_S(f_S) = \min_{f\in\mathcal{F}} R_S(f)$. This is done as follows.

Recall that $f^* = \text{argmin}_{f\in\mathcal{F}} R(f)$. Starting from the inequality

$$R(f^*) = R(f^*) \pm R_S(f^*) \leq R_S(f^*) + \sup_{f\in\mathcal{F}}(R(f) - R_S(f)). \tag{8.14}$$

Combined with $R_S(f^*) - R_S(f_S) \geq 0$ by definition of f_S, we get

$$\begin{aligned} R(f_S) = R(f_S) \pm R(f^*) &\leq \underbrace{R_S(f^*) - R_S(f_S)}_{\geq 0} + R(f_S) - R(f^*) + R(f^*) \\ &\leq 2\sup_{f\in\mathcal{F}} |R(f) - R_S(f)| + R(f^*). \end{aligned} \tag{8.15}$$

Hence with probability at least $1-\delta$,

$$R(f_S) \leq R(f^*) + 2\sqrt{\frac{\log N + \log\frac{2}{\delta}}{2n}}. \tag{8.16}$$

□

8.3 Infinite Case: VC Bound

The previous result can also be extended to the case where the class $\mathcal{F}$ is infinite using VC theory. Vapnik & Chervonenkis argued that what really matters for a sample $S = \{\mathbf{z}_1, \dots \mathbf{z}_n\}$ containing n points is the set

$$\mathcal{F}_{\mathbf{z}_1,\dots\mathbf{z}_n} = \{(f(\mathbf{z}_1), \dots f(\mathbf{z}_n)) | f \in \mathcal{F}\}. \tag{8.17}$$

The size of this set is the total number of possible ways that S can be classified. For binary classification, $\mathcal{F}_{\mathbf{z}_1,\dots\mathbf{z}_n}$ is a set of binary vectors $\subset \{-1, +1\}^n$, see Figure 8.1 below.

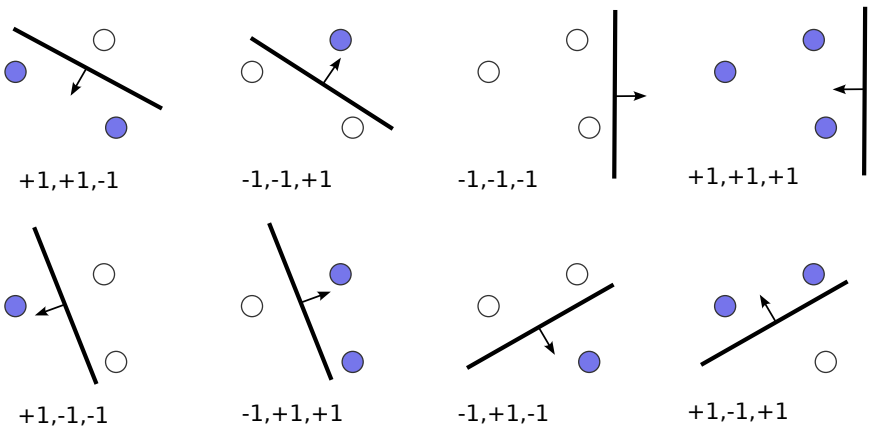


Figure 8.1: Illustration of the concept of growth function.

Definition 27 (Growth Function)**.** *The growth function of $\mathcal{F}$ is equal to*

$$S_{\mathcal{F}}(n) = \sup_{(\mathbf{z}_1 \dots \mathbf{z}_n)} |\mathcal{F}_{\mathbf{z}_1 \dots \mathbf{z}_n}|. \tag{8.18}$$

The growth function can be thought as a measure of the "size "for the class of functions $\mathcal{F}$. Indeed, the size of this set is the number of possible ways in which the data $\{\mathbf{z}_1 \dots \mathbf{z}_n\}$ can be classified using functions in $\mathcal{F}$. An upper bound on this number is 2^n.

Example Consider the set of halfspaces in 2-dimensions, i.e binary functions whose decision boundary is a line. One can easily see that the growth function for one, two, and three points are $2^1, 2^2, 2^3$. However, this exponential trend does not continue past 4 points (exercise: check this yourself).

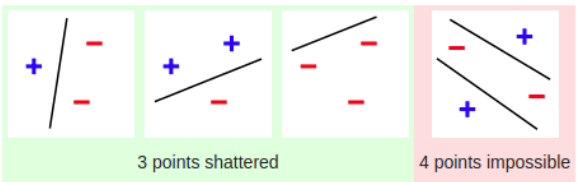


Figure 8.2: Illustration of the concept of shaterring. Source: Wikipedia.

We are now ready to state the VC theorem that gives us a generalization bound that depends on the growth function.

Theorem 60. *For any $\delta > 0$, with probability at least $1 - \delta$,*

$$\forall f \in \mathcal{F}, \;\; R(f) \leq R_S(f) + 2\sqrt{2\frac{\log S_{\mathcal{F}}(2n) + \log \frac{2}{\delta}}{n}},$$

where $S_{\mathcal{F}}(2n)$ is the growth function of the function class $\mathcal{F}$ defined as

$$S_{\mathcal{F}}(n) = \sup_{\mathbf{z}} |\mathcal{F}_{\mathbf{z}_1,\ldots\mathbf{z}_n}|.$$

Proof. Proof omitted. See statistical theory book. □

VC dimension An important concept in VC theory is shaterring. We say that $\mathcal{F}$ shatters S if $|\mathcal{F}_S| = 2^{|S|}$. The VC dimension of a function class $\mathcal{F}$ is the cardinality of the largest set that it can shatter. Importantly this means that we can place the points in the set S arbitrarily, we only need one configuration of these points that can be shattered. Formally, we will define the VC dimension as follows.

Definition 28 (VC dimension)**.** *The VC dimension of a class $\mathcal{F}$ is the largest n such that*

$$S_{\mathcal{F}}(n) = 2^n.$$

Since n points can have 2^n configurations, the VC dimension is the largest number of points that can be shattered (i.e. split arbitrarily) by the function class. For example, the VC dimension of linear classifiers in $\mathbb{R}^d$ is $d + 1$ (see Figure 8.2). How do we prove this? We need to show there exists a configuration of $d + 1$ points that can be shattered, but that there does not exist a configuration of $d + 2$ points that can be shattered.

The following well-known lemma establishes an upper bound on the growth function.

Lemma 61 (Sauer-Shelah lemma)**.** *Let $\mathcal{F}$ be a class of functions with finite VC-dimension h. Then, $\forall n \in \mathbb{N}$,*

$$S_{\mathcal{F}}(n) \leq \sum_{i=0}^{h} \binom{n}{i}, \tag{8.19}$$

and for all $n \geq h$,

$$S_{\mathcal{F}}(n) \leq \left(\frac{en}{h}\right)^h. \tag{8.20}$$

The first bound shows that the growth function grows very fast as a function of n (faster than exponential), but once the number of datapoints n is larger than the

VC dimension h, the growth becomes polynomial.

One can also use this upper bound in combination with Theorem 60 in order to obtain the following generalization bound that depends on h and n:

$$\forall f \in \mathcal{F}, \;\; R(f) \leq R_S(f) + 2\sqrt{2\frac{h(\log(2n)+1) + \log\frac{2}{\delta}}{n}}.$$

Application to neural networks Bartlett et al. (2019) proves bounds on the VC dimension of piecewise linear networks. Letting W be the number of weights and L be the number of layers, they prove that the VC dimension is $\mathcal{O}(WL\log(W))$. VC bounds therefore claim that the number of datapoints required to obtain a small generalization gap has to scale inversely proportionally to $\mathcal{O}(WL\log(W))$. From a practical point of view, these bounds tend to be too large for modern deep neural networks that rely on a large number of parameters. This is partly due to their generality as VC bounds provide guarantees for any probability distribution and for any training algorithm. We will later discuss other alternatives that can yield practically more useful generalization bounds but it is worth emphasizing at this stage that the generalization of deep neural networks is an active area of research.

8.4 Rademacher Bound

A Rademacher random variable σ_i is defined as

$$\sigma_i = \begin{cases} +1, & \text{with prob. } 1/2 \\ -1, & \text{with prob. } 1/2 \end{cases}$$

Definition 29 (Rademacher complexity)**.** *Let $S = \{z_1, \dots z_m\}$ be a set of samples drawn i.i.d. from D. Let $\mathcal{F}$ be a class of functions $f : Z \to \mathbb{R}$. The empirical Rademacher complexity is then defined as*

$$\hat{\mathbb{R}}_S(\mathcal{F}) = \mathbb{E}_\sigma \left[\sup_{f\in\mathcal{F}} \frac{1}{m} \sum_{i=1}^{m} \sigma_i f(z_i) \right]. \tag{8.21}$$

The supremum measures the maximum correlation between $f(z_i)$ and σ_i over all $f \in \mathcal{F}$. Taking the expectation over σ, we can say that the empirical Rademacher complexity measures the ability of functions from $\mathcal{F}$ to fit random noise.

The expected Rademacher complexity is then obtained by taking the expectation of $\hat{\mathbb{R}}_S(\mathcal{F})$ over all samples of size m, i.e.

$$\mathbb{R}_m(\mathcal{F}) = \mathbb{E}_S[\hat{\mathbb{R}}_S(\mathcal{F})], \tag{8.22}$$

which measures the expected noise fitting ability of $\mathcal{F}$ over all datasets S.

In the following, we will use the notation $\hat{\mathbb{E}}_S[f] = \frac{1}{m}\sum_{i=1}^m f(z_i)$ where $S = \{z_1, \ldots, z_m\}$. The following result states a generalization bound in terms of the Rademacher complexity of a given function class.

Theorem 62 (Rademacher-based uniform convergence)**.** *Fix the distribution D and $\delta \in (0,1)$. If $\mathcal{F} \in \{f : Z \to [0,1]\}$ and $S = \{z_1, \ldots z_m\}$. Then with probability at least $1-\delta$ over the draw of S, for all $\mathbf{f} \in \mathcal{F}$, we have*

$$\mathbb{E}_D[f(z)] \leq \hat{\mathbb{E}}_S[f(z)] + 2\mathbb{R}_m(\mathcal{F}) + \sqrt{\frac{\log(1/\delta)}{m}}. \tag{8.23}$$

Proof sketch. We only state the main steps:

1. Define $\phi(S) = \sup_{f\in\mathcal{F}}(\mathbb{E}_D[f] - \hat{\mathbb{E}}_S[f])$. Then

$$\mathbb{E}_D[f(z)] \leq \hat{\mathbb{E}}_S[f(z)] + \phi(S)$$

2. Bound $\phi(S)$ by its expected value $\mathbb{E}_S[\phi(S)]$ using McDiarmid's inequality [2]. McDiarmid's Inequality provides a concentration bound on the deviation of the sample mean of independent random variables from its expected value.

3. Define a ghost sample $S' = \{z'_1, \ldots z'_m\} \sim D$ and show $\mathbb{E}_S[\phi(S)] \leq \mathbb{E}_{S,S'}[\sup_{f\in\mathcal{F}}(\hat{\mathbb{E}}_{S'}[f] - \hat{\mathbb{E}}_S[f])]$
   $\rightsquigarrow$ To prove this, note that $\mathbb{E}_D[f] = \mathbb{E}_{S'}[\hat{\mathbb{E}}_{S'}[f]]$ and $\sup_{f\in\mathcal{F}}\mathbb{E}[f] \leq \mathbb{E}[\sup_{f\in\mathcal{F}} f]$

4. Show $\mathbb{E}_{S,S'}[\sup_{f\in\mathcal{F}}(\hat{\mathbb{E}}_{S'}[f] - \hat{\mathbb{E}}_S[f])] = \mathbb{E}_{S,S'}[\sup_{f\in\mathcal{F}}\sum_i \sigma_i(f(z'_i) - f(z_i))]$
   $\rightsquigarrow$ To prove this, use the fact that z'_i and z_i are interchangeable

5. Show $\mathbb{E}_{S,S'}[\sup_{f\in\mathcal{F}}\sum_i \sigma_i(f(z'_i) - f(z_i))] \leq 2\mathbb{R}_m(\mathcal{F})$
   $\rightsquigarrow$ To prove this, break the sum into two sums, one over z_i and the other over z'_i

□

[2] McDiarmid's Inequality: Let $X_1, X_2, \ldots, X_n$ be independent random variables, and let $f : \mathcal{X}^n \to \mathbb{R}$ be a function. If there exist constants $c_1, c_2, \ldots, c_n > 0$ such that for all $i = 1, 2, \ldots, n$ and $x_i, x'_i \in \mathcal{X}$: $|f(x_1, x_2, \ldots, x_i, \ldots, x_n) - f(x_1, x_2, \ldots, x'_i, \ldots, x_n)| \leq c_i$, then for any $\epsilon > 0$:

$$\Pr\left(\left|\frac{1}{n}\sum_{i=1}^n f(X_i) - \mathbb{E}[f(X)]\right| \geq \epsilon\right) \leq 2e^{-\frac{2\epsilon^2}{\sum_{i=1}^n c_i^2}}.$$

Rademacher bound for Linear Classes

Lemma 63. *Let $\mathcal{F}$ be the class of linear predictors, with the L_2-norm of the weights bounded by a constant $W_2 > 0$. Also assume that with probability one that $\|\mathbf{x}\|_2 \leq D$. Then*

$$\mathbb{R}(\mathcal{F}) \leq \frac{DW_2}{\sqrt{m}}. \tag{8.24}$$

Proof. Let $\mathcal{F}$ be the class

$$\mathcal{F} = \{(\mathbf{w}^\top \mathbf{x}_1, \mathbf{w}^\top \mathbf{x}_2, \ldots, \mathbf{w}^\top \mathbf{x}_m) : \|\mathbf{w}\|_2 \leq W_2\}.$$

The empirical Rademacher complexity of $\mathcal{F}$ can be bounded as follows:

$$\begin{aligned}
\mathbb{R}(\mathcal{F}) &= \mathbb{E}_\sigma \left[\frac{1}{m} \sum_{i=1}^{m} \sup_{f \in \mathcal{F}} \sigma_i f(z_i) \right] \\
&= \frac{1}{m} \mathbb{E}_\sigma \left[\sup_{\mathbf{w}:\|\mathbf{w}\|_2 \leq W_2} \sum_{i=1}^{m} \sigma_i \mathbf{w}^\top \mathbf{x}_i \right] \\
&= \frac{1}{m} \mathbb{E}_\sigma \left[\sup_{\mathbf{w}:\|\mathbf{w}\|_2 \leq W_2} \mathbf{w}^\top (\sum_{i=1}^{m} \sigma_i \mathbf{x}_i) \right].
\end{aligned} \tag{8.25}$$

Since $\|\mathbf{w}\|_2 \leq W_2$ for all $\mathbf{w}$, then

$$\begin{aligned}
\mathbb{R}(\mathcal{F}) &\leq \frac{W_2}{m} \mathbb{E}_\sigma \left[\|(\sum_{i=1}^{m} \sigma_i \mathbf{x}_i)\|_2 \right] \\
&\leq \frac{W_2}{m} \sqrt{\mathbb{E}_\sigma \left[(\sum_{i=1}^{m} \|\sigma_i \mathbf{x}_i\|_2^2) \right]} \\
&= \frac{W_2}{m} \sqrt{\mathbb{E}_\sigma \left[(\sum_{i=1}^{m} \sigma_i^2 \|\mathbf{x}_i\|_2^2) \right]} \\
&= \frac{W_2}{m} \sqrt{\mathbb{E}_\sigma \left[(\sum_{i=1}^{m} \|\mathbf{x}_i\|_2^2) \right]} \\
&\leq \frac{DW_2}{\sqrt{m}}.
\end{aligned} \tag{8.26}$$

□

Note that in the bound derived in Lemma 63, the Lipschitz constant of f w.r.t. to $\mathbf{x}$ appears, since $\|\nabla_{\mathbf{x}} f(\mathbf{z}_i)\|_2 = \|\mathbf{w}\|_2 \leq W_2$.

Another important lemma that is often used to bound the Rademacher complexity of composition of functions is the composition lemma stated below.

Lemma 64 (Composition lemma)**.** *For $\mathcal{F} \in \mathbb{R}^d, \phi : \mathbb{R} \to \mathbb{R}$, let $\mathcal{F}' := \{\phi \circ f : f \in \mathcal{F}\}$. If ϕ is L-Lipschitz continuous, i.e. $|\phi(t) - \phi(t')| \leq L|t - t'|$, then for any m,*

$$\mathbb{R}(\mathcal{F}') \leq L\mathbb{R}(\mathcal{F}). \tag{8.27}$$

Proof. Exercise. □

Application to neural networks Based on Lemma 63 and 64, we can derive a bound on the Rademacher complexity of a one-layer neural network with a Lipschitz activation function, i.e. $f(\mathbf{x}) = \sigma(\mathbf{W}\mathbf{x})$ where σ is Lipschitz. These results can also be extended to multiple layers by relating the Rademacher complexity of a layer to its previous layer (using a Contraction argument) and then use a bound similar to the one above for the first layer, see e.g. details in Neyshabur et al. (2015). This leads to the following theorem.

Theorem 65. *Consider an L-layer ReLU network with activation function $\sigma(t) = \max(0, t)$ with bounded norm:*

$$\mathcal{F}_B := \{\mathbf{x} \to \mathbf{W}_L \sigma(\mathbf{W}_{L-1} \dots \sigma(\mathbf{W}_1 \mathbf{x}) \dots), \mathbf{W}_k \in \mathbb{R}^{m_k \times m_{k-1}}, m_0 = dim(\mathbf{x}), m_L = 1, m_k \in \mathbb{N}, \|\mathbf{W}_i\|_F \leq I$$

Then for a training set $(\mathbf{x}_1, y_1), \dots (\mathbf{x}_n, y_n) \in \mathcal{X} \times \{\pm 1\}$ with $\|\mathbf{x}_i\| \leq 1$, we have that a.s.

$$\mathbb{R}(\mathcal{F}_B) \leq \frac{(2B)^L}{\sqrt{n}}.$$

As a word of caution, existing Rademacher bounds typically depend on the product of the weight matrices of a neural network, which lead to loose or even vacuous bounds (i.e. they have no predictive abilities). Compression arguments such as in Arora et al. (2018) can make these bounds tighter but this is very much an active area of research where significant advances are required to obtain practically useful bounds. To make things worse, the work of Zhang et al. (2021) has shown empirically that for certain datasets, neural networks can fit random labels with zero training error. This indicates that these networks can maximize the Rademacher complexity, therefore highlighting some potentially severe shortcomings of such approaches. In the next lecture, we will see an alternative approach based on PAC-Bayes bounds that

also have long-standing theoretical roots but are also currently seen as empirically promising to evaluate the generalization ability of deep neural networks.

8.5 Exercise: Generalization I

Problem 1 (Rademahcer Complexity):

Given a sample $\mathbf{x}_1, \ldots, \mathbf{x}_n \in \mathbb{R}^d$ drawn from some distribution $\mathcal{D}$, the empirical Rademacher complexity $\hat{\mathcal{R}}(\mathcal{F})$ of the hypothesis class $\mathcal{F}$ of binary classifier is defined as:

$$\hat{\mathcal{R}}(\mathcal{F}) = \mathbb{E}_{\boldsymbol{\sigma}}\left[\sup_{f\in\mathcal{F}} \frac{1}{n}\sum_{i=1}^{n} \sigma_i f(\mathbf{x}_i)\right],$$

where $\boldsymbol{\sigma} = (\sigma_1, \ldots, \sigma_n)$ are independent Rademacher random variables, i.e., $\mathbb{P}(\sigma_i = +1) = \mathbb{P}(\sigma_i = -1) = \frac{1}{2}$. The Rademacher complexity quantifies the richness of the function class $\mathcal{F}$ by measuring how well functions in $\mathcal{F}$ can fit random noise. It is widely used to provide generalization bounds in machine learning: with probability at least $1-\delta$ over sample draws, it holds that

$$\sup_{f\in\mathcal{F}} \left|\mathbb{E}_{x\sim\mathcal{D}}[f(x)] - \frac{1}{n}\sum_{i=1}^{n} f(x_i)\right| \leq 2\hat{\mathcal{R}}(\mathcal{F}) + 3\sqrt{\frac{\log(2/\delta)}{n}}.$$

Now, let's consider the following problem, which involves proving some important properties of Rademacher complexity.

a) For two hypothesis classes $\mathcal{F}, \mathcal{F}'$, prove that the Rademacher complexity of their sum satisfies the following inequality:

$$\hat{\mathcal{R}}(\mathcal{F} + \mathcal{F}') \leq \hat{\mathcal{R}}(\mathcal{F}) + \hat{\mathcal{R}}(\mathcal{F}').$$

b) Consider an L-Lipschitz continuous function $\phi : \mathbb{R} \to \mathbb{R}$, i.e. for all $t, t' \in \mathbb{R}$,

$$|\phi(t) - \phi(t')| \leq L|t - t'|,$$

and define a new class $\mathcal{F}' := \{\phi \circ f : f \in \mathcal{F}\}$. Prove that the Rademacher complexity of $\mathcal{F}'$ is bounded as follows:

$$\hat{\mathcal{R}}(\mathcal{F}') \leq L\hat{\mathcal{R}}(\mathcal{F}).$$

c) Let $\mathcal{F}$ be a class of real-valued functions and let $\ell(x) = \min(1, \max(0, x))$ be the hinged loss. Show that for any $f \in \mathcal{F}$, with probability at least $1-\delta$ over the sample draw, it holds that

$$\mathbb{E}[\ell(f(x), y)] \leq \frac{1}{n}\sum_{i=1}^{n} \ell(f(x_i), y_i) + 2\hat{\mathcal{R}}(\mathcal{F}) + 3\sqrt{\frac{2\log(1/\delta)}{n}}.$$

Problem 2 (Concentration of NTK Eigenvalues):

Consider a set of examples $\{\mathbf{x}_i\}_{i=1}^n$ and let $\mathbf{X} \in \mathbb{R}^{n\times d}$ be a data matrix containing the $\mathbf{x}_i$ as its rows. We consider a two-layer neural network of the following form,

$$f(\mathbf{W}, \mathbf{a}, \mathbf{x}) = \frac{1}{\sqrt{m}} \sum_{k=1}^{m} a_k \sigma(\mathbf{w}_k^\top \mathbf{x}),$$

where we assume that $|a_k| \leq 1$ and the activation function σ is differentiable and $|\sigma(\cdot)| \leq B$.

We define the sampled Gram NTK matrix $\hat{\mathbf{G}} = (\hat{G}_{ij})_{i,j} \in \mathbb{R}^{n\times n}$ and the expected Gram NTK matrix $\mathbf{G} = (G_{ij})_{i,j} \in \mathbb{R}^{n\times n}$ as follows

$$\hat{G}_{ij} := \frac{1}{m} \sum_{k=1}^{m} \mathbf{x}_i^\top \mathbf{x}_j \sigma'(\mathbf{w}_k^\top \mathbf{x}_i)\sigma'(\mathbf{w}_k^\top \mathbf{x}_j), \quad G_{ij} := \mathbb{E}_{\mathbf{w}} \mathbf{x}_i^\top \mathbf{x}_j \sigma'(\mathbf{w}^\top \mathbf{x}_i)\sigma'(\mathbf{w}^\top \mathbf{x}_j).$$

We can view the matrix $\hat{\mathbf{G}}$ as an average of a set of matrices $\mathbf{H}_1, ..., \mathbf{H}_m$, i.e. $\hat{\mathbf{G}} = \frac{1}{m}\sum_{k=1}^m \mathbf{H}_k$, where

$$(\mathbf{H}_k)_{ij} := \mathbf{x}_i^\top \mathbf{x}_j \sigma'(\mathbf{w}_k^\top \mathbf{x}_i)\sigma'(\mathbf{w}_k^\top \mathbf{x}_j), \quad k = 1, ..., m.$$

The goal of this exercise is to bound the deviation between $\hat{\mathbf{G}}$ and $\mathbf{G}$. To do so, we will use a concentration bound that is based from what we have seen in the main lecture. We will use the Frobenius inner product notation $\langle \mathbf{A}, \mathbf{B}\rangle_F = \operatorname{tr}\left(\mathbf{A}^\top \mathbf{B}\right)$ and note that $|\langle \mathbf{A}, \mathbf{B}\rangle| \leq \|\mathbf{A}\|_F \|\mathbf{B}\|_F$.

a) Prove that $\|\mathbf{H}_k\|_F \leq B^2 \|\mathbf{X}\|_F^2$.

b) Define $\mathcal{F} := \{\mathbf{U} \to \langle \mathbf{U}, \mathbf{V}\rangle_F : \|\mathbf{V}\|_F \leq 1\}$, $\mathcal{H} := (\mathbf{H}_1, \dots \mathbf{H}_m)$. The Rademacher complexity of $\mathcal{F}_{|\mathcal{H}}$ is defined as

$$\mathcal{R}(\mathcal{F}_{|\mathcal{H}}) = \frac{1}{m} \mathbb{E}_\epsilon \sup_{\mathbf{V}} \sum_{i=1}^{m} \epsilon_i \langle \mathbf{H}_i, \mathbf{V}\rangle_F.$$

Prove that $\mathcal{R}(\mathcal{F}_{|\mathcal{H}}) \leq \frac{1}{\sqrt{m}} B^2 \|\mathbf{X}\|_F^2$.

c) Recall the concentration bound based on Rademacher complexities:

Theorem 66. *Let $\mathcal{F} : \mathbb{R}^{n\times n} \to \mathbb{R}$ be a function class such that $\forall f \in \mathcal{F}, f(\mathbf{H}) \in [a, b]$ a.s.. Then with probability at least $1 - \delta$ over the draw of $\mathbf{H}_1, ..., \mathbf{H}_m$,*

$$\sup_{f\in\mathcal{F}} f(\mathbf{H}) - \frac{1}{m}\sum_{k=1}^{m} f(\mathbf{H}_k) \leq 2\mathcal{R}(\mathcal{F}_{|\mathcal{H}}) + 3(b-a)\sqrt{\frac{\log(2/\delta)}{2n}}.$$

Prove that with probability at least $1-\delta$ over the draw of $(\mathbf{w}_1, \dots \mathbf{w}_m)$ for $\|\mathbf{u}\|_2 \leq 1$, it holds that:

$$|\mathbf{u}^\top \mathbf{G}\mathbf{u} - \mathbf{u}^\top \hat{\mathbf{G}}\mathbf{u}| \leq \frac{2B^2\|\mathbf{X}\|_F^2}{\sqrt{m}} + 6B^2\|\mathbf{X}\|_F^2\sqrt{\frac{\log(2/\delta)}{2m}}. \tag{8.28}$$

d) Denote $\star := \frac{2B^2\|\mathbf{X}\|_F^2}{\sqrt{m}} + 6B^2\|\mathbf{X}\|_F^2\sqrt{\frac{\log(2/\delta)}{2m}}$ to be the bound in Eq. (8.28). Prove that with probability at least $1-\delta$,

$$\lambda_{\min}(\hat{\mathbf{G}}) \geq \lambda_{\min}(\mathbf{G}) - \star \text{ and } \lambda_{\max}(\hat{\mathbf{G}}) \leq \lambda_{\max}(\mathbf{G}) + \star.$$

Hint: Weyl's inequality bounds the eigenvalues of the sum of Hermitian matrices.

