A Deep Learning Launchpad | 31

by **Mairi Hallman** and **Patrick Boily**, with contributions from **Kevin Cheung**, **Andrew Macfie**, **Smit Patel**, and **Razieh Pourhasan**

31.1 Brief Overview of Tensors 2030
Tensor Products 2031
Tensor Decomposition . . 2035
Python Examples 2036
31.2 Deep Networks 2042
Getting Started 2044
Activation Functions . . . 2046
Weight Initialization . . . 2048
31.3 Regularization 2048
Weight Decay 2048
Early Stopping 2049
Dropout 2049
31.4 Stochastic Descent 2050
Momentum 2050
Nesterov Momentum . . . 2051
AdaGrad 2051
RMSprop 2052
AdaDelta 2052
Adam 2053
Yogi 2053
31.5 CNN 2054
What is Convolution? . . . 2054
How it is Used 2055
Image Class Example . . . 2058
31.6 RNN 2064
Vector-to-Sequence 2065
Sequence-to-Vector 2065
Sequence-to-Sequence . . 2065
Encode-Decode Models . 2066
Bi-Directional RNN 2066
Long-Term Memory 2067
Music Generation 2068
31.7 Specialized Architectures 2076
GAN 2077
Autoencoders 2079
Transformers 2082
31.8 Additional Examples . . . 2083
Learning XOR 2084
Iris Dataset 2086
Boston Dataset 2089
MNIST Dataset 2092
Sunspot Dataset 2100
31.9 Exercises 2105
Chapter References 2107

Recently **artificial intelligence** (AI), **machine learning** (ML), and **deep learning** (DL) have been widely discussed in the media, often featured in articles that promise a future of **intelligent agents** capable of mimicking human cognition by communicating, making decisions, and solving problems based on experience and evidence, much like humans do.

As practitioners in the field, it is essential to understand where deep learning stands in relation to machine learning, how both relate to the broader domain of AI, and why deep learning is currently receiving so much attention. Equally important is the ability to distinguish truly transformative developments from overhyped media claims.

AI, a field that took shape in the 1950s, is an umbrella term encompassing both ML and DL, as well as other approaches that do not involve any form of learning.* It can more broadly be defined as **the effort to automate intellectual tasks normally performed by humans**. We will have more to say on this in Chapter 34; interested readers can consult [20, 21, 26, 34, 44].

Many of these tasks, such as text comprehension, image recognition, and speech processing, are not easily defined by clear, logical rules. They are inherently **complex** and **fuzzy**, making them unsuitable for symbolic approaches and motivating the ML paradigm.

ML emerged from the question: *can a computer learn to perform a specific task just by analyzing data?* This question represents a shift in how we approach programming.†

In essence, ML systems are **trained**, not explicitly programmed. By analyzing many examples related to a task, they discover statistical patterns that enable them to automate the task. An ML **algorithm** learns to generate rules from data, based on **observed input-output pairs**.

Three elements are necessary for training an ML model: **input data**, **examples of the expected output**, and **a way to measure performance**. The model transforms its input data into meaningful output by learning useful **representations**‡ that help it approximate the desired output. These representations are **task-specific**.§

---

* For example, early chess programs relied on hard-coded rules devised by human programmers, an approach known as **symbolic AI**. Symbolic AI, which dominated the field from the 1950s to the late 1980s, is based on explicitly defined rules and logic; it does not qualify as ML.

† In classical programming, developers provide explicit rules and input data to generate output. In contrast, ML involves providing both the data and the expected output, allowing the system to **generate its own rules**.

‡ That is, alternative ways of encoding or viewing the data.

§ What works for one problem may not be useful for another.

The term “learning” in ML refers to the automated search for meaningful data transformations that produce useful representations. However, this search is not unlimited; algorithms typically explore a predefined set of transformations (the **hypothesis space**), which may include operations such as projections, translations, and nonlinear mappings. ML is thus a process of **guided discovery within constrained possibilities**.

DL is a specialized subfield of ML that focuses on learning **multiple layers** of increasingly abstract representations; the term “deep” refers to the **depth** of the involved **neural networks** (NN).[1]

While early DL techniques emphasized **unsupervised learning** and layer-wise training, such as in **autoencoders**, modern DL is largely driven by training deep NN models using **backpropagation**. Common DL architectures include **multilayer perceptrons**, **recurrent neural networks** (RNN), and **convolutional neural networks** (CNN).[2]

To understand these models, one must first grasp the core components of NN: **tensors**, **layers**, **networks**, **loss functions**, and **optimizers**.[3]

1: In a 2016 talk, Jeff Dean (Google AI) summarized the usage of the term as follows: “When you hear the term deep learning, just think of a large deep neural net. Deep refers to the number of layers typically and so this is kind of the popular term that’s been adopted in the press. I think of them as deep neural networks generally.” [17]

2: References for the material/examples found in this chapter include [9, 11, 19, 48].

3: Some of this content is also found in [6, sec. 21.4.3].

31.1 A Brief Overview of Tensors

To begin understanding how neural networks process and transform data, it is essential to first define the foundational data structures on which they operate.

Scalars, **vectors**, and **matrices** can be interpreted as data organized in zero, one, and two dimensions, respectively. These structures can be generalized to **order** N and are collectively referred to as **tensors**:[4]

- a **scalar** is a **zero-order** tensor;
- a **vector** is a **first-order** tensor;
- a **matrix** is a **second-order** tensor;
- a **third-order** tensor (or **tensor cube**) can be visualized as a **stack of matrices**;
- a **fourth-order** tensor is a **vector of third-order tensors**;
- a **fifth-order** tensor is a **matrix of third-order tensors**, etc.

4: These are related but not identical to the objects used in tensor analysis and physics (notably used in general relativity and quantum field theory).

This hierarchical structure continues **recursively**, with each additional order introducing a new **axis**.[5] A symbolic visualization of some of these structures is shown in Figure 31.1.

5: A tensor of order N has N axes.

The **shape** of a tensor is a tuple of integers that describes how many **dimensions** the tensor has along each axis – for instance, a matrix’s shape is described using **two** elements, such as $(3, 5)$,[6] a tensor cube’s shape has **three** elements, such as $(3, 5, 5)$,[7] a vector (1D tensor)’s shape is given by a **single** element, such as (5),[8] and a scalar has an **empty** shape, $( )$. Tensors with the same shape can be added in the natural manner.[9]

6: For an array with 3 rows and 5 columns

7: Which we can view as a 5-deep stack of 3×5 matrices.

8: For a vector with 5 components.

9: The **order** of a tensor is thus simply the number of elements in its shape vector.

Understanding tensors and their dimensional hierarchy is fundamental to the implementation and interpretation of modern DL systems.

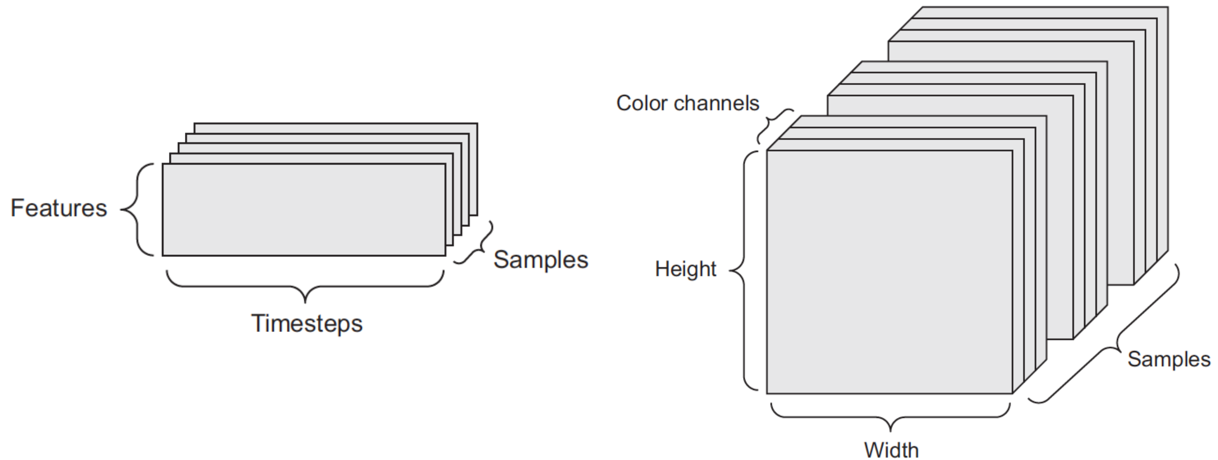


Figure 31.1: A 3D time series data tensor (left) and a 4D image data tensor (right) [9].

31.1.1 Tensor and Matrix Products

We present briefly some of the tensor (and matrix) operations required to make sense of the **tensor decompositions** of Section 31.1.2.

Outer Product ∘

The **outer product** is a key operation in linear algebra and tensor analysis. It enables the construction of **higher-order** tensors from **lower-order** tensors (such as vectors).

Let $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_N$, where $\mathbf{v}_i \in \mathbb{R}^{d_i}$. Their **outer product** is a tensor $\mathbf{T} = \mathbf{v}_1 \circ \mathbf{v}_2 \circ \cdots \circ \mathbf{v}_N \in \mathbb{R}^{d_1 \times d_2 \times \cdots \times d_N}$ of order N, defined by:

$$\mathbf{T}_{i_1 i_2 \ldots i_N} = (\mathbf{v}_1)_{i_1} \cdot (\mathbf{v}_2)_{i_2} \cdot \cdots \cdot (\mathbf{v}_N)_{i_N}.$$

Each element in the resulting tensor is the product of the corresponding entries from each vector.

Examples

1. Let:

$$\mathbf{a} = \begin{bmatrix} 1 \\ 2 \end{bmatrix} \in \mathbb{R}^2, \quad \mathbf{b} = \begin{bmatrix} 3 \\ 4 \\ 5 \end{bmatrix} \in \mathbb{R}^3;$$

Then the outer product $\mathbf{a} \circ \mathbf{b}$ results in a 2×3 matrix:

$$\mathbf{a} \circ \mathbf{b} = \begin{bmatrix} 1 \cdot 3 & 1 \cdot 4 & 1 \cdot 5 \\ 2 \cdot 3 & 2 \cdot 4 & 2 \cdot 5 \end{bmatrix} = \begin{bmatrix} 3 & 4 & 5 \\ 6 & 8 & 10 \end{bmatrix} \in \mathbb{R}^{2 \times 3}.$$

2. Let $\mathbf{a} \in \mathbb{R}^2$, $\mathbf{b} \in \mathbb{R}^3$, $\mathbf{c} \in \mathbb{R}^2$. The outer product $\mathbf{T} = \mathbf{a} \circ \mathbf{b} \circ \mathbf{c} \in \mathbb{R}^{2 \times 3 \times 2}$ is a third-order tensor of shape $2 \times 3 \times 2$, with elements defined by:

$$\mathbf{T}_{ijk} = \mathbf{a}_i \cdot \mathbf{b}_j \cdot \mathbf{c}_k. \qquad \square$$

Kronecker Product ⊗

10: When we view matrices as second-order tensors, we will use this bold-face notation instead of the typical A and B.

Consider two matrices $\mathbf{A}$, $\mathbf{B}$, of shapes (m, n), (p, q), respectively.[10] The **Kronecker product** $\mathbf{A} \otimes \mathbf{B}$ is a matrix of shape (mp, nq) defined by:

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} a_{11}\mathbf{B} & a_{12}\mathbf{B} & \dots & a_{1n}\mathbf{B} \\ a_{21}\mathbf{B} & a_{22}\mathbf{B} & \dots & a_{2n}\mathbf{B} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}\mathbf{B} & a_{m2}\mathbf{B} & \dots & a_{mn}\mathbf{B} \end{bmatrix} \in \mathbb{R}^{mp \times nq}.$$

Assuming that the dimensions are compatible, the Kronecker product satisfies the following properties:

- $(\mathbf{A} \otimes \mathbf{B})(\mathbf{C} \otimes \mathbf{D}) = (\mathbf{AC}) \otimes (\mathbf{BD})$;
- $(\mathbf{A} \otimes \mathbf{B})^\top = \mathbf{A}^\top \otimes \mathbf{B}^\top$.

The Kronecker product is **not necessarily commutative**, however.

Example: Let $\mathbf{A} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ and $\mathbf{B} = \begin{bmatrix} 0 & 5 \\ 6 & 7 \end{bmatrix}$. We have

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} 1 \cdot \mathbf{B} & 2 \cdot \mathbf{B} \\ 3 \cdot \mathbf{B} & 4 \cdot \mathbf{B} \end{bmatrix} = \begin{bmatrix} 0 & 5 & 0 & 10 \\ 6 & 7 & 12 & 14 \\ 0 & 15 & 0 & 20 \\ 18 & 21 & 24 & 28 \end{bmatrix}, \quad \text{but}$$

$$\mathbf{B} \otimes \mathbf{A} = \begin{bmatrix} 0 \cdot \mathbf{A} & 5 \cdot \mathbf{A} \\ 6 \cdot \mathbf{A} & 7 \cdot \mathbf{A} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 5 & 10 \\ 0 & 0 & 15 & 20 \\ 6 & 12 & 7 & 14 \\ 18 & 24 & 21 & 28 \end{bmatrix}. \qquad \square$$

Khatri-Rao Product ⊙

Let $\mathbf{A} \in \mathbb{R}^{m \times n}$ and $\mathbf{B} \in \mathbb{R}^{p \times n}$ be two matrices with the same number of columns. The **Khatri-Rao product** $\mathbf{A} \odot \mathbf{B}$ is a matrix of shape (mp, n), whose columns are given by column-wise Kronecker products:

$$\mathbf{A} \odot \mathbf{B} = \begin{bmatrix} \mathbf{A}_{:,1} \otimes \mathbf{B}_{:,1} & \mathbf{A}_{:,2} \otimes \mathbf{B}_{:,2} & \cdots & \mathbf{A}_{:,k} \otimes \mathbf{B}_{:,k} \end{bmatrix} \in \mathbb{R}^{(mp) \times n};$$

$\mathbf{A}_{:,i}$ and $\mathbf{B}_{:,i}$ represent the ith columns of $\mathbf{A}$ and $\mathbf{B}$, respectively.

Example: let $\mathbf{A} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ and $\mathbf{B} = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$. Then:

$$\mathbf{A}_{:,1} \otimes \mathbf{B}_{:,1} = \begin{bmatrix} 1 \\ 3 \end{bmatrix} \otimes \begin{bmatrix} 5 \\ 7 \end{bmatrix} = \begin{bmatrix} 1 \cdot 5 \\ 1 \cdot 7 \\ 3 \cdot 5 \\ 3 \cdot 7 \end{bmatrix} = \begin{bmatrix} 5 \\ 7 \\ 15 \\ 21 \end{bmatrix},$$

$$\mathbf{A}_{:,2} \otimes \mathbf{B}_{:,2} = \begin{bmatrix} 2 \\ 4 \end{bmatrix} \otimes \begin{bmatrix} 6 \\ 8 \end{bmatrix} = \begin{bmatrix} 2 \cdot 6 \\ 2 \cdot 8 \\ 4 \cdot 6 \\ 4 \cdot 8 \end{bmatrix} = \begin{bmatrix} 12 \\ 16 \\ 24 \\ 32 \end{bmatrix},$$

and so

$$\mathbf{A} \odot \mathbf{B} = \begin{bmatrix} 5 & 12 \\ 7 & 16 \\ 15 & 24 \\ 21 & 32 \end{bmatrix}.$$

On the other hand,

$$\mathbf{B}_{:,1} \otimes \mathbf{A}_{:,1} = \begin{bmatrix} 5 \\ 7 \end{bmatrix} \otimes \begin{bmatrix} 1 \\ 3 \end{bmatrix} = \begin{bmatrix} 5 \cdot 1 \\ 5 \cdot 3 \\ 7 \cdot 1 \\ 7 \cdot 3 \end{bmatrix} = \begin{bmatrix} 5 \\ 15 \\ 7 \\ 21 \end{bmatrix},$$

$$\mathbf{B}_{:,2} \otimes \mathbf{A}_{:,2} = \begin{bmatrix} 6 \\ 8 \end{bmatrix} \otimes \begin{bmatrix} 2 \\ 4 \end{bmatrix} = \begin{bmatrix} 6 \cdot 2 \\ 6 \cdot 4 \\ 8 \cdot 2 \\ 8 \cdot 4 \end{bmatrix} = \begin{bmatrix} 12 \\ 24 \\ 16 \\ 32 \end{bmatrix},$$

and so

$$\mathbf{B} \odot \mathbf{A} = \begin{bmatrix} 5 & 12 \\ 15 & 24 \\ 7 & 16 \\ 21 & 32 \end{bmatrix}. \qquad \square$$

Assuming that the dimensions are compatible (i.e., $m = p = n$), this example shows that the Khatri-Rao is **not necessarily commutative**.

Hadamard Product $*$

Let $\mathbf{A}, \mathbf{B}$ be two matrices of the same shape (m, n). The **Hadamard product** $\mathbf{A} * \mathbf{B} \in \mathbb{R}^{m \times n}$ is given by the element-wise product:

$$(\mathbf{A} * \mathbf{B})_{ij} = \mathbf{A}_{ij}\mathbf{B}_{ij} \quad \text{for all } i = 1, \ldots, m, \ j = 1, \ldots, n.$$

When the dimensions are compatible, the Hadamard product is:

- **commutative** ($\mathbf{A} * \mathbf{B} = \mathbf{B} * \mathbf{A}$),
- **associative** ($(\mathbf{A} * \mathbf{B}) * \mathbf{C} = \mathbf{A} * (\mathbf{B} * \mathbf{C})$), and
- **distributive over tensor addition** ($\mathbf{A} * (\mathbf{B} + \mathbf{C}) = \mathbf{A} * \mathbf{B} + \mathbf{A} * \mathbf{C}$).

Example: let

$$\mathbf{A} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}.$$

Then

$$\mathbf{A} * \mathbf{B} = \begin{bmatrix} 1 \cdot 5 & 2 \cdot 6 \\ 3 \cdot 7 & 4 \cdot 8 \end{bmatrix} = \begin{bmatrix} 5 & 12 \\ 21 & 32 \end{bmatrix}. \qquad \square$$

Mode-n Tensor-Matrix Product

This operation generalizes matrix multiplication to higher-order tensors. Given a tensor $\mathbf{T} \in \mathbb{R}^{M_1 \times \cdots \times M_n \times \cdots \times M_N}$ and a matrix $\mathbf{A} \in \mathbb{R}^{J \times M_n}$, the **mode-$n$ product** is a new tensor $\mathbf{S} = \mathbf{T} \times_n \mathbf{A} \in \mathbb{R}^{M_1 \times \cdots \times M_{n-1} \times J \times M_{n+1} \times \cdots \times M_N}$.

Intuitively, the matrix $\mathbf{A}$ is multiplied with the **mode-n fibres** (i.e., vectors obtained by fixing all indices of $\mathbf{T}$ except the nth):[11]

$$(\mathbf{T} \times_n \mathbf{A})_{m_1,\dots,m_{n-1},j,m_{n+1},\dots,m_N} = \sum_{i=1}^{M_n} \mathbf{T}_{m_1,\dots,i,\dots,m_N} \mathbf{A}_{j,i}.$$

The operation can also be understood in terms of **matricization: unfolding** the tensor $\mathbf{T}$ along mode-n into a matrix $\mathbf{T}_{(n)} \in \mathbb{R}^{M_n \times (M_1 \cdots M_{n-1} M_{n+1} \cdots M_N)}$, and computing the matrix product:

$$\mathbf{S}_{(n)} = \mathbf{A}\mathbf{T}_{(n)}.$$

The result $\mathbf{S}_{(n)}$ is then **refolded** into a tensor of the appropriate shape.

11: Formally, a **fibre** is a vector obtained by fixing all indices of a tensor except one; it is the tensor equivalent of a matrix **row** or **column**. For a 3rd-order tensor $\mathbf{T} \in \mathbb{R}^{P \times Q \times R}$, say:

- the **mode-1 fibre** $\mathbf{T}_{:,q_0,r_0} \in \mathbb{R}^P$ is obtained by fixing $q = q_0, r = r_0$, and letting p vary;
- the **mode-2 fibre** $\mathbf{T}_{p_0,:,r_0} \in \mathbb{R}^Q$ is obtained by fixing $p = p_0, r = r_0$, and letting q vary, and
- the **mode-3 fibre** $\mathbf{T}_{p_0,q_0,:} \in \mathbb{R}^R$ is obtained by fixing $p = p_0, q = q_0$, and letting r vary.

Analogously, a **slice** is a two-dimensional section of a tensor, obtained by fixing all indices except two. For a 3rd-order tensor $\mathbf{T} \in \mathbb{R}^{P \times Q \times R}$, say:

- the **frontal slice** $\mathbf{T}_{:,:,r_0} \in \mathbb{R}^{P \times Q}$ is obtained by fixing $r = r_0$ and letting p, q vary;
- the **lateral slice** $\mathbf{T}_{:,q_0,:} \in \mathbb{R}^{P \times R}$ is obtained by fixing $q = q_0$ and letting p, r vary, and
- the **horizontal slice** $\mathbf{T}_{p_0,:,:} \in \mathbb{R}^{Q \times R}$ is obtained by fixing $p = p_0$ and letting j, k vary.

Examples: let $\mathbf{T} \in \mathbb{R}^{2 \times 3 \times 2}$, $\mathbf{A} \in \mathbb{R}^{4 \times 2}$, $\mathbf{B} \in \mathbb{R}^{2 \times 3}$ be defined as:

$$\mathbf{T}_{:,:,1} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}, \mathbf{T}_{:,:,2} = \begin{bmatrix} 7 & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix}, \mathbf{A} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \\ 2 & 1 \end{bmatrix}, \mathbf{B} = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 1 & 0 \end{bmatrix}.$$

1. The mode-1 product $\mathbf{S} = \mathbf{T} \times_1 \mathbf{A}$ is a tensor $\mathbf{S} \in \mathbb{R}^{4 \times 3 \times 2}$, where:

$$\mathbf{S}_{j,m_2,m_3} = \sum_{i=1}^{2} \mathbf{A}_{j,i} \mathbf{T}_{i,m_2,m_3}.$$

For instance, we have:

$$\mathbf{S}_{3,2,1} = \mathbf{A}_{3,1} \cdot \mathbf{T}_{1,2,1} + \mathbf{A}_{3,2} \cdot \mathbf{T}_{2,2,1} = 1 \cdot 2 + 1 \cdot 5 = 7.$$

Computing all entries yields:

$$\mathbf{S}_{:,:,1} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 5 & 7 & 9 \\ 6 & 9 & 12 \end{bmatrix}, \quad \mathbf{S}_{:,:,2} = \begin{bmatrix} 7 & 8 & 9 \\ 10 & 11 & 12 \\ 17 & 19 & 21 \\ 24 & 27 & 30 \end{bmatrix}.$$

2. The mode-2 product $\mathbf{R} = \mathbf{T} \times_2 \mathbf{B}$ is a tensor $\mathbf{R} \in \mathbb{R}^{2 \times 2 \times 2}$ which can be obtained by matrix-multiplying $\mathbf{B}$ with the mode-2 fibres of $\mathbf{T}$. For instance, we have:

$$\mathbf{T}_{1,:,1} = [1\ 2\ 3]^\top \implies \mathbf{R}_{1,:,1} = \mathbf{B}\mathbf{T}_{1,:,1} = \begin{bmatrix} 1 \cdot 1 + 0 \cdot 2 + (-1) \cdot 3 \\ 2 \cdot 1 + 1 \cdot 2 + 0 \cdot 3 \end{bmatrix} = \begin{bmatrix} -2 \\ 4 \end{bmatrix};$$

$$\mathbf{T}_{2,:,2} = [10\ 11\ 12]^\top \implies \mathbf{R}_{2,:,2} = \mathbf{B}\mathbf{T}_{2,:,2} = \begin{bmatrix} -2 \\ 31 \end{bmatrix}.$$

Computing all entries yields:

$$\mathbf{R}_{:,:,1} = \begin{bmatrix} -2 & -2 \\ 4 & 13 \end{bmatrix}, \quad \mathbf{R}_{:,:,2} = \begin{bmatrix} -2 & -2 \\ 22 & 31 \end{bmatrix}.$$

□

Note that the compatibility rule on the shapes $M_1 \times \cdots M_n \times \cdots M_N$ and $J \times M_n \implies M_1 \times \cdots J \times \cdots M_N$ is a generalization of the **dimension rule** for matrix multiplication.

31.1.2 Tensor Decompositions

DL data is often represented as **high-rank tensors**.[12] As such tensors are typically quite large, it can become **computationally expensive** to store and manipulate them. Instead, we **decompose** these tensors using a set of **smaller**, **structured** tensors that retain the **essential information** found in the original high-rank tensor.[13]

12: Images, videos, etc. (as in Figure 31.1). The **rank** R of a tensor is a subtle quantity to define. For matrices, the rank is the **dimension of the column space** (or the number of non-zero singular values).

13: This works not unlike the **singular value decomposition** of a positive definite matrix A (see [6, sec. 23.4.1], for instance).

This has the effect of **compressing** the models, by reducing the number of neural network parameters (which enables more **efficient deployment** on memory-constrained devices); since decomposed tensors require **fewer operations** to process, the end result is faster DL **training times** and **inference**.

Furthermore, enforcing low-rank tensor structure can help prevent **overfitting** by acting as a form of implicit **regularization** (see [6, sec. 20.2.4] for more information).

Finally, decomposing a tensor can provide **interpretability:** The components from a tensor decomposition often correspond to **latent patterns** in the data, offering insights into the model's **internal representations**.[14]

14: This will be particularly important in natural language and image processing, see Chapters 32 and 35.

We now present a brief introduction to some of the common decomposition techniques.[15]

15: See [45] for more information.

CP Decomposition

The **canonical polyadic** decomposition (CP decomposition)[16] generalizes **matrix factorization** to **higher-order** tensors. It expresses a tensor $\mathbf{T}$ as a sum of outer products of rank-one tensors.

16: Or CANDECOMP/PARAFAC decomposition.

For instance, given an order$-N$ tensor $\mathbf{T} \in \mathbb{R}^{M_1 \times \cdots \times M_N}$, the CP decomposition approximates it as

$$\mathbf{T} \approx \sum_{r=1}^{R} \mathbf{a}_r^{(1)} \circ \cdots \circ \mathbf{a}_r^{(N)},$$

where:

- $\mathbf{a}_r^{(n)} \in \mathbb{R}^{M_n}$, $n = 1, \ldots, N$ are **factor vectors**, and
- R is the (given) **rank** of the decomposition (i.e., the **number of components** in the decomposition).

The result is a **low-rank approximation** of the tensor using a linear combination of outer products. Conceptually, the problem is simple: given a desired rank R, we must find the factor vectors that best approximate $\mathbf{T}$.

Unfortunately, the CP decomposition does not have a **closed form** solution and is **not numerically stable**.[17]

17: The choice of different numerical methods can lead to different solutions.

The **CP rank** R of a tensor $\mathbf{T}$ is the smallest integer R for which we can replace the "$\approx$" in the CP decomposition of $\mathbf{T}$ by a "$=$".

In general, finding the **best** rank approximation of $\mathbf{T}$ using CP decomposition is an **NP-hard problem**.[18]

18: But we can still find an arbitrary rank approximation of $\mathbf{T}$ using algorithms like alternating least squares or PARAFAC, for instance.

Note that an outer product of vectors is, by definition, a tensor of rank 1; the rank R CP decomposition of $\mathbf{T}$ is thus a sum of R rank-one tensors.

Tucker Decomposition

Both the CP and **Tucker** decompositions are a form of higher-order **singular value decomposition**.

The latter decomposition factorizes a tensor $\mathbf{T}$ into a **core tensor** $\mathbf{C}$ contracted along each mode to a set of **factor matrices**.

Given an order$-N$ tensor $\mathbf{T} \in \mathbb{R}^{M_1 \times M_2 \times \cdots \times M_N}$, the Tucker decomposition approximates it as

$$\mathbf{T} \approx \mathbf{C} \times_1 \mathbf{A}^{(1)} \times_2 \mathbf{A}^{(2)} \cdots \times_N \mathbf{A}^{(N)},$$

where:

- $\mathbf{C} \in \mathbb{R}^{R_1 \times R_2 \times \cdots \times R_N}$ is the **core tensor** of the decomposition, and
- $\mathbf{A}^{(n)} \in \mathbb{R}^{M_n \times R_n}$ are the **factor matrices** for mode n.

In the tensorial jargon, $\mathbf{C}$ captures the **interactions between the components across all modes**, while the $\mathbf{A}^{(n)}$ map the **original tensor dimensions to a lower-dimensional latent space**.[19]

19: We promise this will make sense (?) once you know more about tensors.

Tensor Train

The **tensor train** (TT) decomposition represents a high-order tensor as a **sequence** (the “train”) of **third-order** tensors (also called **TT cores**) connected by matrix contractions.[20] Unlike Tucker or CP decompositions, TT scales **linearly** with tensor order, making it suitable for **high-dimensional data**.

20: Obviously, the original tensor must at least be of order 3.

A tensor $\mathbf{T} \in \mathbb{R}^{M_1 \times M_2 \times \cdots \times M_N}$ is represented in TT format as:

$$\mathbf{T}_{i_1, i_2, \ldots, i_N} = \mathbf{G}^{(1)}_{1, i_1, r_1} \mathbf{G}^{(2)}_{r_1, i_2, r_2} \cdots \mathbf{G}^{(N)}_{r_{N-1}, i_N, 1},$$

where each $\mathbf{G}^{(n)} \in \mathbb{R}^{r_{n-1} \times M_n \times r_n}$ is a TT core, and the **TT ranks** r_n are such that $r_0 = r_N = 1$. Each scalar entry in the tensor is computed by **contracting** over the intermediate TT ranks $r_1, \ldots, r_{N-1}$.[21]

21: **Tensor contraction** is an operation that sums over two or more modes (axes); it generalizes **matrix multiplication** to higher-order tensors, reducing the dimensionality of the tensor by eliminating the contracted modes. For instance, if

$$\mathbf{A} \in \mathbb{R}^{M_1 \times M_2 \times M_3} \quad \text{and} \quad \mathbf{B} \in \mathbb{R}^{M_3 \times M_4 \times M_5},$$

then the **contraction** of $\mathbf{A}$ and $\mathbf{B}$ over their **shared mode** (mode 3) is

$$\mathbf{C} = \sum_{i_3} \mathbf{A}_{i_1, i_2, i_3} \mathbf{B}_{i_3, i_4, i_5}.$$

The resulting tensor $\mathbf{C} \in \mathbb{R}^{M_1 \times M_2 \times M_4 \times M_5}$ has a contracted mode-3 dimension.

Contraction can apply to **all** shared modes: if $\mathbf{X} \in \mathbb{R}^{2 \times 3 \times 4}$ and $\mathbf{Y} \in \mathbb{R}^{3 \times 4}$, say, then the contraction of $\mathbf{X}$ and $\mathbf{Y}$ over their shared modes is

$$\mathbf{Z}_i = \sum_{j=1}^{3} \sum_{k=1}^{4} \mathbf{X}_{i,j,k} \mathbf{Y}_{j,k}, \quad i = 1, 2.$$

As with the other decompositions, TT can be computed numerically.

31.1.3 Python Examples

Multiple `Python` modules support the use of tensors for deep learning; throughout this chapter, we will be using `tensorly`, `torch`, and `keras` (TensorFlow is also popular, as is Lightning). We start by showing how to do some of the basic tensor operations and manipulations using `NumPy`; then we move on to the CP and Tucker decompostions using `tensorly`.

Outer Product

```
import numpy as np  # for the following examples

a = np.array([1, 2])
b = np.array([3, 4, 5])
```

```
outer = np.outer(a, b)
print("Outer product:\n", outer)
```

```
Outer product:
 [[ 3  4  5]
 [ 6  8 10]]
```

```
c = np.array([6, 7])
T = np.einsum('i,j,k->ijk', a, b, c)
print("Outer product shape:", T.shape)
print("Outer product:\n", T)
```

```
Outer product shape: (2, 3, 2)
Outer product:
 [[[18 21]
  [24 28]
  [30 35]]

 [[36 42]
  [48 56]
  [60 70]]]
```

Important note: in `NumPy`, tensors are stored in **row-major** (C-style) order, which is not the same as the mathematical (MATLAB-style) order: we think of a tensor $\mathbf{T} \in \mathbb{R}^{2\times3\times2}$ as basically being an array containing two 2×3 matrices, but `NumPy` thinks of a $2 \times 3 \times 2$ tensor as being an array containing two 3×2 matrices (see example above).[22]

22: Proceed with caution: this distinction may yield unexpected numerical outcomes! Read the appropriate documentation to avoid embarassing mistakes.

To display the tensor in the MATLAB-style order, we may use the following command.

```
print(T[:,:,0])
print(T[:,:,1])
```

```
[[18 24 30]
 [36 48 60]]
[[21 28 35]
 [42 56 70]]
```

Kronecker Product

```
A = np.array([[1, 2], [3, 4]])
B = np.array([[0, 5], [6, 7]])

kronAB = np.kron(A, B)
print("Kronecker product:\n", kronAB)

kronBA = np.kron(B,A)
print("Kronecker product:\n", kronBA)
```

```
Kronecker product:
 [[ 0  5  0 10]
 [ 6  7 12 14]
 [ 0 15  0 20]
 [18 21 24 28]]
Kronecker product:
 [[ 0  0  5 10]
 [ 0  0 15 20]
 [ 6 12  7 14]
 [18 24 21 28]]
```

Kathri-Rao Product

```
A = np.array([[1, 2], [3, 4]])     # shape (2, 2)
B = np.array([[5, 6], [7, 8]])     # shape (2, 2)

krpAB = np.stack([np.kron(A[:, i], B[:, i])
  for i in range(A.shape[1])], axis=1)
print("Khatri-Rao product:\n", krpAB)

krpBA = np.stack([np.kron(B[:, i], A[:, i])
  for i in range(B.shape[1])], axis=1)
print("Khatri-Rao product:\n", krpBA)
```

```
Khatri-Rao product:
 [[ 5 12]
 [ 7 16]
 [15 24]
 [21 32]]
Khatri-Rao product:
 [[ 5 12]
 [15 24]
 [ 7 16]
 [21 32]]
```

Hadamard Product

```
X = np.array([[1, 2], [3, 4]])
Y = np.array([[5, 6], [7, 8]])

hadamard = X * Y
print("Hadamard product:\n", hadamard)
```

```
Hadamard product:
 [[ 5 12]
 [21 32]]
```

Mode−n Tensor-Matrix Product

```
T = np.arange(1, 13).reshape(2, 2, 3)  # shape 2 x 3 x 2
A = np.array([[1, 0], [0, 1], [1,- 1], [2, 0]])  # shape 4 x 2
print(T)
```

```
[[[ 1  2]
  [ 3  4]
  [ 5  6]]

 [[ 7  8]
  [ 9 10]
  [11 12]]]
```

```
# Mode-1 product: (2,3,2) x (4,2) => (4,3,2)
T_mode1 = np.tensordot(T, A, axes=([1], [1]))
print("Mode-1 product shape:", T_mode1.shape)
print("Mode-1 product:\n", T_mode1)
```

```
Mode-1 product shape: (2, 3, 4)
Mode-1 product:
 [[[ 1  4  5  6]
  [ 2  5  7  9]
  [ 3  6  9 12]]

 [[ 7 10 17 24]
  [ 8 11 19 27]
  [ 9 12 21 30]]]
```

```
B = np.array([[1, 0,-1], [2, 1,0]])  # shape (2, 3)
# Mode-2 product: (2,3,2) x (2,3) => (2,2,2)
T_mode2 = np.tensordot(T, B, axes=([2], [1]))
print("Mode-2 product shape:", T_mode2.shape)
print("Mode-2 product:\n", T_mode2)
```

```
Mode-2 product shape: (2, 2, 2)
Mode-2 product:
 [[[-2  4]
  [-2 13]]

 [[-2 22]
  [-2 31]]]
```

Compare both of these results with those displayed on p. 2034.

Tensor Contraction

```
A = np.arange(1, 13).reshape(2, 2, 3)  # shape (2, 2, 3)
B = np.array([[1, 0], [0, 1], [1, -1]])  # shape (3, 2)

# Contract over 3rd mode of A and 1st of B
C = np.tensordot(A, B, axes=([2], [0]))
print("Tensor contraction shape:", C.shape)
print("Tensor contraction:\n", C)
```

```
Tensor contraction shape: (2, 2, 2)
Tensor contraction:
 [[[ 4 -1]
  [10 -1]]

 [[16 -1]
  [22 -1]]]
```

CP Decomposition This example uses `tensorly` (based on [14]). We start by randomly generating a tensor with `Python` shape $(4, 5, 5)$.

```
import numpy as np
import tensorly as tl
from tensorly.decomposition import parafac
tensor = tl.tensor(np.random.rand(4, 5, 5))
```

Next, we perform CP decomposition (implemented in the function `parafac()`), with a specified rank.

```
weights, factors = parafac(tensor, rank=2)
print([f.shape for f in factors])
```

```
[(4, 2), (5, 2), (5, 2)]
```

The corresponding factor matrices are shown below.

```
print([f for f in factors])
```

```
[array([[ 2.55938368,  0.49624381],
       [ 1.77254692,  0.57186814],
       [ 2.27188501, -0.46890374],
       [ 2.52875964, -0.71115674]]),
 array([[ 0.51503688, -0.44818875],
       [ 0.47781472,  0.43644334],
       [ 0.41867263, -0.17015981],
       [ 0.36279829, -0.86730986],
       [ 0.44940833, -0.20640827]]),
 array([[ 0.38732292,  0.64199364],
       [ 0.41403377, -0.58954493],
       [ 0.53851071, -0.57951735],
       [ 0.35341253, -0.11034623],
       [ 0.52123097, -0.03739188]])]
```

If we try with a different specified rank, the shape of the factor matrices changes.

```
weights, factors = parafac(tensor, rank=3)
print([f.shape for f in factors])
print([f for f in factors])
```

```
[(4, 3), (5, 3), (5, 3)]
[array([[ 2.42185432,  0.36570674, -0.38130414],
       [ 1.78569396,  0.65348022,  0.35777504],
       [ 2.34544897, -0.38018497,  0.58490281],
       [ 2.46651382, -0.78887973, -0.13466856]]),
 array([[ 0.50832699, -0.51710018, -0.19582142],
       [ 0.46716743,  0.36728412, -0.38438282],
       [ 0.42351606, -0.19013252,  0.66025121],
       [ 0.36600536, -0.94429044,  0.25547303],
       [ 0.46034579, -0.15257433,  0.85992368]]),
 array([[ 0.39012605,  0.60798698, -0.25821908],
       [ 0.43726333, -0.54578806, -0.82441429],
       [ 0.55954273, -0.51872309, -0.52134607],
       [ 0.34910667, -0.18027782,  0.9091495 ],
       [ 0.53087455, -0.03977997, -0.20009663]])]
```

As we did not set a seed for the random tensor generation, the results may vary from one run to the next.

Tucker Decomposition This example also uses `tensorly` and is based on [28]. We start by randomly generating a tensor with `Python` shape $(4, 5, 5)$.

```
from tensorly.decomposition import tucker
import tensorly as tl
import numpy as np

tensor = tl.tensor(np.random.rand(4, 5, 5))
```

We perform the Tucker decomposition using specified ranks.

```
core, factors = tucker(tensor, rank=[2, 3, 3])
print(core.shape)
```

```
(2, 3, 3)
```

The factors have the following shapes.

```
print([f.shape for f in factors])
```

```
[(4, 2), (5, 3), (5, 3)]
```

The core tensor is shown below.

```
print(core)
```

```
[[[ 4.77293138  0.07474678 -0.09817885]
  [-0.01218641  0.09532709 -0.16308812]
  [ 0.01107893  0.81605273  0.24030178]]
```

```
[[ 0.01934867  0.40195788  0.23164684]
 [-1.1010468   0.39874221 -0.4588211 ]
 [-0.19111676 -0.34856158  0.23919615]]]
```

The factor matrices are shown below.

```
print([f for f in factors])
```

```
[array([[ 2.67835607,  0.55550733],
       [ 2.57075435, -0.54887358],
       [ 2.07504396, -0.58550151],
       [ 2.26569094, -0.35003683]]),
 array([[ 0.42558537, -0.56870536],
       [ 0.39767766, -0.78906095],
       [ 0.48824603, -0.95580043],
       [ 0.45437358,  0.17650822],
       [ 0.44939767, -0.08949423]]),
 array([[ 0.35446398,  0.71295441],
       [ 0.44186191, -0.36913502],
       [ 0.46614662,  0.72579271],
       [ 0.46247595, -0.18610834],
       [ 0.44069323,  0.25841722]])]
```

As we did not set a seed for the random tensor generation, the results may vary from one run to the next.

31.2 Deep Networks

As we have seen in [6, sec. 21.4.3], neural networks are essentially functions: they take in **inputs**, something happens "behind the scenes" (**black box**), and out come the **outputs** (as in Figure 31.2).[23]

23: The training/testing paradigm of machine learning (see Chapters 20 and 21 in [6]) still applies.

Simple artificial neural networks (like the two-layer perceptron) can approximate **arbitrary continuous functions** between inputs and outputs, provided that they use a **sufficiently large** number of **hidden units** [16].

Unfortunately, this requirement can lead to impractically **large networks** that require more data than is usually available to **train** them (i.e., to estimate the **network weights** reliably).

Deep networks provide a **scalable** alternative: instead of increasing the number of hidden units, we increase the **depth** of the model (i.e., the number of hidden layers). Each layer can produce multiple outputs, which may be further transformed, for example, into a probability vector using suitable activation functions (see Figure 31.3).

It has been shown that adding layers increases the network's **representational capacity**; a deep neural network can express a **broader** class of functions than a shallower network with the same number of parameters [4, 22].[24] This enhanced expressivity enables deep networks to efficiently capture **complex mappings**.[25]

24: This is due to the number of linear regions in the network growing exponentially with the number of layers.

25: Such as those between the pixel values of a video and a corresponding textual description.

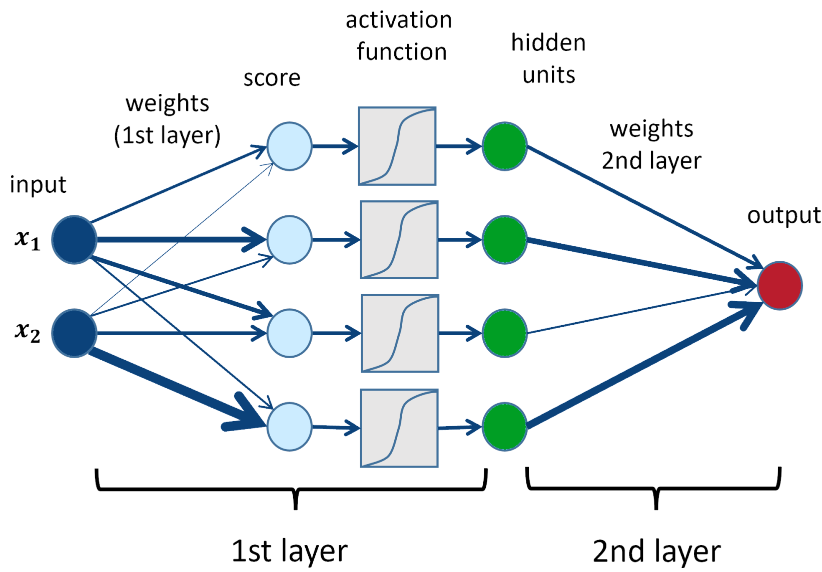


Figure 31.2: The two-layer perceptron is an artificial neural network (an interconnected group of nodes). Each circular node represents an artificial neuron and an arrow represents a connection from the output of one artificial neuron to the input of another; the thickness of the arrow represents the relative magnitude of the contribution of the source node to the corresponding score node (the node signals are combined linearly into each score node). The signal propagates from the input nodes to the hidden layer, where the scores are transformed by nonlinear activation functions before being passed on to the output node [modified from an image by the Lamarr Institute].

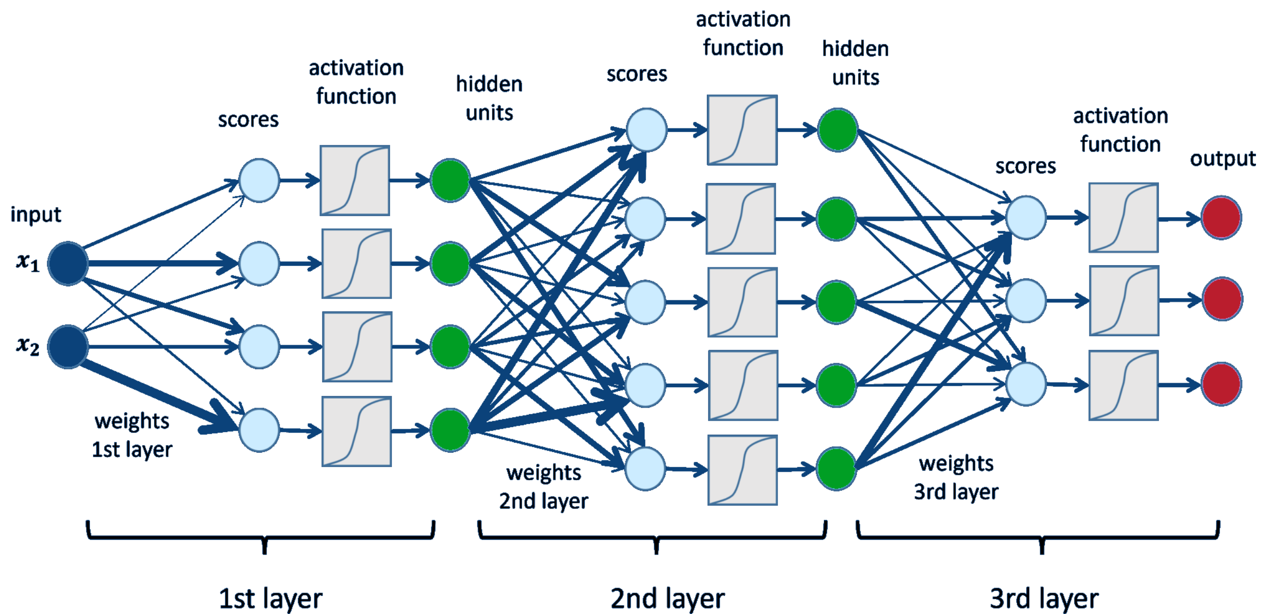


Figure 31.3: A deep network extends the two-layer perceptron by adding multiple hidden layers, as opposed to extending by adding more hidden units in one layer. The principle remains the same [modified from an image by the Lamarr Institute].

By way of illustration, consider a network with 5 layers, each with 10 nodes (for a total of 50 nodes). In the deep learning context, a **linear region** refers to specific portions of the input space where a deep neural network behaves as a linear function.

The number of linear regions that can be learned by the 50-node deep network described above is 10^5; a (shallow) one-layer neural network would require 10^5 nodes to learn the same number of linear regions, so the deep learning "savings" are substantial!

According to the Lamarr Institute's G. Paaß,

> "Deep neural networks for natural language processing, object recognition in images, and speech recognition have a specialized architecture with layers tailored to each application. [...] Overall, their success is based on **four pillars:**
>
> 1. the availability of powerful parallel processors to perform the training;
> 2. the collection and annotation of extensive training data;
> 3. the development of powerful regularization and optimization methods, and
> 4. the availability of toolkits to easily define deep neural networks and automatically compute their gradients."

We have already discussed **parallel processing** in Chapter 30, **data collection** in Chapters 10 and 16, **regularization** in Chapter 20, and **optimization** in Chapter 5; we will visit some of the **specialized architectures** in Chapters 32 (text) and 35 (images); for now, we will get comfortable with **network architecture tools** using `PyTorch` and Lightning.

31.2.1 Getting Started

We will tackle a familiar task, **linear regression**. We start by importing dependencies and setting a seed for reproducibility.

```
import lightning as L
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset

torch.manual_seed(0)
```

Next, we generate artificial data (x, y) according to $y = 3x + 1$.

```
# sample x from a standard normal, multiplied by 10
x = torch.randn(100, 1) * 10
# compute y, add noise
y = 3 * x + 1 + torch.randn(100, 1)
# initialize tensor dataset
dataset = TensorDataset(x, y)
# initialize data loader
train_loader = DataLoader(dataset, batch_size=10, shuffle=True)
```

Once the data loader has been initialized, we define a class for our model.

```
class LinearRegressionModel(L.LightningModule):
    def __init__(self):
        super().__init__()
        self.linear = nn.Linear(1, 1)

    # forward pass
    def forward(self, x):
        return self.linear(x)

    # training step; later examples will also include a validation set.
    def training_step(self, batch, batch_idx):
        x, y = batch
        y_pred = self(x)
        loss = nn.functional.mse_loss(y_pred, y)
        self.log('train_loss', loss)
        return loss

    # stochastic gradient descent for optimization
    def configure_optimizers(self):
        return optim.SGD(self.parameters(), lr=0.01)
```

Finally, we train the model on the artificial data.

```
model = LinearRegressionModel()
trainer = L.Trainer(max_epochs=100)
trainer.fit(model, train_loader)
```

```
INFO:
  | Name   | Type   | Params | Mode
------------------------------------------
0 | linear | Linear | 2      | train
------------------------------------------
2         Trainable params
0         Non-trainable params
2         Total params
0.000     Total estimated model params size (MB)
1         Modules in train mode
0         Modules in eval mode

INFO:lightning.pytorch.callbacks.model_summary:
  | Name   | Type   | Params | Mode
------------------------------------------
0 | linear | Linear | 2      | train
------------------------------------------
2         Trainable params
0         Non-trainable params
2         Total params
0.000     Total estimated model params size (MB)
1         Modules in train mode
0         Modules in eval mode
```

```
/usr/local/lib/python3.10/dist-packages/lightning/pytorch/loops/fit_loop.py:298:
The number of training batches (10) is smaller than the logging interval Trainer(log_every_n_steps=50).
Set a lower value for log_every_n_steps if you want to see logs for the training epoch.
Epoch 99: 100% 10/10 [00:00<00:00, 50.34it/s, v_num=1]
INFO: 'Trainer.fit' stopped: 'max_epochs=100' reached.
INFO:lightning.pytorch.utilities.rank_zero:'Trainer.fit' stopped: 'max_epochs=100' reached.
```

Let's see how the model did.

```
slope, intercept = model.linear.weight.item(), model.linear.bias.item()
print(f'Trained Model Parameters: Slope: {slope:.4f}, Intercept: {intercept:.4f}')
```

```
Trained Model Parameters: Slope: 2.9757, Intercept: 0.9874
```

That's not bad! In the coming sections, we will explore more advanced techniques for training deep networks and for working with different specialized network architectures.

31.2.2 Activation Functions

Within a deep network, different types of hidden units serve different purposes. Most types of hidden units perform an **affine transformation** on the layer's inputs,[26] and then apply an **activation function** $a(z)$ to the **transformed inputs** [11].

26: That is, if the input vector for an observation is $\mathbf{X} = (x_1, \ldots, x_k)$, then the signal that gets sent to a hidden node is $\mathbf{w}^\top \mathbf{X} + b$ for some weight vector $\mathbf{w} = (w_1, \ldots, w_k)$ and scalar b.

At least one activation function within the deep network has to be nonlinear, otherwise the network **collapses to a linear model** [23]. Linear activation functions can be useful to **reduce the number of parameters** in a model, but the discussion surrounding hidden units and their activation functions typically concerns **nonlinear units** [11].[27]

27: A **nonlinear unit** refers to a deep network component (typically a node or activation function) that introduces nonlinearity into the model.

ReLU The default (and most commonly used) activation function for hidden layers is the **rectified linear unit** (ReLU) activation function:

$$a(z) = \max(0, z).$$

ReLU is equivalent to a linear activation function where the negative inputs are "**turned off**" [23].

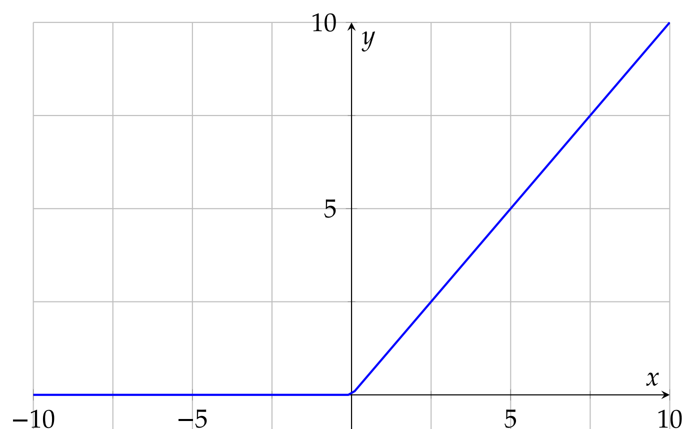


Figure 31.4: The ReLU activation function.

We see from the graph that ReLU is **not differentiable** at the origin; in theory, this should make ReLU useless for **gradient-based learning**, but in practice, most deep networks never achieve **zero loss**, so the problem does not usually arise.

Some variants of ReLU modify the **slope** for **negative inputs** [11]. For a given input z and a pre-selected scalar α, the general form is:

$$a(z, \alpha) = \max(0, z) + \alpha \cdot \min(0, z).$$

Other variants are listed below:

- the **absolute value rectification** uses $\alpha = -1$; it is commonly used for image processing [11];
- the **leaky ReLU** picks $0 < \alpha \ll 1$; it is used in place of standard ReLU to prevent the "**dying ReLU**" problem;[28]
- the **parametric leaky ReLU** uses the deep learning model itself to learn the value of α [5].