Theorem 67 (Weyl's inequality)**.** *Let $\mathbf{A}, \mathbf{B}$ be two Hermitian $n \times n$ matrices. Denote by $\lambda_1(\mathbf{A}) \geq \cdots \geq \lambda_n(\mathbf{A})$ the sorted eigenvalues of $\mathbf{A}$. Then*

$$\lambda_1(\mathbf{A}+\mathbf{B}) \leq \lambda_1(\mathbf{A}) + \lambda_1(\mathbf{B}) \tag{8.29}$$

and

$$\lambda_n(\mathbf{A}+\mathbf{B}) \geq \lambda_n(\mathbf{A}) + \lambda_n(\mathbf{B}) \tag{8.30}$$

Chapter 9

Generalization II: PAC-Bayes Bounds

Recall that in our first attempt at deriving a generalization bound, we derived an upper bound for a fixed hypothesis f and then apply a union bound for all $f \in \mathcal{F}$. In this lecture, we will see that an alternative approach to using a union bound is to use randomized hypotheses. We start with the general formulation of PAC-Bayes bounds and then discuss its application to deep neural networks.

9.1 General formulation PAC-Bayes bounds

We consider a hypothesis space as a set $\mathcal{F}$ of functions from $\mathcal{X} \subseteq \mathbb{R}^d$ to $\mathcal{Y} \subseteq \mathbb{R}$. Given a data distribution $\mathcal{D}$ over $(\mathcal{X}, \mathcal{Y})$, we define the true risk as

$$R(f) = \mathbb{E}_{(\mathbf{x},y)\sim\mathcal{D}}[l((\mathbf{x}, y), f)],$$

where $l((\mathbf{x}, y), f) = \mathbf{1}[f(\mathbf{x}) \neq y]$.

Since we typically do not have access to the exact data distribution but only a sample set $S = \{(\mathbf{x}_i, y_i)\}_{i=1}^N \sim D$, we also define the empirical risk as

$$R_S(f) = \frac{1}{N} \sum_{(x,y)\in S} [l((\mathbf{x}, y), f)].$$

Our goal is to obtain an upper bound on $R(f)$ in terms of $R_S(f)$. More generally, we are interested in the generalization gap:

$$\delta_f^S := \max(0, R(f) - R_S(f)) \leq |R(f) - R_S(f)|.$$

We will define two distributions:

1. Prior distribution P on a hypothesis set $\mathcal{F}$ that does not depend on the data S. This prior is typically "wide", i.e. we won't exclude many functions unless we have a precise idea of what functions $f \in \mathcal{F}$ should be allowed.

2. Posterior distribution $Q := Q(S)$ on the hypothesis set $\mathcal{F}$ that can depend on the data S. As we will see shortly, our model (e.g. a deep neural network) will be assumed to be drawn from the distribution Q.

Note that we will typically assume that $P \gg Q$, i.e. the support of P dominates the support of Q. Formally, this means that

$$Q \ll P \qquad \text{if and only if} \qquad \text{for all } f \in \mathcal{F}, \quad P(f) = 0 \;\text{ implies }\; Q(f) = 0.$$

The general idea behind PAC-Bayes bounds is to construct a stochastic classifier via the distribution Q and bound the expected generalization gap $\mathbb{E}_{f\sim Q}[\delta_f^S]$. This means that we do not choose a specific classifier $f \in \mathcal{F}$ anymore, but a distribution Q over the space of functions $\mathcal{F}$.

KL divergence The Kullback-Leibler (KL) divergence measures how one probability distribution diverges from a second probability distribution. Formally, the KL divergence between two probability distributions P and Q is defined as follows.

For discrete distributions:

$$\mathrm{KL}(P||Q) = \sum_i P(i) \cdot \ln\left(\frac{P(i)}{Q(i)}\right). \tag{9.1}$$

For continuous distributions:

$$\mathrm{KL}(P||Q) = \int P(x) \cdot \ln\left(\frac{P(x)}{Q(x)}\right) dx. \tag{9.2}$$

Note that the KL divergence is non-negative and it is zero if and only if $P = Q$. Note however that this metric is not symmetric in general, i.e. $\mathrm{KL}(Q||P) \neq \mathrm{KL}(P||Q)$.

Change of measure One first problem we need to address is that the distribution Q depends on S, potentially in a complicated way. A key step in the PAC-Bayes analysis is to convert the expectation under Q to an expectation under P. To do so, we will use the following key result from (Donsker and Varadhan, 1975).

Lemma 68 (Change of measure inequality (Donsker and Varadhan, 1975))**.** *For any $P \gg Q$, P-measurable function [a] ϕ (random variable),*

$$\mathbb{E}_{f\sim Q}[\phi(f)] \leq KL(Q||P) + \ln \mathbb{E}_{f\sim P}\left[e^{\phi(f)}\right].$$

[a] Let (X, Σ_X) and (Y, Σ_Y) be measurable spaces, where Σ_X and Σ_Y are sigma-algebras on sets X and Y, respectively. A function $f : X \to Y$ is said to be measurable if, for every measurable set B in Y, the set $\{x \in X : f(x) \in B\}$ is measurable in X. In other words, the inverse image of any measurable set in the codomain is a measurable set in the domain.

Proof. Measure dominance (i) and Jensen's inequality (ii) yield

$$\begin{aligned}
\ln \mathbb{E}_{f\sim P}\left[e^{\phi(f)}\right] &= \ln \mathbb{E}_{f\sim P}\left[e^{\phi(f)}\frac{Q(f)}{Q(f)}\right] \\
&= \ln \int P(f)e^{\phi(f)}\frac{Q(f)}{Q(f)}df \\
&\overset{(i)}{=} \ln \mathbb{E}_{f\sim Q}\left[e^{\phi(f)}\frac{P(f)}{Q(f)}\right] \\
&\overset{(ii)}{\geq} \mathbb{E}_{f\sim Q}\ln\left[e^{\phi(f)}\frac{P(f)}{Q(f)}\right] \\
&= \mathbb{E}_{f\sim Q}[\phi(f)] - \underbrace{\mathbb{E}_{f\sim Q}[\ln Q(f) - \ln P(f)]}_{=\mathrm{KL}(Q||P)},
\end{aligned}$$

where (i) is a change of measure that requires $P \gg Q$ (this guarantees the existence of the Radon-Nikodym derivative which is the density ratio $\frac{dQ}{dP} = \frac{Q(f)}{P(f)}$). □

Note that for the bound to provide a tight estimate, we need $\mathrm{KL}(Q||P)$ to be small, i.e. Q and P should be close to each other.

PAC-Bayesian Theorem We will first prove a general version derived by Bégin et al. (2014) that makes use of a general function Δ to measure the distance between $\mathbb{E}_Q[R(f)]$ and $\mathbb{E}_Q[R_S(f)]$ (we will later use an explicit function Δ). Before we state the theorem, note that since we consider the 0-1 loss and the samples are taken to be i.i.d., then $NR_S(f)$ can only take values from the set $\{0, \dots N\}$ and it follows a binomial distribution $\mathrm{Bin}(N, R(f))$ (we will recall the definition of a binomial distribution below).

Theorem 69 ((Bégin et al., 2014))**.** *For fixed P and any Q, $\epsilon \in (0,1)$ and for any $\Delta : [0,1]\times[0,1] \to \mathbb{R}$ convex function, the following holds with probability greater than $1-\epsilon$ over sample sets S of size N:*

$$\Delta(\mathbb{E}_Q[R(f)], \mathbb{E}_Q[R_S(f)]) \leq \frac{1}{N}\left(KL(Q||P) + \ln\frac{J_\Delta(N)}{\epsilon}\right),$$

where $J_\Delta(N) = \left(\sup_{m\in[0,1]}\sum_{k=0}^{N} Bin(k;N,m)\exp\left(N\Delta\left(m, \frac{k}{N}\right)\right)\right)$.

Proof. We first use Jensen's inequality and apply Lemma 68 with $\phi = N\Delta$:

$$\begin{aligned}
N\Delta(\mathbb{E}_Q[R(f)], \mathbb{E}_Q[R_S(f)]) &\leq \mathbb{E}_Q[N\Delta(R(f), R_S(f))] \quad \text{Jensen's inequality} \\
&\leq \mathrm{KL}(Q||P) + \ln \mathbb{E}_P\left[\exp(N\Delta(R(f), R_S(f)))\right] \quad \text{Change of measure,}
\end{aligned}$$

which implies

$$\Delta(\mathbb{E}_Q[R(f)], \mathbb{E}_Q[R_S(f)]) \leq \frac{1}{N}\left(\mathrm{KL}(Q||P) + \ln \mathbb{E}_P\left[\exp(N\Delta(R(f), R_S(f)))\right]\right).$$

Next, recall that by Markov's inequality:

$$\Pr\left(\ln X \geq \ln \frac{\mathbb{E}[X]}{\epsilon}\right) = \Pr\left(X \geq \frac{\mathbb{E}[X]}{\epsilon}\right) \leq \epsilon.$$

Applying Markov's inequality to the term $\ln \mathbb{E}_P\left[\exp(N\Delta(R(f), R_S(f)))\right]$ where the expectation is taken w.r.t. S, we get that with probability at least $1-\epsilon$,

$$\Delta(\mathbb{E}_Q[R(f)], \mathbb{E}_Q[R_S(f)]) \leq \frac{1}{N}\left(\mathrm{KL}(Q||P) + \ln \mathbb{E}_S\mathbb{E}_P\left[\exp(N\Delta(R(f), R_S(f)))\right] - \ln \epsilon\right).$$

Then, note that $NR_S(f)|f$ is a binomial random variable (where $|$ means conditioned on) with N coin tosses, each with probability $R(f)$. We define the corresponding distribution as

$$\mathrm{Bin}(k; N, R(f)) = \binom{N}{k}(R(f))^k(1-R(f))^{N-k}, \tag{9.3}$$

where k is the number of trials with success $R(f)$.

Therefore,

$$\begin{aligned}\mathbb{E}_{f\sim P}\mathbb{E}_S\left[\exp(N\Delta(R(f), R_S(f)))\right] &= \mathbb{E}_{f\sim P}\sum_{k=0}^{N}\mathrm{Bin}(k; N, R(f))\exp\left(N\Delta\left(R(f), \frac{k}{N}\right)\right) \\ &\leq \sup_{m\in[0,1]}\sum_{k=0}^{N}\mathrm{Bin}(k; N, m)\exp\left(N\Delta\left(m, \frac{k}{N}\right)\right).\end{aligned}$$

Putting things together, we obtain the final result:

$$\Delta(\mathbb{E}_Q[R(f)], \mathbb{E}_Q[R_S(f)]) \leq \frac{1}{N}\left(\mathrm{KL}(Q||P) + \ln\left(\sup_{m\in[0,1]}\sum_{k=0}^{N}\mathrm{Bin}(k; N, m)\exp\left(N\Delta\left(m, \frac{k}{N}\right)\right)\right) - \ln\epsilon\right)$$

□

Variants of the theorem can be derived for specific choices of Δ. We here state a result by (McAllester, 2003) in the case of the square function.

Theorem 70 ((McAllester, 2003)). *For fixed P and any Q, $\epsilon \in (0;1)$ with probability greater than $1-\epsilon$ over sample sets S of size N:*

$$\mathbb{E}_{f\sim Q}[R(f)] - \mathbb{E}_{f\sim Q}[R_S(f)] \leq \sqrt{\frac{2}{N}\left[KL(Q||P) + \ln\left(\frac{2\sqrt{N}}{\epsilon}\right)\right]}$$

Proof idea. The proof is a combination of Theorem 69 where $\Delta = (R(f) - R_S(f))^2$ is the square function.

□

The general rate $\tilde{\mathcal{O}}(1/\sqrt{N})$ [1] is very simple and does not hide complex constants in it. However, the bound holds for a stochastic classifier.

Comparison to VC theory VC bounds discussed earlier in the class depend on the complexity of the model (through the number of parameters) and do not consider any property of the underlying data distribution we are trying to approximate. Deep learning models tend to have a very large number of parameters and it is therefore not surprising that VC bounds yield so-called vacuous bounds (i.e. these bounds do not explain generalization). On the contrary, PAC-Bayes bounds do not depend at all on the model complexity of the underlying function class. Instead, they depend on a prior P and the output of the algorithm encoded in Q, which depends on the data distribution.

9.2 PAC-Bayesian for DNNs

A general recipe used to apply PAC-Bayes bounds to deep neural networks is described in Dziugaite and Roy (2017). Broadly, it consists of the following steps:

1. Choose a Gaussian $P = \mathcal{N}(\theta_0, \lambda\mathbf{I})$, e.g. a simple way should be $\theta_0 = \mathbf{0}$, $\lambda = 1$ but one could also cross-validate over these parameters
2. Choose a Gaussian $Q = \mathcal{N}(\theta, \text{diag}(\sigma))$, where θ are the parameters of our model (e.g. a fully-trained DNN) and where σ_i is the variance in the i-th weight. Choosing σ_i small means that we need more precision for this specific parameter
3. With our choice of P and Q, we can derive a simple expression for $\text{KL}(Q||P)$
4. Minimize PAC-Bayes bound (typically using a surrogate loss)

$$\mathbb{E}_{f\sim Q}[R(f)] - \mathbb{E}_{f\sim Q}[R_S(f)] \leq \sqrt{\frac{2}{N}\left[\text{KL}(Q||P) + \ln\left(\frac{2\sqrt{N}}{\epsilon}\right)\right]}$$

   - achieve small error on sample
   - find wide minima: robust to large parameter perturbations

An experimental evaluation of the resulting bound evaluated on binary version of MNIST is given in Table 1 below (taken from (Dziugaite and Roy, 2017)).

[1] The notation $\tilde{\mathcal{O}}$ is the same as $\mathcal{O}$ with potentially additional ln terms.

Experiment	T-600	T-1200	T-300^2	T-600^2	T-1200^2	T-600^3	R-600
Train error	0.001	0.002	0.000	0.000	0.000	0.000	0.007
Test error	0.018	0.018	0.015	0.016	0.015	0.013	0.508
SNN train error	0.028	0.027	0.027	0.028	0.029	0.027	0.112
SNN test error	0.034	0.035	0.034	0.033	0.035	0.032	0.503
PAC-Bayes bound	0.161	0.179	0.170	0.186	0.223	0.201	1.352
KL divergence	5144	5977	5791	6534	8558	7861	201131
# parameters	471k	943k	326k	832k	2384k	1193k	472k
VC dimension	26m	56m	26m	66m	187m	121m	26m

Table 1: Results for experiments on binary class variant of MNIST. SGD is either trained on (T) true labels or (R) random labels. The network architecture is expressed as N^L, indicating L hidden layers with N nodes each. Errors are classification error. The reported VC dimension is the best known upper bound (in millions) for ReLU networks. The SNN error rates are tight upper bounds (see text for details). The PAC-Bayes bounds upper bound the test error with probability 0.965.

Additional resources: ICML tutorial 2019 by Benjamin Guedj John Shawe-Taylor, see `https://bguedj.github.io/icml2019/material/main.pdf`.

Chapter 10

Architecture

Acknowledgment

Section 10.4 on Transformers is based on the book by Christopher M. Bishop and Hugh Bishop's "Deep Learning: Foundations and Concepts."

10.1 Convolutional Neural Networks (CNNs)

From MLPs to CNNs Unlike Multi-Layer Perceptrons (MLPs), which rely on densely connected layers, Convolutional Neural Networks (CNNs) are structured with alternating convolutional and pooling layers. This makes them particularly effective for handling grid-like data such as images. Convolutional layers extract features using filters that scan across the image, while pooling layers reduce the spatial size of feature maps to lower computational cost and mitigate overfitting. In contrast to MLPs, convolutional layers apply the same filter parameters (i.e., the same weights) at every spatial location in the input – a design choice known as weight sharing.

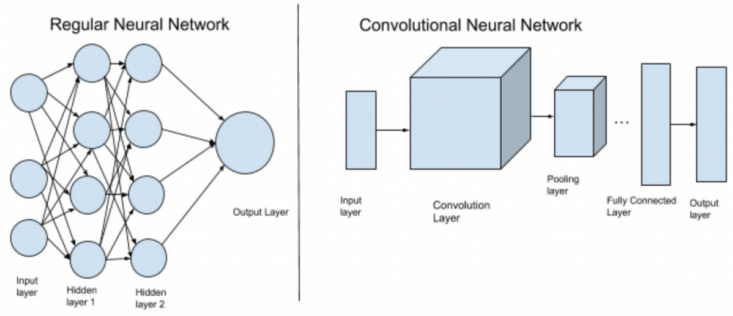


Figure 10.1: MLP vs CNN

Constraining the Function Class In machine learning, good performance often depends on choosing a function class suitable for the task. For image classification, a desirable property is *translation invariance* — the model should recognize an object regardless of its position in the image. CNNs are naturally designed to achieve translation invariance. By applying shared filters across spatial locations and using pooling to retain the most relevant features, they can recognize patterns even when they shift in position.

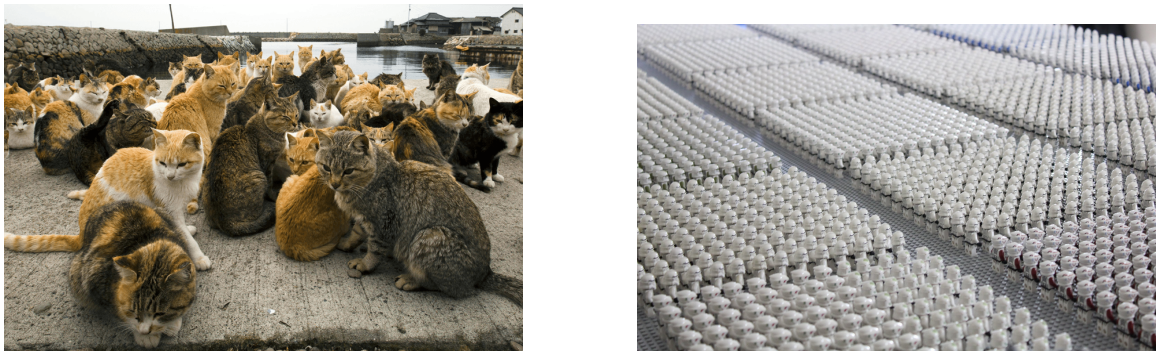


Figure 10.2: Illustration of translation invariance in classification tasks.

Core Components of CNNs A typical Convolutional Neural Network (CNN) is built by stacking layers of the following types:

- **Convolutional layers** that extract local patterns.
- **Pooling layers** that reduce spatial size.
- **Fully-connected layers** that interpret high-level features.
- **Output layers** that produce final predictions, e.g., using sigmoid for binary classification:

$$y_1 = P(Y = 1 \,|\, \mathbf{x}) = \frac{1}{1 + \exp\left[-f(\mathbf{x})\right]},$$

where $\mathbf{x}$ is an input image and $f(\mathbf{x})$ is the corresponding output of the CNN.

10.1.1 Understanding Convolutions

Signals and Operators A signal $f : \mathbb{R} \to \mathbb{R}$ can represent time series, images, or audio. Transformations on signals are described by operators $T : f \mapsto Tf$, often expressed via integral operators:

$$(Tf)(u) = \int_{t_1}^{t_2} H(u,t) f(t) dt.$$

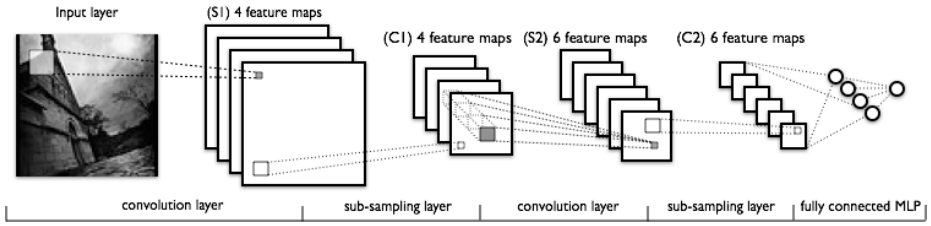


Figure 10.3: CNN Building Blocks. The output of convolving one filter with the input is called a feature map.

Fourier Transform A common example of an operator is the Fourier transform with $H(u,t) = e^{-2\pi itu}$:

$$(\mathcal{F}f)(u) = \int_{-\infty}^{\infty} e^{-2\pi itu} f(t)dt.$$

The convolution operation, denoted by the symbol $*$, combines two functions:

$$(f * h)(u) = \int_{-\infty}^{\infty} h(u-t)f(t)dt = \int_{-\infty}^{\infty} f(u-t)h(t)dt.$$

It is linear, commutative, and shift-equivariant. A 1-dimensional example is shown in Figure 10.4. Convolutions can also be adapted to higher dimensions, a 2-dimensional example is shown in Figure 10.5 and will be discussed in more detail later on.

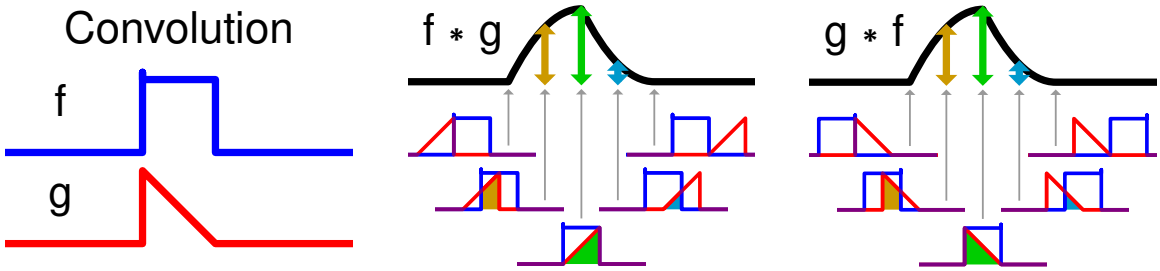


Figure 10.4: Visualization of continuous convolution (source: Wikipedia).

Properties and Examples

- **Shift Equivariance:** Shifting the input shifts the output: $f_\Delta * h = (f * h)_\Delta$.
- **Discrete Convolution:** For sequences $f, h : \mathbb{Z} \to \mathbb{R}$,

$$(f * h)[u] = \sum_{t=-\infty}^{\infty} f[t]h[u-t].$$

Discrete convolution with finite support can be represented as matrix multiplication using Toeplitz matrices (a Toeplitz matrix is one in which each descending diagonal from left to right is constant, i.e., its (i,j)-entry depends only on $i-j$).

- **Gaussian Kernel:** Example with kernel on $[-2, 2]$:

$$h[t] = \frac{1}{16} \begin{cases} 6, & t = 0 \\ 4, & |t| = 1 \\ 1, & |t| = 2 \\ 0, & \text{else} \end{cases}.$$

Cross-Correlation A cross-correlation is a closely related mathematical operation to convolution and only differs in how they handle the ordering of one of the input sequences. It is defined as:

$$(h \star f)[u] = \sum_t h[t] f[u + t], \quad \text{so } h \star f = \bar{h} * f, \text{ with } \bar{h}[t] = h[-t].$$

10.1.2 Two-dimensional convolution

Convolutions extend naturally to multiple dimensions (2D and beyond). For two continuous functions $f(x, y)$ and $g(x, y)$, their 2D convolution is defined as:

$$(f * g)(x, y) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f(a, b)\, g(x - a,\ y - b)\, \mathrm{d}a\, \mathrm{d}b.$$

Equivalently, by commutativity:

$$(f * g)(x, y) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} g(a, b)\, f(x - a,\ y - b)\, \mathrm{d}a\, \mathrm{d}b.$$

The "flip" of g is implicit in writing $g(x - a, y - b)$.

Discrete 2D Convolution

For discrete 2D signals (e.g., an image) $I[m, n]$ and a kernel $K[u, v]$, the (infinite-domain) discrete convolution is

$$(I * K)[i, j] = \sum_{u=-\infty}^{\infty} \sum_{v=-\infty}^{\infty} I[i - u,\ j - v]\ K[u, v].$$

In practice, I and K have finite support. If K is nonzero only for $0 \le u < H_K$ and $0 \le v < W_K$, we re-index with a center (u_0, v_0), often $u_0 = \lfloor H_K/2 \rfloor$, $v_0 = \lfloor W_K/2 \rfloor$. One formulation is:

$$(I * K)[i, j] = \sum_{u=0}^{H_K - 1} \sum_{v=0}^{W_K - 1} I[i - u_0 + u,\ j - v_0 + v]\ K[u, v],$$

where values of $I[\cdot, \cdot]$ outside the domain are handled via padding (e.g., zero-padding) or omitted for "valid" convolution.

For input I of size $H_I \times W_I$ and kernel K of size $H_K \times W_K$, the valid convolution output size is $(H_I - H_K + 1) \times (W_I - W_K + 1)$. As mentioned earlier, padding can also be used to ensure the output preserves the size of the input.

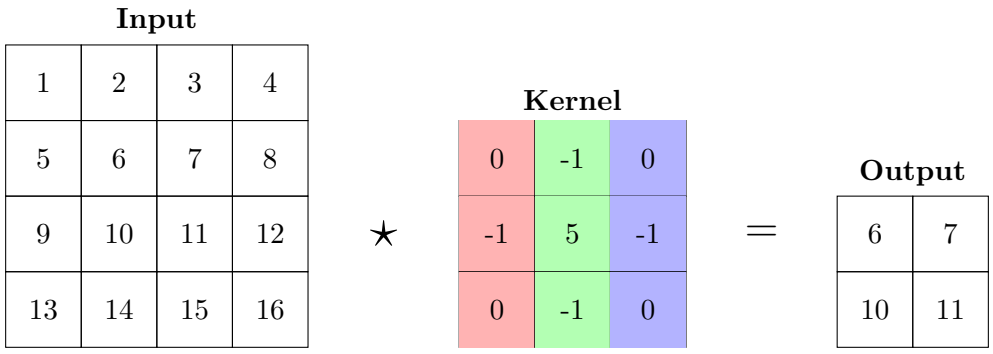


Figure 10.5: Illustration of a 2d convolution of a 4×4 input by a 3×3 kernel.

2D Convolution in CNNs (Multi-Channel)

In convolutional neural networks, inputs and outputs are multi-channel feature maps. Let

$$X \in \mathbb{R}^{H \times W \times C_{\text{in}}}$$

be the input feature map. A convolutional layer has C_{out} kernels, each of size $H_K \times W_K \times C_{\text{in}}$. Denote the m-th kernel by $K^{(m)} \in \mathbb{R}^{H_K \times W_K \times C_{\text{in}}}$, with bias $b^{(m)}$. The output feature map

$$Y \in \mathbb{R}^{H' \times W' \times C_{\text{out}}}$$

is given by, for each output channel m and spatial location (i, j):

$$Y[i, j, m] = \sum_{c=1}^{C_{\text{in}}} \sum_{u=0}^{H_K-1} \sum_{v=0}^{W_K-1} X[i \cdot s_h + u - p_h,\ j \cdot s_w + v - p_w,\ c] \cdot K^{(m)}[H_K - 1 - u,\ W_K - 1 - v,\ c] + b^{(m)},$$

where:

- s_h, s_w are the vertical and horizontal strides.
- p_h, p_w are the vertical and horizontal padding amounts.
- The indices $H_K - 1 - u,\ W_K - 1 - v$ implement the 180° flip.
- Boundary handling: If $(i \cdot s_h + u - p_h,\ j \cdot s_w + v - p_w)$ is outside $[0, H-1] \times [0, W-1]$, one uses padding values (often zero).

Many deep learning frameworks implement the above without explicitly flipping the kernel (i.e., they use cross-correlation in code) but still call it "convolution"; during training, the learned kernel adapts accordingly.