28: Units using ReLU as an activation "**die**" when all of the inputs are non-positive: all negative inputs turn into zero outputs, and the weights cannot be updated [5].

In the early days of deep learning, there were two commonly-used activation functions for hidden units: the **sigmoid** function

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

and the **hyperbolic tangent** function

$$a(z) = \tanh(z).$$

These activation functions are no longer recommended for hidden units because they **saturate**, which is to say that their output values approach a maximum (or a minimum) value and become **insensitive to input changes** in certain regions, where the **derivative** (gradient) of the function is very small (or even zero).

The sigmoid function saturates at 0 and 1; the hyperbolic tangent function at −1 and 1 (see below).

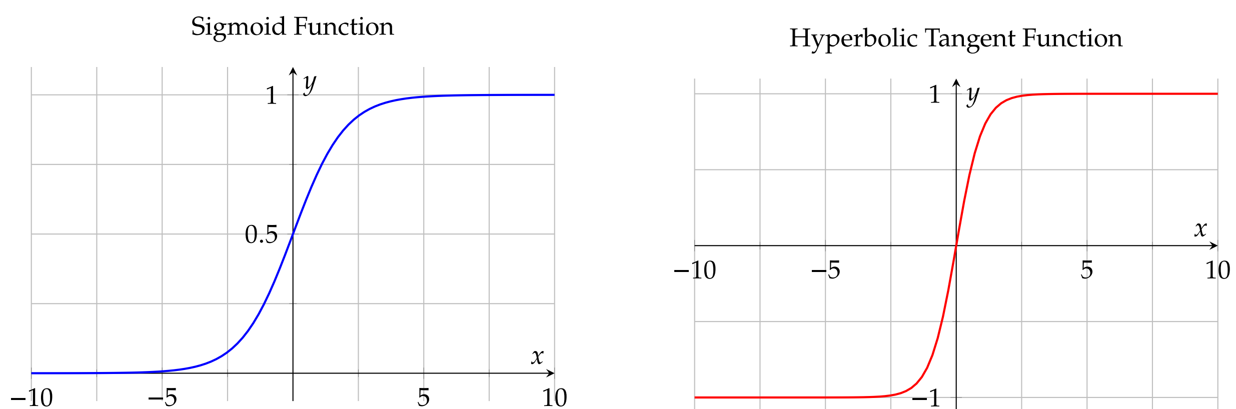


Figure 31.5: The sigmoid and hyperbolic tangent functions.

This has significant implications for training neural networks using gradient-based optimization methods such as **backpropagation** (see Section 21.4.3). If the gradient is near zero, the weights update **very slowly** (if at all); with a negligible gradient, the model does not converge to a solution (i.e., we cannot find the optimal model parameters).[29]

29: This phenomenon is known as the **vanishing gradient problem**.

31.2.3 Weight Initialization

As mentioned in Chapter 21, before a network (shallow or deep) can be trained, its **weights** have to be randomly initialized. But proper **initialization** of the weights is crucial to ensure "**fast**" convergence of the **gradient descent** algorithm (which finds the deep network parameters, i.e., the optimal weights), and to prevent **exploding gradients**.[30]

30: Random weights are often sampled from a normal distribution, but if the variance of this distribution is the same for all layers, gradients may "explode" during the training phase [23].

In practice, experts recommend the following initialization strategies:

- **Glorot initialization**, also known as **Xavier initialization**, sets the variance of the sampling distribution for each layer to $\sigma^2 = \frac{2}{n_{\text{in}}+n_{\text{out}}}$, where n_{in} is the number of **input connections** to a unit in that layer, and n_{out} is the number of **output connections**;[31]
- **LeCun initialization** uses $\sigma^2 = \frac{1}{n_{\text{in}}}$, and is identical to Glorot initialization on layers where the number of input connections is equal to the number of output connections; and
- **He initialization** uses $\sigma^2 = \frac{2}{n_{\text{in}}}$.[32]

31: This is the preferred initialization scheme for sigmoid and hyperbolic tangent activations [23].

32: It is the preferred initialization scheme for ReLU activation [23].

In theory, the closer the initial set of weights is to the optimal training weights, the faster the gradient descent algorithm will converge to the latter; in practice, if the initial weights are already close to the optimal weights, why bother training the model? Of course, in deep learning networks with 1000+ weights, the likelihood that initial randomization of weights would be in near optimal configuration is **negligible**.

31.3 Regularization

But the **large** number of deep learning parameters (often counted in the millions) has another drawback, as it may give DL networks the capacity to **memorize the training data**, including any **noise** and **irrelevant** patterns it may contain. We have seen that **regularization** is used in machine learning to prevent **overfitting** and to improve the model's ability to generalize to **unseen data**.[33]

33: Regularization also improves convergence of the numerical algorithms used to solve the corresponding learning optimization problems.

In general, regularization introduces **additional constraints** or **penalties** into the learning process to discourage overly complex models.

31.3.1 Weight Decay

We discussed **ridge regression** and the **LASSO** in the context of linear models in Section 20.2.4; when generalized beyond such models, these techniques are known respectively as L_2 **regularization** and L_1 **regularization**. In the context of DL, these are also called **weight decay**.

Weight decay applies a regularization term to the objective function $J(\mathbf{w})$. The **regularized objective function** is $\tilde{J}_k(\mathbf{w})$, for $k = 1, 2$, where

$$\tilde{J}_2(\mathbf{w}) = J(\mathbf{w}) + \frac{\lambda}{2}\|\mathbf{w}\|_2^2 = J(\mathbf{w}) + \frac{\lambda}{2}\sum_i w_i^2;$$
$$\tilde{J}_1(\mathbf{w}) = J(\mathbf{w}) + \lambda\|\mathbf{w}\|_1 = J(\mathbf{w}) + \lambda\sum_i |w_i|,$$

and λ is a **hyperparameter** whose optimal value can be found using **cross-validation**. In general, L_2 regularization yields **smaller weights**, while L_1 regularization yields **sparse** models, where "many" of the weights are driven to 0.

L_0 **regularization**, where the regularized objective function

$$\tilde{J}_0(\mathbf{w}) = J(\mathbf{w}) + \lambda\|\mathbf{w}\|_0 = J(\mathbf{w}) + \lambda\sum_i \operatorname{sign}|w_i|,$$

penalizes the number of nonzero parameters in a model, is another approach that yields sparse models [13]. There are drawbacks, however: $\tilde{J}_0$ is **non-differentiable** (and so incompatible with **gradient-based optimization**) and leads to a **combinatorial optimization** problem (i.e., minimizing $\tilde{J}_0$ is **NP-hard** in general).[34]

34: It is nevertheless sometimes used for **feature selection** and for **pruning** edges or nodes from deep networks.

Elastic net regularization combines L_1 and L_2 penalty terms to get the benefits of both L_1 regularization (sparsity, i.e. group-wise feature selection on weights) and L_2 regularization (stability and small weight values); the corresponding regularized objective function

$$\begin{aligned}\tilde{J}(\mathbf{w}) &= J(\mathbf{w}) + \lambda_1\|\mathbf{w}\|_1 + \lambda_2\|\mathbf{w}\|_2^2 \\ &= J(\mathbf{w}) + \lambda\left[\alpha\|\mathbf{w}\|_1 + (1-\alpha)\|\mathbf{w}\|_2^2\right],\end{aligned}$$

where λ_1, λ_2 are model hyperparameters, or alternatively, λ is a hyperparameter and $\alpha \in [0,1]$ is a **mixing parameter**.[35]

35: As implemented in the `R` library `glmnet`.

Neither L_0 regularization nor elastic net regularization is commonly used in the DL context; the main alternatives to weight decay are **early stopping** and **dropout** regularization.

31.3.2 Early Stopping

When a deep learning model is trained for too many **epochs**, the validation loss will typically decrease at first, reach a minimum, and then begin to increase as the model starts to **overfit** the training data.[36]

36: An **epoch** refers to *one complete pass through the entire training data* by the learning algorithm. During an epoch, the model processes all training examples exactly once, usually in smaller groups called **mini-batches**. Training typically requires **multiple** epochs (e.g., 10 to 100 or more), since a single pass is rarely sufficient for the model to converge to an optimal solution. After each epoch, we evaluate the model on **validation data** and monitor performance metrics (**loss**, **accuracy**, etc.). Formally, let

- N be the total number of training samples;
- B be the mini-batch size, and
- E be the number of epochs.

Then the model performs $\frac{N}{B}$ **iterations per epoch**, and so $\frac{E \cdot N}{B}$ **weight updates over the entire training period**. Unlike in traditional numerical analysis where we set the **convergence threshold** ε and a **maximum number of iterations** M, the **number of epochs** E is a hyperparameter and must be chosen carefully to balance underfitting and overfitting.

This can be mitigated using **early stopping**, a regularization technique that monitors **validation performance** during training. In early stopping, the model's parameters are saved only when the validation loss **improves** [11], and training is halted **automatically** if the loss fails to improve (or improves only slightly) over a specified number of iterations.

31.3.3 Dropout

This regularization technique **randomly** sets a given proportion of the model's weights to 0 **after each gradient step**; this means that a "different" model is being trained at each step [39]. This **dropout** approach is similar to **bagging** (see [6, sec. 21.5.1]), but instead of being **independent**, each model's weights form a subset of the **full** deep network's weights [11]. Dropout helps with overfitting by preventing "**lazy**" nodes.

Consider a hockey team consisting of a few star players, many average players, and some weak players; generally, the best players get the **most** ice time. But that means that the other players get **less** ice time, and thus

do not improve **at the same rate**. Suppose that, for each game, a certain number of players were randomly selected and told to **stay home**; the rest of the team would have to **adapt** and learn to play **without** those players. In this scenario, then, the team is less at risk of becoming dependent on a **few** superstars, and the bench-warmers get a chance to **improve** and **contribute**.

In this analogy, we easily recognize that the team plays the role of the **deep network**, and each player represents a **connection weight**. By randomly "dropping out" some weights, the others get more "practice" and become more **valuable assets** to the network [5].[37]

37: The analogy is far from perfect: the team would get quite tired during a game if more than 1 or 2 players were dropped from the line-up, but tiredness does not have an equivalent property in deep learning networks.

We have to remain vigilant, however. Selecting **too high** a dropout proportion can create havoc with the deep network: among the reasons Thanos (see Figure 31.6) was unpopular in the *Marvel Cinematic Universe*, the high proportion of the population he "dropped out" of the Universe was an important driver... perhaps the Avengers wouldn't have gotten up in arms if he had only deleted 1% or 10% of the population. The takeaway is that dropping 50% of the weights is **excessive** and **ill-advised** [1].

Figure 31.6: Consider this meme from Twitter user `@mlinterview`: in the Marvel Cinematic Universe, Thanos is a villain (or a well-intentioned extremist, maybe) whose goal is to make half of all living organisms in the universe disappear (those who disappear are randomly selected). While his motivations are not related to regularization, we can imagine that successfully doing so would force the remaining life forms to adapt... but would it necessarily be for the better?

31.4 Variants of Stochastic Gradient Descent

We discussed an algorithm to find the optimal network weights, namely **stochastic gradient descent** (SGD), in Section 21.4.3. In what follows, we present some common variants that help overcome some of SGD's issues.[38]

38: The notion of algorithmic convergence was covered in [6, ch. 4], as were other details of numerical analysis.

31.4.1 Momentum

One significant drawback of SGD is that it can converge **very slowly** to an optimal solution. SGD with **momentum** helps address this by including a **weighted moving average** of past gradients; this allows the algorithm to converge more quickly when sequential gradients move in the **same direction**, and to slow down if there is a **sudden change** [23].

Let $\mathbf{m}_t$ be the **momentum**, η the **learning rate**, $\mathbf{g}_t = \nabla\mathcal{L}(\boldsymbol{\theta}_t)$ the **gradient at the current output**, and $\boldsymbol{\theta}_t$ the **parameter output**. The parameter β is

the **momentum hyperparameter**, which controls the exponential decay of the weights of past gradients.[39]

$$\begin{aligned}\mathbf{m}_t &= \beta \mathbf{m}_{t-1} + \mathbf{g}_t;\\ \boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t - \eta \mathbf{m}_t.\end{aligned}$$

39: Commonly-used β values include 0.5, 0.9, and 0.99 [23].

Some resources give a different set of formulas for optimization with momentum. Note that the following are equivalent when $\mathbf{m}_0 = 0$:

$$\begin{aligned}\mathbf{m}_t &= \beta \mathbf{m}_{t-1} - \eta \mathbf{g}_t;\\ \boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t + \mathbf{m}_t.\end{aligned}$$

While momentum can speed up convergence, it may also cause the optimizer to miss the minimum by **overshooting** it due to excessive "**acceleration**".

31.4.2 Nesterov Momentum

Nesterov momentum is a variant of momentum designed to reduce the risk of overshooting the minimum during optimization [41]. This algorithm uses the gradient at $\boldsymbol{\theta}_t + \beta \mathbf{m}_{t-1}$ instead of at $\boldsymbol{\theta}_t$. This can be viewed as an "**extrapolation**" step: instead of measuring at the current output, we look slightly ahead in the direction of the momentum, which can allow for faster convergence:

$$\begin{aligned}\mathbf{g}_t &= \nabla \mathscr{L}(\boldsymbol{\theta}_t + \beta \mathbf{m}_{t-1});\\ \mathbf{m}_t &= \beta \mathbf{m}_{t-1} - \eta \mathbf{g}_t;\\ \boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t + \mathbf{m}_t.\end{aligned}$$

Like standard momentum, Nesterov momentum is a **first-order optimization method** that offers improved convergence guarantees over gradient descent in some settings, though this advantage **does not hold** in the stochastic gradient case [11].

31.4.3 AdaGrad

Until now, we have only discussed learning algorithms with **static learning rates**. The **adaptive gradient algorithm** (AdaGrad) modifies the learning rates of the model parameters in a manner that speeds up convergence when the gradient is **less extreme**.

The algorithm is provided below. The variable $\mathbf{s}_t$ represents the **gradient accumulation variable** at step t, with $\mathbf{s}_0 = 0$; η is the **global learning rate**, and ε is a small value that prevents division by 0:

$$\begin{aligned}\mathbf{g}_t &= \nabla \mathscr{L}(\boldsymbol{\theta}_t);\\ \mathbf{s}_t &= \mathbf{s}_{t-1} + \mathbf{g}_t \odot \mathbf{g}_t;\\ \boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t - \eta \operatorname{diag}(\varepsilon \mathbf{I} + \mathbf{s}_t)^{-1/2} \mathbf{g}_t := \boldsymbol{\theta}_t - \eta \frac{1}{\sqrt{\varepsilon + \operatorname{diag}(\mathbf{s}_t)}} \odot \mathbf{g}_t.\end{aligned}$$

One major drawback of AdaGrad is its sensitivity to the **initial parameter conditions**. For example, if the initial gradients are large, the corresponding learning rates can become very small for the remainder of training.

Moreover, since the accumulation of squared gradients in $\mathbf{s}_t$ begins from the first update and is **unweighted**, the effective learning rates may **shrink rapidly**, even before the optimizer has reached a good **local minimum**. As a result, despite providing per-parameter learning rates, AdaGrad remains sensitive to the choice of the **global learning rate** η.[40]

40: References and numerical examples can be found in the Cornell University Computational Optimization Open Textbook and in [23, 48].

Some variants of AdaGrad have been proposed to overcome this (and other problems), including AdaDelta, RMSprop, and Adam.

31.4.4 RMSprop

RMSprop is similar to AdaGrad, but with an **exponentially decaying moving average** gradient accumulation variable [11]. This allows the algorithm to converge more quickly as it approaches a solution. The parameters are the same as for AdaGrad, with the addition of β as the **decay rate**; a common value for β is 0.9 [23]:

$$\begin{aligned}
\mathbf{g}_t &= \nabla\mathcal{L}(\boldsymbol{\theta}_t);\\
\mathbf{s}_t &= \beta\mathbf{s}_{t-1} + (1-\beta)\mathbf{g}_t \odot \mathbf{g}_t;\\
\Delta\boldsymbol{\theta}_t &= -\eta\ \mathrm{diag}(\varepsilon\mathbf{I} + \mathbf{s}_t)^{-1/2}\mathbf{g}_t := -\eta\frac{1}{\sqrt{\varepsilon + \mathrm{diag}(\mathbf{s}_t)}} \odot \mathbf{g}_t;\\
\boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t + \Delta\boldsymbol{\theta}_t.
\end{aligned}$$

In this (and the two following sections), we use the notation

$$\frac{1}{\sqrt{\varepsilon + \mathrm{diag}(\mathbf{s}_t)}} := \mathrm{diag}(\varepsilon\mathbf{I} + \mathbf{s}_t)^{-1/2}.$$

31.4.5 AdaDelta

AdaDelta is a variant of RMSprop that rescales the update $\Delta\boldsymbol{\theta}_t$ using an exponentially weighted moving average of past updates [23, 47, 48]. In addition to $\mathbf{s}_t$, we introduce $\mathbf{u}_t$ as the **update accumulation variable** at step t, with $\mathbf{u}_0 = 0$:[41]

41: Both RMSprop and AdaDelta use trivial initializations for $\mathbf{s}_0$ and $\mathbf{u}_0$.

$$\begin{aligned}
\mathbf{g}_t &= \nabla\mathcal{L}(\boldsymbol{\theta}_t);\\
\mathbf{s}_t &= \beta\mathbf{s}_{t-1} + (1-\beta)\mathbf{g}_t \odot \mathbf{g}_t;\\
\Delta\boldsymbol{\theta}_t &= -\frac{\sqrt{\varepsilon + \mathrm{diag}(\mathbf{u}_{t-1})}}{\sqrt{\varepsilon + \mathrm{diag}(\mathbf{s}_t)}} \odot \mathbf{g}_t;\\
\mathbf{u}_t &= \beta\mathbf{u}_{t-1} + (1-\beta)\Delta\boldsymbol{\theta}_t \odot \Delta\boldsymbol{\theta}_t;\\
\boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t + \Delta\boldsymbol{\theta}_t.
\end{aligned}$$

The search for **efficient DL optimization** is recapped below [48]:

- **stochastic gradient descent (SGD)** improves over standard gradient descent in many scenarios, particularly due to its robustness to redundant data;
- **mini-batch SGD** enhances efficiency further through vectorization, processing larger subsets of data per iteration;[42]
- **momentum** accelerates convergence by incorporating a moving average of past gradients, reducing oscillations and helping the optimizer build velocity in relevant directions;
- **AdaGrad** adapts learning rates per parameter using accumulated squared gradients, effectively serving as a simple yet powerful **preconditioner**.[43]
- **RMSprop** modifies AdaGrad by decoupling the learning rate from the per-parameter scaling, maintaining adaptability without overly diminishing the learning rate over time.

42: An essential approach to parallelize across multiple machines or GPUs.

43: A **preconditioner** is a transformation applied to the gradient (or to the parameter space) that improves the efficiency and stability of the optimization process. This modification to the optimization landscape helps gradient-based methods converge more quickly and reliably. Intuitively, a preconditioner adjusts the **scale** or **orientation** of the gradient updates so that the optimizer does not take overly large steps in some directions and overly small steps in others. This is especially useful when the loss surface is **ill-conditioned** (e.g., it has steep curvature in some directions and shallow curvature in others). Formally, standard gradient descent performs the update:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta \nabla \mathcal{L}(\boldsymbol{\theta}_t);$$

a **preconditioned update** instead takes the form:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta \mathbf{P}_t \nabla \mathcal{L}(\boldsymbol{\theta}_t),$$

where $\mathbf{P}_t$ is a matrix or a vector. **Adaptive** optimization algorithms like AdaGrad, RMSprop, and Adam rely on **diagonal preconditioners** to achieve efficient and stable convergence.

31.4.6 Adam

These ideas culminate in the **adaptive moment estimation** optimizer (Adam) [18], which integrates **momentum** and **adaptive learning rates** into a single efficient and widely used optimization algorithm.[44]

$$\begin{aligned}
\mathbf{g}_t &= \nabla \mathcal{L}(\boldsymbol{\theta}_t);\\
\mathbf{m}_t &= \beta_1 \mathbf{m}_{t-1} + (1-\beta_1)\mathbf{g}_t;\\
\mathbf{s}_t &= \beta_2 \mathbf{s}_{t-1} + (1-\beta_2)\mathbf{g}_t \odot \mathbf{g}_t;\\
\hat{\mathbf{m}}_t &= \frac{\mathbf{m}_t}{1-\beta_1^t};\\
\hat{\mathbf{s}}_t &= \frac{\mathbf{s}_t}{1-\beta_2^t};\\
\Delta\boldsymbol{\theta}_t &= -\eta \frac{1}{\sqrt{\varepsilon + \operatorname{diag}(\hat{\mathbf{s}}_t)}} \odot \hat{\mathbf{m}}_t;\\
\boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t + \Delta\boldsymbol{\theta}_t.
\end{aligned}$$

44: In practice, we correct $\mathbf{m}_t$ and $\mathbf{s}_t$ to prevent bias towards smaller values [23].

Despite its popularity and general effectiveness, Adam is not without shortcomings; there are cases where Adam diverges due to **inadequate variance control** [32].

31.4.7 Yogi

Yogi, a variant of Adam designed to improve stability in such cases, was introduced to address this problem [46]. The only difference is that the Adam update for $\mathbf{s}_t$ is replaced by

$$\mathbf{s}_t = \mathbf{s}_{t-1} + (1-\beta_2)\big(\mathbf{g}_t \odot \mathbf{g}_t\big) \odot \operatorname{sgn}\!\big((\mathbf{g}_t \odot \mathbf{g}_t) - \mathbf{s}_{t-1}\big),$$

where $\mathbf{g}_t \odot \mathbf{g}_t$ is the Hadamard **element-wise square** of the gradient vector $\mathbf{g}_t$.

In practice, of course, we would not typically implement any of the algorithms of this section from scratch. In the next four sections, we discuss commonly used specialized **deep learning architectures:** CNN), RNM), GAM), and Autoencoders.

31.5 Convolutional Neural Networks

Over the years, we have developed deep learning architectures tailored to particular data types; **convolutional neural networks** (CNN) are thought to be particularly effective when working with **image data**.

By design, they can automatically detect and learn patterns from raw pixels, such as **edges**, **textures**, and **shapes**. These patterns are combined at increasing levels of abstraction to enable complex tasks such as **object recognition** or **image segmentation**.[45]

45: Important references for this section include [2, 3, 11, 19, 36, 42, 48, 49].

CNN have been used, among other things, for:

- **image classification** (e.g., distinguishing cats from dogs);
- **object detection** (e.g., identifying pedestrians);
- **medical imaging analysis** (e.g., locating tumors in MRI scans);
- **facial recognition** (e.g., smartphone security);
- **content moderation** (e.g., detecting inappropriate or harmful images in online platforms);
- **artistic style transfer**, and **image synthesis/generation**.

CNN are widely used in **computer vision**, where understanding spatial structure is essential; consequently, we will revisit (and expand on) some of this section's notions in Chapter 35.

31.5.1 What is Convolution?

People who are active on social media often use **filters** on their pictures before posting them. These filters apply different effects to the photo, such as blurring, making the image black and white, or adding cartoon-like effects.

When we apply a filter to a photo, we are in fact using **convolution**, an operation that aggregates the **element-wise product** of an **input data tensor** with a (typically much smaller) **kernel tensor**, and places the outcome in a **feature map**. In the context of image processing, for instance, the original image is the input data, the filter is the kernel, and the filtered image is the feature map.

This is illustrated conceptually in the figure below.

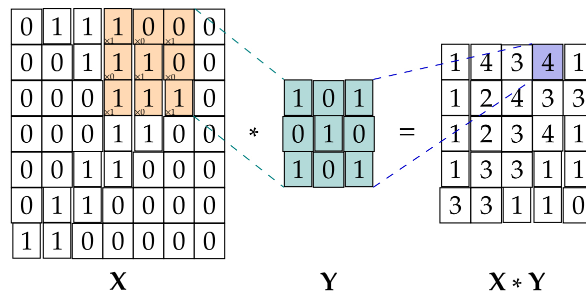


Figure 31.7: The convolution of an input X and kernel Y [33].

Formally, a 1D **discrete** convolution is denoted by

$$\underbrace{s(i)}_{\text{feature map}} = (x * y)(i) = \sum_m \underbrace{x(m)}_{\text{input data}} \underbrace{y(i-m)}_{\text{kernel}};$$

a 1D **continuous** convolution would instead be defined by

$$s(i) = (x * y)(i) = \int_m x(m)y(i-m)dm,$$

where m and i are continuous variables.[46]

46: For the most part, however, we will be primarily concerned with the discrete case.

One of the most common applications of CNN is in **digital image processing**; in the case of a black and white image, the input data is a **tensor of order 2**. Discrete convolution in **two dimensions** can be performed as follows [11]:

$$(\mathbf{X} * \mathbf{K})(i,j) = \sum_m \sum_n X_{m,n} K_{i-m,j-n};$$

since convolution is commutative, the following also holds:

$$(\mathbf{K} * \mathbf{X})(i,j) = \sum_m \sum_n X_{i-m,j-n} K_{m,n}.$$

We see that the two-dimensional discrete convolution is equivalent to **reversing** the row and column indices of the kernel, performing **element-wise multiplication**, and taking the sum of the products.

In the context of deep learning, the term "convolution" often refers to a similar operation called **cross-correlation**; this is equivalent to convolution without the reversing of the matrix indices [11]:

$$(\mathbf{K} \boxtimes \mathbf{X})(i,j) = \sum_m \sum_n X_{i+m,j+n} K_{m,n}.$$

31.5.2 How Convolution is Used in Deep Neural Networks

Basically, a CNN kernel (filter) is a **small sliding matrix of weights** that travels over the input data; during training, the weights of each kernel are **adjusted** so that it becomes **sensitive** to a particular type of **pattern** or **structure** in the input.

Because **each kernel** can only capture a **single** kind of feature,[47] **convolutional layers** (see Section 31.5.3) use **multiple kernels** in parallel to extract **sets of features** from the input. The resulting set of **feature maps** provides a **representation** encoding many aspects of the input data.

47: This means that each kernel becomes **specialized**, or effective at recognizing only one feature category at a time; due to their limited size, each kernel responds strongly to a **specific spatial pattern** and weakly (or not at all) to others.

For example, in the **early layers** of a CNN trained on photographs, some kernels may specialize in detecting **vertical edges**, others **horizontal lines**, other **diagonal lines**, and yet another **colour gradients**, say.

The same principle applies to the network's **deeper layers**, but the features become more **abstract** and **complex**. Early kernels detect **basic** edges or textures; later kernels detect **parts of objects** (e.g., eyes, wheels, facial contours) or **entire shapes**.