10.1.3 Convolutions in Neural Networks

A Convolutional Neural Network (CNN) is a type of deep learning architecture particularly well suited for processing data that has a grid-like topology – most famously, images (2D grids of pixels) but also audio spectrograms, video, and certain time-series. At a high level, a CNN learns to recognize patterns and hierarchies of features (edges $\rightarrow$ textures $\rightarrow$ object parts $\rightarrow$ objects) by stacking special layers that exploit spatial structure. The core building block of a CNN is a convolutional layer whose purpose is to automatically learn and extract spatially local patterns (edges, textures, shapes) from an input (e.g. an image) by applying a set of learnable filters (also called kernels) across the input. At a high-level, convolutional layers aim to:

- Preserve shift equivariance.
- Leverage local and hierarchical structure.
- Learn filters that are optimal for a given dataset (via gradient descent as we will soon discuss).
- Improve efficiency compared to fully-connected layers.

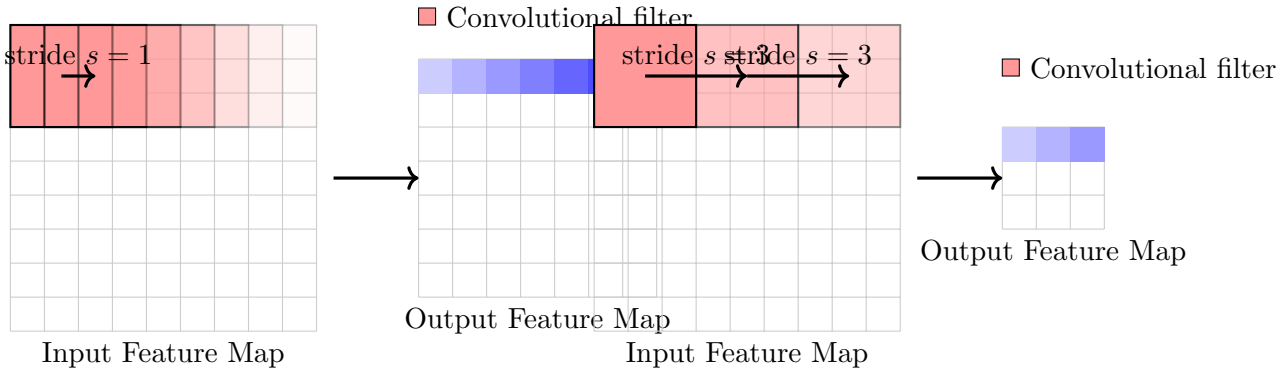
In a convolutional neural network, each filter is convolved over the full spatial dimensions of its input to produce an output known as a feature map (see Figure 10.6). Although convolutional layers inherently preserve the local structure of the input, applying a filter without any padding causes the output's spatial dimensions to shrink – dropping information at the borders. To prevent this loss, one can surround the input with extra pixels (typically set to zero), a technique called zero-padding, which ensures that the resulting feature maps retain the same height and width as the original image (see Figure 10.7).

Learning Filters In contrast to traditional, hand-crafted filters (e.g., Sobel or Gaussian kernels), the convolutional kernels in a CNN are *parameters* learned directly from data. A single convolutional layer maintains a collection of K filters

$$W^{(k)} \in \mathbb{R}^{F \times F \times D}, \quad k = 1, 2, \ldots, K,$$

where F is the spatial size and D is the depth of the input volume (i.e. the number of channels). During training, the network defines a loss function $\mathcal{L}(\theta)$ (for example, cross-entropy for classification) and uses backpropagation to compute the gradient of the loss with respect to each filter weight:

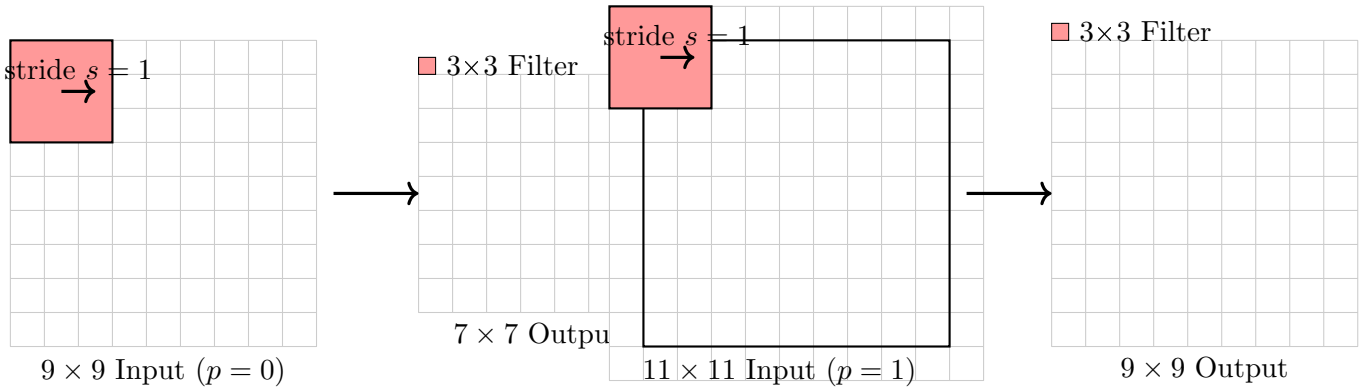
$$\frac{\partial \mathcal{L}}{\partial W^{(k)}_{u,v,d}} \quad \text{for all } u, v, d, k.$$



(a) Stride $s = 1$: overlapping windows $\rightarrow$ 7×7 map.

(b) Stride $s = 3$: non-overlapping 3×3 windows $\rightarrow$ 3×3 map.

Figure 10.6: Comparison of convolution strides on a 9×9 input: (a) stride 1 yields a 7×7 feature map with overlapping receptive fields; (b) stride 3 yields a 3×3 feature map.



(a) No padding ($p = 0$): $9 \times 9 \rightarrow 7 \times 7$.

(b) Zero-padding $p = 1$: $11 \times 11 \rightarrow 9 \times 9$.

Figure 10.7: Effect of zero-padding on convolution output size.

An optimizer (e.g., stochastic gradient descent or Adam) then updates each weight according to

$$W^{(k)} \longleftarrow W^{(k)} - \eta \frac{\partial \mathcal{L}}{\partial W^{(k)}},$$

where η is the learning rate. Over many iterations, the filters adapt to detect the most informative local patterns (edges, textures, object parts, etc.) for the target task.

The convolutional filters that are learned from this process typically extract information about edges (as shown in Figure 10.8) but also texture and other important information.

Figure 10.8: Illustration of learned edge-detection filters applied to an input image. (Left) Two convolutional kernels specialized for detecting horizontal and vertical edges. (Right) The original image alongside the outputs produced by each of these filters.

Receptive Fields and Weight Sharing Deeper layers cover larger regions of the input, while weight sharing ensures filter efficiency.

Efficient Convolution via the Fast Fourier Transform By the Convolution Theorem, the convolution of two signals or functions f and h can be performed more efficiently in the frequency domain:

$$f * h = \mathcal{F}^{-1}\{\,\mathcal{F}\{f\} \cdot \mathcal{F}\{h\}\},$$

where

- $\mathcal{F}\{\cdot\}$ and $\mathcal{F}^{-1}\{\cdot\}$ denote the discrete Fourier transform (DFT) and its inverse,
- the product $\mathcal{F}\{f\} \cdot \mathcal{F}\{h\}$ is taken pointwise in the frequency domain.

Using the Fast Fourier Transform (FFT) to compute each DFT reduces the computational complexity from $\mathcal{O}(N^2)$ (direct convolution) to $\mathcal{O}(N \log N)$, which is especially advantageous for large-scale or high-dimensional data.

10.1.4 The Full Architecture

Each convolutional layer typically includes two additional ingredients:

- An activation function (e.g., ReLU).
- A pooling layer that reduces the spatial resolution, often using max-pooling or average pooling that do downsampling by summarizing local neighborhoods (e.g., selecting the maximum or computing the average value), as illustrated in Figure 10.10.

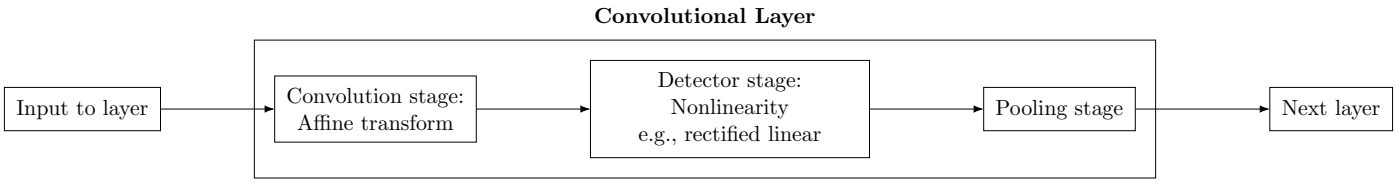


Figure 10.9: Convolutional layer stages: activation and pooling.

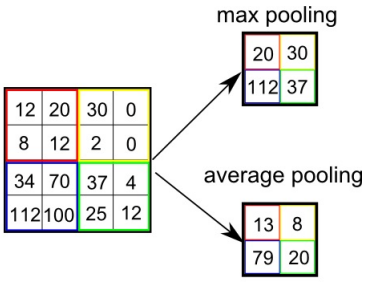


Figure 10.10: Max pooling over 2×2 window.

Channels and Fully-Connected Layers Each convolutional layer can learn multiple filters, generating multi-channel outputs, as shown in Figure 10.11. Toward the end, fully-connected layers integrate high-level features for classification, as shown in Figure 10.12.

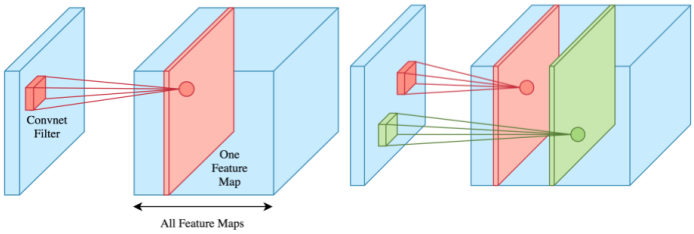


Figure 10.11: Multiple channels in convolutional layers.

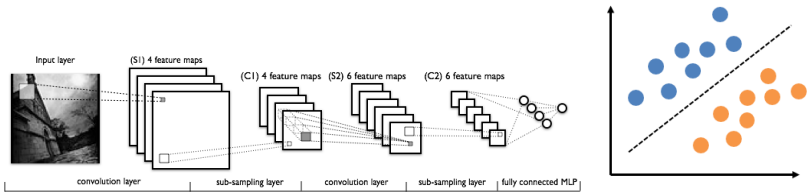


Figure 10.12: Fully-connected layers for classification.

10.2 Residual Layers

Training very deep networks often leads to optimization difficulties, notably vanishing gradients. A residual layer addresses this by explicitly allowing the layer to default to the identity mapping, while learning only the residual modification. Concretely, let

$$g : \mathbb{R}^m \to \mathbb{R}^m$$

be the desired transfer function of a layer. We parameterize it as

$$g(\mathbf{x}) = \mathbf{x} + f(\mathbf{x}; \boldsymbol{\theta}),$$

where $f(\mathbf{x}; \boldsymbol{\theta})$ is a small nonlinear function (e.g., a few convolutional layers). The identity function is the default behavior, ensuring that if $f(\mathbf{x}; \boldsymbol{\theta}) \approx 0$ at initialization, then

$$g(\mathbf{x}) \approx \mathbf{x}.$$

Moreover, the Jacobian of g is

$$J_g = I + J_f \approx I,$$

which preserves the magnitude of backpropagated gradients, facilitating stable gradient flow through many layers.

Linear Residuals When the input and output dimensions of a layer differ, one can still employ a residual connection by introducing a linear projection. Specifically,

$$g(\mathbf{x}) = \mathbf{W}\mathbf{x} + f(\mathbf{x}; \boldsymbol{\theta}),$$

where W is a fixed or learned matrix that matches the dimensions. Initializing $f(\mathbf{x}; \boldsymbol{\theta}) \approx 0$ yields

$$g(\mathbf{x}) \approx \mathbf{W}\mathbf{x},$$

and the Jacobian

$$\mathbf{J}_g = \mathbf{W} + \mathbf{J}_f \approx \mathbf{W}.$$

Thus, gradients can flow through the linear path defined by $\mathbf{W}$, which again ensures well-behaved optimization at the start of training.

Residual Networks (ResNets) The ResNet architecture He et al. (2016) integrates residual layers into a standard deep convolutional network (e.g., VGG-style) by adding "shortcut" branches that skip one or more layers and add directly to the output. Figure 10.13 illustrates a 34-layer ResNet, where arrows denote the residual connections.

Because each block can learn modifications around the identity, training very deep networks (over 100 layers) becomes feasible.

Below, we show how to implement a version of ResNet with 9 layers in pytorch.

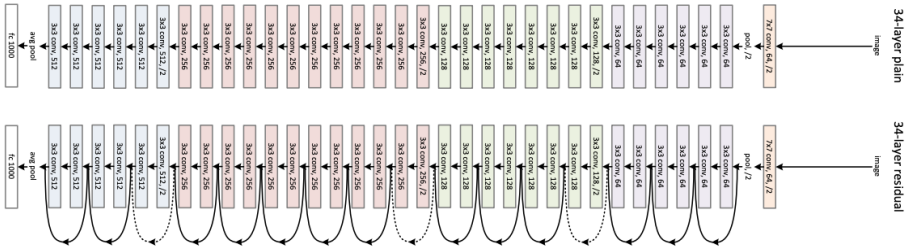


Figure 10.13: 34-layer ResNet architecture with residual connections.

```python
def conv_block(in_channels, out_channels, pool=False):
    """
    Creates a small convolutional block:
      - 3x3 convolution with padding to preserve spatial dimensions
      - Batch normalization for stable training
      - ReLU activation for non-linearity
      - Optional 2x2 max pooling to reduce feature map size
    """
    layers = [
        nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1),  # conv: in->out, preserves HxW
        nn.BatchNorm2d(out_channels),
       # normalize activations per batch
        nn.ReLU(inplace=True)
       # apply ReLU in-place
    ]
    if pool:
        layers.append(nn.MaxPool2d(2))  # downsample by factor of 2
    return nn.Sequential(*layers)       # wrap layers into a single module


class ResNet9(nn.Module):
    """
    A compact ResNet-like architecture with 9 layers:
      - 4 conv blocks with increasing channels
      - 2 residual connections (each summing two conv blocks)
      - Final classifier with global pooling and linear layer
    """
    def __init__(self, in_channels, num_classes):
        super().__init__()

        # Initial conv layers
        self.conv1 = conv_block(in_channels, 64)                # preserves spatial size
        self.conv2 = conv_block(64, 128, pool=True)             # downsample to half

        # First residual block: two conv blocks on 128 channels
```

```
34         self.res1 = nn.Sequential(
35             conv_block(128, 128),                                    #
   identity-mapping conv
36             conv_block(128, 128)                                     #
   another conv block
37         )
38
39         # Deeper conv layers
40         self.conv3 = conv_block(128, 256, pool=True)                 #
   downsample
41         self.conv4 = conv_block(256, 512, pool=True)                 #
   downsample further
42
43         # Second residual block: two conv blocks on 512 channels
44         self.res2 = nn.Sequential(
45             conv_block(512, 512),
46             conv_block(512, 512)
47         )
48
49         # Classification head: global pooling -> flatten -> linear
50         self.classifier = nn.Sequential(
51             nn.MaxPool2d(4),                                         #
   aggregate spatial dims to 1x1
52             nn.Flatten(),                                            #
   flatten to vector
53             nn.Linear(512, num_classes)                              #
   final logits
54         )
55
56     def forward(self, xb):
57         # Apply initial conv layers
58         out = self.conv1(xb)
59         out = self.conv2(out)
60
61         # First residual connection: add input of block to its
   output
62         out = self.res1(out) + out
63
64         # Continue with deeper conv layers
65         out = self.conv3(out)
66         out = self.conv4(out)
67
68         # Second residual connection
69         out = self.res2(out) + out
70
71         # Classification
72         out = self.classifier(out)
73         return out
```

Listing 10.1: ResNet9 with Convolutional Blocks and Residual Connections

Empirical Results on ImageNet He et al. (2016) compared deep networks on the ImageNet dataset to evaluate the effect of depth with and without residual connections. Figure 10.14 shows that without residual connections, deeper networks perform worse due to optimization difficulty, whereas with residual connections, performance improves with depth.

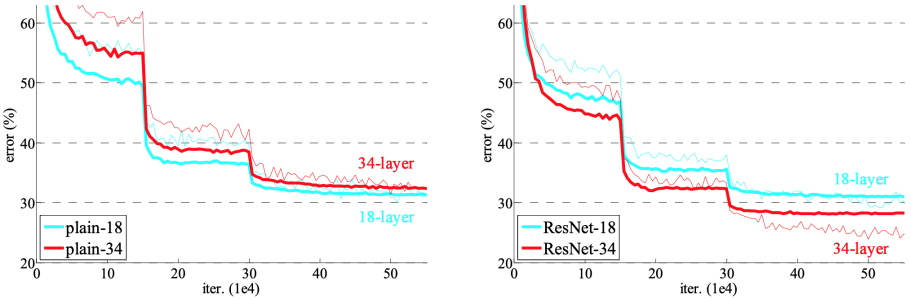


Figure 10.14: Validation error on ImageNet for networks of various depths, with and without residual connections.

Basic Residual Block A basic residual block consists of two (or three) convolutional layers with batch normalization and ReLU activations, wrapped in a residual connection. When dimensions change (e.g., due to stride), the shortcut may include a 1×1 convolution to match dimensions. Such blocks enable the training of networks exceeding 100 layers, which significantly improve accuracy on large-scale tasks.

method	top-1 err.	top-5 err.
VGG [40] (ILSVRC'14)	-	8.43†
GoogLeNet [43] (ILSVRC'14)	-	7.89
VGG [40] (v5)	24.4	7.1
PReLU-net [12]	21.59	5.71
BN-inception [16]	21.99	5.81
ResNet-34 B	21.84	5.71
ResNet-34 C	21.53	5.60
ResNet-50	20.74	5.25
ResNet-101	19.87	4.60
ResNet-152	**19.38**	**4.49**

Figure 10.15: Illustration of a basic residual block (left) and its impact on training deep networks.

Dense Connectivity An alternative to additive shortcuts is *dense connectivity* as introduced in DenseNet Huang et al. (2017). In a dense block, each layer receives as input the concatenation of all feature maps from previous layers:

$$\mathbf{x}_\ell = H_\ell\big([\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_{\ell-1}]\big),$$

where $H_\ell(\cdot)$ denotes the composite operation of batch normalization, ReLU, and convolution. Figure 10.16 depicts this pattern.

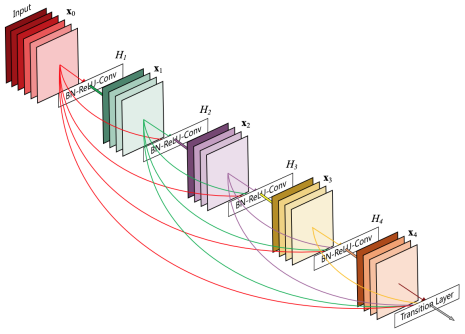


Figure 10.16: Dense block connectivity: each layer concatenates all preceding feature-maps.

DenseNets achieve competitive accuracy with fewer parameters and improve gradient propagation by providing direct paths to all preceding layers.

Residual and dense connectivity patterns have drastically improved the trainability of very deep networks. By facilitating gradient flow and learning only modifications around identity or concatenated features, these architectures have set new benchmarks across vision and other tasks.

10.3 Normalization Techniques

Poorly calibrated dynamic ranges of activations result in ineffective gradient propagation, commonly known as vanishing or exploding gradients. To calibrate these ranges, it is essential to first estimate the variability of the activations. This can be done by computing statistics either across a batch of data points or across features within a single data point which yields two well-known types of normalization techniques:

1. **Batch normalization** computes statistics across multiple examples (e.g., a mini-batch).

2. **Layer normalization** computes statistics across features within the same layer for a single example.

10.3.1 Batch Normalization

Deep networks exhibit strong dependencies between parameters in successive layers, making it difficult to choose a suitable learning rate. For instance, consider a simple

network with a single weight w_l at each layer l, so that its output for a given input x

$$y = \Big(\prod_{l=1}^{L} w_l\Big) x.$$

Under gradient descent with step size η, each weight update introduces terms up to order L, causing products of many gradient factors:

$$y^{\text{new}} = \Big(\prod_{l=1}^{L} (w_l - \eta\, \partial \ell / \partial w_l)\Big)\, x.$$

If individual gradients are smaller than 1, their product vanishes as L grows; if greater than 1, their product explodes. Batch normalization reduces such dependencies by fixing the distribution of activations at each layer.

Proposed by Ioffe and Szegedy (2015), batch normalization computes the mean and variance of each feature across a mini-batch of size $|I|$. For layer l and feature index i:

$$\mu_i^l = \frac{1}{|I|} \sum_{j \in I} z_i^l[j],$$
$$\sigma_i^l = \sqrt{\delta + \frac{1}{|I|} \sum_{j \in I} (z_i^l[j] - \mu_i^l)^2}, \quad \delta > 0,$$

where $z_i^l[j]$ is the pre-activation of feature i on example j. Normalization then yields:

$$\tilde{z}_i^l[j] = \frac{z_i^l[j] - \mu_i^l}{\sigma_i^l},$$

which has zero mean and unit variance across the batch. To retain representational flexibility, each feature is subsequently scaled and shifted:

$$\hat{z}_i^l[j] = \alpha_i^l\, \tilde{z}_i^l[j] + \beta_i^l,$$

where α_i^l, β_i^l are learned parameters.

These operations are differentiable, and the network learns both the original weights and the normalization parameters jointly.

10.3.2 Layer Normalization

Layer normalization applies the same principle at the level of individual examples. For a given example t and layer l, we now compute the statistics across its m^l features:

$$\mu^l[t] = \frac{1}{m^l} \sum_{i=1}^{m^l} z_i^l[t],$$

$$\sigma^l[t] = \sqrt{\delta + \frac{1}{m^l} \sum_{i=1}^{m^l} (z_i^l[t] - \mu^l[t])^2}.$$

Normalization and subsequent scaling/shifting follow as in batch normalization. Layer normalization does not depend on batch size and is thus well suited for recurrent or transformer architectures where batch statistics may be less stable.

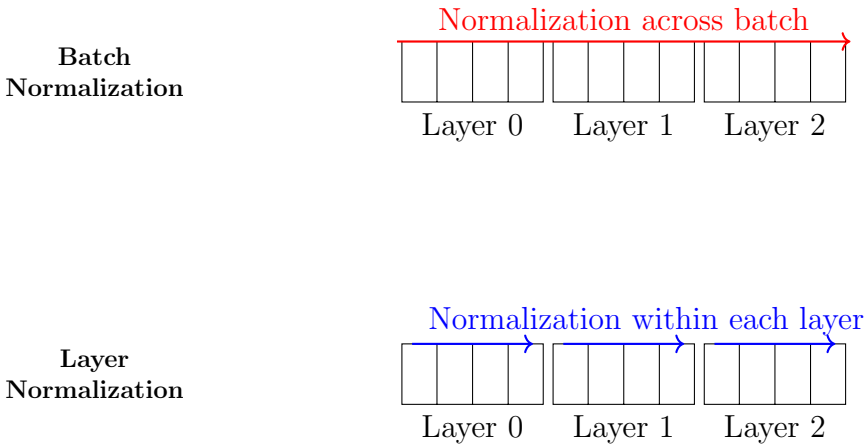


Figure 10.17: Comparison of batch normalization (top) and layer normalization (bottom).

Method	Advantages	Disadvantages
Batch Normalization	(+) Highly effective in convolutional networks (vision) and accelerates convergence.	(–) Performance depends on batch size; small batches yield noisy estimates.
Layer Normalization	(+) Independent of batch size; popular in recurrent and transformer models (NLP).	(–) Gains may be less pronounced in convolutional architectures.

Table 10.1: Comparison of Batch Normalization and Layer Normalization

Batch vs. Layer Normalization To conclude, normalization techniques are key components in modern deep learning, enabling stable and efficient training of very deep models. Understanding their formulation and trade-offs guides practitioners in selecting suitable methods for different architectures.

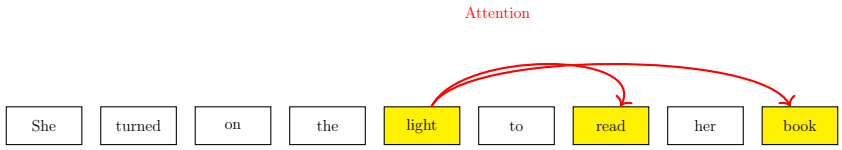
10.4 Transformers

Transformers are a type of deep learning architecture introduced in the paper "Attention is All You Need" by Vaswani et al. (2017). They have become the state-of-the-art approach in many natural language processing (NLP) tasks, such as machine translation and language understanding. They have also found applications in many other domains such as machine translation, image generation, speech recognition, and protein folding.

- Unlike traditional recurrent neural networks (RNNs) or convolutional neural networks (CNNs), Transformers rely solely on *self-attention mechanisms* to process input data.
- Self-attention allows Transformers to capture dependencies between different words in a sentence, making them more effective at modeling long-range dependencies compared to RNNs or CNNs.

10.4.1 Attention Mechanism

Unlike traditional RNNs or CNNs, Transformers rely on a self-attention mechanism to process input data. Self-attention allows the model to weigh the relevance of all other tokens when computing each token's new representation. Consider the following sentence:



One can clearly observe that some words in that sentence are more correlated than others, e.g. "read" and "book" commonly occur in the same sentence.

The input to a Transformer is a set of vectors $\{\mathbf{x}_n\}_{n=1}^N$ of dimension d, called *tokens*. A token might correspond to:

- A word in a sentence,
- A patch in an image,
- An amino acid in a protein.

The entry x_{ni} denotes the i-th feature (or embedding) of the n-th example; arranging these entries for all examples produces the input embedding matrix $\mathbf{X}$ shown in Figure 10.18.

Figure 10.18: Input Embedding matrix

At a high level, the self-attention mechanism allows each token to attend to (i.e., gather information from) all other tokens by computing relevance-based weights, thereby enabling the model to capture dependencies across the entire input. Let's try to formalize this in more detail next.

Formalization Mathematically, attention maps input tokens $\mathbf{x}_1, \ldots, \mathbf{x}_N$ to a new set of vectors $\mathbf{y}_1, \ldots, \mathbf{y}_N$ that aim to capture rich semantics. The attention mechanism will focus on the $\mathbf{x}_m$'s that are important in the sentence. Mathematically, the $\mathbf{y}$ vectors are defined as weighted combinations of the input vectors $\{\mathbf{x}\}_{i=1}^N$:

$$\mathbf{y}_n = \sum_{m=1}^{N} a_{nm}\, \mathbf{x}_m, \quad a_{nm} = \frac{\exp(\mathbf{x}_n^\top \mathbf{x}_m)}{\sum_{k=1}^{N} \exp(\mathbf{x}_n^\top \mathbf{x}_k)}, \tag{10.1}$$

where the coefficients a_{nm} are attention weights which are defined using a softmax function to transform the inner products between tokens and also to normalize the contribution of the attention weights.

Note that Eq. 10.1 can be rewritten in a matrix form as:

$$\mathbf{Y} = \text{Softmax}(\mathbf{X}\mathbf{X}^\top)\,\mathbf{X},$$

where Softmax normalizes each row.