To control how convolution behaves and how efficiently it runs, CNN rely on three key design choices:

- the **stride** determines how far the kernel moves across the input during each step, as illustrated in Figures 31.8 and 31.9;[48]

48: A stride of 1 means the kernel moves one pixel at a time; a stride of 2 skips every other pixel. Larger strides reduce the output size and computational cost but may result in a loss of detail.

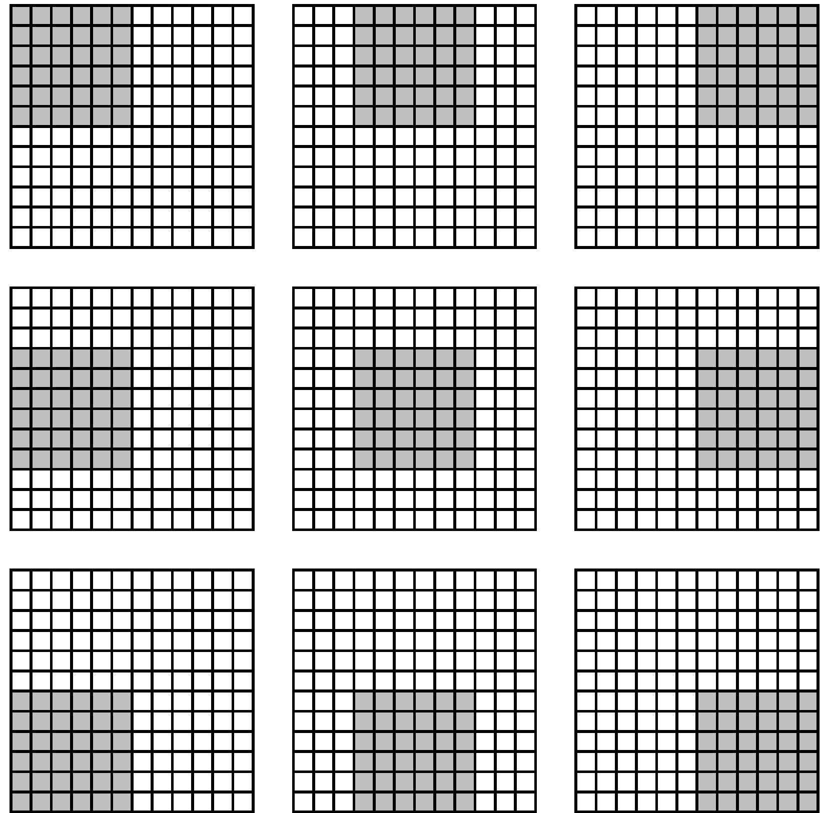


Figure 31.8: Stride of 3 skips; kernel in grey.

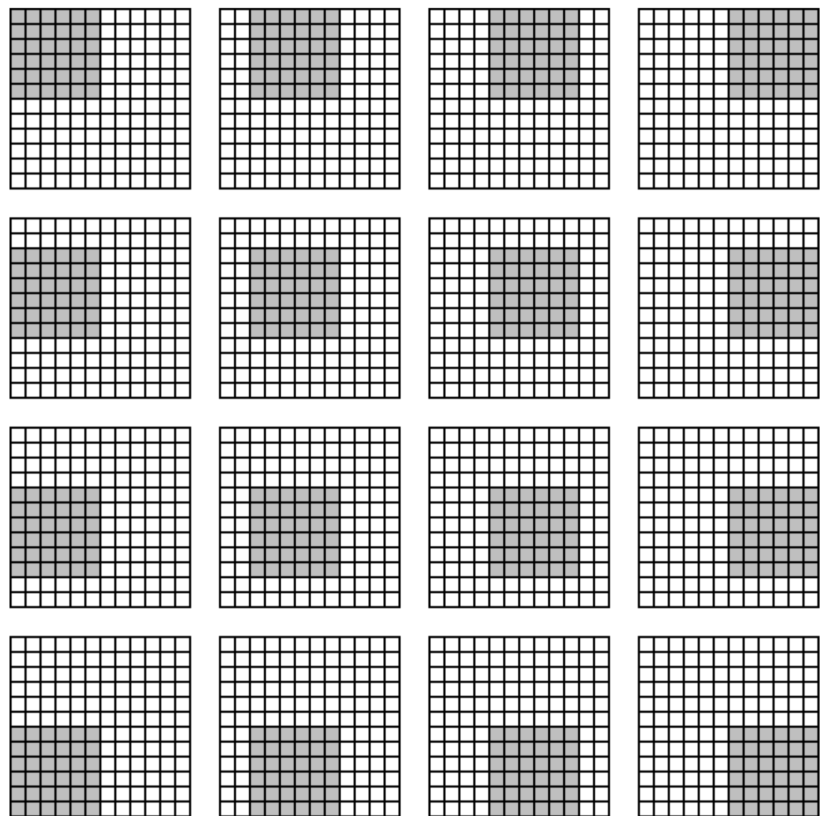


Figure 31.9: Stride of 2 skips; kernel in grey.

- the **padding** adds zeros around the border of the input to preserve spatial dimensions, as illustrated in Figure 31.10; [49]

49: Without padding, each convolution reduces the output size, potentially discarding edge information too early.

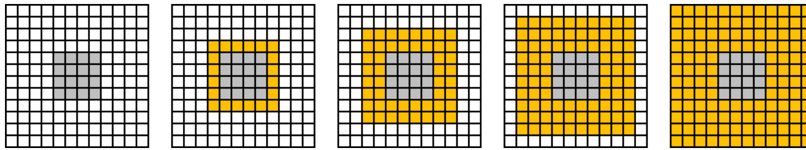


Figure 31.10: Various kernel paddings; kernel in grey, padding zeros in orange.

- the **kernel size** determines the spatial extent of the region each kernel examines, as illustrated in Figure 31.11.[50]

50: Smaller kernels (e.g., 3×3) capture fine details, while larger kernels (e.g., 7×7) detect broader patterns but are more computationally expensive.

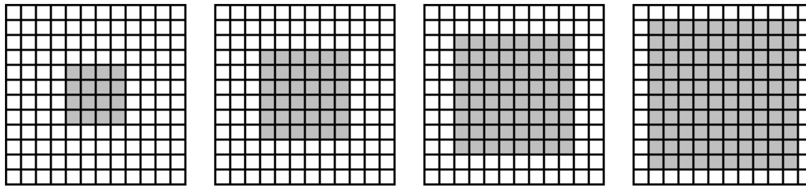


Figure 31.11: Various kernel sizes in grey.

These components are combined with **nonlinear activation functions** (e.g., ReLU) and **pooling layers** to build deep, hierarchical representations of the input.

31.5.3 The Convolutional Layer

A convolutional layer has three steps [11]:

1. several parallel convolutions yield a set of **linear activations**;
2. each linear activation is passed through a **nonlinear activation function** (such as ReLU),[51]
3. a **pooling function** replaces the output at a given location with a summary statistic of nearby outputs (one common pooling function is the **max pooling** function, which provides the maximum output within a rectangular neighbourhood); pooling is useful when we only care about the presence and/or absence of a feature and not its specific location or when we are processing different-sized inputs.[52]

51: As mentioned previously, this introduces nonlinearity and gives the network more flexibility.

52: It also helps compress the data size into manageable chunks.

The output of each layer is then used as input to the next layer.

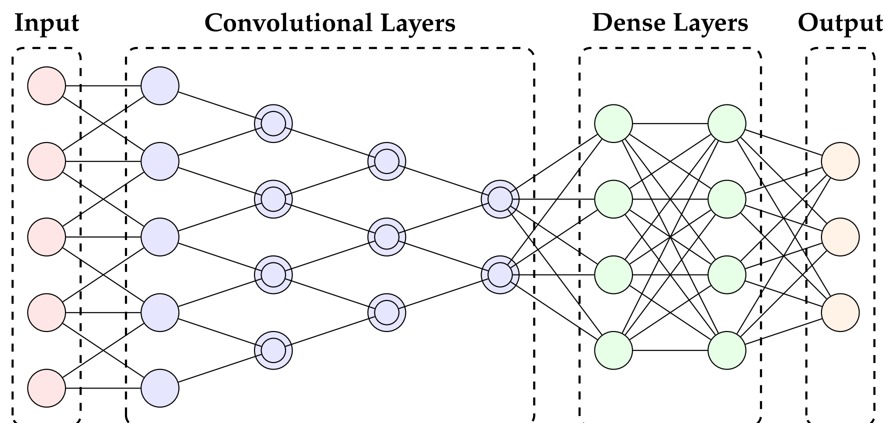


Figure 31.12: A deep convolutional network with four convolutional layers and two dense layers; the double-circle nodes represent pooling nodes [25].

31.5.4 Image Classification Example

We use convolutional neural networks to perform **image classification** on the Rock, Paper, Scissors dataset (available with TensorFlow).[53]

53: This example uses TensorFlow and Keras, but we will provide other examples using PyTorch, the dominant Python framework for machine learning.

We import the appropriate modules and set a reproducibility seed.

```
import tensorflow as tf
from tensorflow.keras import layers, models, Input
from tensorflow.keras.callbacks import EarlyStopping
import tensorflow_datasets as tfds
import matplotlib.pyplot as plt
import numpy as np

tf.keras.utils.set_random_seed(42)
```

The dataset is only partitioned into training and test sets, so we set aside 30% of the training set as a validation set.[54]

54: Typical images look like

The task is to classify an image as **rock** (0), **paper** (1), or **scissors** (2).

```
(ds_train, ds_val, ds_test) = tfds.load(
    'rock_paper_scissors',
    split=['train[:70%]','train[70%:]','test'],
    shuffle_files=True,
    as_supervised=True
)
```

Next, we normalize the images by dividing the RGB channels by 255 (so each channel is represented by a value in $[0, 1]$) and we resize each image to 128×128. We also shuffle the training data to ensure that the model doesn't learn the order of the images, and batch the training, validation, and test data.

```
def preprocess(image, label):

    # normalize images
    image = tf.cast(image, tf.float32) / 255.0

    # resize images
    image = tf.image.resize(image, [128, 128])

    return image, label

BATCH_SIZE = 32
SHUFFLE_BUFFER_SIZE = 1000

# shuffle so the model doesn't learn the order of the training data
ds_train = ds_train.map(preprocess).shuffle(SHUFFLE_BUFFER_SIZE).batch(BATCH_SIZE)
ds_val = ds_val.map(preprocess).batch(BATCH_SIZE)
ds_test = ds_test.map(preprocess).batch(BATCH_SIZE)
```

We start with one convolutional layer with 16 filters, ReLU activations, a 3×3 kernel, and max pooling, using Adam optimization.

```
input_shape = (128, 128, 3)

model = models.Sequential([
    layers.Input(shape=input_shape),

    # convolution step
    layers.Conv2D(16, (3, 3), activation='relu'),

    # pooling step
    layers.MaxPooling2D((2, 2)),

    # flattening for classification
    layers.Flatten(),

    # dense layer before final classification layer
    layers.Dense(128, activation='relu'),

    layers.Dense(3)  # 3 classes
])


model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy']
              )
```

Before fitting the model, we add an early stop mechanism to prevent overfitting. We then train the model over 10 epochs.

```
early_stop = EarlyStopping(
    monitor='val_loss',
    min_delta=0.001,
    patience=3,
    verbose=1,
    mode='auto',
    restore_best_weights=True
)

history = model.fit(ds_train,
                    epochs=10,
                    validation_data=(ds_val),
                    callbacks=[early_stop]
                    )
```

```
Epoch 1/10
56/56 - 10s 85ms/step - loss: 1.0240 - accuracy: 0.7137 - val_loss: 0.2209 - val_accuracy: 0.9563
Epoch 2/10
56/56 - 5s 61ms/step - loss: 0.0743 - accuracy: 0.9881 - val_loss: 0.0240 - val_accuracy: 0.9947
Epoch 3/10
56/56 - 4s 37ms/step - loss: 0.0131 - accuracy: 0.9989 - val_loss: 0.0148 - val_accuracy: 0.9947
Epoch 4/10
56/56 - 4s 37ms/step - loss: 0.0045 - accuracy: 1.0000 - val_loss: 0.0062 - val_accuracy: 0.9987
```

```
Epoch 5/10
56/56 - 5s 58ms/step - loss: 0.0020 - accuracy: 1.0000 - val_loss: 0.0042 - val_accuracy: 1.0000
Epoch 6/10
56/56 - 4s 38ms/step - loss: 0.0013 - accuracy: 1.0000 - val_loss: 0.0039 - val_accuracy: 1.0000
Epoch 7/10
56/56 - 4s 38ms/step - loss: 0.0010 - accuracy: 1.0000 - val_loss: 0.0041 - val_accuracy: 0.9987
Epoch 8/10
53/56 - ETA: 0s - loss: 8.3814e-04 - accuracy: 1.0000

Restoring model weights from the end of the best epoch: 5.
56/56 - 5s 59ms/step - loss: 8.2481e-04 - accuracy: 1.0000 - val_loss: 0.0032 - val_accuracy: 0.9987
Epoch 8: early stopping
```

The training accuracy quickly shoots up to 100% and the validation accuracy hovers around 99%, which just screams “overfitting”!

Let’s evaluate the model on the test data.

```
model.evaluate(ds_test)
```

```
12/12 - 1s 104ms/step - loss: 1.3345 - accuracy: 0.7151
[1.3345420360565186, 0.7150537371635437]
```

As expected, the model does not perform that well on new data. We repeat the experience, but this time, we add a second convolutional layer.

```
model = models.Sequential([
    layers.Input(shape=input_shape),

    # first convolution step
    layers.Conv2D(16, (3, 3), activation='relu'),

    # first pooling step
    layers.MaxPooling2D((2, 2)),

    # second convolution step
    layers.Conv2D(32, (3, 3), activation='relu'),

    # second pooling step
    layers.MaxPooling2D((2, 2)),

    # flattening for classification
    layers.Flatten(),

    # dense layer before final classification layer
    layers.Dense(128, activation='relu'),

    layers.Dense(3)  # 3 classes
])

model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
```

```
history = model.fit(ds_train,
                    epochs=10,
                    validation_data=(ds_val),
                    callbacks=[early_stop]
                    )
```

```
Epoch 1/10
56/56 - 6s 43ms/step - loss: 1.4043 - accuracy: 0.5890 - val_loss: 0.4816 - val_accuracy: 0.9206
Epoch 2/10
56/56 - 5s 54ms/step - loss: 0.2150 - accuracy: 0.9552 - val_loss: 0.0609 - val_accuracy: 0.9881
Epoch 3/10
56/56 - 4s 38ms/step - loss: 0.0441 - accuracy: 0.9909 - val_loss: 0.0182 - val_accuracy: 0.9987
Epoch 4/10
56/56 - 4s 39ms/step - loss: 0.0143 - accuracy: 0.9977 - val_loss: 0.0076 - val_accuracy: 0.9987
Epoch 5/10
56/56 - 5s 60ms/step - loss: 0.0053 - accuracy: 0.9994 - val_loss: 0.0030 - val_accuracy: 1.0000
Epoch 6/10
56/56 - 4s 41ms/step - loss: 0.0022 - accuracy: 1.0000 - val_loss: 0.0019 - val_accuracy: 1.0000
Epoch 7/10
56/56 - 4s 39ms/step - loss: 0.0016 - accuracy: 1.0000 - val_loss: 0.0013 - val_accuracy: 1.0000
Epoch 8/10
56/56 - 5s 60ms/step - loss: 0.0009 - accuracy: 1.0000 - val_loss: 0.0011 - val_accuracy: 1.0000
Epoch 9/10
56/56 - 4s 38ms/step - loss: 0.0006 - accuracy: 1.0000 - val_loss: 0.0008 - val_accuracy: 1.0000
Epoch 10/10
56/56 - 4s 38ms/step - loss: 0.0005 - accuracy: 1.0000 - val_loss: 0.0007 - val_accuracy: 1.0000
```

And how does it perform on the test data?

```
model.evaluate(ds_test)
```

```
12/12 - 0s 16ms/step - loss: 1.0663 - accuracy: 0.7634
[1.0662583112716675, 0.7634408473968506]
```

An improvement, but still not amazing. In the first exercise of section 31.9, we ask learners to implement a 3rd and 4th additional convolutional layers, to see if we can further improve the CNN's performance.

On the test data, the model with 3 convolutional layers performs roughly as well as the models with 1 and 2 convolutional layers:

```
12/12 - 0s 10ms/step - loss: 1.1395 - accuracy: 0.7930
[1.1395031213760376, 0.7930107712745667]
```

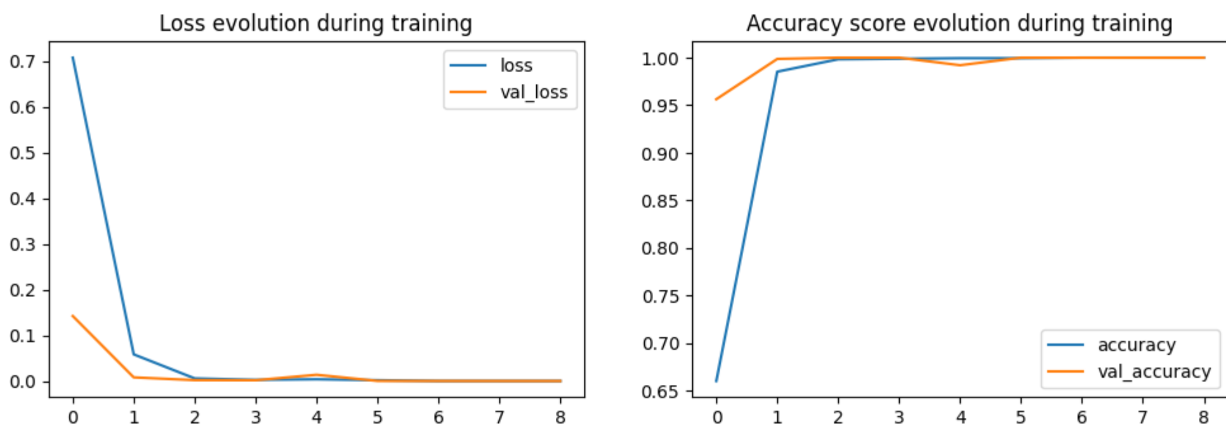
The model with 4 convolutional layers is much better, however:

```
12/12 - 0s 10ms/step - loss: 0.1825 - accuracy: 0.9409
[0.1824917197227478, 0.9408602118492126]
```

Now that we have crossed an acceptable accuracy threshold, let's plot the **learning curves** for the 4-layer model.[55]

55: A learning curve is a plot that shows how a model's performance changes over time (or training progress), typically as a function of the number of **epochs** or **iterations**.

```
def plot_learning_curves(history):
    plt.figure(figsize=(12, 8))

    plt.subplot(2, 2, 1)
    plt.plot(history.history['loss'], label='loss')
    plt.plot(history.history['val_loss'], label='val_loss')
    plt.title('Loss evolution during training')
    plt.legend()

    plt.subplot(2, 2, 2)
    plt.plot(history.history['accuracy'], label='accuracy')
    plt.plot(history.history['val_accuracy'], label='val_accuracy')
    plt.title('Accuracy score evolution during training')
    plt.legend();

plot_learning_curves(history)
```



We can get a sense for where the model fails and where it succeeds by displaying some of its correct and incorrect predictions.

```
test_images, test_labels = [], []
for images, labels in ds_test:
    test_images.append(images.numpy())
    test_labels.append(labels.numpy())
test_images = np.concatenate(test_images)
test_labels = np.concatenate(test_labels)

predictions = model.predict(test_images)
predicted_labels = np.argmax(predictions, axis=1)

# identify correct and incorrect predictions
correct_predictions = predicted_labels == test_labels
incorrect_predictions = predicted_labels != test_labels

# get indices for correct and incorrect predictions
correct_indices = np.where(correct_predictions)[0]
incorrect_indices = np.where(incorrect_predictions)[0]
```

```
# plot correct predictions
plot_predictions(test_images, test_labels, predicted_labels, correct_indices,
                    title="Correct Predictions")

# plot incorrect predictions
plot_predictions(test_images, test_labels, predicted_labels, incorrect_indices,
                    title="Incorrect Predictions")
```

Look at the images that were classified incorrectly: it seems that the model's most significant weakness is misclassifying scissors as paper. Do you notice anything about these scissors images compared to the scissors and paper images in the correct predictions? How could the model potentially be improved?

31.6 Recurrent Neural Networks

To this point, we have been working with **feed-forward networks** (FFNNN), that is, deep networks whose computation flows in a single direction: **forward**. These networks are limited in that they can only take in and output sequences of a **predetermined size**.[56]

56: Such as images with the same dimensions, say.

In certain applications, we need networks that can handle inputs of **varying lengths** that arise in **sequential data**,[57] such as text, speech, video, and time series data, say.

57: This refers to any data where the **order of the elements matters**; the **sequence itself** conveys important information that would be lost if the data were shuffled or randomly selected into a training dataset. Sequential data exhibits **dependencies across time or position**.

Recurrent neural networks (RNN) are designed to handle such sequential data. They incorporate **memory** of previous inputs, making them suitable for applications where the order and temporal dynamics of the data are important.[58]

58: Important references for this section include [5, 11, 23, 37, 38, 48].

RNN have been used, among other things, for:

- **speech recognition** (e.g., transcribing spoken words)
- **text generation** (e.g., predictive typing or creative writing tools)
- **machine translation** (e.g., English to French)
- **time-series forecasting** (e.g., predicting stock prices)
- **sequential generation** (e.g., chatbots or composing music)

In the 2010s, RNN were **widely used** in **natural language processing** (NLP), where understanding **context** over a sequence of words or events is critical (see Chapter 32).[59]

59: Variants such as **long short-term memory** networks (LSTM) and **gated recurrent units** (GRUs), which addressed limitations linked to vanishing or exploding gradients, were particularly popular [8, 15]. Since 2018, however, the emergence of **transformers** has shifted the playground; models such as **BERT**, **GPT**, and **T5** have largely replaced RNN in state-of-the-art NLP applications [29, 7, 27, 31, 10, 30, 40, 43].

In a RNN, the **output** of a node is used as an input to the **next layer**, as in a FFNNN, but it may also be used as an input to the **same node at the next time step** t [5]. More formally, a node's **hidden state** $\mathbf{h}_t$ at time t is a function of its new inputs $\mathbf{X}_t$ and its previous hidden state $\mathbf{h}_{t-1}$ [11].

Depending on the type of RNN, the hidden state may or may not be equal to the output at the previous time step:

$$\mathbf{h}_t = f(\mathbf{h}_{t-1}, \mathbf{X}_t, \boldsymbol{\theta}).$$

RNN are trained *via* **backpropagation through time**, which involves "unrolling" the network across time steps and performing backpropagation as described in Section 21.4.3.

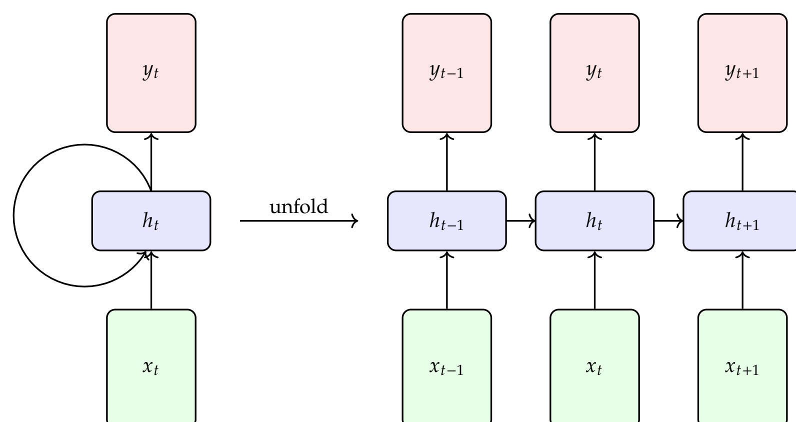


Figure 31.13: "Unrolling" in a recurrent network.

The four main types of RNN are **sequence-to-sequence**, **sequence-to-vector**, **vector-to-sequence**, and **encoder-decoder**.

31.6.1 Vector-to-Sequence RNN

Vector-to-sequence RNN are used to generate sequences of variable lengths from vectors of a fixed length. At each time step t, we take a sample of the outputs from $\mathbf{h}_t$ and use these as inputs for the next time step to get $\mathbf{h}_{t+1}$. This can be expressed using the **conditional generative model**, where T is the length of the output sequence [23]:

$$
\begin{aligned}
p(\mathbf{y}_{1:T} \mid \mathbf{X}) &= \sum_{\mathbf{h}_{1:T}} p(\mathbf{y}_{1:T}, \mathbf{h}_{1:T} \mid \mathbf{X}) \\
&= \sum_{\mathbf{h}_{1:T}} \prod_{t=1}^{T} p(\mathbf{y}_t \mid \mathbf{h}_t) \times p(\mathbf{h}_t \mid \mathbf{h}_{t-1}, \mathbf{y}_{t-1}, \mathbf{X}).
\end{aligned}
$$

The **hidden state** conditional probability is:

$$
\begin{aligned}
p(\mathbf{h}_t \mid \mathbf{h}_{t-1}, \mathbf{y}_{t-1}, \mathbf{X}) &= \mathcal{I}(\mathbf{h}_t = f(\mathbf{h}_{t-1}, \mathbf{y}_{t-1}, \mathbf{X})) \\
&= \begin{cases} 1, & \text{if } \mathbf{h}_t = f(\mathbf{h}_{t-1}, \mathbf{y}_{t-1}, \mathbf{X}) \\ 0, & \text{if } \mathbf{h}_t \neq f(\mathbf{h}_{t-1}, \mathbf{y}_{t-1}, \mathbf{X}), \end{cases}
\end{aligned}
$$

and the **hidden state update** is $\mathbf{h}_t = \varphi(\mathbf{w}_{xh}[\mathbf{X}; \mathbf{y}_{t-1}] + \mathbf{w}_{hh}\mathbf{h}_{t-1} + \mathbf{b}_h)$, where φ is the activation function.

For **categorical** outputs, we take

$$p(\mathbf{y}_t \mid \mathbf{h}_t) = \text{Categorical}(\mathbf{y}_t \mid \text{softmax}(\mathbf{w}_{hy}\mathbf{h}_t + \mathbf{b}_y));$$

for **continuous** outputs, we take

$$p(\mathbf{y}_t \mid \mathbf{h}_t) = \mathcal{N}(\mathbf{y}_t \mid \mathbf{w}_{hy}\mathbf{h}_t + \mathbf{b}_y, \sigma^2 \mathbf{I}).$$

31.6.2 Sequence-to-Vector RNN

Sequence-to-vector RNN process input sequences of variable length and produce a single output vector of fixed length. Unlike vector-to-sequence models, they receive new input at each time step but **suppress** the output until the **final step**.

This architecture is commonly used for tasks such as **classification**,[60] where the goal is to assign a single label to an entire input sequence. For example, a sequence-to-vector model can be used to classify emails as either **ham** (legitimate) or **spam**.

60: See [6, ch. 21].

The model can be expressed as a **conditional generative model** [23], where the output depends only on the final hidden state $\mathbf{h}_T$:

$$p(\mathbf{y} \mid \mathbf{X}_{1:T}) = \text{Categorical}(\mathbf{y} \mid \text{softmax}(\mathbf{w}\mathbf{h}_T + \mathbf{b})).$$

31.6.3 Sequence-to-Sequence RNN

Sequence-to-sequence RNN are used to model problems where both the input and output are sequences, typically of the same length. These models process an input sequence step by step and produce a corresponding output at each time step. A common example is time-aligned

sequence transformation, such as phoneme-to-phoneme conversion or aligned language translation.

Assuming the input and output sequences have equal length T, the model can be described using a **conditional generative model** [23]:

$$p(\mathbf{y}_{1:T} \mid \mathbf{x}_{1:T}) = \sum_{\mathbf{h}_{1:T}} \prod_{t=1}^{T} p(\mathbf{y}_t \mid \mathbf{h}_t) \times \mathcal{I}(\mathbf{h}_t = f(\mathbf{h}_{t-1}, \mathbf{x}_t)).$$

The **initial hidden state** is typically computed as

$$\mathbf{h}_1 = f(\mathbf{h}_0, \mathbf{x}_1) = f_0(\mathbf{x}_1).$$

31.6.4 Encoder-Decoder Models

In many applications, the input and output sequences of a sequence-to-sequence model are not of equal length. To handle this more general case, we use an **encoder-decoder architecture**. The **encoder** is a sequence-to-vector RNN that processes the input sequence and compresses it into a fixed-length representation. This vector is then passed to the **decoder**, a vector-to-sequence RNN that generates the output sequence one element at a time.

The encoder transforms the input sequence $\mathbf{X}_{1:T}$ into a final hidden state $\mathbf{h}_T^{\text{enc}}$, which is then used to initialize the hidden state of the decoder:

$$\mathbf{h}_1^{\text{dec}} = \mathbf{h}_T^{\text{enc}}.$$

The decoder then produces the output sequence $\mathbf{y}_{1:T'}$, where T' may differ from T, by iteratively computing:

$$\mathbf{h}_t^{\text{dec}} = f(\mathbf{h}_{t-1}^{\text{dec}}, \mathbf{y}_{t-1}), \quad p(\mathbf{y}_t \mid \mathbf{h}_t^{\text{dec}}) = \text{Categorical}(\mathbf{y}_t \mid \text{softmax}(\mathbf{w}\mathbf{h}_t^{\text{dec}} + \mathbf{b}_t)).$$

31.6.5 Bi-Directional RNN

In some tasks, it is beneficial for the output at time step t to depend on both **past** and **future** inputs. To capture this full temporal context, we use a **bi-directional RNN**, which consists of two separate RNN: a **forward RNN** that processes the sequence from **left to right**, and a **backward RNN** that processes the sequence from **right to left** [5, 23].

At each time step t, the **forward** and **backward** hidden states are computed as:

$$\mathbf{h}_t^{\rightarrow} = \varphi(\mathbf{w}_{xh}^{\rightarrow}\mathbf{X}_t + \mathbf{w}_{hh}^{\rightarrow}\mathbf{h}_{t-1}^{\rightarrow} + \mathbf{b}_h^{\rightarrow}),$$
$$\mathbf{h}_t^{\leftarrow} = \varphi(\mathbf{w}_{xh}^{\leftarrow}\mathbf{X}_t + \mathbf{w}_{hh}^{\leftarrow}\mathbf{h}_{t+1}^{\leftarrow} + \mathbf{b}_h^{\leftarrow}).$$

The final hidden representation at time t is obtained by concatenating the two directions:

$$\mathbf{h}_t = [\mathbf{h}_t^{\rightarrow}, \mathbf{h}_t^{\leftarrow}].$$

31.6.6 Long-Term Memory

Standard RNN often struggle to retain information from far in the past due to the **vanishing gradient problem**.

To address this limitation, specialized recurrent units have been developed that allow the network to learn which information to **retain for the long term** and which to **discard** [5].

Long Short-Term Memory (LSTM) Units

LSTM units extend the standard RNN architecture by introducing a separate **memory cell** that enables long-term information storage [23].

The hidden state $\mathbf{H}_t$ represents **short-term memory**, while the memory cell $\mathbf{C}_t$ serves as **long-term memory**. Information flow is controlled by three gates:

- the **input gate** $\mathbf{I}_t$ controls which new information is added to the memory;
- the **forget gate** $\mathbf{F}_t$ determines what information to remove, and
- the **output gate** $\mathbf{O}_t$ determines what part of the memory is output.

The update equations for LSTM units are:

$$\begin{aligned}
\mathbf{I}_t &= \sigma(\mathbf{X}_t\mathbf{w}_{xi} + \mathbf{H}_{t-1}\mathbf{w}_{hi} + \mathbf{b}_i) && \text{input gate}\\
\mathbf{F}_t &= \sigma(\mathbf{X}_t\mathbf{w}_{xf} + \mathbf{H}_{t-1}\mathbf{w}_{hf} + \mathbf{b}_f) && \text{forget gate}\\
\mathbf{O}_t &= \sigma(\mathbf{X}_t\mathbf{w}_{xo} + \mathbf{H}_{t-1}\mathbf{w}_{ho} + \mathbf{b}_o) && \text{output gate}\\
\tilde{\mathbf{C}}_t &= \tanh(\mathbf{X}_t\mathbf{w}_{xc} + \mathbf{H}_{t-1}\mathbf{w}_{hc} + \mathbf{b}_c) && \text{candidate memory cell}\\
\mathbf{C}_t &= \mathbf{F}_t * \mathbf{C}_{t-1} + \mathbf{I}_t * \tilde{\mathbf{C}}_t && \text{updated memory cell}\\
\mathbf{H}_t &= \mathbf{O}_t * \tanh(\mathbf{C}_t) && \text{final hidden state}
\end{aligned}$$

where σ is an appropriate activation function.

Gated Recurrent Units (GRUs)

GRUs are a simplified alternative to LSTM units that eliminate the need for a separate memory cell [5, 23]. Instead of distinct input and forget gates, GRUs use a single **update gate** $\mathbf{Z}_t$. A **reset gate** $\mathbf{R}_t$ is used to control how much of the previous hidden state contributes to the candidate activation. The output gate is omitted entirely.

The GRU update equations are:

$$\begin{aligned}
\mathbf{R}_t &= \sigma(\mathbf{X}_t\mathbf{w}_{xr} + \mathbf{H}_{t-1}\mathbf{w}_{hr} + \mathbf{b}_r) && \text{reset gate}\\
\mathbf{Z}_t &= \sigma(\mathbf{X}_t\mathbf{w}_{xz} + \mathbf{H}_{t-1}\mathbf{w}_{hz} + \mathbf{b}_z) && \text{update gate}\\
\tilde{\mathbf{H}}_t &= \tanh(\mathbf{X}_t\mathbf{w}_{xh} + (\mathbf{R}_t * \mathbf{H}_{t-1})\mathbf{w}_{hh} + \mathbf{b}_h) && \text{candidate hidden state}\\
\mathbf{H}_t &= \mathbf{Z}_t * \mathbf{H}_{t-1} + (1 - \mathbf{Z}_t) * \tilde{\mathbf{H}}_t && \text{final hidden state}
\end{aligned}$$

Conceptual illustrations of these architectures (and many more besides) can be found on p. 2076.

31.6.7 Music Generation Example

In this example, we will train a RNN with a LSTM layer on MIDI data,[¶] and then use the trained network to generate new music. First, we import the required dependencies and set a seed to ensure reproducibility.

```
import os
import glob
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
import pretty_midi
from torch.utils.data import Dataset, DataLoader, random_split
from lightning.pytorch.callbacks.early_stopping import EarlyStopping
import lightning as L
import torch.nn.functional as F

L.seed_everything(seed=42)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False
```

The MIDI data will be represented as a sequence of **notes**, each characterized by three features:

- **pitch**, a categorical variable with 128 classes corresponding to the standard MIDI pitch values;
- **step**, a continuous variable indicating the number of beats between the onset of the current note and the previous one,[61]
- **duration**, a continuous variable representing how long the note is held, quantized using the same resolution as above.

61: Following music theory conventions, this is quantized to a resolution of 0.125 beats.

The zipped folder `classical-piano.zip` contains 75 MIDI tracks (see the DUDADS' GitHub repository).[62]

62: Data handling for this example is more complicated; data management **classes** are provided in order to provide complete instructions, but this step may be skipped if we are more interested in the network **implementation** itself.

The `MidiDataset` class loads all of the MIDI files in a given directory into a dataset. For each file, overlapping instrument tracks are combined into one acoustic grand piano track. Pitches and durations are quantized, and the sequences are concatenated.

```
class MidiDataset(Dataset):
    def __init__(self, midi_dir: str, seq_length: int, quantize_step=0.125,
        quantize_duration=0.125, vocab_size=128):
        self.midi_dir = midi_dir
        self.seq_length = seq_length
        self.quantize_step = quantize_step
        self.quantize_duration = quantize_duration
        self.vocab_size = vocab_size
        self.data = self.load_and_process_data()
```

[¶] **Musical instrument digital interface** (MIDI) is a standard protocol and file format for transmitting musical instructions, such as pitch, timing, and velocity, between electronic instruments, computers, and audio devices.

```
    def load_and_process_data(self):
        midi_files = glob.glob(os.path.join(self.midi_dir, '*.mid'))
        data = []
        for midi_file in midi_files:
            merged_midi_data = self.merge_tracks(midi_file)
            notes = sorted(merged_midi_data.instruments[0].notes,
                            key=lambda note: note.start)
            processed_notes = self.process_notes(notes)
            for i in range(len(processed_notes) - self.seq_length):
                data.append(processed_notes[i:i + self.seq_length + 1])
        return data

    def merge_tracks(self, file_path):
        midi_data = pretty_midi.PrettyMIDI(file_path)
        merged_midi = pretty_midi.PrettyMIDI()
        merged_instrument = pretty_midi.Instrument(program=0)

        all_notes = []
        for instrument in midi_data.instruments:
            all_notes.extend(instrument.notes)

        sorted_notes = sorted(all_notes, key=lambda note: note.start)
        merged_instrument.notes.extend(sorted_notes)
        merged_midi.instruments.append(merged_instrument)

        return merged_midi

    def process_notes(self, notes):
        processed = []
        for i in range(1, len(notes)):
            pitch = notes[i].pitch
            step = round((notes[i].start - notes[i-1].start) / self.quantize_step) *
                            self.quantize_step
            duration = round((notes[i].end - notes[i].start) / self.quantize_duration) *
                            self.quantize_duration
            processed.append([pitch, step, duration])
        return processed

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        sequence = self.data[idx]
        inputs = torch.tensor(sequence[:-1], dtype=torch.float32)
        pitch_target = torch.tensor(sequence[-1][0], dtype=torch.long)
        step_target = torch.tensor(sequence[-1][1], dtype=torch.float32)
        duration_target = torch.tensor(sequence[-1][2], dtype=torch.float32)
        return inputs, pitch_target, step_target, duration_target
```

The `MidiDataModule` class allows us to handle the data with Lightning, and handles the train and validation splits.

Why aren't we using a test set? The goal is not to predict a "correct" outcome; rather, we are trying to generate outputs that sound "nice".[63] A test set is therefore not necessary.

63: Or at the very least, "interesting".

```
class MidiDataModule(L.LightningDataModule):
    def __init__(self, midi_dir: str, seq_length: int, batch_size: int, val_split=0.2,
                quantize_step=0.125, quantize_duration=0.125):
        super().__init__()
        self.midi_dir = midi_dir
        self.seq_length = seq_length
        self.batch_size = batch_size
        self.val_split = val_split
        self.quantize_step = quantize_step
        self.quantize_duration = quantize_duration

    def setup(self, stage=None):
        full_dataset = MidiDataset(self.midi_dir, self.seq_length, self.quantize_step,
                                    self.quantize_duration)
        val_size = int(len(full_dataset) * self.val_split)
        train_size = len(full_dataset) - val_size
        self.train_dataset, self.val_dataset = random_split(full_dataset, [train_size, val_size])

    def train_dataloader(self):
        return DataLoader(self.train_dataset, batch_size=self.batch_size, shuffle=True)

    def val_dataloader(self):
        return DataLoader(self.val_dataset, batch_size=self.batch_size)
```

We can now define the model class. Pitch is **categorical**, so we use **cross entropy** for the pitch loss. Step and duration are continuous, so for those we use **MSE**.

The first layer is an LSTM layer, followed by a **dropout layer** for **regularization**; each feature has a corresponding output layer. We also implement **weight decay** with the Adam optimizer.

```
class MIDIModel(L.LightningModule):
    def __init__(self, input_size, hidden_size, output_size, dropout_rate=0.1,
                weight_decay=1e-5):
        super(MIDIModel, self).__init__()

        # LSTM layer
        self.rnn = nn.LSTM(input_size, hidden_size, batch_first=True)

        # dropout layer
        self.dropout = nn.Dropout(dropout_rate)

        # pitch output
        self.fc_pitch = nn.Linear(hidden_size, output_size)

        # step output
        self.fc_step = nn.Linear(hidden_size, 1)
```

```
        # duration output
        self.fc_duration = nn.Linear(hidden_size, 1)

        # pitch loss
        self.loss_fn_pitch = nn.CrossEntropyLoss()

        # step and duration loss
        self.loss_fn_step_duration = nn.MSELoss()
        self.weight_decay = weight_decay

    def forward(self, x):
        rnn_out, _ = self.rnn(x)
        rnn_out = self.dropout(rnn_out)
        pitch_out = self.fc_pitch(rnn_out[:, -1, :])
        step_out = self.fc_step(rnn_out[:, -1, :])
        duration_out = self.fc_duration(rnn_out[:, -1, :])

        return {'pitch': pitch_out, 'step': step_out, 'duration': duration_out}

    def training_step(self, batch, batch_idx):
        inputs, pitch_target, step_target, duration_target = batch
        predictions = self(inputs)

        loss_pitch = self.loss_fn_pitch(predictions['pitch'], pitch_target)
        loss_step = self.loss_fn_step_duration(predictions['step'], step_target.unsqueeze(-1))
        loss_duration = self.loss_fn_step_duration(predictions['duration'],
                        duration_target.unsqueeze(-1))

        loss = loss_pitch + loss_step + loss_duration

        # log the training loss
        self.log('train_loss', loss, prog_bar=True, logger=True)

        return loss

    def validation_step(self, batch, batch_idx):
        inputs, pitch_target, step_target, duration_target = batch
        predictions = self(inputs)

        loss_pitch = self.loss_fn_pitch(predictions['pitch'], pitch_target)
        loss_step = self.loss_fn_step_duration(predictions['step'], step_target.unsqueeze(-1))
        loss_duration = self.loss_fn_step_duration(predictions['duration'],
                        duration_target.unsqueeze(-1))

        loss = loss_pitch + loss_step + loss_duration
        self.log('val_loss', loss, prog_bar=True, logger=True) # log the validation loss

        return loss

    def configure_optimizers(self):
        optimizer = optim.Adam(self.parameters(), lr=0.001, weight_decay=self.weight_decay)

        return optimizer
```

Next, we define the necessary variables and initialize the data module.

```
# replace with the path to the folder containing your
# MIDI files
midi_dir = "/my-midi-directory"
seq_length = 25
batch_size = 64
vocab_size = 128
input_size = 3
hidden_size = 128
output_size = vocab_size

data_module = MidiDataModule(midi_dir=midi_dir,
                             seq_length=seq_length,
                             batch_size=batch_size
                             )
```

We can now initialize and train the model.[64] Before choosing a seed sequence, we verify that it is not part of the training or validation set.

64: In the interest of preserving cloud computation resources, we only train for 20 epochs, but the **early stopping** callback would facilitate longer training if desired.

```
model = MIDIModel(input_size=input_size,
                  hidden_size=hidden_size,
                  output_size=output_size)

early_stop_callback = EarlyStopping(monitor="val_loss",
                                    min_delta=0.00,
                                    patience=3,
                                    verbose=False,
                                    mode="min")

trainer = L.Trainer(max_epochs=20,
                    callbacks=[early_stop_callback],
                    accelerator="gpu" if torch.cuda.is_available() else "cpu",
                    devices=1)

trainer.fit(model, data_module)
```

```
INFO: LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]
INFO: lightning.pytorch.accelerators.cuda: LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]
INFO:
  | Name                  | Type             | Params | Mode
-------------------------------------------------------------------
0 | rnn                   | LSTM             | 68.1 K | train
1 | dropout               | Dropout          | 0      | train
2 | fc_pitch              | Linear           | 16.5 K | train
3 | fc_step               | Linear           | 129    | train
4 | fc_duration           | Linear           | 129    | train
5 | loss_fn_pitch         | CrossEntropyLoss | 0      | train
6 | loss_fn_step_duration | MSELoss          | 0      | train
-------------------------------------------------------------------
```

```
84.9 K    Trainable params
0         Non-trainable params
84.9 K    Total params
0.339     Total estimated model params size (MB)
7         Modules in train mode
0         Modules in eval mode

INFO: lightning.pytorch.callbacks.model_summary:
  | Name                  | Type             | Params | Mode
-------------------------------------------------------------------
0 | rnn                   | LSTM             | 68.1 K | train
1 | dropout               | Dropout          | 0      | train
2 | fc_pitch              | Linear           | 16.5 K | train
3 | fc_step               | Linear           | 129    | train
4 | fc_duration           | Linear           | 129    | train
5 | loss_fn_pitch         | CrossEntropyLoss | 0      | train
6 | loss_fn_step_duration | MSELoss          | 0      | train
-------------------------------------------------------------------
84.9 K    Trainable params
0         Non-trainable params
84.9 K    Total params
0.339     Total estimated model params size (MB)
7         Modules in train mode
0         Modules in eval mode
Epoch 19: 100% 1885/1885 [00:17<00:00, 109.42it/s, v_num=0, train_loss=3.410, val_loss=3.590]
```

Now that our model is trained, we can use it to generate new music. The `get_seed_sequence` function allows us to sample a seed sequence from a different MIDI file to be used as input.[65]

65: Alternatively, a custom sequence could be defined manually.

```
def get_seed_sequence(midi_file_path, device, num_notes=25, skip_notes=25,
                quantize_step=0.125, quantize_duration=0.125):
    midi_data = pretty_midi.PrettyMIDI(midi_file_path)

    merged_notes = []
    for instrument in midi_data.instruments:
        merged_notes.extend(instrument.notes)

    notes = sorted(merged_notes, key=lambda note: note.start)

    if len(notes) < (skip_notes + num_notes):
        print(f"Not enough notes after skipping the first {skip_notes}. Using all
                available notes after skipping.")
        start_index = skip_notes if skip_notes < len(notes) else 0
        num_notes = len(notes) - start_index
    else:
        start_index = skip_notes

    seed_sequence = []
    previous_start = notes[start_index].start if notes else 0.0
```

```
    for note in notes[start_index:start_index + num_notes]:
        pitch = note.pitch

        step = note.start - previous_start
        duration = note.end - note.start

        step = round(step / quantize_step) * quantize_step
        duration = round(duration / quantize_duration) * quantize_duration

        seed_sequence.append([pitch, step, duration])
        previous_start = note.start

    seed_sequence_tensor = torch.tensor(seed_sequence, dtype=torch.float32).to(device)
    return seed_sequence_tensor

midi_file_path = "path-to/seed-file.mid"

seed_sequence = get_seed_sequence(midi_file_path, device=device, num_notes=25,
                                  quantize_step=0.125, quantize_duration=0.125)
```

We are now ready to generate music! We will use probabilistic **pitch generation**, with the randomness level controlled by the **temperature** parameter;[66] **step** and **duration** generation are deterministic.

66: Higher temperature leads to more randomness.

```
def generate_notes(model, device, seed_sequence, sequence_length, vocab_size=128,
            temperature=1.0, quantize_step=0.125, quantize_duration=0.125, max_silence=1.0):

    # set model to evaluation mode
    model.eval()
    generated_sequence = []

    # initialize with the seed sequence
    current_sequence = seed_sequence  # Initialize with the seed sequence

    for _ in range(sequence_length):
        current_sequence_tensor = current_sequence.unsqueeze(0).to(device)

        with torch.no_grad():
            predictions = model(current_sequence_tensor)

        # apply temperature scaling
        pitch_logits = predictions['pitch'].squeeze() / temperature
        pitch_probs = F.softmax(pitch_logits, dim=-1)

        # sample pitch from multinomial distribution
        predicted_pitch = torch.multinomial(pitch_probs, num_samples=1).item()
        predicted_step = predictions['step'].squeeze().item()
        predicted_duration = predictions['duration'].squeeze().item()

        # max silence
        predicted_step = min(predicted_step, max_silence)

        predicted_step = round(predicted_step / quantize_step) * quantize_step
```

```
        predicted_duration = round(predicted_duration / quantize_duration) *
                                        quantize_duration

        generated_sequence.append([predicted_pitch, predicted_step, predicted_duration])

        new_note = torch.tensor([[predicted_pitch, predicted_step, predicted_duration]],
                    dtype=torch.float32).to(device)
        current_sequence = torch.cat((current_sequence[1:], new_note))

    return generated_sequence

model.to(device)
generated_sequence = generate_notes(model, device=device, seed_sequence=seed_sequence,
                sequence_length=400, temperature=1)
```

Finally, the big reveal... we output the generated sequence to a MIDI file, which can then be played back.

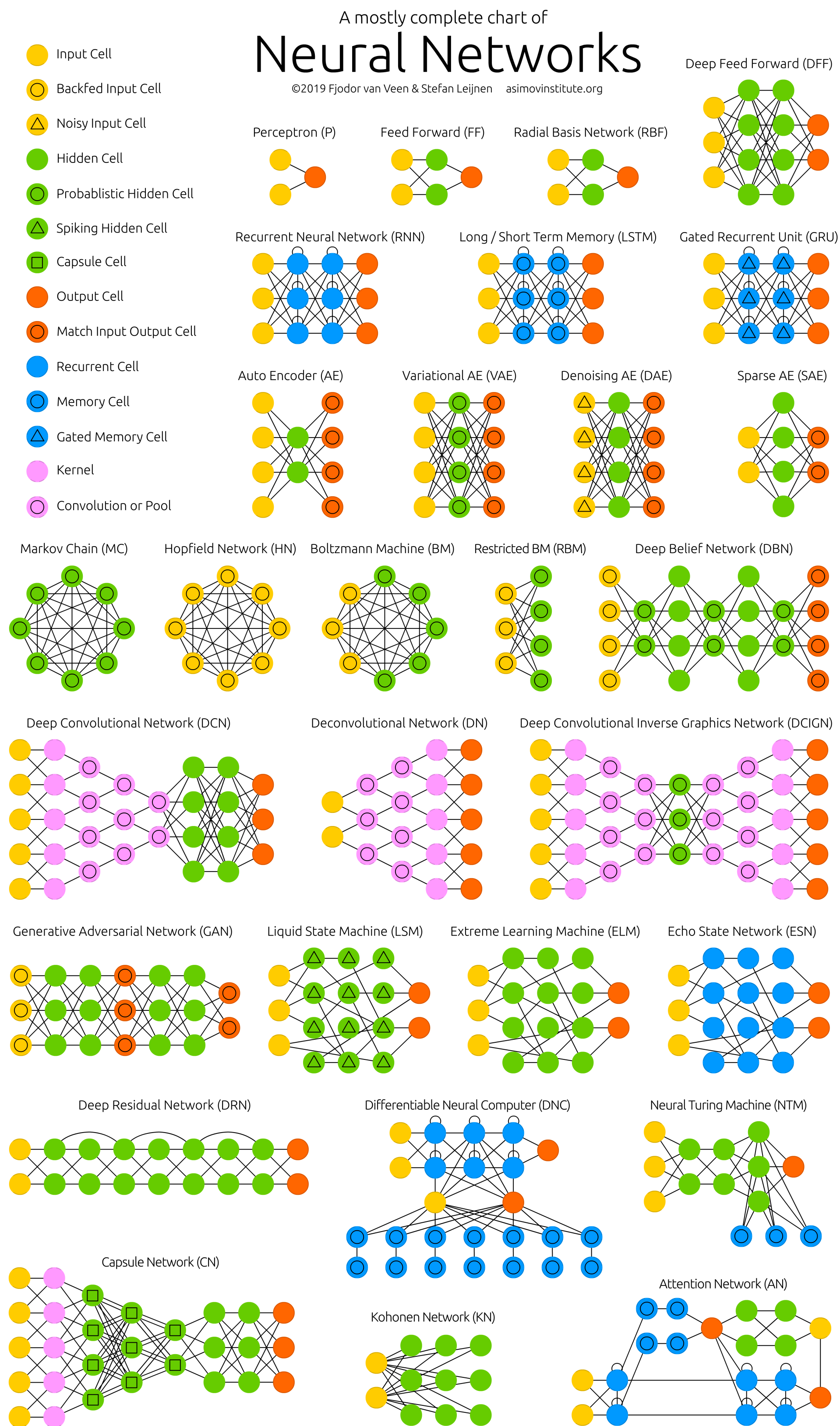
```
def notes_to_midi(generated_sequence, output_path="generated_music.mid"):
    midi = pretty_midi.PrettyMIDI()
    instrument = pretty_midi.Instrument(program=0)

    start_time = 0

    for note in generated_sequence:
        pitch = int(note[0])
        step = note[1]
        duration = note[2]

        start_time += step
        end_time = start_time + duration

        midi_note = pretty_midi.Note(
            velocity=100,
            pitch=pitch,
            start=start_time,
            end=end_time
        )
        instrument.notes.append(midi_note)

    midi.instruments.append(instrument)
    midi.write(output_path)

notes_to_midi(generated_sequence, output_path="generated_music.mid")
```

We seeded one run of the RNN with Sonata in G Major Op. 37 No. 2: I. Allegro, by Muzio Clementi; the output is stored in the DUDADS GitHub repository (`generated_music_8.mid`).

Take a listen to the short track; does it sound like music? Is it **good** music? How could the model be improved?

31.7 Specialized Architectures

The *Asimov Institute* has a chart of specialized network architectures.



31.7.1 Generative Adversarial Networks (GAN)

Generative adversarial networks (GAN) were first introduced in 2014 [12]. These models are driven by competition between a deceiver and a detector. Just as individuals become better liars over time by learning from **past successes** and **failures**, a GAN learns through adversarial training between two components: a **generator** and a **discriminator**.

The **generator** produces synthetic samples $\mathbf{X} = g(\mathbf{z}; \boldsymbol{\theta}^{(g)})$, where $\mathbf{z}$ is a latent random vector and $\boldsymbol{\theta}^{(g)}$ are the generator's parameters. These generated samples, along with real data, are passed to the **discriminator**, which assigns each input a probability $d(\mathbf{X}; \boldsymbol{\theta}^{(d)})$ indicating how likely the input is to come from the true data distribution rather than the generator. The output $d(\mathbf{X}; \boldsymbol{\theta}^{(d)})$ is then used to update $\boldsymbol{\theta}^{(g)}$ for the next iteration.