Adjustable Parameters So far, we have presented an attention mechanism without any learnable parameters. One way to introduce such learnable parameters is to multiply the data matrix $\mathbf{X}$ by a learnable matrix $\mathbf{U} \in \mathbb{R}^{d\times d}$:

$$\mathbf{Y} = \text{Softmax}\big(\mathbf{X}\,\mathbf{U}\,\mathbf{U}^\top\,\mathbf{X}^\top\big)\mathbf{X}\mathbf{U}.$$

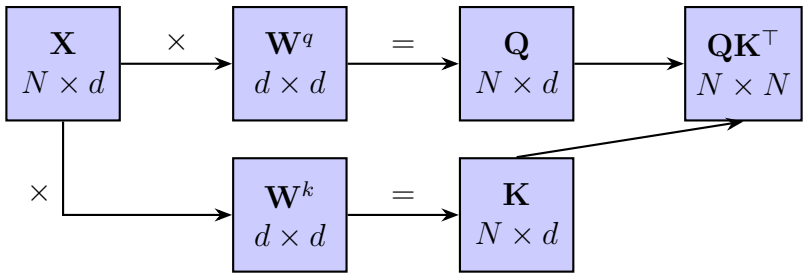


Figure 10.19: Computation graph for the computation of the matrix $\mathbf{QK}^\top$.

An even more flexible and widely used formulation employs separate *query*, *key*, and *value* projection matrices. Concretely, define

$$\mathbf{Q} = \mathbf{X}\,\mathbf{W}^q, \quad \mathbf{K} = \mathbf{X}\,\mathbf{W}^k, \quad \mathbf{V} = \mathbf{X}\,\mathbf{W}^v,$$

where $\mathbf{W}^q, \mathbf{W}^k \in \mathbb{R}^{d\times d_k}, \mathbf{W}^v \in \mathbb{R}^{d\times d_v}$ (for simplicity, we will assume $d_k = d_v = d$) are learnable weight matrices. This separate projection formulation enables the model to learn distinct representations for computing similarities (*queries* and *keys*) and for constructing the output (*values*), thereby enhancing expressiveness and allowing the network to capture richer dependencies among tokens.

The matrix $\mathbf{QK}^\top$ is first computed (as shown in Figure 10.19) before passing through the Softmax function.

The attention output then becomes:

$$\mathbf{Y} = \text{Softmax}\left(\frac{1}{\sqrt{d}}\mathbf{Q}\,\mathbf{K}^\top\right)\,\mathbf{V},$$

where the scaling factor $\frac{1}{\sqrt{d}}$ inside the Softmax is crucial for stabilizing the dot-product attention. Indeed, the softmax function becomes highly peaked when its inputs are large in magnitude. This leads to gradients that are very small (vanishing) during backpropagation, making training unstable or slow. Dividing by $\sqrt{d}$ ensures that the dot products $\mathbf{q}_i^\top \mathbf{k}_j$ have variance approximately 1, making the softmax operate in a reasonable numerical range even as d increases.

Remark 8 (Note on Nomenclature)**.** *The terminology "queries", "keys", and "values" originates in the information retrieval literature and denotes:*

- Queries: *vectors representing the information sought.*
- Keys: *vectors used to evaluate relevance against queries.*
- Values: *vectors containing the information retrieved based on key–query similarity.*

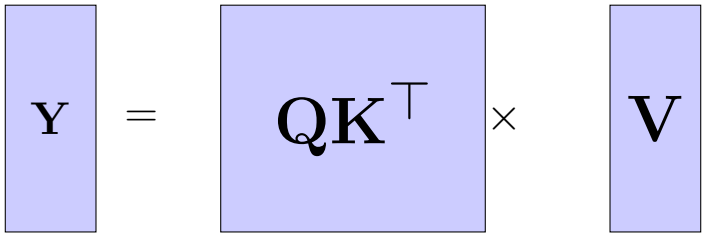
Python code The Python code to implement the attention mechanism is shown in Listing 10.2.

```
def scaled_dot_product_attention(Q, K, V):
    """
    Q, K, V: (batch_size, seq_len, embed_dim)
    Returns:
      output: (batch_size, seq_len, embed_dim)
      attn_weights: (batch_size, seq_len, seq_len)
    """
    d_k = Q.size(-1)  # embed_dim in single-head scenario

    scores = torch.matmul(Q, K.transpose(-2, -1))/(d_k**0.5)
    attn_weights = torch.softmax(scores, dim=-1)

    output = torch.matmul(attn_weights, V)

    return output, attn_weights

class SingleHeadSelfAttention(nn.Module):
    def __init__(self, embed_dim):
        super().__init__()
        self.embed_dim = embed_dim

        # Linear projections for Q, K, V (all output embed_dim)
        self.W_Q = nn.Linear(embed_dim, embed_dim)
        self.W_K = nn.Linear(embed_dim, embed_dim)
        self.W_V = nn.Linear(embed_dim, embed_dim)

        # Final linear projection
        self.W_O = nn.Linear(embed_dim, embed_dim)

    def forward(self, x):
        """
        x: (batch_size, seq_len, embed_dim)
        Returns:
          out: (batch_size, seq_len, embed_dim)
          attn_weights: (batch_size, seq_len, seq_len)
        """
        # Project inputs x
        Q = self.W_Q(x)
        K = self.W_K(x)
        V = self.W_V(x)

        # Scaled dot-product attention (single head)
        attn_output, attn_weights = scaled_dot_product_attention(Q,
   K, V)

        # Final projection
        out =  self.W_O(attn_output) # (batch_size, seq_len,
   embed_dim)
```

```
47
48          return out, attn_weights
```

Listing 10.2: Dot product attention

Attention vs. CNN Unlike fixed-weight multiplications in CNNs, activations in the attention mechanism interact multiplicatively via data-dependent attention coefficients.



10.4.2 Multi-Head Attention

A single *attention head* focuses on one pattern. In order to further increase the expressivity of the model, we can use multiple heads by concatenating them as follows:

$$\text{MultiHead}(\mathbf{x}) = \text{Concat}\big(\text{head}_1, \ldots, \text{head}_h\big)\, \mathbf{W}^O,$$

where $\mathbf{W}^O \in \mathbb{R}^{hd \times d_{\text{model}}}$ is another matrix of learnable parameters.

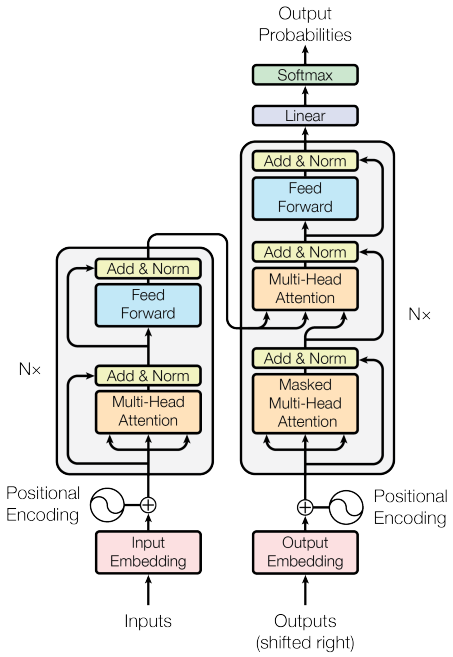
Multi-head attention allows Transformers to attend to information from different representation subspaces simultaneously, enabling the model to capture diverse patterns such as syntax and long-range dependencies. This parallel attention mechanism improves expressiveness, generalization, and training efficiency.

10.4.3 Transformer Model

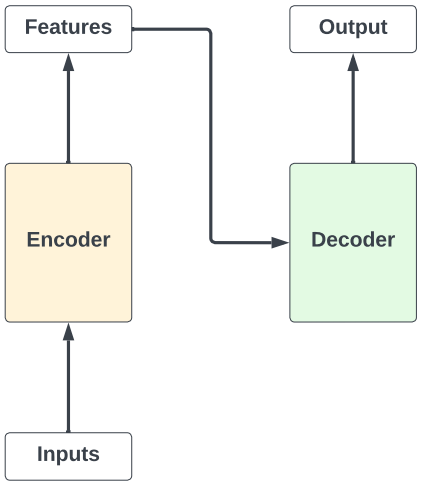
Building on the self-attention mechanism, the full Transformer architecture is realized by stacking multiple identical blocks. Figure 10.20 depicts one such block, which consists of two primary components:

- **Encoder:** Processes the input sequence (e.g., tokenized text) to produce contextualized feature representations.
- **Decoder:** Consumes the features and generates an output sequence (typically one token at a time).

Each encoder block applies multi-head self-attention followed by a position-wise feed-forward network, while each decoder block adds an additional cross-attention layer to incorporate encoder information.



(a) A single Transformer block as depicted by Vaswani et al. (2017).



(b) A simplified view of a single Transformer block with an encoder–decoder structure.

Figure 10.20: Overview of a single block of the Transformer architecture.

Block of an Encoder The encoder first uses multi-head self-attention with residual connections and layer normalization to integrate information across all tokens. It then applies a position-wise feed-forward network, again with residual connections and normalization, to transform each token's embedding into a contextually enriched representation.

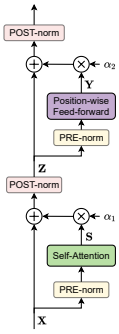


Figure 10.21: Block of an encoder

We next formalize the encoder computation. Given an input sequence $\mathbf{X} \in \mathbb{R}^{n \times d_v}$, with n tokens of dimension d_v, a (single-head) self-attention is defined as:

$$\mathbf{S}^\ell = \mathbf{A}^\ell \, \mathbf{X}^\ell \mathbf{W}^V, \quad \mathbf{A}^\ell = \mathrm{softmax}\Big(\tfrac{1}{\sqrt{d_k}} \mathbf{X}^\ell \mathbf{W}^Q (\mathbf{x}^\ell \mathbf{W}^K)^\top\Big).$$

The encoder block is then built as follows:

$$\begin{aligned} \mathbf{Z}^\ell &= \alpha_1 \, \mathbf{S}^\ell + \mathbf{x}^\ell, \\ \mathbf{Y}^\ell &= \sigma(\mathbf{Z}^\ell \mathbf{W}^{F_1}) \, \mathbf{W}^{F_2}, \\ \mathbf{X}^{\ell+1} &= \alpha_2 \, \mathbf{Y}^\ell + \mathbf{Z}^\ell, \end{aligned}$$

with residual weights α_1, α_2 and feed-

Block of a Decoder In a Transformer, the encoder processes the entire input sequence, using self-attention and feed-forward layers to produce contextualized representations for each input token. The decoder, in contrast, generates the output sequence autoregressively: it first uses masked self-attention to attend only to earlier output positions, then uses cross-attention to incorporate information from the encoder's representations, and finally applies a feed-forward layer—enabling it to produce each output token conditioned on both prior outputs and the encoded input.

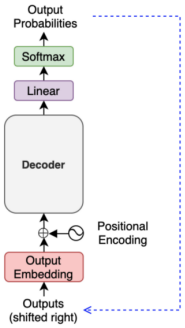


Figure 10.22: The decoder generates tokens sequentially. Source: `kikaben.com`.

Lising 10.3 shows a pytorch implementation of a simple decoder block.

```
class SingleTransformerBlock(nn.Module):
    def __init__(self, embed_dim, ff_dim):
        super().__init__()

        # Single-head self-attention
        self.self_attn = SingleHeadSelfAttention(embed_dim)

        # Feed-forward layers
        self.linear1 = nn.Linear(embed_dim, ff_dim)
        self.relu = nn.ReLU()
        self.linear2 = nn.Linear(ff_dim, embed_dim)

        # LayerNorms
        self.norm1 = nn.LayerNorm(embed_dim)
        self.norm2 = nn.LayerNorm(embed_dim)

    def forward(self, x):
        # --- Self-Attention ---
        attn_out, _ = self.self_attn(x)          # (batch_size, seq_len, embed_dim)
        x = self.norm1(x + attn_out)             # Residual connection

        # --- Feed-Forward ---
        ff_out = self.linear2(self.relu(self.linear1(x)))
        x = self.norm2(x + ff_out)               # Residual connection

```

```
26          return x
```

Listing 10.3: Example of a decoder block

Positional Encodings As shown in Figure 10.20a, positional encodings in Transformers are added to the input embeddings to provide information about the order of tokens in a sequence, since the Transformer architecture itself is permutation-invariant (i.e., it does not inherently process inputs in a sequential order like RNNs or CNNs).

There are two common types:

- **Sinusoidal positional encodings** (used in the original Transformer):

$$\mathrm{PE}_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d}}\right), \quad \mathrm{PE}_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d}}\right),$$

  where pos is the position and i indexes the embedding dimension. These allow the model to generalize to longer sequences.

- **Learned positional embeddings**, where each position has a trainable vector added to the token embedding.

In both cases, positional encodings are added to the input embeddings before being fed into the Transformer, allowing the model to distinguish between different token positions.

10.4.4 Sampling Strategies

Problem Setup A decoder Transformer outputs a probability distribution over the next token given the past tokens, i.e. $p(\mathbf{y}_n \mid \mathbf{y}_{1:n-1})$. In order to find the most probable sequence of length N, we would ideally maximize:

$$\prod_{n=1}^{N} p\big(\mathbf{y}_n \mid \mathbf{y}_{1:n-1}\big).$$

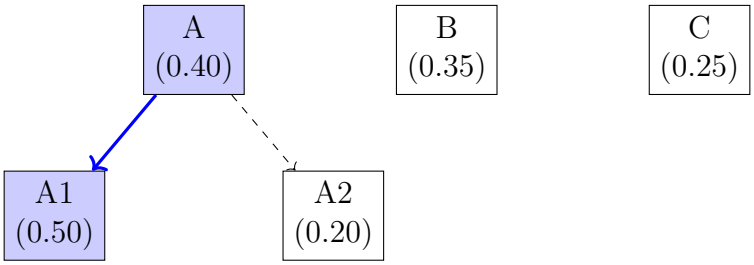
However, exhaustive search has a prohibitive time complexity of order $\mathcal{O}(K^N)$ for a vocabulary of size K. Instead, we will consider different types of search.

Greedy Search

Greedy search selects the highest-probability token at each step (deterministic), which is not guaranteed to be globally optimal but has a cheap cost equal to $\mathcal{O}(NK)$, since it evaluates K candidates at each of the N steps.

Greedy Search (GS) Example Given tokens A (0.4), B (0.35), C (0.25):

- Step 1: choose A.
- Step 2 after A: A1 (0.50), A2 (0.20) $\implies$ final sequence A, A1 (prob. 0.4×0.5=0.2).

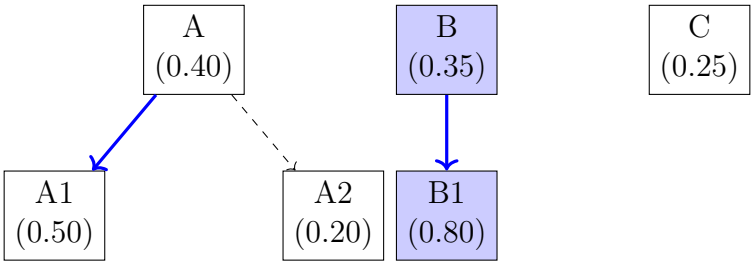


Beam Search

Beam Search (width B) maintains the top-B hypotheses at each step, expanding each with all next tokens and pruning to top-B, resulting in a total cost of $\mathcal{O}(NBK)$.

Beam Search Example ($B = 2$)

- Step 1: keep A (0.4) and B (0.35).
- Step 2: from A $\rightarrow$ A1 (0.5, seq 0.2), A2 (0.2); from B $\rightarrow$ B1 (0.8, seq 0.28).
- Final: choose B, B1.




```
def beam_search(initial_probs, expansions, beam_width, steps):
    """
    initial_probs: dict[token, float] of starting token
    probabilities
    expansions: dict[token, dict[next_token, float]] of conditional
    probabilities
    beam_width: int, how many hypotheses to keep at each step
    steps: int, total number of time steps (including the initial
    one)
    """
    # Initialize beam with the first-step tokens
```

```
    beam = [([token], prob) for token, prob in initial_probs.items()]
    # Keep only the top beam_width
    beam = sorted(beam, key=lambda x: x[1], reverse=True)[:beam_width]

    # Expand for the remaining steps
    for _ in range(1, steps):
        all_candidates = []
        for seq, seq_prob in beam:
            last = seq[-1]
            # For each possible next token from 'last'
            for next_token, next_prob in expansions.get(last, {}).items():
                new_seq = seq + [next_token]
                new_prob = seq_prob * next_prob
                all_candidates.append((new_seq, new_prob))
        # Select the top beam_width sequences
        beam = sorted(all_candidates, key=lambda x: x[1], reverse=True)[:beam_width]

    return beam

if __name__ == "__main__":
    # --- Example setup ---
    # Step 1 probabilities
    initial = {
        "A": 0.40,
        "B": 0.35,
        "C": 0.25
    }
    # Step 2 expansions
    expansions = {
        "A": {"A1": 0.50, "A2": 0.20},
        "B": {"B1": 0.80},
        # "C" has no expansions in this example
    }

    beam_width = 2
    total_steps = 2

    final_beam = beam_search(initial, expansions, beam_width, total_steps)

    print("Final beam (top 2 sequences):")
    for seq, prob in final_beam:
        print(f"  Sequence: {seq}, Probability: {prob:.2f}")

    # And the best sequence is:
    best_seq, best_prob = final_beam[0]
    print(f"\nBest sequence: {best_seq} with probability {best_prob
```

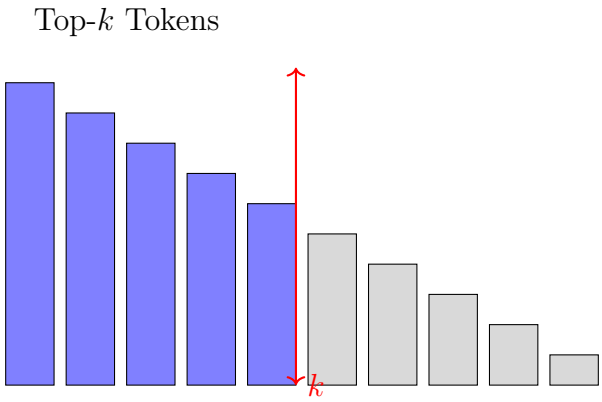
```
    :.2f}")
```

Listing 10.4: Beam Search

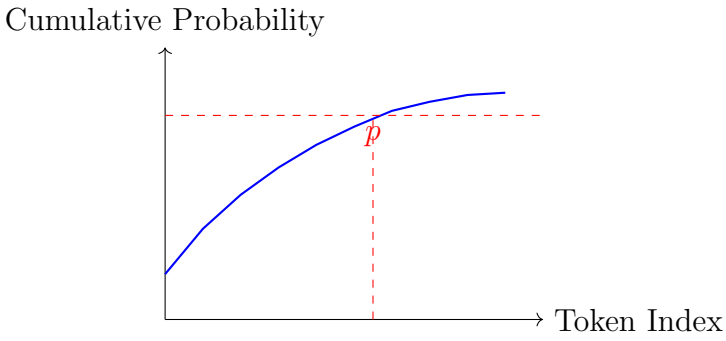
Stochastic Sampling

Stochastic sampling generates each token by sampling from the model's softmax distribution and comes into two flavor known as top-k sampling and top-p (Nucleus) sampling.

Top-k Sampling At each step, it keeps exactly the k highest-probability tokens while others are discarded.



Top-p Sampling Select the minimal set of tokens whose cumulative probability exceeds p.



Comparison

- Top-k sampling retains exactly the k most probable tokens at each decoding step, discarding the rest.

- Top-p sampling, in contrast, selects the smallest set of tokens whose cumulative probability exceeds a threshold p, allowing for a more flexible and context-sensitive choice.

10.4.5 Applications

Natural Language Processing

Tokenization Tokenization is the process of converting raw text into a sequence of discrete units (called *tokens*) that can be processed by a language model. These tokens may correspond to characters, subwords, or full words, depending on the tokenizer used. Tokenization reduces the vocabulary size while preserving enough granularity for the model to understand and generate language effectively. Subword-based tokenization strikes a balance between character-level flexibility and word-level efficiency.

The steps of Tokenization are:

1. **Preprocessing:** The raw input string is first normalized, which may include lowercasing, stripping whitespace, or Unicode normalization.
2. **Token Splitting:** The normalized string is split into tokens using a specific tokenization strategy. For example:
   - **Byte-Pair Encoding (BPE):** Frequently used in GPT-style models.
   - **WordPiece:** Used in BERT.
   - **Unigram Language Model:** Used in SentencePiece.
3. **Mapping to IDs:** Each token is mapped to an integer ID from a vocabulary $\mathcal{V}$. The final output is a sequence:

$$\text{Input text} \quad \rightarrow \quad [t_1, t_2, \ldots, t_n] \quad \rightarrow \quad [i_1, i_2, \ldots, i_n]$$

   where t_j is a token and $i_j = \text{ID}(t_j)$.

As an example, Listing 10.5 shows how to tokenize a sentence using the GPT2 tokenizer.

```
import torch
import matplotlib.pyplot as plt
import numpy as np
from transformers import GPT2Tokenizer, GPT2Model

input_text = "Like a fish out of water"

def show_token_embedding_norms_and_vectors_from_text(text):
    # Load GPT-2 tokenizer and model
```

```
    tokenizer = GPT2Tokenizer.from_pretrained("gpt2")
    model = GPT2Model.from_pretrained("gpt2")

    # Tokenize the input text
    inputs = tokenizer(text, return_tensors="pt", add_special_tokens=False)
    input_ids = inputs["input_ids"][0]

    # Get input embeddings for the tokens
    with torch.no_grad():
        embeddings = model.wte(input_ids)  # shape: (T, d)
        norms = torch.norm(embeddings, dim=1).tolist()
        embeddings_list = embeddings.tolist()

    # Display result
    print(f"Input text: {repr(text)}\n")
    print(f"{'Token':<20} {'Token ID':<10} {'L2 Norm':<10} {'Embedding Vector'}")
    print("-" * 90)
    for token, token_id, norm, vector in zip(tokenizer.convert_ids_to_tokens(input_ids), input_ids, norms, embeddings_list):
        vector_str = "[" + ", ".join(f"{v:.3f}" for v in vector[:8]) + ("..." if len(vector) > 8 else "") + "]"
        print(f"{token:<20} {token_id:<10} {norm:<10.4f} {vector_str}")

show_token_embedding_norms_and_vectors_from_text(input_text)
```

Listing 10.5: Tokenization example

The output of Listing 10.5 is shown in Table 10.2.

Token	Token ID	L2 Norm	Embedding Vector
Like	7594	3.2995	[−0.166, −0.080, 0.086, −0.043, −0.061, −0.129, −0.217, −0.110,
a	257	2.5727	[−0.051, 0.006, 0.047, −0.059, −0.063, −0.030, −0.272, 0.015, ...]
fish	5916	3.2591	[0.027, 0.100, 0.066, 0.051, −0.156, −0.080, −0.364, −0.158, ...]
out	503	2.7112	[−0.048, −0.044, 0.034, −0.141, 0.091, −0.064, −0.282, −0.014, ...]
of	286	2.5936	[−0.057, 0.018, 0.033, 0.041, 0.012, −0.040, −0.253, 0.002, ...]
water	1660	3.0460	[0.065, 0.020, 0.023, 0.141, −0.052, −0.019, −0.331, −0.144, ...]

Table 10.2: Token Embeddings: Norms and Partial Vectors

Masking Masking is a crucial mechanism in large language models (LLMs) such as Transformers, particularly during training for autoregressive language modeling. It ensures that, when predicting the next token, the model only attends to the previous tokens and not future ones.

Given a sequence of input embeddings $\mathbf{X} \in \mathbb{R}^{n \times d}$, the attention scores are computed as:

$$\mathbf{A} = \frac{1}{\sqrt{d_k}} \mathbf{Q}\mathbf{K}^\top, \quad \text{where} \quad \mathbf{Q} = \mathbf{X}\mathbf{W}^Q,\ \mathbf{K} = \mathbf{X}\mathbf{W}^K$$

To prevent a token at position i from attending to tokens at positions $j > i$, a causal (or autoregressive) mask $\mathbf{M} \in \mathbb{R}^{n \times n}$ is applied, where:

$$\mathbf{M}_{ij} = \begin{cases} 0, & \text{if } j \leq i \\ -\infty, & \text{if } j > i \end{cases}$$

The masked attention scores become:

$$\tilde{\mathbf{A}} = \mathbf{A} + \mathbf{M}$$

Then the attention weights are computed with softmax:

$$\mathbf{W} = \text{softmax}(\tilde{\mathbf{A}})$$

This masking ensures that token i cannot access future tokens during self-attention, preserving the autoregressive property required for next-token prediction.

An illustration of the masking process is shown in Figure 10.23.

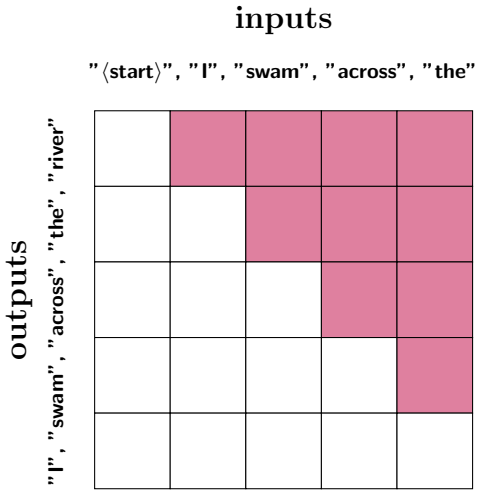


Figure 10.23: Illustration of the mask matrix used in masked self-attention, where attention weights for the purple entries are set to zero. Note the addition of a start token at the beginning of the input to ensure a consistent alignment between input and output tokens.

Vision Transformers

A Vision Transformer (ViT) is a neural network architecture that applies the transformer model, originally developed for natural language processing, to image data. Instead of processing sequences of words, a ViT divides an image into fixed-size patches, embeds them into vectors, and treats them as a sequence of tokens, just like words in a sentence. This allows it to leverage the transformer's self-attention mechanism to model global relationships across the entire image.

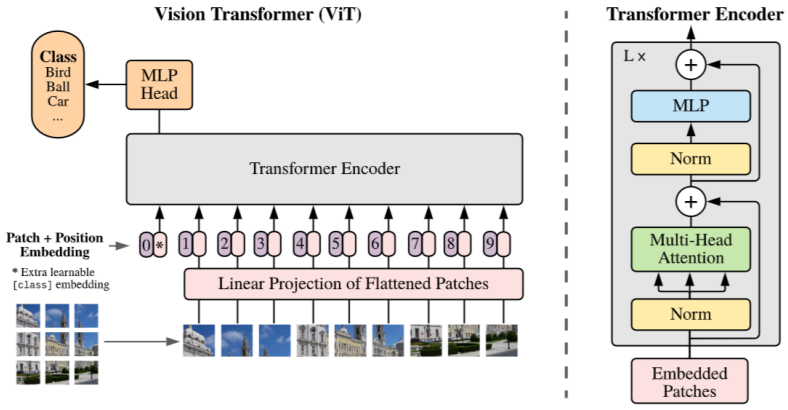


Figure 10.24: Illustration of a VIT Dosovitskiy et al. (2020).