The objective function of the discriminator [11] when using binary cross-entropy loss is:

$$v(\boldsymbol{\theta}^{(g)}, \boldsymbol{\theta}^{(d)}) = \mathrm{E}_{x \sim p_{\text{data}}}[\log d(x)] + \mathrm{E}_{z \sim p_z}[\log(1 - d(g(z)))].$$

The generator aims to minimize the discriminator's ability to **distinguish fake data from real** while the discriminator simultaneously tries to maximize this ability. The **equilibrium** of this game corresponds to a generator that produces **data indistinguishable from the true distribution**.

The training process involves solving the following **minimax optimization** problem.[67]

$$\begin{aligned} g^* &= \arg\min_g \left\{ \max_d \; v(g, d) \right\} \\ &= \arg\min_g \left\{ \max_d \; \mathrm{E}_{x \sim p_{\text{data}}}[\log d(x)] + \mathrm{E}_{z \sim p_z}[\log(1 - d(g(z)))] \right\}. \end{aligned}$$

67: Minimax optimization is a strategy from game theory and optimization in which two agents operate in opposition: one seeks to **maximize** a function, while the other seeks to **minimize** it. Formally, the general form is:

$$\min_\theta \max_\phi \; f(\theta, \phi).$$

The **inner maximization** $\max_\phi$ identifies the best response of the first player (e.g., the discriminator), given the current strategy θ. The **outer minimization** $\min_\theta$ finds the best counter-strategy for the second player (e.g., the generator), anticipating that the first player will respond optimally.

Explicitly, the discriminator loss is[68]

$$L^{(d)} = -\mathrm{E}_{x \sim p_{\text{data}}}[\log d(x)] - \mathrm{E}_{z \sim p_z}[\log(1 - d(g(z)))],$$

and the generator loss is

$$L^{(g)} = \mathrm{E}_{z \sim p_z}[\log(1 - d(g(z)))].$$

68: Here $L^{(d)} \equiv L^{(d)}(\boldsymbol{\theta}^{(g)}, \boldsymbol{\theta}^{(d)})$ and $L^{(g)} \equiv L^{(g)}(\boldsymbol{\theta}^{(g)}, \boldsymbol{\theta}^{(d)})$.

Training GAN

We provide an overview of best practices for training GAN, as well as common challenges.

Nonsaturating Loss

This formulation treats GAN training as a **zero-sum problem**; in early training, this makes it difficult for the generator to learn.

During the first pass, the output of the generator depends entirely on the **random initialization**. No insight from the discriminator has

been provided, resulting in poor outputs. This makes it easy for the discriminator to **identify fake outputs**, i.e., $\log(1 - d(g(z))) \approx 0$. If none of the generator's outputs are "good", then the discriminator will not provide any insight about which synthetic data points were good, and the generator **cannot incorporate any new knowledge** to update its parameters.

We can address this *via* the generator's **nonsaturating loss** [12]:

$$L^{(g)} = -\mathrm{E}_{z\sim p_z}[\log(d(g(z)))],$$

which helps avoid vanishing gradients early in training.[69]

69: Since the loss is no longer zero-sum, the minimax formulation is no longer technically correct, and the inner maximization and outer minimization should be written separately, not in a nested format.

Gradient Updates

Re-updating the generator's parameters after each update of the discriminator is computationally expensive. In practice, the discriminator is updated k times before the generator is updated again.

The following steps [24] detail how GAN are generally trained,[70] with n representing the number of training iterations and k the number of discriminator updates between each generator update.

70: The updates in steps 4 and 5 depend on the choice of optimizer.

1. Initialize $\boldsymbol{\theta}^{(g)}$ and $\boldsymbol{\theta}^{(d)}$.
2. For i in $1, \ldots, n$:
3.   For j in $1, \ldots, k$:
4.   Perform the gradient update for $\boldsymbol{\theta}^{(d)}$.
5.   Perform the gradient update for $\boldsymbol{\theta}^{(g)}$.
6. Return $\boldsymbol{\theta}^{(g)}$, $\boldsymbol{\theta}^{(d)}$.

Minibatch Discrimination

One challenge of training GAN is that the generator may converge to an **underfit** distribution which does not reflect all of the modes present in the original data, known as **mode collapse** [24].[71] One way to overcome this problem is to use **minibatch discrimination**.

71: If the discriminator eventually learns to distinguish data at this mode, the generator may gradually move towards another mode. This cycle of mode collapse is referred to as **mode hopping**.

When performing minibatch discrimination, the discriminator attempts to classify **sample groups**, rather than individual points, as real or fake. Samples that are extremely homogeneous compared to the original dataset thus receive lower scores [35].

Feature Matching

Feature matching aims to improve stability during training by narrowing the scope of the generator. Rather than trying to generate samples as close to the real data as possible, the generator attempts to generate samples with **similar summary statistics**.[72] Empirical results indicate that feature matching improves convergence when instability is a concern [35].

72: We can choose any statistic of interest.

Further details can be found in [11, sec. 20.10.4] and [24, ch. 26].

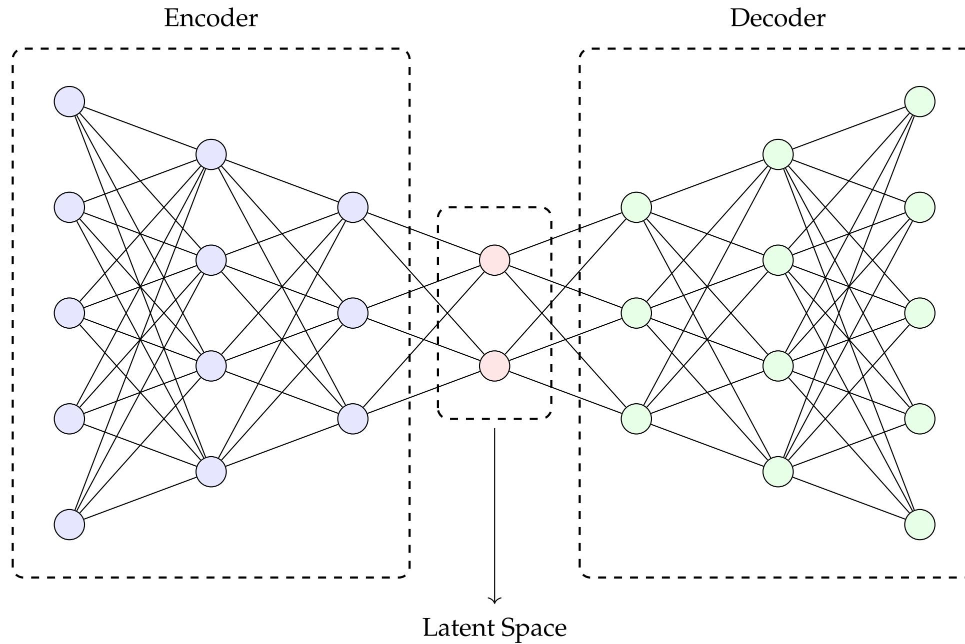


Figure 31.14: Schematics for a simple autoencoder.

31.7.2 Autoencoders and Variational Autoencoders

Autoencoders [11, ch. 14] are neural networks that attempt to output an approximate representation of a given input, as shown schematically in Figure 31.14.[73]

73: See also [6, vol. 4, ch. 26, fig. 26.11].

In a standard deterministic autoencoder, the **encoder** function f maps the input $\mathbf{x}$ to a latent representation $\mathbf{h} = f(\mathbf{x})$, while the **decoder** function g reconstructs the input as $\mathbf{r} = g(\mathbf{h})$.

The network attempts to minimize the loss function

$$\mathscr{L}(\mathbf{x}, g(f(\mathbf{x}))),$$

for some appropriate $\mathscr{L}$.

Undercomplete Autoencoders

A summary that keeps all of the content from the original piece of writing is not a very useful summary; an autoencoder that just copies and pastes the input as output is rather unhelpful. Autoencoders have commonly been used for **dimension reduction** and **feature selection**.[74] One way of doing this is by imposing a **dimensionality constraint** on $\mathbf{h}$ to ensure that the learned representation is **compressed** relative to the input.

74: See [6, vol. 3, ch. 23].

Undercomplete autoencoders (UE) constrain the output to be of lesser dimension than the input; this forces the autoencoder to select the **most important** features in order to still achieve a "good" representation.

If g is linear and $\mathscr{L}$ is MSE, then a UE is essentially performing **principal component analysis** (PCA).[75] Incorporating nonlinearities into f and g can provide better approximations than PCA, but this may result in outputs that are too near the original inputs to be useful.

75: See [6, vol. 3, ch. 23, sec. 23.2.3].

Regularized Autoencoders

Regularization is another way to restrain autoencoders from producing outputs that are too similar to their inputs; **regularized autoencoders** penalize the loss function to discourage **overly specific outputs**.

Even if the output dimensions are equal to or greater than the input dimensions, regularization allows autoencoders to produce useful outputs [11, ch. 14.2].

Sparse autoencoders in particular use a penalty that encourages **sparsity**.[76] Ideally, this helps the encoder select useful features for the outputs.

76: This refers to situations where most model elements (weights, activations) are zero or inactive.

The loss function of a sparse autoencoder can be expressed as

$$\mathscr{L}(\mathbf{x}, g(f(\mathbf{x}))) + \Omega(\mathbf{h}, \mathbf{x}),$$

where $\Omega(\mathbf{h}, \mathbf{x})$ is the **sparsity penalty**.

A common sparsity constraint is based on the **Kullback-Leibler** (KL) divergence. Let:

- ρ be the target average activation of a hidden unit;[77]
- $a_j^{(i)}$ be the activation of hidden unit j for input i, and
- $\hat{\rho}_j = \frac{1}{m} \sum_{i=1}^{m} a_j^{(i)}$ be the average activation of hidden unit j over m training examples.

77: Typically, $\rho = 0.05$.

The KL divergence between ρ and $\hat{\rho}_j$ is

$$\mathrm{KL}(\rho \parallel \hat{\rho}_j) = \rho \log\left(\frac{\rho}{\hat{\rho}_j}\right) + (1-\rho)\log\left(\frac{1-\rho}{1-\hat{\rho}_j}\right).$$

The total sparsity penalty term over all k hidden units is:

$$\Omega_{\text{sparsity}} = \beta \sum_{j=1}^{k} \mathrm{KL}(\rho \parallel \hat{\rho}_j),$$

where β controls the importance of the sparsity penalty.[78]

78: An alternative sparsity penalty is the **L1 norm** of the hidden activations:

$$\Omega_{\text{sparsity}} = \lambda \sum_{i=1}^{m} \sum_{j=1}^{k} \left| a_j^{(i)} \right|,$$

where λ is the regularization strength for the L1 penalty.

De-noising Autoencoders

We have primarily discussed autoencoders in the context of feature selection; **de-noising autoencoders** input noisy data and output "clean" data.[79] We train de-noising autoencoders by first defining a **noise-addition process** $C(\tilde{\mathbf{x}} \mid \mathbf{x})$;[80] the goal is to learn a **reconstruction distribution** $p_{\text{rec}}(\mathbf{x} \mid \tilde{\mathbf{x}})$.

79: Such as may be used to remove noise from images or audio signals, for instance.

80: Given a sample $\mathbf{x}$, this is the conditional distribution of noise-injected samples $\tilde{\mathbf{x}}$ [11, pages 507–508].

One common approach is to model reconstruction through the decoder distribution $p_{\text{dec}}(\mathbf{x} \mid \tilde{\mathbf{h}})$, where $\tilde{\mathbf{h}} = f(\tilde{\mathbf{x}})$, and to minimize the negative log-likelihood

$$\mathscr{L}(\mathbf{x}, g(f(\tilde{\mathbf{x}}))) = -\log p_{\text{dec}}(\mathbf{x} \mid \tilde{\mathbf{h}} = f(\tilde{\mathbf{x}})),$$

where $\tilde{\mathbf{x}}$ is the noisy input and $\mathbf{x}$ is the original data. Training proceeds as follows:

1. repeatedly draw a sample $\mathbf{x}$ and a noisy sample $\tilde{\mathbf{x}}$ from $C(\tilde{\mathbf{x}} \mid \mathbf{x})$;
2. learn $p_{\text{rec}}(\mathbf{x} \mid \tilde{\mathbf{x}})$ by minimizing $\mathscr{L}$ over the training examples $(\mathbf{x}, \tilde{\mathbf{x}})$.

Variational Autoencoders

Variational autoencoders (VAE) are a **probabilistic extension** of traditional autoencoders. Instead of encoding the input to a fixed point in **latent space**, the encoder outputs a **distribution** over latent variables:

$$\mathbf{z} \sim q(\mathbf{z} \mid \mathbf{x}) = \mathcal{N}\big(\mathbf{z} \mid \boldsymbol{\mu}(\mathbf{x}), \text{diag}(\sigma^2(\mathbf{x}))\big),$$

where both the mean $\boldsymbol{\mu}(\mathbf{x})$ and the covariance matrix $\text{diag}(\sigma^2(\mathbf{x}))$ are learned functions of the input.

Sampling directly from $\mathbf{z} \sim q(\mathbf{z} \mid \mathbf{x})$ would make it impossible to perform backpropagation; instead, we **re-parameterize** to express the sampling operation deterministically.

First, sample $\boldsymbol{\varepsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$; then re-parameterize as follows:

$$\mathbf{z} \sim q(\mathbf{z} \mid \mathbf{x}) = \mathcal{N}\big(\mathbf{z} \mid \boldsymbol{\mu}(\mathbf{x}), \text{diag}(\sigma^2(\mathbf{x}))\big),$$
$$\mathbf{z} = \boldsymbol{\mu}(\mathbf{x}) + \boldsymbol{\sigma}(\mathbf{x}) \odot \boldsymbol{\varepsilon}.$$

After reparameterization, $\mathbf{z}$ is passed through the decoder to generate an output $\mathbf{r} = g(\mathbf{z})$.

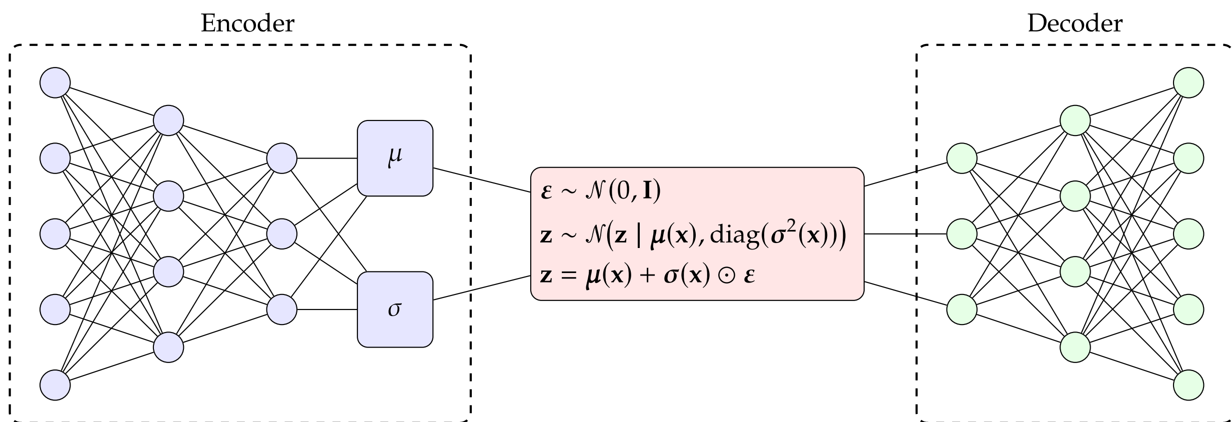


Figure 31.15: A variational autoencoder.

VAEs are trained to maximize the **evidence lower bound**, or ELBO:

$$\text{ELBO}(\mathbf{x}) = \mathrm{E}_{q_{\text{enc}}(\mathbf{z}|\mathbf{x})}[\log p_{\text{dec}}(\mathbf{x} \mid \mathbf{z})] - \text{KL}\big(q_{\text{enc}}(\mathbf{z} \mid \mathbf{x}) \,\|\, p(\mathbf{z})\big).$$

VAEs differ from traditional autoencoders in that they are **generative models**, capable of producing plausible outputs from random inputs drawn from a latent distribution [24]. While standard autoencoders are typically used for representation learning tasks such as feature extraction or de-noising, VAEs are designed to **generate entirely new data**. In image processing, this allows VAEs to create novel images rather than simply improve or reconstruct existing ones.

This summary offers only a high-level overview. Additional details, including the **re-parameterization** trick and derivation of the training objective, can be found in [11, ch. 14, sec. 20.10.3] and [24, ch. 21].

31.7.3 Transformers

Transformers are a class of neural network architectures designed to model sequential data without relying on recurrence. Introduced by [43], the transformer replaces recurrence with a self-attention mechanism that allows each position in a sequence to attend to all others simultaneously. This design enables parallel processing and improves the modeling of long-range dependencies.

To preserve information about token positions (lost due to the absence of recurrence), transformers incorporate **positional encodings**. These are added to the input embeddings and inject information about the position of each element in the sequence. A common choice is to use sinusoidal encodings:

$$\mathrm{PE}_{(\mathrm{pos},2i)} = \sin\left(\frac{\mathrm{pos}}{10000^{2i/d_{\mathrm{model}}}}\right), \quad \mathrm{PE}_{(\mathrm{pos},2i+1)} = \cos\left(\frac{\mathrm{pos}}{10000^{2i/d_{\mathrm{model}}}}\right),$$

where pos is the token position and d_{model} is the embedding dimension.

Attention and Multi-Head Mechanisms The core operation in a transformer is **scaled dot-product attention**, defined for a query matrix Q, key matrix K, and value matrix V as:

$$\mathrm{Attention}(Q, K, V) = \mathrm{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V,$$

where d_k is the dimensionality of the key vectors. This allows each position to attend to others proportionally to their similarity. Transformers use **multi-head attention**, which performs multiple attention operations in parallel and concatenates their outputs:

$$\mathrm{MultiHead}(Q, K, V) = \mathrm{concat}(\mathrm{head}_1, \ldots, \mathrm{head}_h)W^O,$$

where

$$\mathrm{head}_i = \mathrm{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

and $W^O \in \mathbb{R}^{(h \cdot d_v) \times d_{\mathrm{model}}}$ is a learned **output projection matrix**.[81]

81: Each head$_i$ outputs a vector of dimension d_v, and the outputs from all h heads are concatenated into a single matrix of dimension $h \cdot d_v$. The projection matrix W^O maps this concatenated output back to the model's embedding dimension d_{model}, allowing the multi-head attention mechanism to be integrated seamlessly into the surrounding architecture. Each transformer block also contains a **position-wise feed-forward network** and employs residual connections with layer normalization to ensure stable training. Each head i also uses learned **input projection matrices**:

$$W_i^Q \in \mathbb{R}^{d \times d_q}, W_i^K \in \mathbb{R}^{d \times d_k}, W_i^V \in \mathbb{R}^{d \times d_v}.$$

Encoder and Decoder Structure A transformer consists of an **encoder** and a **decoder**, each composed of multiple stacked layers:

- the **encoder** processes the input sequence through layers of multi-head self-attention and feed-forward sublayers; each position attends to all others in the sequence;
- the **decoder** generates the output sequence autoregressively; each layer includes masked self-attention, cross-attention (which attends to encoder outputs), and a feed-forward sublayer; the masking ensures causality during training by preventing attention to future tokens.

Training Objective The model is trained to minimize the **cross-entropy loss** between the predicted token distributions and the ground truth sequence. Given an input sequence $\mathbf{x}_{1:T}$ and a target sequence $\mathbf{y}_{1:T'}$, the loss is:

$$\mathcal{L}(\boldsymbol{\theta}) = -\sum_{t=1}^{T'} \log p(\mathbf{y}_t \mid \mathbf{y}_{<t}, \mathbf{x}; \boldsymbol{\theta}),$$

where $p(\mathbf{y}_t \mid \cdot)$ is the decoder's predicted probability of the correct token at time t, and $\boldsymbol{\theta}$ denotes model parameters. During training, the model uses **teacher forcing**, conditioning on the true previous tokens $\mathbf{y}_{<t}$. At inference time, the model generates outputs one token at a time using previously predicted tokens.

Extensions

Label Smoothing Rather than treating the target token as a one-hot vector, label smoothing distributes a small portion of the probability mass across all vocabulary entries. This prevents the model from becoming overconfident and can improve generalization. For a smoothing parameter ε, the target distribution is adjusted as:

$$\tilde{y}_i = \begin{cases} 1 - \varepsilon + \frac{\varepsilon}{V}, & \text{if } i = \text{correct class}, \\ \frac{\varepsilon}{V}, & \text{otherwise}, \end{cases}$$

where V is the vocabulary size.

Scheduled Sampling During training, instead of always using the ground truth token as input at each decoding step, the model may occasionally use its own prediction. This encourages robustness to its own errors and better approximates inference-time behaviour. A scheduling function determines the probability of using a sampled token versus the true one at each step.

Beam Search Decoding Instead of greedily selecting the most probable token at each step, **beam search** maintains a set of the top k candidate sequences (beams), expanding them at each step and retaining only the k most probable. This enables the model to explore multiple decoding paths and often improves the quality of the generated sequence, especially for long or ambiguous outputs.

Transformers have become the foundation for numerous state-of-the-art architectures, including BERT, GPT, T5, and Vision Transformers (ViT). For further details, see [43] and related discussions in [11, sec. 20.10].

31.8 Additional Examples

We provide examples of deep learning in action using common frameworks. We will revisit the use of deep learning networks for natural language processing and images in Chapters 32 and 35, respectively.

31.8.1 Learning the XOR Function

The exclusive OR function is a simple binary operation defined by:

$$\mathrm{XOR}(x_1, x_2) = \begin{cases} 1 & \text{if } x_1 \neq x_2 \\ 0 & \text{if } x_1 = x_2 \end{cases}$$

x_1	x_2	$\mathrm{XOR}(x_1, x_2)$
0	0	0
0	1	1
1	0	1
1	1	0

The critical property of the XOR function is that it is **not linearly separable**: there exists no straight line (in two dimensions) that can separate the output classes (0 and 1) of XOR.

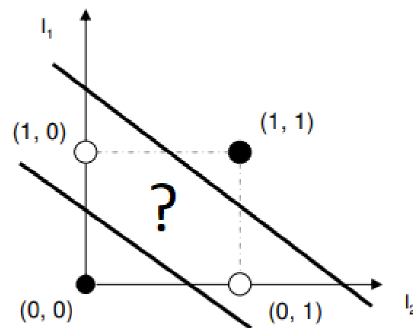


Figure 31.16: Linear non-separability of the XOR function: no straight line can separate the white discs from the black discs [author unknown].

The XOR function illustrates the fundamental limitations of **linear models** and the expressive power of **deep nonlinear architectures**: any model worth its salt should be able to learn XOR. In this example, we will use a small feed-forward neural network (FFNNN), implemented in Keras, to learn the XOR function. We start by setting up the architecture.

```
from keras import models
from keras import layers

model = models.Sequential()

# One hidden layer with 6 units
model.add(layers.Dense(6, input_shape=(2,), activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))

# Set up for training with optimizer ADAM with default parameters
model.compile(optimizer='adam', loss='mean_squared_error', metrics=['binary_accuracy'])
```

When it is not specified, Keras can still figure out the input shape.[82] There are many possible choices for the loss function.[83] In this instance, we used `mean_squared_error` because, well, because there was no compelling reason to use other loss functions.

82: Everything still works if the `input_shape` option is removed; equivalently, one can specify `input_dim = 2` instead.

83: See the keras documentation for a complete list.

Next, we create the input (table) for the XOR dataset.

```
import numpy as np

# The four possible inputs of XOR
x = np.array([[0,0],[0,1],[1,0],[1,1]])
# The corresponding expected outputs of XOR
y = np.array([[0],[1],[1],[0]])
```

We train the model for 2000 passes.

```
history = model.fit(x,y,epochs=2000)
```

```
Epoch 1/2000
1/1 ========= 0s 174ms/step - binary_accuracy: 0.50 - loss: 0.26
Epoch 2/2000
1/1 ========= 0s  13ms/step - binary_accuracy: 0.25 - loss: 0.26
...
Epoch 1999/2000
1/1 ========= 0s  14ms/step - binary_accuracy: 1.00 - loss: 0.03
Epoch 2000/2000
1/1 ========= 0s  13ms/step - binary_accuracy: 1.00 - loss: 0.03
```

We can see how well the model approximates XOR.

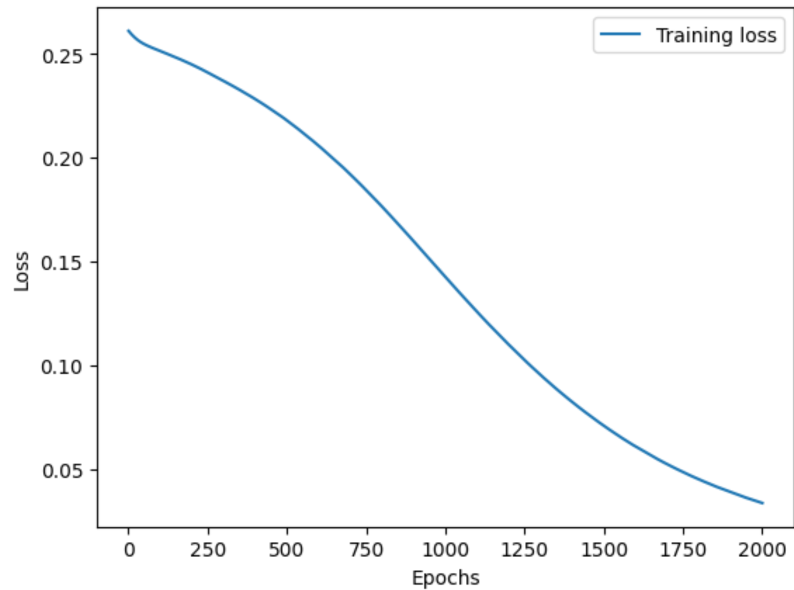
```
print(model.predict(x))
```

```
1/1 ========= 0s 18ms/step
[[0.20125844]
 [0.8208007 ]
 [0.8231141 ]
 [0.175821  ]]
```

This is indeed quite a good approximation of $(0, 1, 1, 0)^\top$.