10.4.6 Shortcomings of Transformer Architectures

For reasoning tasks (logical, mathematical, commonsense, causal), Transformers still seem to struggle with:

- Multi-step reasoning
- Symbolic/math reasoning
- Lack of explicit reasoning modules
- Commonsense/world-knowledge integration
- Logical consistency
- Causal/counterfactual inference

10.5 Exercise: Neural Network Architectures

Problem 1 (Convolutions as Dense Operations):
The goal of this exercise is to better understand the term "parameter-sharing" in the context of convolutional neural networks.

a) Consider the 1-d convolution of an input $\mathbf{x} \in \mathbb{R}^D$ with a kernel $\mathbf{h} \in \mathbb{R}^K$:

$$(\mathbf{x} * \mathbf{h})_k = \sum_{i=1}^{K} x_{K+k-i} h_i \quad \text{for } k \in \{1, \dots, D-K+1\}$$

For fixed $\mathbf{h}$ we can thus view $\mathbf{x} * \mathbf{h}$ as a 1-d convolutional layer in a neural network

$$f_{\mathbf{h}} : \mathbb{R}^D \to \mathbb{R}^{D-K+1},\ \mathbf{x} \mapsto \mathbf{x} * \mathbf{h}$$

Rewrite this operation as a dense layer, namely find a matrix $\mathbf{W_h} \in \mathbb{R}^{(D-K+1)\times D}$ such that you can write

$$f(\mathbf{x}) = \mathbf{W_h x}\ \forall \mathbf{x}$$

Hint: Use Toeplitz matrix.

b) Compare the layer $f_{\mathbf{h}} : \mathbb{R}^D \to \mathbb{R}^{D-K+1},\ \mathbf{x} \mapsto \mathbf{W_h x}$ with a general dense layer $f : \mathbb{R}^D \to \mathbb{R}^{D-K+1},\ \mathbf{x} \mapsto \mathbf{W x}$
for some matrix $\mathbf{W} \in \mathbb{R}^{(D-K+1)\times D}$.

What are the number of free parameters in f and in $f_{\mathbf{h}}$? Why does $f_{\mathbf{h}}$ have less free parameters? Think about "parameter-sharing".

Problem 2 (Batch Normalization):

Assume we have a fully connected neural network f with input $\mathbf{x} \in \mathbb{R}^d$ described by the recursive system

- $\mathbf{h}^{(0)}(\mathbf{x}) = \mathbf{x} \in \mathbb{R}^d$
- $\mathbf{h}^{(l+1)}(\mathbf{x}) = \mathbf{W}^{(l+1)}\tilde{\mathbf{h}}^{(l)}(\mathbf{x}) \in \mathbb{R}^{d_{l+1}}$ where $\mathbf{W}^{(l+1)} \in \mathbb{R}^{d_{l+1} \times d_l}$
- $\tilde{\mathbf{h}}^{(l+1)}(\mathbf{x}) = \phi\left(\mathbf{h}^{(l+1)}(\mathbf{x})\right) \in \mathbb{R}^{d_{l+1}}$ where $\phi : \mathbb{R} \to \mathbb{R}$ is a non-linearity applied component-wise
- $f(\mathbf{x}) = \left(\mathbf{W}^{(L+1)}\right)^T \tilde{\mathbf{h}}^{(L)}(\mathbf{x}) \in \mathbb{R}$ where $\mathbf{W}^{(L)} \in \mathbb{R}^{d_L}$

We can describe the batch norm operation at layer $l+1$ as follows. Fix a dataset $\{\mathbf{x}_1, \ldots, \mathbf{x}_n\} \subset \mathbb{R}^d$ and take a mini batch $\mathcal{B} = \{\mathbf{x}_1, \ldots, \mathbf{x}_B\}$ which we assume for simplicity is just the first B examples.
During the forward-pass for one step of SGD we calculate the pre-activations $\mathbf{h}^{(l+1)}(\mathbf{x}_1), \ldots, \mathbf{h}^{(l+1)}(\mathbf{x}_B)$. Given those values, we calculate mean and variance statistics as

- $\boldsymbol{\mu}^{(l+1)}(\mathcal{B}) = \frac{1}{B}\sum_{i=1}^{B} \mathbf{h}^{(l+1)}(\mathbf{x}_i) \in \mathbb{R}^{d_{l+1}}$
- $\boldsymbol{\sigma}^{(l+1)}(\mathcal{B}) \in \mathbb{R}^{d_{l+1}}$ such that $\sigma_j^{(l+1)} = \sqrt{\frac{1}{B-1}\sum_{i=1}^{B}\left(h_j^{(l+1)}(\mathbf{x}_i) - \mu_j^{(l+1)}\right)^2}$

Instead of propagating $h^{(l+1)}(\mathbf{x}_i)$ through the non-linearity ϕ for $i = 1, \ldots, B$, we pass

$$\mathbf{z}_j^{(l+1)}(\mathbf{x}) = \gamma_j^{(l+1)} \frac{h_j^{(l+1)}(\mathbf{x}) - \mu_j^{(l+1)}}{\sigma_j^{(l)}} + \beta_j^{(l+1)} \quad \text{for } j = 1, \ldots, d_{l+1}$$

where $\boldsymbol{\gamma}^{(l+1)}, \boldsymbol{\theta}^{(l+1)} \in \mathbb{R}^{d_{l+1}}$ are learnable parameters. The equations for a layer thus become:

- $\mathbf{h}^{(l+1)}(\mathbf{x}) = \mathbf{W}^{(l+1)}\tilde{\mathbf{h}}^{(l)}(\mathbf{x}) \in \mathbb{R}^{d_{l+1}}$
- $\mathbf{z}^{(l+1)}(\mathbf{x}) = \boldsymbol{\gamma}^{(l+1)} \odot \frac{\mathbf{h}^{(l+1)}(\mathbf{x}) - \boldsymbol{\mu}^{(l+1)}(\mathcal{B})}{\boldsymbol{\sigma}^{(l+1)}(\mathcal{B})} + \boldsymbol{\theta}^{(l+1)} \in \mathbb{R}^{d_{l+1}}$
- $\tilde{\mathbf{h}}^{(l+1)}(\mathbf{x}) = \phi\left(\mathbf{z}^{(l+1)}(\mathbf{x})\right) \in \mathbb{R}^{d_{l+1}}$

It is very important to realize that $\boldsymbol{\mu}^{(l)}(\mathcal{B})$ and $\boldsymbol{\sigma}^{(l)}(\mathcal{B})$ are a function of the batch $\mathcal{B}$ and thus vary in different forward passes, while $\boldsymbol{\gamma}^{(l)}$ and $\boldsymbol{\theta}^{(l)}$ do not.

In this exercise we want to understand the role of the rescaling operation through $\boldsymbol{\gamma}^{(l)}$ better.

a) Where lies the difference between

- Standardizing the layer and rescaling it with learnable parameters (batch normalization).
- Not modifying the layer at all.

You can argue very informally.

b) Why do we even rescale again with $\boldsymbol{\gamma}^{(l)}$? To that end, consider a very simple neural network

$$f(x) = w^{(2)}\phi(w^{(1)}x)$$

where $x \in \mathbb{R}$, $w^{(1)} \in \mathbb{R}$ and $w^{(2)} \in \mathbb{R}$. What happens to the representation power of f if we apply batch norm on the first layer without applying the rescale operation?

Chapter 11

Regularization

In machine learning, over-parametrized models are thought to overfit to the training set, i.e. their performance on the test set can be poor despite achieving a low training error [1]. One technique that is commonly used to avoid/control over-fitting is that of regularization. A classical regularization approach in machine learning involves adding a penalty term to the loss function in order to discourage the norm of the model parameters from reaching large values (the ℓ_1 and ℓ_2 norm are for instance among the most popular choices). Students with a background in machine learning should already be familiar with such concepts (which will be reviewed in the exercise sheet) but might be surprised to learn that deep learning models trained with (stochastic) gradient descent exhibit a very different behavior in practice. This is illustrated in Figure 11.1 where we see that while the training error reaches zero, the test error keeps decreasing, without ever observing any sign of overfitting. It seems that some sort of regularization is at play, if so, where does it come from?

In these notes, we will take a close look at this phenomenon. Specifically, we will study two different ways to enforce regularization when training deep learning models. The first approach is due to the use of gradient descent methods that exhibit an implicit bias, while the second approach, known as Dropout, consists of randomly dropping neurons by sampling them in an i.i.d. fashion from a Bernoulli distribution.

11.1 Implicit Bias of Gradient Descent

The parameters of deep neural networks are almost exclusively trained using some sort of (stochastic) gradient descent procedure. In these notes, we will study the properties of the solution that is found by such procedures. One question of special interest is whether the solution has desirable properties in terms of generalization. We start with the simple linear regression case and then discuss logistic regression.

[1] As a side note, we will see later in this lecture that this picture is somewhat more complicated (see Double Descent phenomenon)

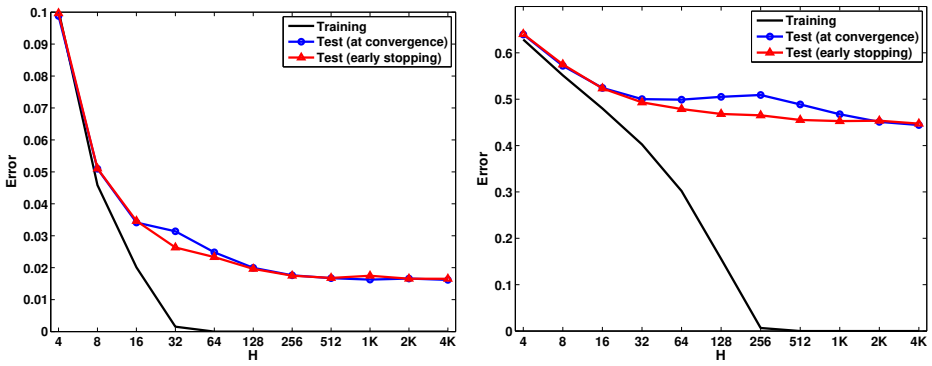


Figure 11.1: The training error and the test error based on different stopping criteria when 2-layer NNs with different numbers of hidden units are trained on MNIST and CIFAR-10. See details in Neyshabur et al. (2014).

11.1.1 Linear Regression

Consider a training dataset $D = (\mathbf{x}_i, y_i)_{i=1}^n$ where $\mathbf{x}_i \in \mathbb{R}^d$ are some given features and $y_i \in \{-1, +1\}$ are the corresponding labels. The least-squares objective is then defined as:

$$\mathcal{L}(\mathbf{w}) = \frac{1}{n} \sum_{i=1}^{n} (y_i - \mathbf{w}^\top \mathbf{x}_i)^2, \tag{11.1}$$

where $\mathbf{w} \in \mathbb{R}^d$ are the parameters to learn. We can also rewrite the equation above in a vector form as

$$\mathcal{L}(\mathbf{w}) = \frac{1}{n} \left\| \mathbf{y} - \mathbf{X}\mathbf{w} \right\|^2, \tag{11.2}$$

where $\mathbf{y} = [y_1 \dots y_n] \in \mathbb{R}^n$ and $\mathbf{X}$ is matrix whose rows are the $\mathbf{x}_i's$, i.e. $\mathbf{X} = [\mathbf{x}_1 \dots \mathbf{x}_n]^\top \in \mathbb{R}^{n \times d}$.

Minimum-norm solution found by gradient descent Assume that $d \gg n$ (over-parametrized setting), then many $\mathbf{w}$'s satisfy $\mathbf{X}\mathbf{w} = \mathbf{y}$ and all have $L(\mathbf{w}) = 0$ and are global minima of the problem, see illustration in Figure 11.2.

Which solution do we converge to? The following theorem shows that gradient flow converges to the global minimum which is the closest to the initialization in terms of ℓ_2 distance.

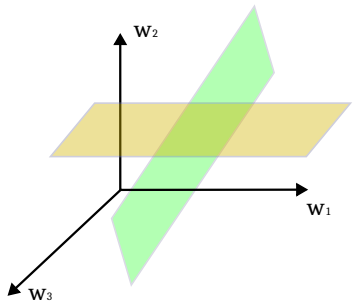


Figure 11.2: Over-parametrized model with $n = 2, d = 3$ (this is an undetermined system with more parameters than datapoints). There is a manifold of solutions corresponding to the intersection of the planes shown above.

Theorem 71 (Gunasekar et al. (2018)). *Consider the iterate of gradient descent on the least-square regression loss. Then, if $d > n$,*

$$\mathbf{w}_t \to \underset{\mathbf{w}|\mathbf{X}\mathbf{w}=\mathbf{y}}{\operatorname{argmin}} \|\mathbf{w} - \mathbf{w}_0\|_2. \tag{11.3}$$

Therefore, if $\mathbf{w}_0 = 0$, $\mathbf{w}$ converges to the minimum norm solution.

Proof. First, we note that

$$\nabla_{\mathbf{w}}\mathcal{L}(\mathbf{w}_t) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \mathbf{w}_t^\top \mathbf{x}_i)\mathbf{x}_i, \tag{11.4}$$

i.e. the updates lie on a low dimensional affine manifold $\Delta\mathbf{w}_t \in \operatorname{span}(\{\mathbf{x}_i\}_{i=1}^n)$. Within this low dimensional manifold, there is a unique global minimizer that satisfies the linear constraints in $G = \{\mathbf{w} \mid \mathbf{w}^\top \mathbf{x}_i = y_i, \forall i = [n]\}$.

In order to complete the proof, we need to show that there is only one solution in this subspace.

Let's first prove uniqueness. Assume that there are two solutions $\mathbf{w}_1^*$ and $\mathbf{w}_2^*$ both in the span of $\{\mathbf{x}_i\}_{i=1}^n$. We have $\mathbf{x}_i^\top \mathbf{w}_1^* = \mathbf{x}_i^\top \mathbf{w}_2^* = y_i \ \forall i = 1, \dots n$. Therefore

$$(\mathbf{w}_1^* - \mathbf{w}_2^*)^\top \mathbf{x}_i = 0 \quad \forall i, \tag{11.5}$$

which in matrix notation can be written as

$$\mathbf{X}(\mathbf{w}_1^* - \mathbf{w}_2^*) = 0. \tag{11.6}$$

Since we assumed that $d > n$, $\mathbf{X}$ is full rank, and therefore $\mathbf{w}_1^* = \mathbf{w}_2^*$.

We next turn to prove the existence of a solution in the span of $\{\mathbf{x}_i\}_{i=1}^n$. We write the minimum norm solution as $\mathbf{w}^*$ and split it into two components $\mathbf{w}^* = \mathbf{w}_\mathbf{x}^* + \mathbf{w}_c^*$, where $\mathbf{w}_\mathbf{x}^*$ is the component into the span of $\{\mathbf{x}_i\}_{i=1}^n$, and $\mathbf{w}_c^*$ is its complement. Then, we have

$$\begin{aligned}
\|\mathbf{w}^*\|_2^2 &= \|\mathbf{w}_\mathbf{x}^* + \mathbf{w}_c^*\|_2^2 && (11.7)\\
&= \|\mathbf{w}_\mathbf{x}^*\|_2^2 + 2\langle \mathbf{w}_\mathbf{x}^*, \mathbf{w}_c^*\rangle + \|\mathbf{w}_c^*\|_2^2 && (11.8)\\
&= \|\mathbf{w}_\mathbf{x}^*\|_2^2 + \|\mathbf{w}_c^*\|_2^2 && (11.9)\\
&\geq \|\mathbf{w}_\mathbf{x}^*\|_2^2, && (11.10)
\end{aligned}$$

where we used $\langle \mathbf{w}_\mathbf{x}^*, \mathbf{w}_c^*\rangle = 0$ since the two spaces are the complement of each other.

However, since $\mathbf{w}^*$ is the minimum norm solution in G, we have $\|\mathbf{w}_c^*\|_2^2 = 0$ and therefore $\|\mathbf{w}^*\|_2^2 = \|\mathbf{w}_\mathbf{x}^*\|_2^2$. We also have the additional constraint that $\mathbf{x}_i^\top \mathbf{w}_c^* = 0$ for all $\mathbf{x}_i$ and we therefore conclude that $\mathbf{w}_c^* = 0$. This implies that the solution $\mathbf{w}^*$ lies in the span of the data.

□

11.1.2 Logistic regression

We have seen that in a regression setting, gradient descent converges to the minimum-norm solution. Next, we will see that in a classification setting (logistic regression), gradient descent with arbitrary initialization converges to the maximum margin classifier on separable data. Notably, this convergence happens without any sort of explicit regularization.

Assumption 2. *The dataset is linearly separable:* $\exists\ \mathbf{w}_*$ *such that* $\mathbf{w}_*^\top \mathbf{x}_i > 0\ \forall i \in [n]$.

Assumption 3. $\ell(u)$ *is a positive, differentiable, monotonically decreasing to zero*[a]*, (so* $\forall u : \ell(u) > 0, \ell'(u) < 0$, $\lim_{u\to\infty} \ell(u) = \lim_{u\to\infty} \ell'(u) = 0$*), a* β*-smooth function,* i.e. *its derivative is* β*-Lipshitz and* $\lim_{u\to-\infty} \ell'(u) \neq 0$.

[a] The requirement of non-negativity and that the loss asymptotes to zero is purely for convenience. It is enough to require the loss is monotone decreasing and bounded from below. Any such loss asymptotes to some constant, and is thus equivalent to one that satisfies this assumption, up to a shift by that constant.

Assumption 3 includes many common loss functions, including the logistic and exponential loss.

The main question we ask is: can we characterize the direction of $\mathbf{w}(t)$ as $t \to \infty$? That is, does the limit $\lim_{t\to\infty} \mathbf{w}(t) / \|\mathbf{w}(t)\|$ always exist, and if so, what is it?

In order to analyze this limit, we will need to make a further assumption on the tail of the loss function:

Definition 30. *A function $f(u)$ has a "tight exponential tail", if there exist positive constants c, a, μ_+, μ_-, u_+ and u_- such that*

$$\forall u > u_+ : f(u) \leq c\left(1 + \exp\left(-\mu_+ u\right)\right) e^{-au}$$
$$\forall u > u_- : f(u) \geq c\left(1 - \exp\left(-\mu_- u\right)\right) e^{-au}.$$

Assumption 4. *The negative loss derivative $-\ell'(u)$ has a tight exponential tail (Definition 30).*

Examples For example, the exponential loss $\ell(u) = e^{-u}$ and the commonly used logistic loss $\ell(u) = \log(1 + e^{-u})$ both follow this assumption with $a = c = 1$.

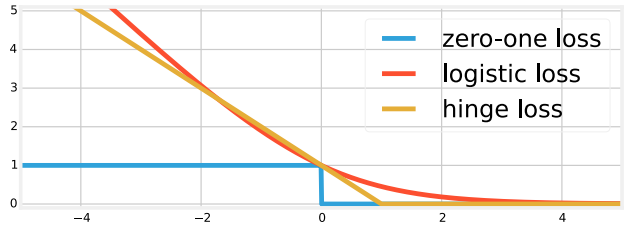


Figure 11.3: Logistic loss $\ell(u) = \log(1 + e^{-u})$ where $u = \mathbf{w}^\top \mathbf{x}_i$. Note that we observe lower values of the loss for larger values of $u = \mathbf{w}^\top \mathbf{x}_i$. This intuitively explains why gradient descent enforces $\mathbf{w}(t) \to \infty$ in order to minimize the loss. Source: `https://fa.bianp.net`.

Karush-Kuhn-Tucker (KKT) conditions We briefly introduce the concept of KKT conditions which we will be using in our forthcoming main result. Consider the following constrained optimization problem:

$$\begin{aligned} \text{Minimize} \quad & f(\mathbf{x}) \\ \text{subject to} \quad & g_i(\mathbf{x}) \leq 0, \quad i = 1, 2, \ldots, m \\ & h_j(\mathbf{x}) = 0, \quad j = 1, 2, \ldots, p \end{aligned}$$

where $\mathbf{x}$ is the vector of decision variables, $f(\mathbf{x})$ is the objective function, $g_i(\mathbf{x})$ are inequality constraints, and $h_j(x)$ are equality constraints.

The KKT conditions for a solution $\mathbf{x}^*$ to be optimal are given by:

1. $\nabla f(\mathbf{x}^*) + \sum_{i=1}^{m} \lambda_i \nabla g_i(\mathbf{x}^*) + \sum_{j=1}^{p} \mu_j \nabla h_j(\mathbf{x}^*) = 0$ (Stationarity)
2. $g_i(\mathbf{x}^*) \leq 0, \quad i = 1, 2, \ldots, m$ (Primal Feasibility)
3. $\lambda_i \geq 0, \quad i = 1, 2, \ldots, m$ (Dual Feasibility)
4. $\lambda_i g_i(\mathbf{x}^*) = 0, \quad i = 1, 2, \ldots, m$ (Complementary Slackness)
5. $h_j(\mathbf{x}^*) = 0, \quad j = 1, 2, \ldots, p$ (Equality Constraints)

where λ_i and μ_j are the Lagrange multipliers associated with the inequality and equality constraints, respectively.

Theorem 72 (Soudry et al. (2018)). *For any dataset which is linearly separable (Assumption 2), any β-smooth decreasing loss function (Assumption 3) with an exponential tail (Assumption 4), any stepsize $\eta < 2\beta^{-1}\sigma_{\max}^{-2}(\mathbf{X})$ and any starting point $\mathbf{w}(0)$, the gradient descent iterates will behave as:*

$$\mathbf{w}(t) = \hat{\mathbf{w}} \log t + \boldsymbol{\rho}(t) , \tag{11.11}$$

where $\hat{\mathbf{w}}$ is the L_2 max-margin vector (the solution to the hard margin SVM):

$$\hat{\mathbf{w}} = \underset{\mathbf{w} \in \mathbb{R}^d}{\operatorname{argmin}} \|\mathbf{w}\|^2 \text{ s.t. } \mathbf{w}^\top \mathbf{x}_i \geq 1, \tag{11.12}$$

and the residual grows at most as $\|\boldsymbol{\rho}(t)\| = \mathcal{O}(\log\log(t))$, and so

$$\lim_{t \to \infty} \frac{\mathbf{w}(t)}{\|\mathbf{w}(t)\|} = \frac{\hat{\mathbf{w}}}{\|\hat{\mathbf{w}}\|}.$$

Furthermore, for almost all data sets (all except measure zero), the residual $\rho(t)$ is bounded.

Proof sketch. 1) If $\mathbf{w}(t) / \|\mathbf{w}(t)\|$ converges to some limit $\mathbf{w}_\infty$, then we can write $\mathbf{w}(t) = g(t)\mathbf{w}_\infty + \boldsymbol{\rho}(t)$ where $g : \mathbb{R} \to \mathbb{R}$ is such that $g(t) \to \infty$, $\forall n : \mathbf{x}_i^\top \mathbf{w}_\infty > 0$, and $\lim_{t\to\infty} \boldsymbol{\rho}(t)/g(t) = 0$.

2) The gradient of the loss can then be written as:

$$-\nabla \mathcal{L}(\mathbf{w}) = \sum_{i=1}^{n} \exp\left(-\mathbf{w}(t)^\top \mathbf{x}_i\right) \mathbf{x}_i = \sum_{i=1}^{n} \exp\left(-g(t)\mathbf{w}_\infty^\top \mathbf{x}_i\right) \exp\left(-\boldsymbol{\rho}(t)^\top \mathbf{x}_i\right) \mathbf{x}_i . \tag{11.13}$$

As $g(t) \to \infty$ and the exponents become more negative, only those samples with the largest (*i.e.*, least negative) exponents will contribute to the gradient. These are precisely the samples with the smallest margin $\operatorname{argmin}_i \mathbf{w}_\infty^\top \mathbf{x}_i$, aka the "support vectors".

The negative gradient (Eq. 11.13) would then *asymptotically become a non-negative linear combination of support vectors.*

3) The limit $\mathbf{w}_\infty$ will then be dominated by these gradients, since any initial conditions become negligible as $\|\mathbf{w}(t)\| \to \infty$ (proof omitted). Therefore, $\mathbf{w}_\infty$ will also be a non-negative linear combination of support vectors, and so will its scaling $\hat{\mathbf{w}} = \frac{\mathbf{w}_\infty}{\left(\min_i \mathbf{w}_\infty^\top \mathbf{x}_i\right)}$. We conclude that:

$$\begin{cases} \hat{\mathbf{w}} = \sum_{i=1}^n \alpha_i \mathbf{x}_i \\ \alpha_i \geq 0 \text{ if } \mathbf{x}_i \in \mathcal{S} \\ \alpha_i = 0 \text{ if } \mathbf{x}_i \notin \mathcal{S} \end{cases}$$

These are precisely the KKT conditions for the SVM problem (Eq. (11.12)) and we can conclude that $\hat{\mathbf{w}}$ is indeed its solution and $\mathbf{w}_\infty$ is thus proportional to it.

□

Detailed proof for a special case (optional reading). For simplicity, we focus on the special case where $\ell(u) = e^{-u}$ (exponential loss) and analyze gradient flow:

$$\dot{\mathbf{w}}(t) = -\nabla \mathcal{L}(\mathbf{w}(t)) .$$

Recall that if $\mathbf{w}(t) / \|\mathbf{w}(t)\|$ converges to some limit $\mathbf{w}_\infty$, then we can write $\mathbf{w}(t) = g(t)\mathbf{w}_\infty + \boldsymbol{\rho}(t)$ such that $g(t) \to \infty$, $\forall n : \mathbf{x}_i^\top \mathbf{w}_\infty > 0$, and $\lim_{t\to\infty} \boldsymbol{\rho}(t)/g(t) = 0$. In our case, we take the limit to be the max-margin solution $\mathbf{w}_\infty = \hat{\mathbf{w}}$ and $g(t) = \log t$.

Therefore, we would like to prove that $\boldsymbol{\rho}(t)$ is bounded by a constant. To do so, we define

$$\mathbf{r}(t) := \mathbf{w}(t) - \hat{\mathbf{w}} \log t - \tilde{\mathbf{w}}, \tag{11.14}$$

where $\hat{\mathbf{w}}$ is the L_2 max-margin vector and $\tilde{\mathbf{w}}$ is a vector that satisfies

$$\forall i \in \mathcal{S} : \eta \exp\left(-\mathbf{x}_i^\top \tilde{\mathbf{w}}\right) = \alpha_i , \tag{11.15}$$

where $\mathcal{S}$ denotes the set of support vectors.

In order to prove that the residual $\boldsymbol{\rho}(t) = \mathbf{r}(t) + \tilde{\mathbf{w}}$ is bounded, we will show that $\|\mathbf{r}(t)\|$ is bounded.