As a final step, we plot the loss against epochs to see how quickly the training error decreases.

```
%matplotlib inline

import matplotlib.pyplot as plt

loss = history.history['loss']
epochs = range(1, len(loss) + 1)

plt.plot(epochs, loss, label='Training loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.show()
```



31.8.2 Classification in the Iris Dataset

In this example, we build a classifier from the classic Iris dataset (accessible through the `seaborn` module).[84]

84: See [6, sec. 19.5.6] for more information.

```
import seaborn as sns

iris = sns.load_dataset("iris")

print(iris.head())
print(iris.shape)
```

```
   sepal_length  sepal_width  petal_length  petal_width species
0           5.1          3.5           1.4          0.2  setosa
1           4.9          3.0           1.4          0.2  setosa
2           4.7          3.2           1.3          0.2  setosa
3           4.6          3.1           1.5          0.2  setosa
4           5.0          3.6           1.4          0.2  setosa

(150, 5)
```

The goal is to build a **classifier** that takes in `sepal_length`, `sepal_width`, `petal_length`, and `petal_width` as input data and outputs the `species`.[85] First, we list the existing unique values of `species`.

85: Described *via* **one-hot** species **encoding** (see Chapter 32).

```
d = iris.species.unique()

numSpecies = d.shape[0]
print(d)
```

```
['setosa' 'versicolor' 'virginica']
```

We write a function that yields the index of a species in the list of distinct values.

```
def toIndex (values, species):
    idx = -1
    n = values.shape[0]  # number of distinct values
    for i in range(n):
        if species == values[i]:
            idx = i
            break
    return idx
```

We check if the function is doing the “right thing”.

```
for s in d:
    print(toIndex(d, s))
```

```
0
1
2
```

Next, we set up a training set and a testing set.[86]

86: We will not normalize the data since the values do not seem to have drastically different magnitudes.

```
from keras.utils import to_categorical

# fix a seed for replication
np.random.seed(2018)
# random permutation of the rows
iris = iris.reindex(np.random.permutation(iris.index))

y = iris.species
# convert targets to the one-hot encodings
y = (y.apply( lambda s: toIndex(d, s))).values
# turn indices to one-hot encodings acceptable to keras
y = to_categorical(y)
# convert the first columns into a numpy array
x = iris.drop('species',axis = 1).values

# take first 120 as training set and
# the rest as the test set
trainCount = 120
train_x = x[0:trainCount-1]
train_y = y[0:trainCount-1]

test_x = x[trainCount:]
test_y = y[trainCount:]
```

We will build a small fully connected feed-forward network with two hidden layers.

```
speciesModel = models.Sequential()
speciesModel.add(layers.Dense(4, input_shape=(4,), activation='relu'))
speciesModel.add(layers.Dense(4, activation='relu'))
speciesModel.add(layers.Dense(numSpecies, activation='softmax'))

# categorical_crossentropy is used for multi-category classification
speciesModel.compile(optimizer='rmsprop', loss='categorical_crossentropy',
                metrics=['accuracy'])
```

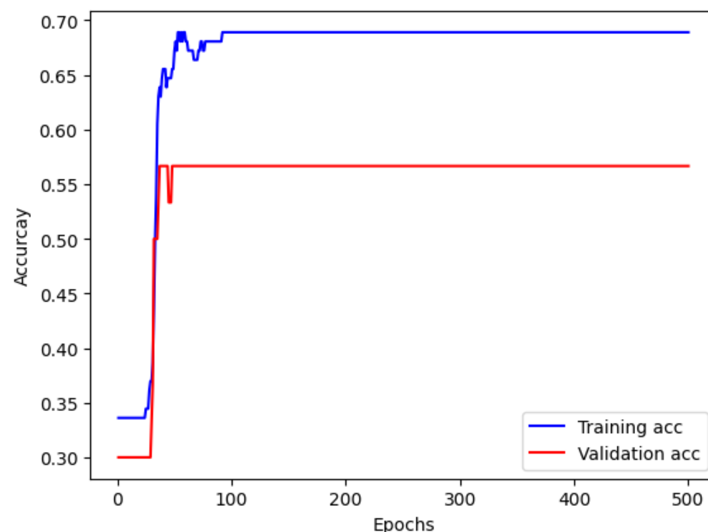
We train for 500 epochs in batch mode (i.e., each pass goes through all the training observations).

```
irisHistory = speciesModel.fit(train_x, train_y, epochs=500,
                validation_data=(test_x, test_y))
```

```
Epoch 1/500
4/4 ========= 0s 24ms/step - accuracy: 0.30 - loss: 2.60 - val_accuracy: (
...
Epoch 500/500
4/4 ========= 0s  7ms/step - accuracy: 0.73 - loss: 0.45 - val_accuracy: (
```

We plot the training and validation accuracy.

```
acc = irisHistory.history['accuracy']
val_acc = irisHistory.history['val_accuracy']

epochs = range(1, len(acc) + 1)

plt.plot(epochs, acc, 'b', label='Training acc')
plt.plot(epochs, val_acc, 'r', label='Validation acc')
plt.xlabel('Epochs')
plt.ylabel('Accurcay')
plt.legend()

plt.show()
```



31.8.3 Regression in the Boston Dataset

We build a feed-forward neural network for **regression** on the well-known Boston Housing dataset; we will also perform *k***-fold cross-validation**.[87]

87: See [6, ch. 20].

```
from keras.datasets import boston_housing

(train_x, train_y), (test_x, test_y) = boston_housing.load_data()

print(train_x.shape)
print(test_x.shape)
```

```
(404, 13)
(102, 13)
```

We take a look at the targets, which are median values of homes (in thousands of dollars).

```
train_y
```

```
array([15.2, 42.3, 50. , 21.1, 17.7, 18.5, 11.3, 15.6, 15.6, 14.4, 12.1, 17.9, 23.1, 19.9,
       15.7,  8.8, 50. , 22.5, 24.1, 27.5, 10.9, 30.8, 32.9, 24. , 18.5, 13.3, 22.9, 34.7,
       16.6, 17.5, 22.3, 16.1, 14.9, 23.1, 34.9, 25. , 13.9, 13.1, 20.4, 20. , 15.2, 24.7,
       22.2, 16.7, 12.7, 15.6, 18.4, 21. , 30.1, 15.1, 18.7,  9.6, 31.5, 24.8, 19.1, 22. ,
       14.5, 11. , 32. , 29.4, 20.3, 24.4, 14.6, 19.5, 14.1, 14.3, 15.6, 10.5,  6.3, 19.3,
       19.3, 13.4, 36.4, 17.8, 13.5, 16.5,  8.3, 14.3, 16. , 13.4, 28.6, 43.5, 20.2, 22. ,
       23. , 20.7, 12.5, 48.5, 14.6, 13.4, 23.7, 50. , 21.7, 39.8, 38.7, 22.2, 34.9, 22.5,
       31.1, 28.7, 46. , 41.7, 21. , 26.6, 15. , 24.4, 13.3, 21.2, 11.7, 21.7, 19.4, 50. ,
       22.8, 19.7, 24.7, 36.2, 14.2, 18.9, 18.3, 20.6, 24.6, 18.2,  8.7, 44. , 10.4, 13.2,
       21.2, 37. , 30.7, 22.9, 20. , 19.3, 31.7, 32. , 23.1, 18.8, 10.9, 50. , 19.6,  5. ,
       14.4, 19.8, 13.8, 19.6, 23.9, 24.5, 25. , 19.9, 17.2, 24.6, 13.5, 26.6, 21.4, 11.9,
       22.6, 19.6,  8.5, 23.7, 23.1, 22.4, 20.5, 23.6, 18.4, 35.2, 23.1, 27.9, 20.6, 23.7,
       28. , 13.6, 27.1, 23.6, 20.6, 18.2, 21.7, 17.1,  8.4, 25.3, 13.8, 22.2, 18.4, 20.7,
       31.6, 30.5, 20.3,  8.8, 19.2, 19.4, 23.1, 23. , 14.8, 48.8, 22.6, 33.4, 21.1, 13.6,
       32.2, 13.1, 23.4, 18.9, 23.9, 11.8, 23.3, 22.8, 19.6, 16.7, 13.4, 22.2, 20.4, 21.8,
       26.4, 14.9, 24.1, 23.8, 12.3, 29.1, 21. , 19.5, 23.3, 23.8, 17.8, 11.5, 21.7, 19.9,
       25. , 33.4, 28.5, 21.4, 24.3, 27.5, 33.1, 16.2, 23.3, 48.3, 22.9, 22.8, 13.1, 12.7,
       22.6, 15. , 15.3, 10.5, 24. , 18.5, 21.7, 19.5, 33.2, 23.2,  5. , 19.1, 12.7, 22.3,
       10.2, 13.9, 16.3, 17. , 20.1, 29.9, 17.2, 37.3, 45.4, 17.8, 23.2, 29. , 22. , 18. ,
       17.4, 34.6, 20.1, 25. , 15.6, 24.8, 28.2, 21.2, 21.4, 23.8, 31. , 26.2, 17.4, 37.9,
       17.5, 20. ,  8.3, 23.9,  8.4, 13.8,  7.2, 11.7, 17.1, 21.6, 50. , 16.1, 20.4, 20.6,
       21.4, 20.6, 36.5,  8.5, 24.8, 10.8, 21.9, 17.3, 18.9, 36.2, 14.9, 18.2, 33.3, 21.8,
       19.7, 31.6, 24.8, 19.4, 22.8,  7.5, 44.8, 16.8, 18.7, 50. , 50. , 19.5, 20.1, 50. ,
       17.2, 20.8, 19.3, 41.3, 20.4, 20.5, 13.8, 16.5, 23.9, 20.6, 31.5, 23.3, 16.8, 14. ,
       33.8, 36.1, 12.8, 18.3, 18.7, 19.1, 29. , 30.1, 50. , 50. , 22. , 11.9, 37.6, 50. ,
       22.7, 20.8, 23.5, 27.9, 50. , 19.3, 23.9, 22.6, 15.2, 21.7, 19.2, 43.8, 20.3, 33.2,
       19.9, 22.5, 32.7, 22. , 17.1, 19. , 15. , 16.1, 25.1, 23.7, 28.7, 37.2, 22.6, 16.4,
       25. , 29.8, 22.1, 17.4, 18.1, 30.3, 17.5, 24.7, 12.6, 26.5, 28.7, 13.3, 10.4, 24.4,
       23. , 20. , 17.8,  7. , 11.8, 24.4, 13.8, 19.4, 25.2, 19.4, 19.4, 29.1])
```

Next, we normalize the data; note that normalization parameters should be obtained from the training set excluding the test set.

```
mean = train_x.mean(axis = 0)
train_x -= mean
std = train_x.std(axis = 0)
train_x /= std

test_x -= mean
test_x /= std
```

We build a small network with two hidden layers, the first with 128 nodes, and the second with 48. We write a function for that since we will be training different instances of the same model.

```
def build_model():
    model = models.Sequential()
    model.add(layers.Dense(128, activation='relu'))
    model.add(layers.Dense(48, activation='relu'))
    model.add(layers.Dense(1)) # No activation function for arbitrary output values

    # Use MSE for loss as it is common for regression
    model.compile(optimizer='rmsprop', loss='mean_squared_error', metrics=['mean_absolute_error'])

    return model
```

One problem with model validation using only one validation set is that the validation error can have high variance. In other words, depending on how the examples are split into training and validation sets, the validation error we obtain can vary a lot. To obtain a more accurate picture of validation error, we use k-fold cross-validation.

To set up k-fold cross-validation, the examples are partitioned into k folds of roughly equal size. The model is then trained k times, once with each of these k folds left out as validation set. The validation error on each of these hold-out sets is computed; the average of the k validation errors is the score we report.

The following code carries out this procedure on the training examples for the Boston Housing data. We will compare the cross-validation error score with the validation error on the test set.

```
k = 4 # 4-fold validation. Common values: 4, 5, 10

folds_x = np.array_split(train_x, k)
folds_y = np.array_split(train_y, k)

num_epochs = 200 # Training can take a while
batch_size = 25
```

We initialize the errors and run the model.

```
val_err_scores = np.zeros(k)
mae_history = []
```

```
for i in range(k):

    partial_x = np.concatenate( np.delete(folds_x, i, axis = 0) )
    partial_y = np.concatenate( np.delete(folds_y, i, axis = 0) )

    model = build_model()
    model.fit(partial_x, partial_y, epochs=num_epochs, batch_size = batch_size,
              verbose = 0) # Train silently

    val_mse, val_mae = model.evaluate(folds_x[i], folds_y[i], verbose = 0)
    val_err_scores[i] = val_mae

print(val_err_scores)
np.mean(val_err_scores)
```

```
[2.26713371 2.36012745 2.38808918 2.41208768]
np.float64(2.356859505176544)
```

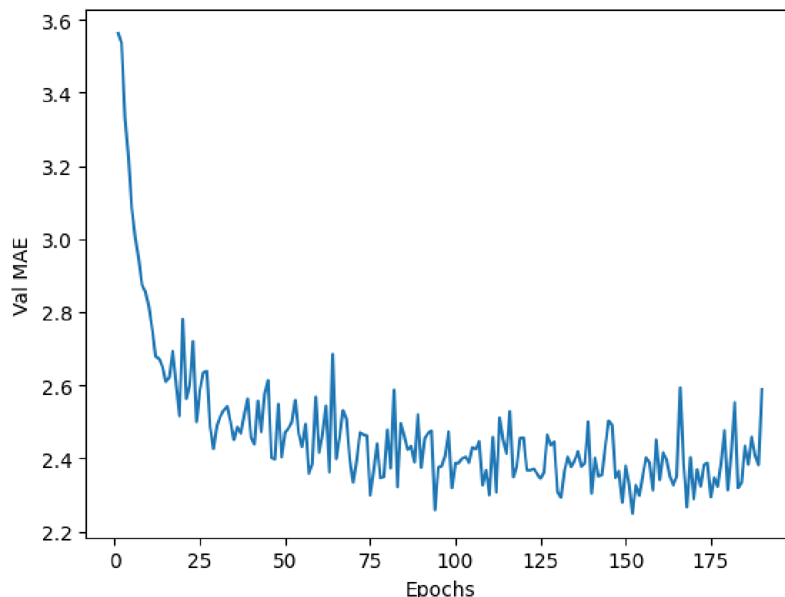
As we ran without specifying a seed, the results of repeated runs could be different (although conceivably of comparable magnitudes).

We can display the average per-epoch MAE scores across all folds:

```
mae_history = np.zeros(num_epochs)

for i in range(k):

    partial_x = np.concatenate( np.delete(folds_x, i, axis = 0) )
    partial_y = np.concatenate( np.delete(folds_y, i, axis = 0) )

    model = build_model()
    history = model.fit(partial_x, partial_y, epochs=num_epochs,
                        validation_data=(folds_x[i], folds_y[i]),
                        batch_size = batch_size, verbose = 0) # Train silently
    mae_history += history.history['val_mean_absolute_error']

avg_mae_history = mae_history / k

to_plot = avg_mae_history[10:] # Ignore the first 10 epochs
plt.plot(range(1, len(to_plot)+1), to_plot)
plt.xlabel('Epochs')
plt.ylabel('Val MAE')
plt.show()
```

We see on the next page that the average validation MAE pretty much stops improving after 80 or so epochs. We now train the model on the full training set and see what the test MAE score is.

```
model = build_model()
model.fit(train_x, train_y, epochs=80, batch_size=batch_size, verbose = 0)
test_mse, test_mae = model.evaluate(test_x, test_y)
print(test_mae)
```




```
4/4 [=] 0s 2ms/step - loss: 13.8429 - mean_absolute_error: 2.5445
2.765582323074341
4/4 [=] 0s 2ms/step - loss: 13.8429 - mean_absolute_error: 2.5445
2.765582323074341
```

31.8.4 MNIST Dataset

The Modified National Institute of Standards and Technology (MNIST) database is a collection of images of handwritten digits. Each image consists of a 28×28 grayscale pixel grid; its label is a digit.[88]

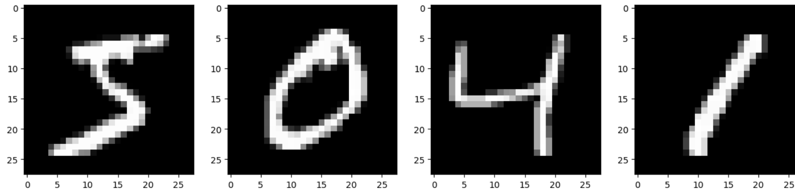
88: Accessing the MNIST database is very simple; it is one of the Keras module's built-in datasets.

```
from keras.datasets import mnist

# Data set already split into training and testing sets
(train_x, train_y), (test_x, test_y) = mnist.load_data()
print(train_x.shape)
print(test_x.shape)
```

```
(60000, 28, 28)
(10000, 28, 28)
```

Let's take a look at the first few training images.

```
%matplotlib inline
import matplotlib.pyplot as plt
fig = plt.figure(figsize=(16,16))
n = 4
for i in range(n):
    fig.add_subplot(1, n, i+1)
    plt.imshow(train_x[i], cmap='gray')
```



Taking a look at the corresponding labels shows that they are simply the values for the digits associated with the corresponding images.

```
train_y[0:n]
```

```
array([5, 0, 4, 1], dtype=uint8)
```

For simplicity, we scale the image values from the range $[0, 255]$ to $[0, 1]$ and convert the labels to their one-hot encodings.

```
from keras.utils import to_categorical

dim = train_x.shape[1] * train_x.shape[2]

# vectorize the matrices
train_x = train_x.reshape( (train_x.shape[0], dim) )
train_x = train_x.astype('float32') / 255.0
test_x = test_x.reshape( (test_x.shape[0], dim) )
test_x = test_x.astype('float32') / 255.0

train_y = to_categorical(train_y)
test_y = to_categorical(test_y)
```

Classification In this example, we build a small FFNNN with two hidden layers with the goal of recognizing handwritten digits.

```
model = models.Sequential()
model.add(layers.Dense(256, input_dim = dim, activation='relu'))
model.add(layers.Dense(128, activation='relu'))
model.add(layers.Dense(10, activation='softmax')) # 10 units, one for each digit
```

We use a customized optimizer.

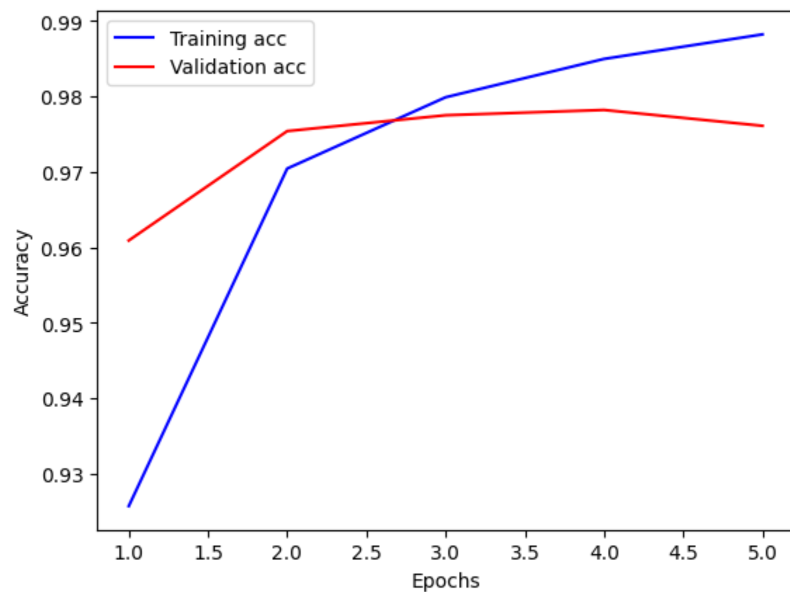
```
myRMSprop= optimizers.RMSprop(learning_rate = 0.001)
model.compile(optimizer=myRMSprop, loss='categorical_crossentropy', metrics=['accuracy'])
```

The model takes a while to train.

```
history = model.fit( train_x, train_y, epochs = 5, batch_size = 100,
                    validation_data=( test_x, test_y ), verbose = 0)
```

Next, we plot the training and validation accuracy.

```
acc = history.history['accuracy']
val_acc = history.history['val_accuracy']

epochs = range(1, len(acc) + 1)

plt.plot(epochs, acc, 'b', label='Training acc')
plt.plot(epochs, val_acc, 'r', label='Validation acc')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

plt.show()
```



Not too shabby!

Regularization In this section, we implement deep learning regularization. In comparison to the classification FFNNN of the previous section, we use a larger network.

```
model = models.Sequential()
model.add(layers.Dense(400, activation='relu'))
model.add(layers.Dense(400, activation='relu'))
model.add(layers.Dense(400, activation='relu'))
model.add(layers.Dense(200, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))
```

We make use of Keras' **early stopping** feature: we instruct Keras to stop after 3 epochs without improvement. The best model is then saved to the file `best_model.keras`.

```
from keras.callbacks import EarlyStopping, ModelCheckpoint
callbacks = [EarlyStopping(monitor='val_loss', patience=3),
    ModelCheckpoint(filepath='best_model.keras', monitor='val_loss', save_best_only=True)]
model.summary()
```

```
Model: "sequential"

 Layer (type)                     Output Shape                  Param #
 dense (Dense)                    ?                         0 (unbuilt)
 dense_1 (Dense)                  ?                         0 (unbuilt)
 dense_2 (Dense)                  ?                         0 (unbuilt)
 dense_3 (Dense)                  ?                         0 (unbuilt)
 dense_4 (Dense)                  ?                         0 (unbuilt)

 Total params: 0 (0.00 B)
 Trainable params: 0 (0.00 B)
 Non-trainable params: 0 (0.00 B)
```

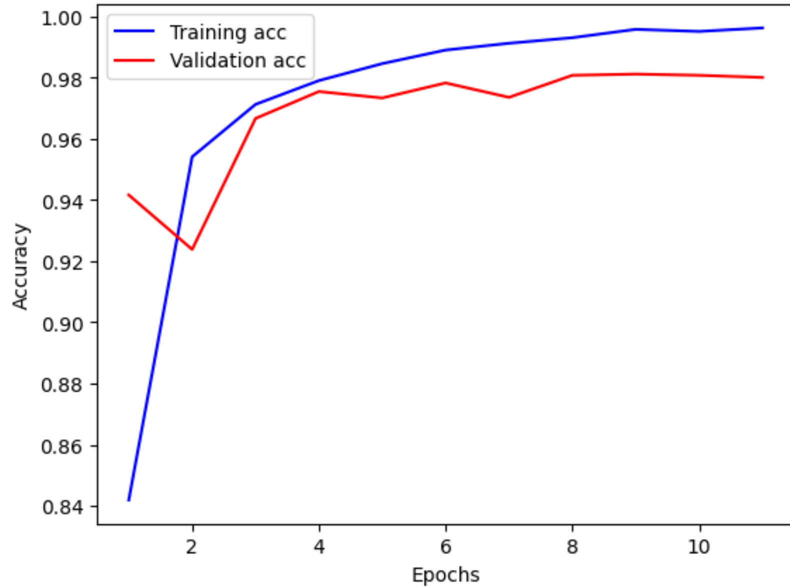
We now start the training process, with a limit of 25 epochs. It will take a while, and we most likely will not make it to 25 epochs.

```
myRMSprop= optimizers.RMSprop(learning_rate = 0.001)
model.compile(optimizer=myRMSprop, loss='categorical_crossentropy', metrics=['accuracy'])

history = model.fit( train_x, train_y, epochs = 25, batch_size = 500,
                    callbacks = callbacks,
                    validation_data=( test_x, test_y ), verbose = 1)
```

```
Epoch 1/25
120/120 [=] 1s 6ms/step - accuracy: 0.706 - loss: 0.899 - val_accuracy: 0.942 - val_loss: 0.195
Epoch 2/25
120/120 [=] 1s 7ms/step - accuracy: 0.949 - loss: 0.169 - val_accuracy: 0.924 - val_loss: 0.248
...
Epoch 10/25
120/120 [=] 1s 8ms/step - accuracy: 0.996 - loss: 0.012 - val_accuracy: 0.981 - val_loss: 0.079
Epoch 11/25
120/120 [=] 1s 7ms/step - accuracy: 0.996 - loss: 0.011 - val_accuracy: 0.980 - val_loss: 0.086
```

Next, we plot the training and validation accuracy and see how many epochs we have traversed.

```
%matplotlib inline

import matplotlib.pyplot as plt

acc = history.history['accuracy']
val_acc = history.history['val_accuracy']

epochs = range(1, len(acc) + 1)

plt.plot(epochs, acc, 'b', label='Training acc')
plt.plot(epochs, val_acc, 'r', label='Validation acc')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

plt.show()
```



Keras also accommodates L_1 **regularization**; we implement an FFNNN with 4 hidden layers.

```
from keras import regularizers

model = models.Sequential()
model.add(layers.Dense(400, input_dim = dim, activation='relu',
                       kernel_regularizer=regularizers.l1(0.0001)))
model.add(layers.Dense(400, input_dim = dim, activation='relu',
                      kernel_regularizer=regularizers.l1(0.0001)))
model.add(layers.Dense(400, input_dim = dim, activation='relu',
                      kernel_regularizer=regularizers.l1(0.0001)))
model.add(layers.Dense(200, activation='relu',
                      kernel_regularizer=regularizers.l1(0.0001)))
model.add(layers.Dense(10, activation='softmax'))
```

We use the same training settings as in the early stopping example.

```
import tensorflow as tf
tf.config.run_functions_eagerly(True)

myRMSprop = tf.keras.optimizers.RMSprop(learning_rate=0.001) # Mairi added

model.compile(optimizer=myRMSprop, loss='categorical_crossentropy', metrics=['accuracy'])

history_l1 = model.fit( train_x, train_y, epochs = 25, batch_size = 500,
                    validation_data=( test_x, test_y ), verbose = 1)
```

```
...
Epoch 24/25
120/120 [=] 3s 23ms/step - accuracy: 0.986 - loss: 0.171 - val_accuracy: 0.978 - val_loss: 0.192
Epoch 25/25
120/120 [=] 3s 22ms/step - accuracy: 0.987 - loss: 0.168 - val_accuracy: 0.979 - val_loss: 0.190
```

Finally, we experiment with **dropout**, which is readily available in Keras. We use the same architecture with two dropout layers.

```
from keras.layers import Dropout

model = models.Sequential()
model.add(layers.Dense(400, input_dim = dim, activation='relu'))
model.add(layers.Dense(400, input_dim = dim, activation='relu'))
model.add(Dropout(0.5))
model.add(layers.Dense(400, input_dim = dim, activation='relu'))
model.add(Dropout(0.5))
model.add(layers.Dense(200, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))

myRMSprop = tf.keras.optimizers.RMSprop(learning_rate=0.001) # Mairi added
model.compile(optimizer=myRMSprop, loss='categorical_crossentropy', metrics=['accuracy'])

history_dropout = model.fit( train_x, train_y, epochs = 25, batch_size = 500,
                    validation_data=( test_x, test_y ), verbose = 1)
```

```
Epoch 1/25
120/120 [=] 3s 23ms/step - accuracy: 0.679 - loss: 0.947 - val_accuracy: 0.936 - val_loss: 0.217
...
Epoch 24/25
120/120 [=] 3s 23ms/step - accuracy: 0.997 - loss: 0.009 - val_accuracy: 0.982 - val_loss: 0.098
Epoch 25/25
120/120 [=] 3s 23ms/step - accuracy: 0.997 - loss: 0.010 - val_accuracy: 0.983 - val_loss: 0.089
```