First, Eq. (11.14) implies that

$$\dot{\mathbf{r}}(t) = \dot{\mathbf{w}}(t) - \frac{1}{t}\hat{\mathbf{w}} = -\nabla \mathcal{L}(\mathbf{w}(t)) - \frac{1}{t}\hat{\mathbf{w}}. \tag{11.16}$$

Therefore

$$\begin{aligned}\frac{1}{2}\frac{d}{dt}\left\|\mathbf{r}\left(t\right)\right\|^{2} &= \dot{\mathbf{r}}^{\top}\left(t\right)\mathbf{r}\left(t\right)\\ &= \sum_{i=1}^{n}\exp\left(-\mathbf{x}_{i}^{\top}\mathbf{w}(t)\right)\mathbf{x}_{i}^{\top}\mathbf{r}(t)-\frac{1}{t}\hat{\mathbf{w}}^{\top}\mathbf{r}\left(t\right)\\ &= \left[\sum_{i\in\mathcal{S}}\exp\left(-\log\left(t\right)\hat{\mathbf{w}}^{\top}\mathbf{x}_{i}-\tilde{\mathbf{w}}^{\top}\mathbf{x}_{i}-\mathbf{x}_{i}^{\top}\mathbf{r}(t)\right)\mathbf{x}_{i}^{\top}\mathbf{r}(t)-\frac{1}{t}\hat{\mathbf{w}}^{\top}\mathbf{r}\left(t\right)\right]\\ &\quad+\left[\sum_{i\notin\mathcal{S}}\exp\left(-\log(t)\hat{\mathbf{w}}^{\top}\mathbf{x}_{i}-\tilde{\mathbf{w}}^{\top}\mathbf{x}_{i}-\mathbf{x}_{i}^{\top}\mathbf{r}(t)\right)\mathbf{x}_{i}^{\top}\mathbf{r}(t)\right], \qquad (11.17)\end{aligned}$$

where in the last equality we used $\mathbf{w}(t)=\mathbf{r}(t)+\hat{\mathbf{w}}\log t+\tilde{\mathbf{w}}$ and decomposed the sum over support vectors $\mathcal{S}$ and non-support vectors. We examine both bracketed terms. Recall that $\hat{\mathbf{w}}^{\top}\mathbf{x}_{i}=1$ for $i\in\mathcal{S}$, and that we defined (in Eq. (11.15)) $\tilde{\mathbf{w}}$ so that $\sum_{i\in\mathcal{S}}\exp\left(-\tilde{\mathbf{w}}^{\top}\mathbf{x}_{i}\right)\mathbf{x}_{i}=\hat{\mathbf{w}}$. Thus, the first bracketed term in Eq. (11.17) can be written as

$$\begin{aligned}&\frac{1}{t}\sum_{i\in\mathcal{S}}\exp\left(-\tilde{\mathbf{w}}^{\top}\mathbf{x}_{i}-\mathbf{x}_{i}^{\top}\mathbf{r}\left(t\right)\right)\mathbf{x}_{i}^{\top}\mathbf{r}(t)-\frac{1}{t}\sum_{i\in\mathcal{S}}\exp\left(-\tilde{\mathbf{w}}^{\top}\mathbf{x}_{i}\right)\mathbf{x}_{i}^{\top}\mathbf{r}(t)\\ =&\frac{1}{t}\sum_{i\in\mathcal{S}}\exp\left(-\tilde{\mathbf{w}}^{\top}\mathbf{x}_{i}\right)\left(\exp\left(-\mathbf{x}_{i}^{\top}\mathbf{r}\left(t\right)\right)-1\right)\mathbf{x}_{i}^{\top}\mathbf{r}(t)\leq 0, \qquad (11.18)\end{aligned}$$

since $\forall z,\ z\left(e^{-z}-1\right)\leq 0$.

We now focus on the second bracketed term in Eq. (11.17). We define the minimum margin to a non-support vector as:

$$\theta=\min_{i\notin\mathcal{S}}\mathbf{x}_{i}^{\top}\hat{\mathbf{w}}>1\,, \qquad (11.19)$$

Since $\forall z\ e^{-z}z\leq 1$ and $\theta=\min_{i\notin\mathcal{S}}\mathbf{x}_{i}^{\top}\hat{\mathbf{w}}>1$ (Eq. (11.19)), the second bracketed term in Eq. (11.17) can be upper bounded by

$$\sum_{i\notin\mathcal{S}}\exp\left(-\log\left(t\right)\hat{\mathbf{w}}^{\top}\mathbf{x}_{i}-\tilde{\mathbf{w}}^{\top}\mathbf{x}_{i}\right)\exp\left(-\mathbf{x}_{i}^{\top}\mathbf{r}(t)\right)\mathbf{x}_{i}^{\top}\mathbf{r}(t)\leq\frac{1}{t^{\theta}}\sum_{i\notin\mathcal{S}}\exp\left(-\tilde{\mathbf{w}}^{\top}\mathbf{x}_{i}\right)\,. \qquad (11.20)$$

Substituting Eq. (11.18) and Eq. (11.20) into Eq.(11.17) and integrating, we obtain, that $\exists C, C'$ such that

$$\forall t_{1},\forall t>t_{1}:\left\|\mathbf{r}\left(t\right)\right\|^{2}-\left||\mathbf{r}(t_{1})\right||^{2}\leq C\int_{t_{1}}^{t}\frac{dt}{t^{\theta}}\leq C'<\infty\,,$$

since $\theta > 1$ (Eq. (11.19)). Thus, we showed that $\mathbf{r}(t)$ is bounded, which completes the proof for the special case.

□

We have shown that the iterates of gradient descent asymptotically converge to the max-margin SVM solution, but what about the rate of convergence? It turns out to be a very slow rate equal to

$$\left\| \frac{\mathbf{w}(t)}{\|\mathbf{w}(t)\|} = \frac{\hat{\mathbf{w}}}{\|\hat{\mathbf{w}}\|} \right\| = \mathcal{O}\left(\frac{1}{\log t}\right).$$

In contrast, the rate of convergence of the loss is of the order $\mathcal{O}\left(\frac{1}{t}\right)$, which is a much faster rate.

Validation error lower bound An important practical consequence of the theory is that although the margin of $\mathbf{w}(t)$ keeps improving, and so we can expect the population (or test) misclassification error of $\mathbf{w}(t)$ to improve for many datasets, the same cannot be said about the expected population loss (or test loss)! At the limit, the direction of $\mathbf{w}(t)$ will converge toward the max-margin predictor $\hat{\mathbf{w}}$. Although $\hat{\mathbf{w}}$ has zero training error, it will not generally have zero misclassification error on the population, or on a test or a validation set. Since the norm of $\mathbf{w}(t)$ will increase, if we use the logistic loss or any other convex loss, the loss incurred on those misclassified points will also increase.

Recall that $\mathcal{V}$ is a set of indices for validation set samples. We calculate the validation loss for logistic loss if the error of the L_2 max margin vector has some classification errors on the validation, *i.e.,* $\exists k \in \mathcal{V} : \hat{\mathbf{w}}^\top \mathbf{x}_k < 0$*:*

$$\begin{aligned}
\mathcal{L}_{\text{val}}(\mathbf{w}(t)) &= \sum_{i \in \mathcal{V}} \log\left(1 + \exp\left(-\mathbf{w}(t)^\top \mathbf{x}_i\right)\right) \\
&\geq \log\left(1 + \exp\left(-\mathbf{w}(t)^\top \mathbf{x}_k\right)\right) \\
&= \log\left(1 + \exp\left(-\left(\boldsymbol{\rho}(t) + \hat{\mathbf{w}} \log t\right)^\top \mathbf{x}_k\right)\right) \\
&= \log\left(\exp\left(-\left(\boldsymbol{\rho}(t) + \hat{\mathbf{w}} \log t\right)^\top \mathbf{x}_k\right)\left(1 + \exp\left(\left(\boldsymbol{\rho}(t) + \hat{\mathbf{w}} \log t\right)^\top \mathbf{x}_k\right)\right)\right) \\
&= -\left(\boldsymbol{\rho}(t) + \hat{\mathbf{w}} \log t\right)^\top \mathbf{x}_k + \log\left(1 + \exp\left(\left(\boldsymbol{\rho}(t) + \hat{\mathbf{w}} \log t\right)^\top \mathbf{x}_k\right)\right) \\
&\geq -\log t \; \hat{\mathbf{w}}^\top \mathbf{x}_k - \boldsymbol{\rho}(t)^\top \mathbf{x}_k
\end{aligned}$$

Thus, for all datasets $\mathcal{L}_{\text{val}}(\mathbf{w}(t)) = \Omega(\log(t))$.

Extensions to neural networks Extensions of this analysis to neural networks have been derived in Ji and Telgarsky (2018, 2019) for deep linear networks and in Lyu and Li (2019); Ji and Telgarsky (2020); Phuong and Lampert (2020) for two-hidden

layers ReLU networks. We will next discuss the result presented in Lyu and Li (2019); Ji and Telgarsky (2020).

We consider the following depth-2 ReLU neural network:

$$f_{\boldsymbol{\theta}}(\mathbf{x}) = \sum_{j\in[m]} v_j \phi(\mathbf{w}_j^\top \mathbf{x} + b_j), \tag{11.21}$$

where $\mathbf{x} \in \mathbb{R}^d$ is a data vector, $\boldsymbol{\theta}$ is the vector obtained by concatenating the vectors $\mathbf{w}_1, \ldots, \mathbf{w}_m, \mathbf{b}, \mathbf{v}$ and $\phi(z) = \max\{0, z\}$ is a ReLU activation function.

We will use the following property that is satisfied by depth-2 ReLU networks as defined above are homogeneous (with $L = 2$).

Definition 31 (Homogeneous network)**.** *We say that a network is homogeneous if there exists $L > 0$ such that for every $\alpha > 0$ and $\boldsymbol{\theta}, \mathbf{x}$ we have $f(\alpha\boldsymbol{\theta}; \mathbf{x}) = \alpha^L f(\boldsymbol{\theta}; \mathbf{x})$.*

Let $S = \{(\mathbf{x}_i, y_i)\}_{i=1}^n \subseteq \mathbb{R}^d \times \{-1, 1\}$ be a binary classification training dataset. Let $f(\boldsymbol{\theta}; \cdot) : \mathbb{R}^d \to \mathbb{R}$ be a neural network parameterized by $\boldsymbol{\theta}$. For a loss function $\ell : \mathbb{R} \to \mathbb{R}$ the empirical loss of $f(\boldsymbol{\theta}; \cdot)$ on the dataset S is

$$\mathcal{L}(\boldsymbol{\theta}) := \frac{1}{n} \sum_{i=1}^n \ell(y_i f(\boldsymbol{\theta}; \mathbf{x}_i)) \ . \tag{11.22}$$

Theorem 73 (Paraphrased from Lyu and Li (2019); Ji and Telgarsky (2020))**.** *Let $f(\boldsymbol{\theta}; \cdot)$ be a homogeneous ReLU neural network parameterized by $\boldsymbol{\theta}$. Consider minimizing either the exponential or the logistic loss over a binary classification dataset $\{(\mathbf{x}_i, y_i)\}_{i=1}^n$ using gradient flow. Assume that there exists time t_0 such that $\mathcal{L}(\boldsymbol{\theta}(t_0)) < \frac{1}{n}$ (and thus $y_i f(\boldsymbol{\theta}(t_0); \mathbf{x}_i) > 0$ for every $\mathbf{x}_i$). Then, gradient flow converges in direction to a first-order stationary point (KKT point) of the following maximum margin problem in parameter space:*

$$\min_{\boldsymbol{\theta}} \frac{1}{2} \|\boldsymbol{\theta}\|^2 \quad s.t. \quad \forall i \in [n] \ \ y_i f(\boldsymbol{\theta}; \mathbf{x}_i) \geq 1 \ . \tag{11.23}$$

Moreover, $\mathcal{L}(\boldsymbol{\theta}(t)) \to 0$ and $\|\boldsymbol{\theta}(t)\| \to \infty$ as $t \to \infty$.

Recall once more that KKT points can at a high level be understood as critical points that satisfy a set of conditions known as the Karush-Kuhn-Tucker conditions. Note however that KKT points are only critical points and are not a sufficient condition for global optimality.

11.2 Dropout

Dropout is a popular regularization technique for neural networks proposed by Hinton et al. (2012). As shown in Figure 11.4, the key idea is to randomly "drop" subsets

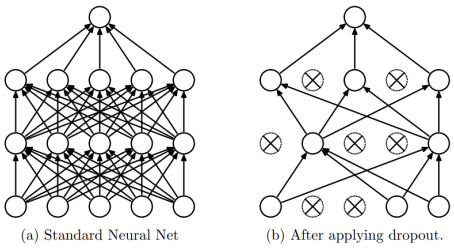


Figure 11.4: Dropout removes units at random during the forward pass. Figure from Srivastava et al. (2014).

of units in the network. More precisely, we define a "keep" probability π_i^l for each unit i in the layer l of the network. A typical choice is $\pi_i^0 = 0.8$ (inputs) and $\pi_i^{l\geq 1} = 0.5$ (hidden units). This can be realized by sampling a bit mask and zeroing out activations, so that standard backpropagation applies.

The original motivation given in Hinton et al. (2012) is as follows: ... *"overfitting is greatly reduced by randomly omitting half of the feature detectors on each training case. This prevents complex co-adaptations in which a feature detector is only helpful in the context of several other specific feature detectors. Instead, each neuron learns to detect a feature that is generally helpful for producing the correct answer given the combinatorially large variety of internal contexts in which it must operate".*

Yet another intuitive explanation of Dropout is that it can be viewed as training an ensemble of multiple models, where each model is a subset of the full network with different sets of neurons active. This ensemble effect helps improve the generalization of the model.

Analysis: the regression case In the following, we will see that one can interpret Dropout as adding a regularization term to the loss function. We will focus on the regression case and we will follow the presentation discussed in the book of Arora et al. (2020). Specifically, the model we consider is a non-linear neural network with k layers:

$$f_{\mathbf{W}}(\mathbf{x}) = \mathbf{W}_k\sigma(\mathbf{W}_{k-1}\sigma(\dots\mathbf{W}_2\sigma(\mathbf{W}_1\mathbf{x})\dots)),$$

where $\mathbf{W}_i \in \mathbb{R}^{d_i\times d_{i-1}}$ are the weight matrices, $\mathbf{x} \in \mathbb{R}^{d_0}$ is the input and $\sigma(\cdot)$ is a entrywize activation function.

For simplicity, we will consider the case where we apply Dropout to the top layer of the network. Let $\mathbf{B}$ be a diagonal random matrix with i.i.d. diagonal elements drawn from a Bernoulli distribution, i.e. $B_{ii} \sim \frac{1}{1-p}\text{Ber}(1-p), i \in [d_{k-1}]$ for some dropout rate p.

We are given n training examples $(\mathbf{x}_i, \mathbf{y}_i)_{i=1}^n$ where each $(\mathbf{x}_i, \mathbf{y}_i)$ is drawn from a probability distribution $\mathcal{D}$ defined on $\mathcal{X}\times\mathcal{Y}$ and a loss function $\ell : \mathcal{Y}\times\mathcal{Y} \to \mathbb{R}$. Our goal is to find $\mathbf{W}^*$ that minimizes the population risk $\mathbb{E}_D[\ell(f_{\mathbf{W}}(\mathbf{x}), \mathbf{y})]$.

We will choose ℓ to be the square loss function. The population risk corresponding to the scenario without Dropout is then

$$\mathcal{L}(\mathbf{W}) := \frac{1}{n}\sum_{i=1}^{n} \|\mathbf{y}_i - \mathbf{W}_k\sigma(\mathbf{W}_{k-1}\sigma(\dots \mathbf{W}_2\sigma(\mathbf{W}_1\mathbf{x}_i)\|^2. \tag{11.24}$$

In contrast, the population risk corresponding to the scenario with Dropout is

$$\mathcal{L}_{Drop}(\mathbf{W}) := \frac{1}{n}\sum_{i=1}^{n} \mathbb{E}_{\mathbf{B}}\|\mathbf{y}_i - \mathbf{W}_k\mathbf{B}\sigma(\mathbf{W}_{k-1}\sigma(\dots \mathbf{W}_2\sigma(\mathbf{W}_1\mathbf{x}_i)\|^2. \tag{11.25}$$

Our goal is to understand the explicit regularization introduced by Dropout, which is captured by the term $\mathcal{L}_{Drop}(\mathbf{W}) - \mathcal{L}(\mathbf{W})$. Let $a_{i,j} \in \mathbb{R}$ denote the output of the i-th hidden node in the j-th hidden layer on an input vector $\mathbf{x}$. Also let vector $\mathbf{a}_j \in \mathbb{R}^{d_j}$ denote the activation of the j-th layer on the input $\mathbf{x}$. We can then rewrite the Dropout objective as

$$\mathcal{L}_{Drop}(\mathbf{W}) := \frac{1}{n}\sum_{i=1}^{n} \mathbb{E}_{\mathbf{B}}\|\mathbf{y}_i - \mathbf{W}_k\mathbf{B}\mathbf{a}_{k-1}(\mathbf{x}_i)\|^2. \tag{11.26}$$

The following theorem will show that the Dropout objective $\mathcal{L}_{Drop}(\mathbf{W})$ is a regularized version of the original objective $\mathcal{L}(\mathbf{W})$.

Theorem 74 (Dropout regularizer in deep regression)**.** *The population risk in the Dropout scenario can be written as*

$$\mathcal{L}_{Drop}(\mathbf{W}) = \mathcal{L}(\mathbf{W}) + R(\mathbf{W}),$$

where $R(\mathbf{W})$ is a regularizer term defined as

$$R(\mathbf{W}) = \lambda \sum_{j=1}^{d_{k-1}} \|\mathbf{W}_k(:,j)\|^2 \hat{a}_j^2,$$

where $\hat{a}_j = \left(\frac{1}{n}\sum_{i=1}^{n} a_{j,k-1}(\mathbf{x}_i)^2\right)^{1/2}$ and $\lambda = \frac{p}{1-p}$.

Proof. Since $B_{ii} \sim \frac{1}{1-p}\text{Ber}(1-p), i \in [d_{k-1}]$, we have $\mathbb{E}[B_{ii}] = 1$ and $\text{var}[B_{ii}] = \frac{p}{1-p}$.

For simplicity, we consider a single sample $(\mathbf{x}, \mathbf{y})$ and denote the j-th coordinate of $\mathbf{y}$ as y_j. Then,

$$\mathbb{E}_{\mathbf{B}}\|\mathbf{y} - \mathbf{W}_k\mathbf{B}\mathbf{a}_{k-1}(\mathbf{x})\|^2 = \sum_{j=1}^{d_k} \mathbb{E}_{\mathbf{B}}[y_j - \mathbf{W}_k(j,:)^\top \mathbf{B}\mathbf{a}_{k-1}]^2. \tag{11.27}$$

For each summand,

$$\mathbb{E}_{\mathbf{B}}[y_j - \mathbf{W}_k(j,:)^\top \mathbf{B}\mathbf{a}_{k-1}]^2 = \left(\mathbb{E}_{\mathbf{B}}[y_j - \mathbf{W}_k(j,:)^\top \mathbf{B}\mathbf{a}_{k-1}]\right)^2 + \text{var}[y_j - \mathbf{W}_k(j,:)^\top \mathbf{B}\mathbf{a}_{k-1}],$$

based on the definition of the variance.

Then, since $\mathbb{E}[\mathbf{B}] = \mathbf{I}$, the first term on the RHS is equal to

$$\left(\mathbb{E}_{\mathbf{B}}[y_j - \mathbf{W}_k(j,:)^\top \mathbf{B}\mathbf{a}_{k-1}]\right)^2 = \left([y_j - \mathbf{W}_k(j,:)^\top \mathbf{a}_{k-1}]\right)^2.$$

The second term on the RHS can me manipulated as follows

$$\begin{aligned}
\text{var}[y_j - \mathbf{W}_k(j,:)^\top \mathbf{B}\mathbf{a}_{k-1}] &= \text{var}[\mathbf{W}_k(j,:)^\top \mathbf{B}\mathbf{a}_{k-1}] \\
&\overset{(i)}{=} \text{var}[\sum_{l=1}^{d_{k-1}} \mathbf{W}_k(j,l)\mathbf{B}_{ll} a_{l,k-1}] \\
&\overset{(ii)}{=} \sum_{l=1}^{d_{k-1}} (\mathbf{W}_k(j,l) a_{l,k-1})^2 \,\text{var}[\mathbf{B}_{ll}] \\
&= \frac{p}{1-p} \sum_{l=1}^{d_{k-1}} (\mathbf{W}_k(j,l) a_{l,k-1})^2
\end{aligned}$$

where (i) uses the fact that $\mathbf{B}$ is diagonal, (ii) uses $\text{var}[aX] = a^2 \,\text{var}[X]$ and $\text{var}\left[\sum_{i=1}^n X_i\right] = \sum_{i=1}^n \text{var}[X_i]$ for uncorrelated X_i's.

Putting things together, we obtain

$$\mathbb{E}_{\mathbf{B}}\|\mathbf{y} - \mathbf{W}_k \mathbf{B}\mathbf{a}_{k-1}(\mathbf{x})\|^2 = \|\mathbf{y} - \mathbf{W}_k \mathbf{a}_{k-1}(\mathbf{x})\|^2 + \frac{p}{1-p} \sum_{l=1}^{d_{k-1}} \|\mathbf{W}_k(:,l)\|^2 a_{l,k-1}^2.$$

Finally, we conclude the proof by averaging over all samples.

□

11.3 Exercise: Regularization

Problem 1 (Regularization warm-up with Ridge Regression):

Consider a data matrix $\mathbf{X} \in \mathbb{R}^{n \times d}$ that contains n datapoints $\mathbf{x}_i \in \mathbb{R}^d$ for $i = 1, \ldots, n$, as well as the corresponding targets $y_i \in \mathbb{R}$. The ridge regression solution for a linear model is given by

$$\hat{\mathbf{w}} = \underset{\mathbf{w} \in \mathbb{R}^d}{\operatorname{argmin}} \|\mathbf{X}\mathbf{w} - \mathbf{y}\|_2^2 + \frac{\lambda}{2}\|\mathbf{w}\|_2^2, \tag{11.28}$$

where $\lambda > 0$ and $\mathbf{y} = (y_1, \ldots y_n) \in \mathbb{R}^n$.

With the singular value decomposition $\mathbf{X} = \mathbf{U}\mathbf{D}\mathbf{V}^\top$, show that the prediction $\hat{y} := \mathbf{X}\hat{\mathbf{w}}$ has the following form:

$$\hat{\mathbf{y}} = \sum_{i=1}^{d} \mathbf{u}_i \frac{d_i^2}{d_i^2 + \lambda} \mathbf{u}_i^\top \mathbf{y}. \tag{11.29}$$

How does the above differ from the predictions made by a non-regularized linear estimator? What is the effect of λ?

Problem 2 (Connection between early stopping and L^2 regularization):

Early stopping is a regularization method, used when training a learner with an iterative method, in which the parameters from an earlier iteration (rather than the last one) are returned. In most cases, the criterion used to decide the "stop time" is the error on the validation set.

Formally, we are interested in optimizing the risk function $\mathcal{R}(\mathbf{w})$. We assume we can approximate it using the second-order approximation around an optimal $\mathbf{w}^*$, i.e.

$$\mathcal{R}(\mathbf{w}) \approx \mathcal{R}(\mathbf{w}^*) + \frac{1}{2}(\mathbf{w} - \mathbf{w}^*)^\top \mathbf{H}(\mathbf{w} - \mathbf{w}^*). \tag{11.30}$$

Our goal will be to identify a connection between early stopping and L^2 regularization for a linear model with parameters $\mathbf{w}$. To do so, we will follow these steps:

a) Starting from the quadratic approximation of $\mathcal{R}(\mathbf{w})$, compute its gradient and write down the update rule for $\mathbf{w}_k$ given by gradient descent. Derive a recursive equation for the difference vector $\mathbf{w}_k - \mathbf{w}^*$.

b) Re-write the equation you derived in step 1 using the eigen-decomposition $\mathbf{H} = \mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^\top$.

c) Collapse the recursive equation (i.e., expand steps $k-1, \ldots, 1, 0$) to get a simplified expression for $\mathbf{w}_k$. You should get to an expression that depends on the term $(\mathbf{w}_0 - \mathbf{w}^*)$. Then simplify the obtained expression by assuming that $\mathbf{w}_0 = \mathbf{0}$.

d) Now consider the regularizer risk $\mathcal{R}(\mathbf{w}) + \lambda\|\mathbf{w}\|_2^2$. Show that the first-order optimality condition can be written as

$$\mathbf{Q}^\top \mathbf{w} = \left[\mathbf{I} - \lambda(\mathbf{\Lambda} + \lambda\mathbf{I})^{-1}\right]\mathbf{Q}^\top \mathbf{w}^*. \tag{11.31}$$

Hint: recall that if $\mathbf{A}, \mathbf{B}$ are two invertible matrices, then $(\mathbf{AB})^{-1} = \mathbf{B}^{-1}\mathbf{A}^{-1}$.

e) Conclude by matching the two formulas and by deriving a connection between the weight decay factor λ, the learning rate η, and the (early) stopping time τ. (You can use approximations, e.g., make some assumptions about the eigenvalues relative to these parameters to show how they depend on one another.)

Chapter 12

Adversarial Examples

While neural networks have recently achieved staggering performance at some tasks (such as image classification), they also exhibit robustness failures. For instance, in a classification setting, Szegedy et al. (2013) show that one can slightly perturb an image to trick the model into predicting a wrong class. This work triggered a lot of interest in developing adversarial attacks as well as countermeasures to make neural networks more robust to these attacks.

12.1 Adversarial Examples

We consider the following typical classification setting encountered in machine learning. We are given a training set $\mathcal{S} = (\mathbf{x}_i, y_i)_{i=1}^n$ consisting of pairs of feature vectors $\mathbf{x}_i \in \mathbb{R}^d$ and corresponding labels $y_i \in \{1, \dots k\}$ where k is the number of classes. We define a model $f_{\mathbf{w}}(\mathbf{x}) : \mathbb{R}^d \to \mathbb{R}^k$ that is parametrized by $\mathbf{w} \in \mathbb{R}^d$. The parameter vector $\mathbf{w}$ is trained by optimizing a loss function $\ell : \mathbb{R}^k \times \mathbb{Z}_+ \to \mathbb{R}_+$. A typical loss function used to train deep neural networks is the cross-entropy:

$$\ell(f_{\mathbf{w}}(\mathbf{x}), y) = \log\left(\sum_{j=1}^{k} \exp(f_{\mathbf{w}}(\mathbf{x}))_j\right) - f_{\mathbf{w}}(\mathbf{x})_y,$$

where $f_{\mathbf{w}}(\mathbf{x})_j$ denotes the j-th entry of the vector $f_{\mathbf{w}}(\mathbf{x})$.

Definition of an adversarial example Let's start with an informal definition of an adversarial example. An adversarial example is an input to a machine learning model that was specifically designed to cause the model to predict the wrong class. But how do we create such an example given a pair $(\mathbf{x}, y)$? What we want is to find an example $\bar{\mathbf{x}}$ that is "close" (we will soon define this more precisely) to $\mathbf{x}$ and such

that the corresponding loss $\ell(f_{\mathbf{w}}(\bar{\mathbf{x}}), y)$ is large. We could simply frame this as an optimization problem, i.e. find the perturbation $\bar{\boldsymbol{\delta}}$ such that $\bar{\mathbf{x}} = \mathbf{x} + \boldsymbol{\delta}$, where

$$\bar{\boldsymbol{\delta}} = \underset{\boldsymbol{\delta} \in \Delta}{\operatorname{argmax}}\, \ell(f_{\mathbf{w}}(\mathbf{x} + \boldsymbol{\delta}), y). \tag{12.1}$$

We still need to precisely define the set Δ (in order to explain what we mean by "close" perturbations). Ideally, we would like to have some notion of distance that ensures that the perturbation $\boldsymbol{\delta}$ does not significantly deviate from $\mathbf{x}$. A common choice is to use the L_∞ norm, i.e.

$$\Delta = \{\boldsymbol{\delta} : \|\boldsymbol{\delta}\|_\infty \leq \epsilon\}, \tag{12.2}$$

for a small $\epsilon > 0$.

One obvious open question is how do we optimize Eq. (12.1)? One can simply use (stochastic) gradient descent to maximize Eq. (12.1). This means that we can simply update $\boldsymbol{\delta}$ as follows:

$$\boldsymbol{\delta}_{k+1} = \boldsymbol{\delta}_k + \eta \nabla_{\boldsymbol{\delta}} \ell(f_{\mathbf{w}}(\mathbf{x} + \boldsymbol{\delta}), y), \tag{12.3}$$

where $\eta > 0$ is a step-size parameter and $\boldsymbol{\delta}_0 \in \Delta$. After each iteration k, we project $\boldsymbol{\delta}$ back into the norm ball defined by $\|\boldsymbol{\delta}\|_\infty \leq \epsilon$. Projecting onto this norm ball involves clipping values of $\boldsymbol{\delta}$ to lie within the range $[-\epsilon, \epsilon]$. A surrogate update involves taking the sign of the gradient [1]:

$$\boldsymbol{\delta}_{k+1} = \boldsymbol{\delta}_k + \eta\, \text{sign}(\nabla_{\boldsymbol{\delta}} \ell(f_{\mathbf{w}}(\mathbf{x} + \boldsymbol{\delta})), y). \tag{12.4}$$

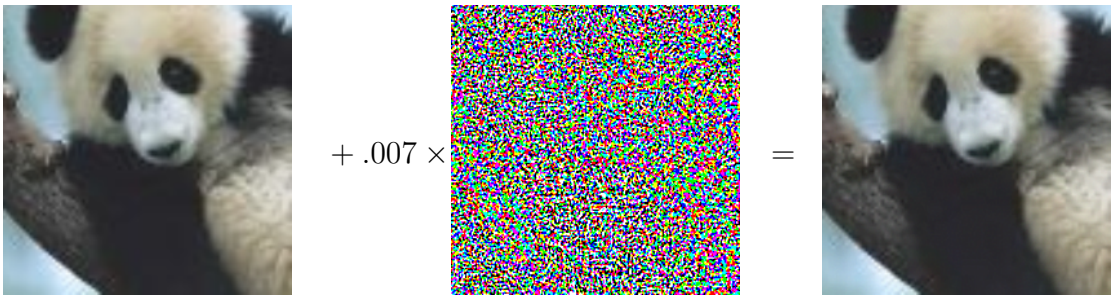
This is the so-called Fast Gradient Sign Method (FGSM) introduced in Goodfellow et al. (2014). We show an illustration of an adversarial example created by FGSM in Figure. 12.1.

We differentiate between different types of attacks:

- White-box attacks: the attacker is assumed to have access to the model parameters. This is a rather strong assumption.
- Black-box attacks: The attacker does not have access to the model parameters, and is instead only able to query the model.

In the case of black-box attacks, the attacker is not able to directly compute the gradients of the models required in Eq. (12.3). However, these derivatives can be estimated using a finite difference approximation.

[1] If we take the step size large enough, then we observe that the relative size of the entries does not matter, and one can therefore simply consider the sign of the entries, thus dropping the magnitude of the gradient vector.



$\mathbf{x}$
"panda"
57.7% confidence

$\text{sign}(\nabla_{\boldsymbol{\delta}} \ell(f_{\mathbf{w}}(\mathbf{x} + \boldsymbol{\delta})), y)$
"nematode"
8.2% confidence

$\mathbf{x} + \eta \, \text{sign}(\nabla_{\boldsymbol{\delta}} \ell(f_{\mathbf{w}}(\mathbf{x} + \boldsymbol{\delta})), y)$
"gibbon"
99.3 % confidence

Figure 12.1: A demonstration of fast adversarial example generation applied to GoogLeNet on ImageNet. By adding an imperceptibly small perturbation δ whose elements are equal to the sign of the elements of the gradient of the cost function with respect to the input, we can change GoogLeNet's classification of the image. Here $\eta = 0.007$ corresponds to the magnitude of the smallest bit of an 8-bit image encoding after GoogLeNet's conversion to real numbers.
Source: figure from Goodfellow et al. (2014).

12.2 Existence of Robust Networks

Consider a two-layer neural network of width k defined by

$$f(\mathbf{x}) = \sum_{j=1}^{k} a_j \sigma(\mathbf{w}_j^\top \mathbf{x} + b_j), \tag{12.5}$$

where $\mathbf{x} \in \mathbb{R}^d$ is the input, $\mathbf{w}_j \in \mathbb{R}^d$ and $b_j \in \mathbb{R}$ are the weight vectors and offset of the first layer, while $a_j \in \mathbb{R}$ are the weights of the second layer. We will assume that the input vectors $\mathbf{x}_1, \dots \mathbf{x}_n$ are on the sphere of radius $\sqrt{d}$, denoted by $\sqrt{d}\mathbb{S}^{d-1}$. Then, the distance between every two inputs is at most $\mathcal{O}(\sqrt{d})$, which implies a perturbation of size $\mathcal{O}(\sqrt{d})$ is sufficient to flip the sign of the input (assuming $\exists i, j$ s.t. $f(\mathbf{x}_i) > 0$ and $f(\mathbf{x}_j) < 0$).

The following theorem derived by Vardi et al. (2022) explicitly constructs an example of a neural network that is $\sqrt{d}$-robust, i.e. it requires a large perturbation of size $\mathcal{O}(\sqrt{d})$ to change the predicted class. This is illustrated in Figure 12.2.

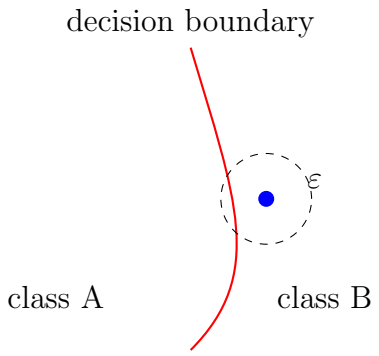


Figure 12.2: Being robust means that the size of the ϵ-ball is large, i.e. one needs a large perturbation to change the class of a datapoint.

Theorem 75 (Vardi et al. (2022))**.** *Let $\{(\mathbf{x}_i, y_i)\}_{i=1}^m \subseteq (\sqrt{d} \cdot \mathbb{S}^{d-1}) \times \{-1, 1\}$ be a dataset. Let $0 < c < 1$ be a constant independent of d and suppose that $|\langle \mathbf{x}_i, \mathbf{x}_j \rangle| \leq c \cdot d$ for every $i \neq j$. Then, there exists a depth-2 ReLU network f of width m such that $y_i f(\mathbf{x}_i) \geq 1$ for every $i \in [m]$, and for every $\mathbf{x}_i$ flipping the sign of the output requires a perturbation of size larger than $\frac{1-c}{4} \cdot \sqrt{d}$. Thus, f is $\sqrt{d}$-robust w.r.t. $\mathbf{x}_1, \ldots, \mathbf{x}_m$.*

Proof. Consider the network $f(\mathbf{x}) = \sum_{j=1}^m a_j \sigma(\mathbf{w}_j^\top \mathbf{x} + b_j)$ such that for every $j \in m$ we have $a_j = y_j$, $\mathbf{w}_j = \frac{2\mathbf{x}_j}{d(1-c)}$ and $b_j = -\frac{1+c}{1-c}$. For every $i \in [m]$ we have

$$\begin{aligned} \mathbf{w}_i^\top \mathbf{x}_i + b_i &= \frac{2\left\|\mathbf{x}_i\right\|^2}{d(1-c)} - \frac{1+c}{1-c} \\ &= \frac{2d}{d(1-c)} - \frac{1+c}{1-c} = 1, \end{aligned}$$

and for every $i \neq j$ we have

$$\mathbf{w}_j^\top \mathbf{x}_i + b_j = \frac{2\mathbf{x}_j^\top \mathbf{x}_i}{d(1-c)} - \frac{1+c}{1-c} \leq \frac{2c}{1-c} - \frac{1+c}{1-c} = -1,$$

where we used the condition $|\langle \mathbf{x}_i, \mathbf{x}_j \rangle| \leq c \cdot d$ in the inequality (i.e. the correlation between every pair of examples is not too large).
Hence, since σ is a ReLU function, we get $f(\mathbf{x}_i) = a_i = y_i$ for every $i \in [m]$.

We now prove that f is $\sqrt{d}$-robust. Let $i \in [m]$ and let $\mathbf{x}_i' \in \mathbb{R}^d$ such that $\|\mathbf{x}_i - \mathbf{x}_i'\| \leq \frac{1-c}{4} \cdot \sqrt{d}$. We show that $\operatorname{sign}(f(\mathbf{x}_i')) = \operatorname{sign}(f(\mathbf{x}_i))$. Indeed, we have

$$\mathbf{w}_i^\top \mathbf{x}_i' + b_i = \mathbf{w}_i^\top (\mathbf{x}_i' - \mathbf{x}_i) + \underbrace{\mathbf{w}_i^\top \mathbf{x}_i + b_i}_{=1} \geq -\|\mathbf{w}_i\| \cdot \|\mathbf{x}_i' - \mathbf{x}_i\| + 1 \geq \frac{-2}{\sqrt{d}(1-c)} \cdot \frac{\sqrt{d}(1-c)}{4} + 1 = \frac{1}{2}.$$

Also, for every $i \neq j$ we have

$$\mathbf{w}_j^\top \mathbf{x}_i' + b_j = \mathbf{w}_j^\top (\mathbf{x}_i' - \mathbf{x}_i) + \underbrace{\mathbf{w}_j^\top \mathbf{x}_i + b_j}_{=-1} \leq \|\mathbf{w}_j\| \cdot \|\mathbf{x}_i' - \mathbf{x}_i\| - 1 \leq \frac{2}{\sqrt{d}(1-c)} \cdot \frac{\sqrt{d}(1-c)}{4} - 1 = -\frac{1}{2}.$$

Therefore, $\text{sign}(f(\mathbf{x}_i')) = \text{sign}(a_i) = \text{sign}(y_i) = \text{sign}(f(\mathbf{x}_i))$. □

Despite the existence of robust networks, gradient descent is not guaranteed to converge to them (see for instance Vardi et al. (2022) for a proof based on gradient flow).

12.3 Adversarial Examples at Initialization

We will now present a result by Bubeck et al. (2021) that shows that a single step of gradient descent at initialization suffices to find an adversarial example. We consider a two-layer network similar to the one defined in Eq. (12.5) but where we set $b_j = 0$ for all j and with a slightly different scaling:

$$f(\mathbf{x}) = \frac{1}{\sqrt{k}} \sum_{j=1}^{k} a_j \sigma(\mathbf{w}_j^\top \mathbf{x}). \tag{12.6}$$

We initialize the weights $\mathbf{w}_j \sim \mathcal{N}\left(0, \frac{1}{d}\mathbf{I}_d\right)$ and $a_j \in \mathbb{R}$ are independent from the $\mathbf{w}_j$'s and i.i.d. uniformly distributed in $\{-1, +1\}$. With this parametrization, the central limit theorem states that, for $\mathbf{x} \in \sqrt{d} \cdot \mathbb{S}^{d-1}$ (so that $\mathbf{w}_j^\top \mathbf{x} \sim \mathcal{N}(0,1)$) and large width k, the distribution of $f(\mathbf{x})$ is approximately a centered Gaussian with variance equal to

$$\mathbb{E}_{X \sim \mathcal{N}(0,1)}[\sigma(X)^2] = \mathcal{O}(1). \tag{12.7}$$

12.3.1 Preliminaries

Concentration of random variables In the following, we will make use of Bernstein's inequality which we restate below for convenience (see e.g. Theorem 2.10 in Boucheron et al. (2013)):

Theorem 76 (Bernstein's inequality)**.** *Let (X_j) be i.i.d. centered random variables such that there exists $\sigma, c > 0$ such that for all integers $q \geq 2$,*

$$\mathbb{E}[|X_j|^q] \leq \frac{q!}{2}\sigma^2 c^{q-2}\,.$$

Then with probability at least $1-\gamma$ one has:

$$\sum_{j=1}^{k} X_j \leq \sqrt{2\sigma^2 k \log(1/\gamma)} + c\log(1/\gamma)\,.$$

Proof sketch. We first bound the moment generating function (MGF) using the condition $\mathbb{E}[|X_j|^q] \leq \frac{q!}{2}\,\sigma^2 c^{q-2}$. For $\lambda \in [0, 1/c)$, we have

$$\mathbb{E}\left[e^{\lambda X_j}\right] = 1 + \sum_{q=2}^{\infty} \frac{\lambda^q \mathbb{E}[X_j^q]}{q!} \leq 1 + \frac{\sigma^2}{2}\sum_{q=2}^{\infty} \lambda^q c^{q-2}.$$

Simplify the sum:

$$\mathbb{E}\left[e^{\lambda X_j}\right] \leq 1 + \frac{\sigma^2\lambda^2}{2(1-\lambda c)}.$$

Using $1 + x \leq e^x$, we get

$$\mathbb{E}\left[e^{\lambda X_j}\right] \leq \exp\left(\frac{\sigma^2\lambda^2}{2(1-\lambda c)}\right).$$

Since the X_j are independent, the MGF of their sum is

$$\mathbb{E}\left[e^{\lambda \sum_{j=1}^{k} X_j}\right] \leq \exp\left(\frac{k\sigma^2\lambda^2}{2(1-\lambda c)}\right).$$

Using Markov's inequality:

$$\Pr\left(\sum_{j=1}^{k} X_j \geq t\right) \leq e^{-\lambda t}\mathbb{E}\left[e^{\lambda \sum_{j=1}^{k} X_j}\right] \leq \exp\left(-\lambda t + \frac{k\sigma^2\lambda^2}{2(1-\lambda c)}\right).$$

To conclude the bound, we have to optimize the RHS over λ (this can be done by setting the derivative to zero).

□

We will also use the following concentration of χ^2 (chi-square) random variables (see e.g., (2.19) in Wainwright (2019)). Recall that the probability density function (pdf) of a chi-square random variable X with k degrees of freedom is given by:

$$f(x;k) = \frac{1}{2^{k/2}\Gamma(k/2)} x^{(k/2)-1} e^{-x/2}$$

where $x \geq 0$ is the variable, k is the degrees of freedom, $\Gamma(\cdot)$ is the gamma function, and $k/2$ represents the shape parameter. The mean and variance of a chi-square random variable are given by $\mathbb{E}(X) = k$ and variance $\text{var}(X) = 2k$. The name "chi-square" reflects the fact that this distribution arises from summing the squares of independent standard normal variables.

Lemma 77 (Concentration χ^2)**.** *Let $X_1, \dots, X_k$ be i.i.d. standard Gaussians, then with probability at least $1 - \gamma$, one has:*

$$\left| \sum_{j=1}^{k} X_j^2 - k \right| \leq 4\sqrt{k \log(2/\gamma)}\,. \tag{12.8}$$

Proof idea. This result follows from the fact that $\sum_{j=1}^{k} X_j^2$ is a Chi-squared random variable (with k degrees random of freedom) that concentrates around its mean (which is equal to k). □

Asymptotic analysis We will also use various mathematical notations, $\mathcal{O}, o, \Omega$ that allow us to relate the growth of various functions, which we briefly review below.

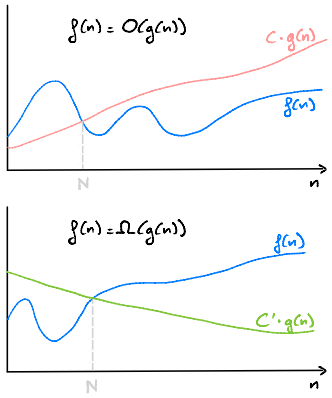


Figure 12.3: $\mathcal{O}$ vs Ω notation.

- $f(n) = o(g(n))$ if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$. So as n gets larger, $f(n)$ becomes an insignificant portion of $g(n)$.
- $f(n) = \mathcal{O}(g(n))$ if there exists constants N and C such that $|f(n)| < Cg(n)$ for all $n > N$.
- $f(n) = \Omega(g(n))$ if there exists constants N and C' such that $|f(n)| > C'g(n)$ for all $n > N$.

Examples

- $f(n) = 6n^4 - 2n^3 + 5 = \mathcal{O}(n^4)$
- $f(n) = 3n + 4 = \Omega(n)$
- $f(n) = n^2, g(n) = n^3$, then $f(n) = o(g(n))$ since $\lim_{n\to\infty} \frac{n^2}{n^3} = 0$.

12.3.2 Main Result

We will conduct the analysis under the following assumption on the activation function σ.

Assumption 5. *Let σ be differentiable almost everywhere, and assume that there exists $\sigma', c' > 0$ such that for all integers $q \geq 2$,*

$$\mathbb{E}_{X\sim\mathcal{N}(0,1)}[|\sigma'(X)|^{2q}] \leq \frac{q!}{2}\sigma'^2 c'^{q-2}\,.$$

We are now reading to state our formal main result that shows that one step of gradient descent is sufficient to flip the class predicted by our neural network.

Theorem 78. *Let $\gamma \in (0,1)$ and σ be non-constant, Lipschitz and with Lipschitz derivative. Assume $k \geq \mathcal{O}(\log^3(1/\gamma))$ and $d \geq \mathcal{O}(\log(k/\gamma)\log(1/\gamma))$. Then, there exists $\eta > 0$ such that with probability at least $1-\gamma$ one has:*

$$\operatorname{sign}(f(\mathbf{x})) \neq \operatorname{sign}(f(\mathbf{x} + \eta\nabla f(\mathbf{x})))\,.$$

We will give a proof sketch of the theorem that will rely on the following proposition, which shows that the norm of the gradient is independent of the dimension d.

Proposition 79. *Assume σ satisfies Assumption 5. Then with high probability, for any $\mathbf{x} \in \sqrt{d}\mathbb{S}^{d-1}$,*

$$\|\nabla f(\mathbf{x})\| = \Omega(1).$$

Proof sketch of Prof. 79. For a smooth activation function σ, we have

$$\nabla f(\mathbf{x}) = \frac{1}{\sqrt{k}}\sum_{j=1}^{k} a_j \mathbf{w}_j \sigma'(\mathbf{w}_j^\top \mathbf{x}).$$

Let $P = \mathbf{I}_d - \frac{\mathbf{x}\mathbf{x}^\top}{d}$ be the projection on the orthogonal complement of the span of $\mathbf{x}$. Since this projection does not increase the norm, we have $\|\nabla f(\mathbf{x})\| \geq \|P\nabla f(\mathbf{x})\|$. Moreover $a_j P\mathbf{w}_j$ is distributed as $\mathcal{N}\left(0, \frac{1}{d}\mathbf{I}_{d-1}\right)$ and is independent of $\mathbf{w}_j^\top\mathbf{x}$, and thus conditioning on the values $(\mathbf{w}_j^\top\mathbf{x})_{j\in[k]}$ we obtain:

$$P\nabla f(\mathbf{x}) = \frac{1}{\sqrt{k}}\sum_{j=1}^{k} a_j P\mathbf{w}_j\sigma'(\mathbf{w}_j^\top\mathbf{x}) \overset{(d)}{=} \left(\sqrt{\frac{1}{kd}\sum_{j=1}^{k}\sigma'(\mathbf{w}_j^\top\mathbf{x})^2}\right)\mathbf{y} \quad \text{where } \mathbf{y}\sim\mathcal{N}\left(0,\mathbf{I}_{d-1}\right),$$

where the notation $\overset{(d)}{=}$ means both sides are equal in distribution.

Using (12.8) we have that with probability at least $1-\gamma$:

$$\|\mathbf{y}\|^2 = \sum_{i=1}^{d} y_i^2 \geq d - 1 - 4\sqrt{d\log(2/\gamma)} \geq d - 5\sqrt{d\log(2/\gamma)}\,.$$

where we used that $d \geq 1$ and $\gamma < 2/e$.

Then with probability at least $1-\gamma$ for $0 < \gamma < 2/e$ one has:

$$\|\nabla f(\mathbf{x})\| \geq \|P\nabla f(\mathbf{x})\| \geq \left(1 - 5\sqrt{\frac{\log(2/\gamma)}{d}}\right)\sqrt{\frac{1}{k}\sum_{j=1}^{k}\sigma'(\mathbf{w}_j^\top \mathbf{x})^2}\,.$$

Finally, the term $\left(\sqrt{\frac{1}{k}\sum_{j=1}^{k}\sigma'(\mathbf{w}_j^\top \mathbf{x})^2}\right)$ can be shown to be close to a constant (w.h.p.) using Assumption 5 and Bernstein's inequality (by first centering the random variable). This last step is left as an exercise.

□

Proof idea of Theorem 78. Consider a simple linear model of the form $f(\mathbf{x}) = \mathbf{w}^\top \mathbf{x}$, where $\mathbf{w} \in \mathbb{R}^d$ is a set of parameters and $\mathbf{x} \in \mathbb{R}^d$ is a given input. We consider the case where the parameters are initialized at random, $\mathbf{w} \sim \mathcal{N}(0, \mathbf{I}_d)$.

Consider perturbations Δ of $\mathbf{x}$ in the direction $-f(\mathbf{x})\nabla f(\mathbf{x})$:

$$\mathbf{x}_+ = \mathbf{x} - \eta f(\mathbf{x})\nabla f(\mathbf{x}), \tag{12.9}$$

where $\eta > 0$ is a step size that controls the magnitude of the perturbation applied to $\mathbf{x}$.

By Taylor expanding $f(\mathbf{x})$, we get

$$\begin{aligned} f(\mathbf{x}_+) &= f(\mathbf{x}) + \nabla f(\mathbf{x})^\top(\mathbf{x}_+ - \mathbf{x}) \\ &= f(\mathbf{x}) - \nabla f(\mathbf{x})^\top(\eta f(\mathbf{x})\nabla f(\mathbf{x})) \\ &= f(\mathbf{x})(1 - \eta\|\nabla f(\mathbf{x})\|^2). \end{aligned}$$

This implies that $f(\mathbf{x})$ moves towards zero as long as $\|\nabla f(\mathbf{x})\| \neq 0$. In fact, one can check that if we choose a perturbation Δ such that $\|\Delta\| = \frac{2|f(\mathbf{x})|}{\|\nabla f(\mathbf{x})\|}$, the sign of $f(\mathbf{x})$ flips. By Proposition 79, recall that $\|\nabla f(\mathbf{x})\| = \Omega(1)$, which implies that we can change the prediction of the function $f(\mathbf{x})$ using a perturbation of constant size.

□

12.4 Adversarial Training

Next, we discuss how to train a classifier that is robust to adversarial samples (in the sense that we don't want close adversarial perturbations to exist). To do so, we once

again consider the standard risk setting encountered in machine learning. We define the true risk as the expected loss under the true distribution of the samples denoted by $\mathcal{D}$, i.e.

$$R(f_{\mathbf{w}}) = \mathbb{E}_{(\mathbf{x},y)\sim\mathcal{D}}[\ell(f_{\mathbf{w}}(\mathbf{x}), y)]. \tag{12.10}$$

The distribution $\mathcal{D}$ is typically unknown, as we instead only have access to a finite set of samples $\mathcal{S} = (\mathbf{x}_i, y_i)_{i=1}^n$. We thus consider the empirical risk defined as

$$\hat{R}(f_{\mathbf{w}}) = \frac{1}{|\mathcal{S}|} \sum_{(\mathbf{x},y)\in\mathcal{S}} \ell(f_{\mathbf{w}}(\mathbf{x}), y). \tag{12.11}$$

As we have seen, the problem with this approach is that the model obtained by optimizing the empirical risk might be vulnerable to adversarial attacks. One can however purposefully modify the risk to make the model more robust to adversarial examples. This can be achieved by modifying the loss to consider the whole $\Delta(\mathbf{x})$ region around each sample as follows:

$$R_{\text{adv}}(f_{\mathbf{w}}) = \mathbb{E}_{(\mathbf{x},y)\sim\mathcal{D}} \left[\max_{\boldsymbol{\delta}\in\Delta(\mathbf{x})} \ell(f_{\mathbf{w}}(\mathbf{x}+\boldsymbol{\delta}), y) \right]. \tag{12.12}$$

The empirical counterpart can simply be defined as

$$\hat{R}_{\text{adv}}(f_{\mathbf{w}}) = \frac{1}{|\mathcal{S}|} \sum_{(\mathbf{x},y)\sim\mathcal{S}} \max_{\boldsymbol{\delta}\in\Delta(\mathbf{x})} \ell(f_{\mathbf{w}}(\mathbf{x}+\boldsymbol{\delta}), y). \tag{12.13}$$

Stochastic Gradient Descent (SGD) Let's now turn to the question of optimizing the risk defined in Eq. 12.13. A natural candidate to solve this optimization problem is SGD whose update step is

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \frac{\eta}{|\mathcal{B}|} \sum_{(\mathbf{x},y)\sim\mathcal{B}} \nabla_{\mathbf{w}} \max_{\boldsymbol{\delta}\in\Delta(\mathbf{x})} \ell(f_{\mathbf{w}}(\mathbf{x}+\boldsymbol{\delta}), y) \tag{12.14}$$

where $\eta > 0$ and $\mathcal{B} \subseteq \mathcal{S}$ is a mini-batch and $\eta > 0$ is a chosen step-size.

Observe that the update of SGD involves the gradient of the maximum over perturbations $\boldsymbol{\delta}$. How do we compute such a gradient? Fortunately, there is a simple answer provided by Danskin's theorem that states that the gradient of the maximum of functions is the gradient of the function evaluated at the maximum, thus

$$\nabla_{\mathbf{w}} \max_{\boldsymbol{\delta}\in\Delta(\mathbf{x})} \ell(f_{\mathbf{w}}(\mathbf{x}+\boldsymbol{\delta}), y) = \nabla_{\mathbf{w}} \ell(f_{\mathbf{w}}(\mathbf{x}+\boldsymbol{\delta}^*), y). \tag{12.15}$$

Additional references

See the *Adversarial Robustness - Theory and Practice* tutorial by by Zico Kolter and Aleksander Madry at `https://adversarial-ml-tutorial.org/`.

Chapter 13

Double Descent & Bias-variance Decomposition

13.1 Double Descent Phenomenon

Let's consider the typical machine learning setting where a model $h \in \mathcal{H}$ tries to predict an output $y \in \mathcal{Y} \subseteq \mathbb{R}$ from a given input $\mathbf{x} \in \mathcal{X} \subseteq \mathbb{R}^d$, where the pairs $(\mathbf{x}, y)$ are drawn from some unknown distribution $\mathcal{D}$.

While $\mathcal{D}$ is unknown, we assume we have access to a finite training set of n i.i.d. data pairs denoted by $S = \{(\mathbf{x}_i, y_i)\}_{i=1}^n$.

The model h is parameterized by a weight vector $\mathbf{w} \in \mathbb{R}^p$ (we will write $h_\mathbf{w}$ to highlight this dependence when needed) and it should ideally be learned by minimizing the following population risk:

$$\mathcal{R}(\mathbf{w}) = \mathbb{E}_{(\mathbf{x},y)\sim\mathcal{D}}\, \ell(h_\mathbf{w}(\mathbf{x}), y)\,, \tag{13.1}$$

for some loss function $\ell : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}$. Some example of model $h_\mathbf{w}$ could be a linear model ($h_\mathbf{w}(\mathbf{x}) = \mathbf{w}^\top \mathbf{x}$) or other feature-based models, e.g. $h_\mathbf{w}(\mathbf{x}) = \mathbf{w}^\top \phi(\mathbf{x})$, where $\phi(\mathbf{x})$ are features extracted from a neural network. We denote by $\mathcal{R}^*$ the minimum value of $\mathcal{R}(\mathbf{w})$.

The traditional view in machine learning is that larger models tend to overfit to the training data, yielding poor generalization to unseen data. This is illustrated in Figure 13.1 (a) on the left side. However, it turns out this is an incomplete picture as shown by Belkin et al. (2019). As one increases the size of a deep learning model [1] (entering the so-called over-parametrized regime where the number of parameters exceeds the number of training datapoints), one observes that the test error starts decreasing, even though the training error goes to zero. We see there is a peak at the interpolation threshold when the model capacity becomes equal to the number of training datapoints. We will explain this phenomenon in this lecture.

[1] We will talk about the assumption on the model in slightly mode details in Section 13.3

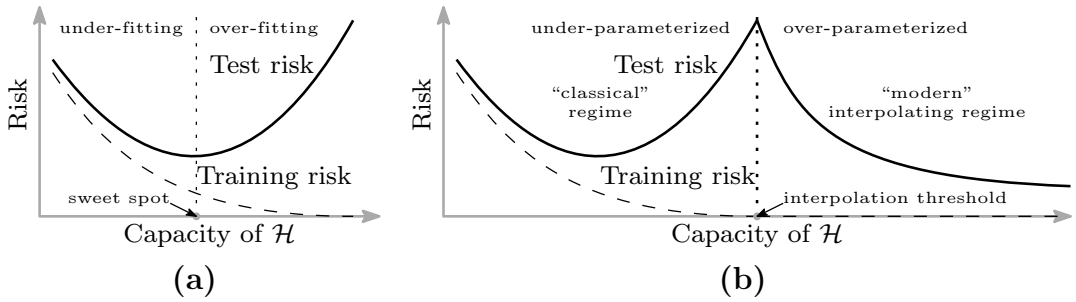


Figure 13.1: **Curves for training risk (dashed line) and test risk (solid line). Figure from Belkin et al. (2019)** (**a**) The classical *U-shaped risk curve* arising from the bias-variance trade-off. (**b**) The *double descent risk curve*, which incorporates the U-shaped risk curve (i.e., the "classical" regime) together with the observed behavior from using high capacity function classes (i.e., the "modern" interpolating regime), separated by the interpolation threshold. The predictors to the right of the interpolation threshold have zero training risk.

It turns out that the double descent phenomenon in the case of a square loss (regression setting) can be explained from the point of view of the bias-variance decomposition, which we review in the next section.

13.2 Bias-variance Decomposition

We briefly recall the standard bias-variance decomposition in the case of the square loss. We work in the context of regression [2] where we assume that there exists $\mathbf{w}_* \in \mathbb{R}^p$ such that the true labels are given by $y = h_{\mathbf{w}_*}(\mathbf{x}) + \epsilon \in \mathbb{R}$ (planted model assumption) where ϵ is some zero-mean noise independent of $\mathbf{x}$ and with variance $\mathbb{E}[\epsilon^2] = \sigma^2$ [3].

Linear model For linear models, the function $h(\mathbf{x})$ takes the form $h_{\mathbf{w}}(\mathbf{x}) = \mathbf{w}^\top \phi(\mathbf{x})$.

[2] A similar analysis can be performed for classification, where each class $k \in \{1 \cdots K\}$ can be represented by a one-hot vector in $\mathbb{R}^K$ and the predictor outputs a score or probability vector in $\mathbb{R}^K$.

[3] Notation: recall $(\mathbf{x}_i, y_i)$ denote a pair from the training set, while $(\mathbf{x}, y)$ is a test pair.

Theorem 80 (Risk decomposition (see Proposition 3.9 by Bach (2021))). *Under the planted model assumption, for any $\mathbf{w} \in \mathbb{R}^p$, we have $\mathcal{R}^* = \sigma^2$ and*

$$\mathcal{R}(\mathbf{w}) - \mathcal{R}^* = \|\mathbf{w} - \mathbf{w}_*\|_{\mathbf{\Sigma}}^2,$$

where $\mathbf{\Sigma} = \mathbb{E}_{\mathbf{x}}[\phi(\mathbf{x})\phi(\mathbf{x})^\top]$ is the (non-centered) covariance matrix and $\|\mathbf{w}\|_{\mathbf{\Sigma}}^2 = \mathbf{w}^\top \mathbf{\Sigma} \mathbf{w}$.

If $\bar{\mathbf{w}}$ is a random variable (e.g. an estimator of $\mathbf{w}_*$), then we have the following decomposition:

$$\mathbb{E}[\mathcal{R}(\bar{\mathbf{w}}) - \mathcal{R}^*] = \underbrace{\|\mathbb{E}[\bar{\mathbf{w}}] - \mathbf{w}_*\|_{\mathbf{\Sigma}}^2}_{\textit{Bias}} + \underbrace{\mathbb{E}\|\bar{\mathbf{w}} - \mathbb{E}[\bar{\mathbf{w}}]\|_{\mathbf{\Sigma}}^2}_{\textit{Variance}}.$$

Proof. For the first part of the statement, we simply use the definition of $\mathcal{R}(\mathbf{w})$:

$$\begin{aligned}
\mathcal{R}(\mathbf{w}) &= \mathbb{E}_{(\mathbf{x},y)\sim\mathcal{D}}\, \ell(h_{\mathbf{w}}(\mathbf{x}), y) \\
&= \mathbb{E}_{(\mathbf{x},y)\sim\mathcal{D}}[(\mathbf{w}^\top \phi(\mathbf{x}) - y)^2] \\
&= \mathbb{E}_{(\mathbf{x},\epsilon)\sim\mathcal{D}}[(\mathbf{w}^\top \phi(\mathbf{x}) - \mathbf{w}_*^\top \phi(\mathbf{x}) - \epsilon)^2] \\
&\overset{(i)}{=} \mathbb{E}_{\mathbf{x}}[(\mathbf{w}^\top \phi(\mathbf{x}) - \mathbf{w}_*^\top \phi(\mathbf{x}))^2] + \mathbb{E}_{\epsilon}[\epsilon^2] \\
&= (\mathbf{w} - \mathbf{w}_*)^\top \mathbf{\Sigma} (\mathbf{w} - \mathbf{w}_*) + \sigma^2,
\end{aligned}$$

where we used $\mathbb{E}[\epsilon] = 0$ in (i).

For the second part of the statement, we perform the usual bias-variance decomposition:

$$\begin{aligned}
\mathbb{E}[\mathcal{R}(\bar{\mathbf{w}}) - \mathcal{R}^*] &= \mathbb{E}\left[\|\bar{\mathbf{w}} \pm \mathbb{E}[\bar{\mathbf{w}}] - \mathbf{w}_*\|_{\mathbf{\Sigma}}^2\right] \\
&= \mathbb{E}\left[\|\bar{\mathbf{w}} - \mathbb{E}[\bar{\mathbf{w}}]\|_{\mathbf{\Sigma}}^2\right] + 2\mathbb{E}[(\bar{\mathbf{w}} - \mathbb{E}[\bar{\mathbf{w}}])^\top \mathbf{\Sigma} (\mathbb{E}[\bar{\mathbf{w}}] - \mathbf{w}_*)] + \mathbb{E}\left[\|\mathbb{E}[\bar{\mathbf{w}}] - \mathbf{w}_*\|_{\mathbf{\Sigma}}^2\right] \\
&= \mathbb{E}\left[\|\bar{\mathbf{w}} - \mathbb{E}[\bar{\mathbf{w}}]\|_{\mathbf{\Sigma}}^2\right] + 0 + \|\mathbb{E}[\bar{\mathbf{w}}] - \mathbf{w}_*\|_{\mathbf{\Sigma}}^2.
\end{aligned}$$

□

Since we typically do not have access to the distribution $\mathcal{D}$, we find an estimate of $\mathbf{w}$ by minimizing the empirical risk

$$\bar{\mathcal{R}}(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n} \ell(h_{\mathbf{w}}(\mathbf{x}_i), y_i) = \frac{1}{n}\|\mathbf{y} - \boldsymbol{\phi}\mathbf{w}\|_2^2, \tag{13.2}$$

where $\mathbf{y} = (y_1, \dots y_n)$ and $\boldsymbol{\phi} = \begin{pmatrix} \vdots & & \vdots \\ \phi(\mathbf{x}_1) & \dots & \phi(\mathbf{x}_n) \\ \vdots & & \vdots \end{pmatrix} \in \mathbb{R}^{n\times p}$.

The optimum $\bar{\mathbf{w}}$ has a closed-form expression given by $\bar{\mathbf{w}} = \frac{1}{n}\bar{\mathbf{\Sigma}}^{-1}\boldsymbol{\phi}^\top \mathbf{y}$ where $\bar{\mathbf{\Sigma}} = \frac{1}{n}\boldsymbol{\phi}^\top\boldsymbol{\phi} \in \mathbb{R}^{p\times p}$ (this is the so-called OLS estimator).

Theorem 81. *The variance of the expected risk of the OLS estimator* $\bar{\mathbf{w}} = \frac{1}{n}\bar{\mathbf{\Sigma}}^{-1}\boldsymbol{\phi}^\top \mathbf{y}$ *is equal to*

$$\frac{\sigma^2}{n}\mathbb{E}[\mathrm{tr}(\mathbf{\Sigma}\bar{\mathbf{\Sigma}}^{-1})].$$

Proof. First note that we can write the OLS estimator as $\bar{\mathbf{w}} = \frac{1}{n}\bar{\mathbf{\Sigma}}^{-1}\boldsymbol{\phi}^\top\mathbf{y} = \frac{1}{n}\bar{\mathbf{\Sigma}}^{-1}\boldsymbol{\phi}^\top(\boldsymbol{\phi}\mathbf{w}_* + \boldsymbol{\epsilon}) = \mathbf{w}_* + \frac{1}{n}\bar{\mathbf{\Sigma}}^{-1}\boldsymbol{\phi}^\top\boldsymbol{\epsilon}$.

Therefore

$$\begin{aligned}
\mathbb{E}\left[\|\mathbb{E}[\bar{\mathbf{w}}] - \mathbf{w}_*\|_{\mathbf{\Sigma}}^2\right] &= \mathbb{E}\left[\left(\frac{1}{n}\bar{\mathbf{\Sigma}}^{-1}\boldsymbol{\phi}^\top\boldsymbol{\epsilon}\right)^\top \mathbf{\Sigma}\left(\frac{1}{n}\bar{\mathbf{\Sigma}}^{-1}\boldsymbol{\phi}^\top\boldsymbol{\epsilon}\right)\right] \\
&= \mathbb{E}\,\mathrm{tr}\left[\mathbf{\Sigma}\left(\frac{1}{n}\bar{\mathbf{\Sigma}}^{-1}\boldsymbol{\phi}^\top\boldsymbol{\epsilon}\right)\left(\frac{1}{n}\bar{\mathbf{\Sigma}}^{-1}\boldsymbol{\phi}^\top\boldsymbol{\epsilon}\right)^\top\right] \\
&= \frac{1}{n^2}\mathbb{E}\,\mathrm{tr}\left[\mathbf{\Sigma}\bar{\mathbf{\Sigma}}^{-1}\boldsymbol{\phi}^\top\boldsymbol{\epsilon}\boldsymbol{\epsilon}^\top\boldsymbol{\phi}\bar{\mathbf{\Sigma}}^{-1}\right] \\
&\overset{(i)}{=} \frac{\sigma^2}{n^2}\mathbb{E}\,\mathrm{tr}\left[\mathbf{\Sigma}\bar{\mathbf{\Sigma}}^{-1}\boldsymbol{\phi}^\top\boldsymbol{\phi}\bar{\mathbf{\Sigma}}^{-1}\right] \\
&= \frac{\sigma^2}{n}\mathbb{E}\,\mathrm{tr}\left[\mathbf{\Sigma}\bar{\mathbf{\Sigma}}^{-1}\right],
\end{aligned}$$

where we used $\mathbb{E}[\boldsymbol{\epsilon}\boldsymbol{\epsilon}^\top] = \sigma^2\mathbf{I}$ in (i).

□

In Figure 13.2 (taken from Dar et al. (2021)), we plot the bias and variance as we increase the number of parameters of the model. We can see that a peak appears in the variance at the interpolation threshold. What is causing this peak? By inspecting the expression of the variance derived in Theorem 81, we see that the inverse of the covariance matrix appears. It turns out that at the interpolation threshold, this matrix is badly conditioned, which makes the inverse to be ill-defined.

GD vs OLS estimator Note that we have here assumed that we compute the exact OLS closed-form solution but in practice, one often relies on gradient descent to optimize the empirical risk. However, one can show that gradient descent approximates the closed-form solution (Yao et al., 2007) and we therefore obtain the same result if we perform a large number of iterations of gradient descent.

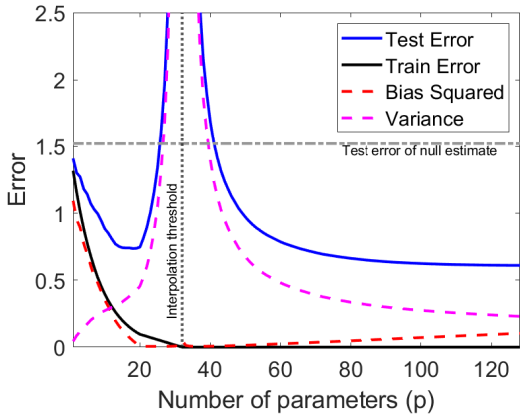


Figure 13.2: Illustration of bias-variance trade-off for a problem where $d = 128$ and $n = 32$, and where we vary the number of parameters p to be between 1 and d. The interpolation threshold is located at $p = n$. Source: Dar et al. (2021).

Kernel ridge regression One can generalize the analysis of ridge regression to kernel ridge regression for *finite-dimensional* feature spaces (common infinite dimensional kernels are not directly covered and require a more involved analysis).

Consider a positive-definite real-valued kernel $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ on a non-empty set $\mathcal{X}$ defined as

$$k(\mathbf{x}, \mathbf{x}') = \langle \phi(\mathbf{x}), \phi(\mathbf{x}') \rangle$$

The kernel k as an RKHS (Reproducing Kernel Hilbert Space) $\mathcal{H}_k$ associated with it, which is a Hilbert space of functions $f : \mathcal{X} \to \mathbb{R}$ that satisfies some properties. We refer the reader to Bach (2021) for further details.

Given a set of n i.i.d. observations $(x_i, y_i) \in \mathcal{X} \times \mathbb{R}$, $i = 1, \ldots, n$, we now aim to minimize, for $\lambda > 0$,

$$\mathcal{R}(\mathbf{w}) = \frac{1}{n} \sum_{i=1} (y_i - f_{\mathbf{w}}(x_i))^2 + \lambda \|f_{\mathbf{w}}\|_{\mathcal{H}}^2.$$

Here, $\|f\|_{\mathcal{H}}$ denotes the norm in a Hilbert space $\mathcal{H}$, which is defined for a function $f \in \mathcal{H}$ as:

$$\|f\|_{\mathcal{H}} = \sqrt{\langle f, f \rangle_{\mathcal{H}}},$$

where $\langle \cdot, \cdot \rangle_{\mathcal{H}} = \langle f, g \rangle = \int_{-\infty}^{\infty} f(\mathbf{x}) g(\mathbf{x}) \, \mathrm{d}\mathbf{x}$ denotes the inner product in the Hilbert space $\mathcal{H}$.

By the Representer theorem (see NTK Chapter), any minimizer of the empirical risk is

$$f_{\mathbf{w}}(\mathbf{x}) = \sum_{i=1}^{n} w_i k(\mathbf{x}, \mathbf{x}_i),$$

with

$$\mathbf{w} = (\mathbf{K} + n\lambda\mathbf{I})^{-1}\mathbf{y},$$

where $\mathbf{K} \in \mathbb{R}^{n\times n}$ is the kernel matrix.

One can then adapt the analysis of ridge regression by noting that f is simply a function of the form $f_{\mathbf{w}}(\mathbf{x}) = \mathbf{w}^\top k(\mathbf{x}, \cdot)$ which has a similar form to the ridge regression case.

13.3 Brief Discussion about Different Regimes of Overfitting

Certain interpolating methods, including some types of neural networks, are able to fit (noisy) training data without catastrophically bad test performance, going against the more classical belief that overparametrized models overfit to the training data. A recent work by Mallinar et al. (2022) shows that the picture is in fact more complicated as there exist different regimes of overfitting that are illustrated in Figure 13.3 (taken from Mallinar et al. (2022)).

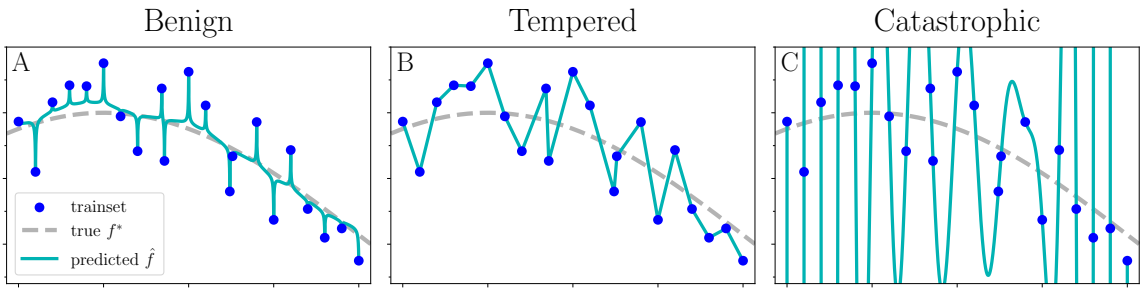


Figure 13.3: **As $n \to \infty$, interpolating methods can exhibit three types of overfitting. (A)** In *benign overfitting*, the predictor asymptotically approaches the ground-truth, Bayes-optimal function. Nadaraya-Watson kernel smoothing with a singular kernel, shown here, is asymptotically benign. **(B)** In *tempered overfitting*, the predictor approaches a constant test risk greater than the Bayes-optimal risk. Piecewise-linear interpolation is asymptotically tempered. **(C)** In *catastrophic overfitting*, the predictor generalizes arbitrarily poorly. Rank-n polynomial interpolation is asymptotically catastrophic.

Bibliography

Zeyuan Allen-Zhu, Yuanzhi Li, and Zhao Song. A convergence theory for deep learning via over-parameterization. *arXiv preprint arXiv:1811.03962*, 2018.

Raman Arora, Sanjeev Arora, Joan Bruna, Nadav Cohen, Simon Du, Rong Ge, Suriya Gunasekar, C Jin, Jason Lee, Tengyu Ma, et al. Theory of deep learning, 2020.

Sanjeev Arora, Rong Ge, Behnam Neyshabur, and Yi Zhang. Stronger generalization bounds for deep nets via a compression approach. In *International Conference on Machine Learning*, pages 254–263. PMLR, 2018.

Francis Bach. Breaking the curse of dimensionality with convex neural networks. *The Journal of Machine Learning Research*, 18(1):629–681, 2017.

Francis Bach. Learning theory from first principles. *Draft of a book, version of Sept*, 6:2021, 2021.

Pierre Baldi and Kurt Hornik. Neural networks and principal component analysis: Learning from examples without local minima. *Neural networks*, 2(1):53–58, 1989.

Andrew R Barron. Universal approximation bounds for superpositions of a sigmoidal function. *IEEE Transactions on Information theory*, 39(3):930–945, 1993.

Peter L Bartlett, Nick Harvey, Christopher Liaw, and Abbas Mehrabian. Nearly-tight vc-dimension and pseudodimension bounds for piecewise linear neural networks. *The Journal of Machine Learning Research*, 20(1):2285–2301, 2019.

Luc Bégin, Pascal Germain, Fran**c**cois Laviolette, and Jean-Francis Roy. Pac-bayesian theory for transductive learning. In *Artificial Intelligence and Statistics*, pages 105–113. PMLR, 2014.

Mikhail Belkin, Daniel Hsu, Siyuan Ma, and Soumik Mandal. Reconciling modern machine-learning practice and the classical bias–variance trade-off. *Proceedings of the National Academy of Sciences*, 116(32):15849–15854, 2019.

Stéphane Boucheron, Gábor Lugosi, and Pascal Massart. *Concentration inequalities: A nonasymptotic theory of independence*. Oxford university press, 2013.

Sébastien Bubeck, Yeshwanth Cherapanamjeri, Gauthier Gidel, and Remi Tachet des Combes. A single gradient step finds adversarial examples on random two-layers neural networks. *Advances in Neural Information Processing Systems*, 34:10081–10091, 2021.

Wojciech Marian Czarnecki, Simon Osindero, Razvan Pascanu, and Max Jaderberg. A deep neural network's loss surface contains every low-dimensional pattern. *arXiv preprint arXiv:1912.07559*, 2019.

Yehuda Dar, Vidya Muthukumar, and Richard G Baraniuk. A farewell to the bias-variance tradeoff? an overview of the theory of overparameterized machine learning. *arXiv preprint arXiv:2109.02355*, 2021.

Marc Peter Deisenroth, A Aldo Faisal, and Cheng Soon Ong. *Mathematics for machine learning*. Cambridge University Press, 2020.

Monroe D Donsker and SR Srinivasa Varadhan. Asymptotics for the wiener sausage. *Communications on Pure and Applied Mathematics*, 28(4):525–565, 1975.

Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

Simon S Du, Xiyu Zhai, Barnabas Poczos, and Aarti Singh. Gradient descent provably optimizes over-parameterized neural networks. *arXiv preprint arXiv:1810.02054*, 2018.

Gintare Karolina Dziugaite and Daniel M Roy. Computing nonvacuous generalization bounds for deep (stochastic) neural networks with many more parameters than training data. *arXiv preprint arXiv:1703.11008*, 2017.

Ronen Eldan and Ohad Shamir. The power of depth for feedforward neural networks. In *Conference on learning theory*, pages 907–940. PMLR, 2016.

Rong Ge, Furong Huang, Chi Jin, and Yang Yuan. Escaping from saddle points—online stochastic gradient for tensor decomposition. In *Conference on learning theory*, pages 797–842. PMLR, 2015.

Saeed Ghadimi and Guanghui Lan. Stochastic first-and zeroth-order methods for nonconvex stochastic programming. *SIAM Journal on Optimization*, 23(4):2341–2368, 2013.

Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256. JMLR Workshop and Conference Proceedings, 2010.

Gene H Golub and Charles F Van Loan. *Matrix computations*. JHU press, 2013.

Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.

Suriya Gunasekar, Jason Lee, Daniel Soudry, and Nathan Srebro. Characterizing implicit bias in terms of optimization geometry. In *International Conference on Machine Learning*, pages 1832–1841. PMLR, 2018.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

Geoffrey E Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*, 2012.

Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

Gao Huang, Zhuang Liu, and Kilian Q. Weinberger. Densely connected convolutional networks. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2261–2269, 2017.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. pmlr, 2015.

Ziwei Ji and Matus Telgarsky. Gradient descent aligns the layers of deep linear networks. *arXiv preprint arXiv:1810.02032*, 2018.

Ziwei Ji and Matus Telgarsky. The implicit bias of gradient descent on nonseparable data. In *Conference on Learning Theory*, pages 1772–1798. PMLR, 2019.

Ziwei Ji and Matus Telgarsky. Directional convergence and alignment in deep learning. *Advances in Neural Information Processing Systems*, 33:17176–17186, 2020.

Kenji Kawaguchi. Deep learning without poor local minima. In *Advances in Neural Information Processing Systems*, pages 586–594, 2016.

Jaehoon Lee, Lechao Xiao, Samuel Schoenholz, Yasaman Bahri, Roman Novak, Jascha Sohl-Dickstein, and Jeffrey Pennington. Wide neural networks of any depth evolve as linear models under gradient descent. *Advances in neural information processing systems*, 32:8572–8583, 2019.

Jaehoon Lee, Samuel S. Schoenholz, Jeffrey Pennington, Ben Adlam, Lechao Xiao, Roman Novak, and Jascha Sohl-Dickstein. Finite versus infinite neural networks: an empirical study. *34th Conference on Neural Information Processing Systems (NeurIPS)*, 2020.

Moshe Leshno, Vladimir Ya Lin, Allan Pinkus, and Shimon Schocken. Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. *Neural networks*, 6(6):861–867, 1993.

Kaifeng Lyu and Jian Li. Gradient descent maximizes the margin of homogeneous neural networks. *arXiv preprint arXiv:1906.05890*, 2019.

Neil Mallinar, James B Simon, Amirhesam Abedsoltan, Parthe Pandit, Mikhail Belkin, and Preetum Nakkiran. Benign, tempered, or catastrophic: A taxonomy of overfitting. *arXiv preprint arXiv:2207.06569*, 2022.

David McAllester. Simplified pac-bayesian margin bounds. In *Learning theory and Kernel machines*, pages 203–215. Springer, 2003.

James R Munkres. Topology, 2000.

Behnam Neyshabur, Ryota Tomioka, and Nathan Srebro. In search of the real inductive bias: On the role of implicit regularization in deep learning. *arXiv preprint arXiv:1412.6614*, 2014.

Behnam Neyshabur, Ryota Tomioka, and Nathan Srebro. Norm-based capacity control in neural networks. In *Conference on Learning Theory*, pages 1376–1401. PMLR, 2015.

Mihai Nica. Notes on random feature regression and wide neural networks. 2021.

Mary Phuong and Christoph H Lampert. The inductive bias of relu networks on orthogonally separable data. In *International Conference on Learning Representations*, 2020.

Allan Pinkus. Approximation theory of the mlp model in neural networks. *Acta numerica*, 8:143–195, 1999.

Gilles Pisier. Remarques sur un résultat non publié de b. maurey. *Séminaire Analyse fonctionnelle (dit*, pages 1–12, 1981.

Carl Edward Rasmussen. Gaussian processes in machine learning. In *Summer school on machine learning*, pages 63–71. Springer, 2003.

Sashank J Reddi, Ahmed Hefny, Suvrit Sra, Barnabas Poczos, and Alex Smola. Stochastic variance reduction for nonconvex optimization. In *International conference on machine learning*, pages 314–323. PMLR, 2016.

Walter Rudin et al. *Principles of mathematical analysis*, volume 3. McGraw-hill New York, 1976.

Itay Safran and Ohad Shamir. Spurious local minima are common in two-layer relu neural networks. In *International Conference on Machine Learning*, pages 4433–4441. PMLR, 2018.

Daniel Soudry, Elad Hoffer, Mor Shpigel Nacson, Suriya Gunasekar, and Nathan Srebro. The implicit bias of gradient descent on separable data. *The Journal of Machine Learning Research*, 19(1):2822–2878, 2018.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013.

Matus Telgarsky. Benefits of depth in neural networks. In *Conference on learning theory*, pages 1517–1539. PMLR, 2016.

Matus Telgarsky. Deep learning theory lecture notes. `https://mjt.cs.illinois.edu/dlt/`, 2021. Version: 2021-10-27 v0.0-e7150f2d (alpha).

Gal Vardi, Gilad Yehudai, and Ohad Shamir. Gradient methods provably converge to non-robust networks. *arXiv preprint arXiv:2202.04347*, 2022.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

BA Vostrecov and MA Kreines. Approximation of continuous functions by superpositions of plane waves. In *Dokl. Akad. Nauk SSSR*, volume 140, pages 1237–1240, 1961.

Martin J Wainwright. *High-dimensional statistics: A non-asymptotic viewpoint*, volume 48. Cambridge University Press, 2019.

E Weinan, Chao Ma, and Lei Wu. Barron spaces and the compositional function spaces for neural network models. *arXiv preprint arXiv:1906.08039*, 2019.

Yuan Yao, Lorenzo Rosasco, and Andrea Caponnetto. On early stopping in gradient descent learning. *Constructive Approximation*, 26(2):289–315, 2007.

Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. Understanding deep learning (still) requires rethinking generalization. *Communications of the ACM*, 64(3):107–115, 2021.