We save the dropout model in HDF5 format.

```
model.save("my_model.keras")
```

CNN In this section, we train a simple CNN on the MNIST dataset, using the functionality available in Keras.

```
from keras import models, layers

model = models.Sequential()
model.add(layers.Conv2D(32, (3,3), activation='relu',
                    input_shape=(28,28,1)))
model.add(layers.MaxPooling2D((2,2)))
model.add(layers.Conv2D(64, (3,3), activation='relu'))
model.add(layers.Flatten())
model.add(layers.Dense(32, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))
```

Next, we compile the model.

```
model.compile(optimizer='rmsprop',
              loss = 'categorical_crossentropy',
              metrics=['accuracy'])
```

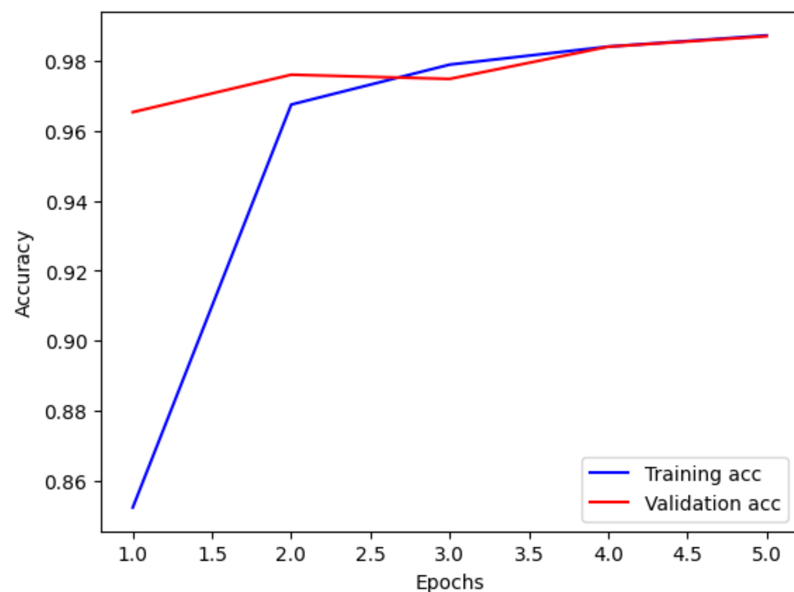
The training phase can take a while.

```
history = model.fit(train_x, train_y, epochs=5,
                    batch_size=500,
                    validation_data=( test_x, test_y ),
                    verbose = 1)
```

```
Epoch 1/5
120/120 [=] 4s 29ms/step - accuracy: 0.700 - loss: 0.896 - val_accuracy: 0.965 - val_loss: 0.123
...
Epoch 4/5
120/120 [=] 3s 28ms/step - accuracy: 0.985 - loss: 0.052 - val_accuracy: 0.984 - val_loss: 0.048
Epoch 5/5
120/120 [=] 3s 27ms/step - accuracy: 0.989 - loss: 0.036 - val_accuracy: 0.987 - val_loss: 0.042
```

Next, we plot the training and validation accuracy and see how many epochs we have traversed.

```
%matplotlib inline

import matplotlib.pyplot as plt

acc = history.history['accuracy']
val_acc = history.history['val_accuracy']

epochs = range(1, len(acc) + 1)

plt.plot(epochs, acc, 'b', label='Training acc')
plt.plot(epochs, val_acc, 'r', label='Validation acc')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

plt.show()
```



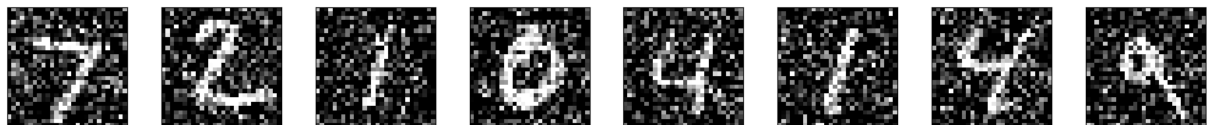
De-Noising In this example, we use a de-noising autoencoder to classify noisy images; it is modified from a Keras autoencoder blog post.

First, we generate digits with artificial noise by applying a Gaussian noise matrix to each image; we also clip the images between 0 and 1.

```
from keras.datasets import mnist
import numpy as np

(x_train, _), (x_test, _) = mnist.load_data()

x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.

# adapt this if using `channels_first` image data format
x_train = np.reshape(x_train, (len(x_train), 28, 28, 1))
x_test = np.reshape(x_test, (len(x_test), 28, 28, 1))
noise_factor = 0.4

x_train_noisy = x_train + noise_factor * np.random.normal(loc=0.0, scale=1.0,
                size=x_train.shape)
x_test_noisy = x_test + noise_factor * np.random.normal(loc=0.0, scale=1.0,
                size=x_test.shape)

x_train_noisy = np.clip(x_train_noisy, 0., 1.)
x_test_noisy = np.clip(x_test_noisy, 0., 1.)
```

Let's take a look at a few of these noisy digits.

```
%matplotlib inline

import matplotlib.pyplot as plt

n = 8
plt.figure(figsize=(20, 2))
for i in range(n):
    ax = plt.subplot(1, n, i+1)
    plt.imshow(x_test_noisy[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
plt.show()
```



We build and train an autoencoder for de-noising these images. Note that the training can take a long time.[89]

89: At least half an hour on a recent mid-grade desktop computer.

```
from keras.layers import Input, MaxPooling2D, UpSampling2D, Conv2D
from keras.models import Model
```

```
input_img = Input(shape=(28, 28, 1))

x = Conv2D(32, (3, 3), activation='relu', padding='same')(input_img)
x = MaxPooling2D((2, 2), padding='same')(x)
x = Conv2D(32, (3, 3), activation='relu', padding='same')(x)
encoded = MaxPooling2D((2, 2), padding='same')(x)

# at this point the representation is (7, 7, 32)

x = Conv2D(32, (3, 3), activation='relu', padding='same')(encoded)
x = UpSampling2D((2, 2))(x)
x = Conv2D(32, (3, 3), activation='relu', padding='same')(x)
x = UpSampling2D((2, 2))(x)
decoded = Conv2D(1, (3, 3), activation='sigmoid', padding='same')(x)

autoencoder = Model(input_img, decoded)
autoencoder.compile(optimizer='adadelta', loss='binary_crossentropy')
```

Ideally, we should train for 50 epochs, but we have neither the hardware nor the patience for this, so we will train only for 5 epochs (hardly sufficient, we know).

```
autoencoder.fit(x_train_noisy, x_train,
                epochs=5, batch_size=600,
                shuffle=True,
                validation_data=(x_test_noisy, x_test))
```

```
Epoch 1/5
100/100 [=] 8s 78ms/step - loss: 0.7015 - val_loss: 0.6971
Epoch 2/5
100/100 [=] 8s 77ms/step - loss: 0.6956 - val_loss: 0.6915
Epoch 3/5
100/100 [=] 8s 80ms/step - loss: 0.6900 - val_loss: 0.6859
Epoch 4/5
100/100 [=] 8s 75ms/step - loss: 0.6843 - val_loss: 0.6799
Epoch 5/5
100/100 [=] 8s 76ms/step - loss: 0.6780 - val_loss: 0.6728
```

31.8.5 Sunspot Dataset

In this example, we use neural networks to predict monthly sunspot activity.[90] First, we import the required modules.

90: The file `monthly-sunspot.csv` is in the DUDADS GitHub repository.

```
import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np
import os
import pandas as pd
import tensorflow as tf
```

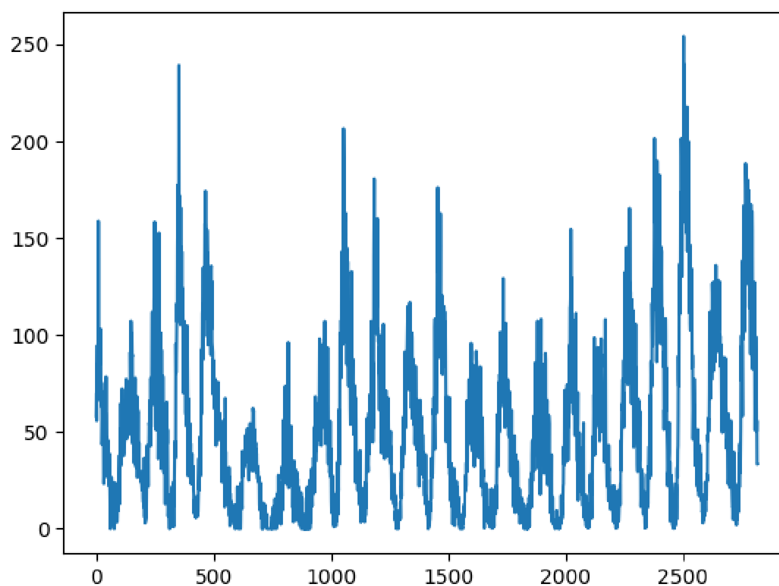
The data contains monthly intensity, from Jan 1749 to Dec 1983.

```
df = pd.read_csv('monthly-sunspots.csv')
df.head()
df.tail()
```

```
     Month   Sunspots
0    1749-01 58.0
1    1749-02 62.6
2    1749-03 70.0
3    1749-04 55.7
4    1749-05 85.0

     Month   Sunspots
2815 1983-08 71.8
2816 1983-09 50.3
2817 1983-10 55.8
2818 1983-11 33.3
2819 1983-12 33.4
```

The data's periodicity is easier to spot when it is plotted.

```
data_seq = df['Sunspots']
plt.plot(data_seq)
plt.show()
```



Preparing the Data for Training and Testing We will use the first part of the data (up to a splitting index) for training, and the rest will be used for testing.[91]

91: Classical time series analysis would approach the problem differently, see [6, vol. 1, ch. 9].

```
split_index = int(0.8 * len(data_seq))
```

We normalize the data so that it has mean 0 and variance 1.

```
data_seq_norm = np.array((data_seq - data_seq[:split_index].mean())
                         / data_seq[:split_index].std())
```

The following function is used to create the training and testing data. It generates a list of all pairs $(x_{i-\text{input length}}, x_{i-\text{input length}+1}), \ldots, (x_{i-1}, x_i)$ within `data[start_index:end_index]`.[92]

92: In ML, we sample training observations randomly from the available data; for ML time series forecasting, however, we want to select (contiguous) training **segments**, since the time series covariance typically plays a role in the analysis of the data.

As the outputs are reals and TensorFlow expects vectors, we apply the mapping $x \longrightarrow [x]$.

```
def nn_data(
    data, start_index, end_index, input_length
):
  inputs = []
  outputs = []

  i_start = start_index + input_length
  if end_index is None:
    end_index = len(data)

  for i in range(i_start, end_index):
    indices = slice(i-input_length, i)
    # Reshape data from (input_length,) to (input_length, 1)
    inputs.append(np.reshape(data[indices], (input_length, 1)))
    outputs.append(data[i])
  return np.array(inputs), np.array(outputs)
```

We generate input sequences of length `input_length`.

```
input_length = 100
x_train, y_train = nn_data(
    data_seq_norm, 0, split_index, input_length
)
x_test, y_test = nn_data(
    data_seq_norm, split_index, None, input_length
)
```

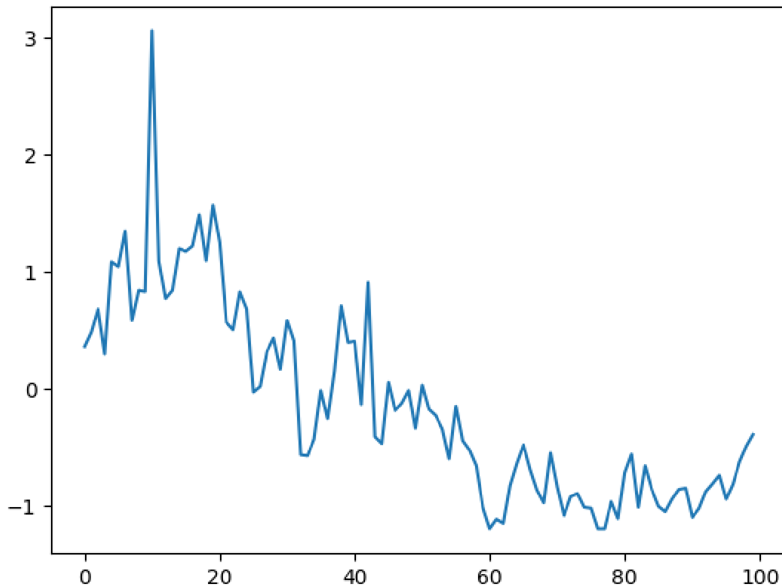
The training observation predictor vector **x** is of length 100; the response y is the next value immediately after the segment has ended. The 1st training observation is displayed below.

```
x_train[0]
```

```
array([[ 0.35827485],
       [ 0.48186036],
       [ 0.68067184],
       [ 0.29648209],
       [ 1.08366807],
       [ 1.04336845],
       [ 1.34695895],
       [ 0.5812661 ],
       [ 0.83918369],
       [ 0.82843712],
       [ 3.06103627],
       [ 1.08904136],
       [ 0.76933101],
       [ 0.83918369],
       [ 1.19650702],
       ...
       ...
       [-1.00922571],
       [-1.05489862],
       [-0.94743296],
       [-0.86414707],
       [-0.8534005 ],
       [-1.10325817],
       [-1.0280322 ],
       [-0.88295356],
       [-0.81578752],
       [-0.7432482 ],
       [-0.94743296],
       [-0.8211608 ],
       [-0.63040925],
       [-0.49607717],
       [-0.39398479]])
```

Of course, we might get more information by plotting the predictor vector as a time series segment.

```
plt.plot(x_train[0])
```



The corresponding response is found below.

```
y_train[0]
```

```
np.float64(-0.17636682500608733)
```

There are 2156 training samples, in total.

```
x_train.shape
```

```
(2156, 100, 1)
```

We have one final processing step to tackle: the Keras API expects the set of training samples to already be partitioned into batches.[93] To effectively disable batching, we should set the batch size to be at least as small as the total number of training observations.

93: The training algorithm works with a single batch at a time in order to improve performance, but it comes at the cost of some accuracy.

Shuffling the training data (the set of labeled examples, not the individual input sequences) is also a prerequisite.

```
nn_train = tf.data.Dataset.from_tensor_slices(
    (x_train, y_train)
).shuffle(len(x_train)).batch(batch_size=len(data_seq))
nn_test = tf.data.Dataset.from_tensor_slices(
    (x_test, y_test)
).batch(batch_size=len(data_seq))
```

Model, Training, and Testing We use (and define) a network with one LSTM layer and one affine layer that maps the LSTM final state to a one-dimensional output; nonlinearity comes from the activation function applied by the LSTM layer.[94]

94: In the following chunk of code, `units` represents the dimensionality of the hidden layer state; in `input_shape`, `None` means sequences of any length, and `1` means that the sequence terms are one-dimensional.

```
model = tf.keras.models.Sequential([
    tf.keras.layers.LSTM(
        units=5,
        name='lstm',
        input_shape=[None, 1],
        # only output the final state term
        return_sequences=False,
    ),
    tf.keras.layers.Dense(units=1, name='out'),
])

# set optimization algorithm and loss function
model.compile(optimizer='adam', loss='mae')
```

We fit the model and pass the test set in one step.[95]

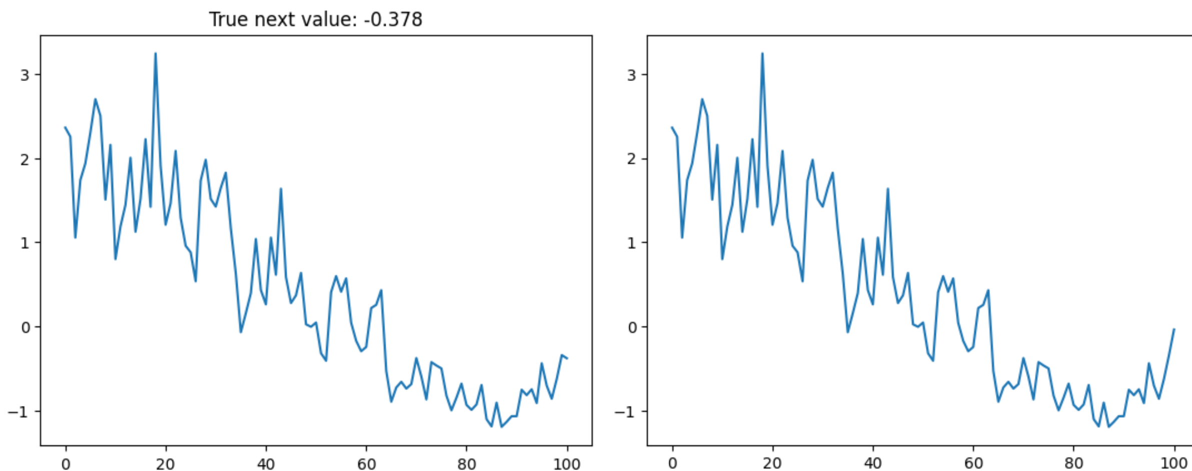
95: So we get the error metric for both training and testing.

```
model.fit(nn_train, validation_data=nn_test)
```

```
1/1 [=] 0s 497ms/step - loss: 0.7013 - val_loss: 1.2558
```

Predictions Here is one test case with the actual final observation, and the same test case with the predicted final observation.

```
plt.plot(np.append(x_test[0].flatten(), y_test[0]))
plt.title(f"True next value: {y_test[0]:.3f}")
plt.show()

for x, y in nn_test.take(1):
    plt.plot(list(x[0]) + [model.predict(x)[0]])
```



31.9 Exercises

1. Implement the 3rd and 4th convolutional layers in the example of section 31.5.4.
2. Try experimenting with different training data, regularization techniques, additional layers, and investigate the parameter losses individually in the example of section 31.6.7.
3. Consider the example of section 31.8.1.
   a) Try the alternative `model.add()` calls suggested in sidenote 82.
   b) Try altering the number of units in the first layer: how small a network can we get to still learn the XOR function? (We might need to increase the number of epochs if the loss decreases slowly.)
   c) Try adding an extra layer with 6 units and see if we can obtain an accurate model with a much smaller epoch value.
   d) Experiment with different optimizers: `rmsprop`, `sgd`, `adagrad`, `adadelta`, etc.
4. Consider the example of section 31.8.2.
   a) Experiment with different split ratios for the training and testing sets.
   b) Experiment with different FFNNN architectures and see how fast we can train a good classifier.
   c) Choose your own dataset and build a DL classifier for it.
5. Consider the example of section 31.8.3.
   a) Experiment with different architectures and parameters.
   b) Perform a similar k-fold cross-validation for regression on datasets of your choice.
6. Consider the example of section 31.8.4.
   a) Experiment with different architectures and various parameters in the classification example.
   b) Create a 28×28 black-and-white image of your handwriting (in white against a black background) of one of the ten digits and save it in PNG format as `mydigit.png`. Run the following code and see if the model recognizes your digit correctly. Note that the answer is a vector of 10 values between 0 and 1. We take the number with the largest value as the prediction made by the model.

```
import numpy as np
my_digit = plt.imread('mydigit.png')
# Take a look at the image
plt.imshow(my_digit, cmap='gray')
d = model.predict(np.array(
                [my_digit.reshape(dim)]))
print(d) # Print the softmax scores
np.argmax(d[0]) # Extract the max val index
```

   c) Using the same `mydigit.png` file and `best_model` from the early stopping example, run the following code and see if the model recognizes your digit correctly.

```
from keras.models import load_model
best_model = load_model('best_model.keras')
import numpy as np
my_digit = plt.imread('mydigit.png')
d = best_model.predict(np.array(
                [my_digit.reshape(dim)]))
print(d) # Print the softmax scores
np.argmax(d[0]) # Extract the max val index
```

   d) Using the same `mydigit.png` file from the previous examples and the L_1 regularization model, make a prediction regarding your digit.
   e) Repeat the above experiment using L_2 regularization by replacing `l1` with `l2` in the code.

f) Using the same `mydigit.png` file from the previous examples and `my_model.keras`, make a dropout prediction regarding your digit.
g) Experiment with different probability values in the dropout layers.
h) Combine early stopping with dropout to try to improve the validation error.
i) Using the same `mydigit.png` file and the convolutional model, run the following code and see if the model recognizes your digit correctly.

```
import numpy as np
my_digit = plt.imread('mydigit.png')

d = model.predict(np.array(
                [my_digit.reshape(28,28,1)]))
print(d) # Print the softmax scores
np.argmax(d[0]) # Extract the max val index
```

j) Experiment with different epoch and batch size values in the convolutional example. Modify the network and see how accuracy and training time change.
k) Test the autoencoder on `mydigit.png`. First add some noise as follows.

```
import numpy as np
my_digit = plt.imread('mydigit.png')
my_digit_noisy = my_digit + 0.7 * noise_factor *
    np.random.normal(loc=0.0, scale=1.0,
    size=my_digit.shape)
my_digit_noisy = np.clip(my_digit_noisy, 0., 1.)
# Take a look at the noisy image
plt.imshow(my_digit_noisy)
```

Then, obtain a clean version of the image using the following code.

```
cleaned = autoencoder.predict(np.array(
        [my_digit_noisy.reshape(28,28,1)]))
plt.imshow(cleaned[0].reshape(28, 28))
```

l) Create other images, add noise to them, and de-noise them with the trained autoencoder.
m) Does the de-noising autoencoder work if it is trained for only one epoch? Try different batch sizes.
n) Train the de-noising autoencoder for 10 epochs. Re-run the last three exercises. Do you notice a difference? What about if you use 20 epochs? 50 epochs?

7. Regularization might reduce overfitting in the example of section 31.8.5. Implement various regularization strategies, following this tutorial. Display predictions for Jan 1984 to Dec 2028.
8. We would like to extend the work done on the MNIST dataset and build a classifier that distinguishes the 10 digits and the 26 uppercase characters of the English alphabet. In particular, we want to build a model that would distinguish a handwritten "8" from a "B", a "1" from an "I", etc.

   a) Generate a training and testing set from a few existing images (such as can be found in `B1.png`, `B2.png`, and `B3.png` in the DUDADS GitHub repository) using **data augmentation**.
   b) Lump the generated images (for the 26 uppercase characters) with a reasonably sized subset of MNIST (be sure to separate the resulting data into a training/testing set).
   c) Using an approach similar to the early stopping example, train the model (ideally, you should train for at least 25 epochs; this will take quite a lot of computing time). Use different regularization techniques to fine tune the model.
   d) Use the model to convert a short handwritten English paragraph (consisting of digits and uppercase letters) into an alphanumeric string (you will need to write a routine to scan and process this text).

9. Visit TensorFlow's Neural Network Playground. In light of what we have discussed in this chapter, tinker with the settings; do you recognize what is happening?
10. **Double descent** is a modern development in ML and DL. It provides an interesting picture of what we sometimes observe in the over-parameterized regime (when there are substantially more parameters than training data points). Consult MLU-Explain's Double Descent Part 1: A Visual Introduction and Double Descent Part 2: A Mathematical Explanation; are the concepts introduced there likely to affect the examples we have presented in this chapter? Discuss why or why not.

Chapter References

[1] M. Abadi et al. 'Tensorflow: A system for large-scale machine learning'. In: *12th USENIX symposium on operating systems design and implementation (OSDI 16)*. 2016, pp. 265–283.

[2] L. Alzubaidi et al. 'Review of deep learning: concepts, CNN architectures, challenges, applications, future directions'. In: *J. Big Data* 8.1 (Mar. 2021).

[3] C. Anumol. 'Advancements in CNN architectures for computer vision: A comprehensive review'. In: *2023 Annual International Conference on Emerging Research Areas: International Conference on Intelligent Systems (AICERA/ICIS)*. IEEE, Nov. 2023.

[4] P. Baldi and R. Vershynin. 'The capacity of feedforward neural networks'. In: *Neural Networks* 116 (2019), pp. 288–311. DOI: https://doi.org/10.1016/j.neunet.2019.04.009.

[5] B. Boehmke and B. Greenwell. *Hands on Machine Learning with R*. CRC Press.

[6] P. Boily. *Data Understanding, Data Analysis, and Data Science (Course Notes)*. Data Action Lab, 2022.

[7] T.B. Brown et al. 'Language Models are Few-Shot Learners'. In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 1877–1901.

[8] K. Cho et al. 'Learning phrase representations using RNN encoder–decoder for statistical machine translation'. In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 2014, pp. 1724–1734.

[9] F. Chollet. *Deep Learning with Python*. 1st. USA: Manning Publications Co., 2017.

[10] J. Devlin et al. 'BERT: Pre-training of deep bidirectional transformers for language understanding'. In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics (NAACL)*. 2019, pp. 4171–4186.

[11] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016.

[12] I. Goodfellow et al. *Generative Adversarial Networks*. 2014. URL: https://arxiv.org/abs/1406.2661.

[13] T. Hastie, T. Tibshirani, and M. Wainwright. *Statistical Learning with Sparsity: The LASSO and Generalizations*. CRC Press, 2015.

[14] A. Hilary. 'CP Decomposition: Approximating Tensors using collection of Vectors'. In: (2024).

[15] S. Hochreiter and J. Schmidhuber. 'Long short-term memory'. In: *Neural Computation* 9.8 (1997), pp. 1735–1780.

[16] K. Hornik, M. Stinchcombe, and H. White. 'Multilayer feedforward networks are universal approximators'. In: *Neural Networks* 2.5 (1989), pp. 359–366.

[17] *Deep Learning for Building Intelligent Computer Systems*.

[18] D.P. Kingma and J. Ba. 'Adam: A Method for Stochastic Optimization'. In: *arXiv preprint* (2014).

[19] Y. LeCun, Y. Bengio, and G. Hinton. 'Deep Learning'. In: *Nature* 521 (2015), pp. 436–444.

[20] J. Lighthill. 'Artificial Intelligence: A General Survey'. In: *Artificial Intelligence: a paper symposium* (1973). Presented to the Science Research Council in the UK.

[21] J. McCarthy. *What is Artificial Intelligence?* 2006.

[22] G. Montúfar et al. 'On the number of linear regions of deep neural networks'. In: *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*. Cambridge, MA, USA: MIT Press, 2014, pp. 2924–2932.

[23] K.P. Murphy. *Probabilistic Machine Learning (volume 1): an Introduction*. MIT Press, 2022.

[24] K.P. Murphy. *Probabilistic Machine Learning (volume 2): Advanced Topics*. MIT Press, 2023.

[25] I. Neutelings. *Neural networks*. 2022. URL: https://tikz.net/neural_networks/#full_code.

[26] N.J. Nilsson. *The Quest for Artificial Intelligence*. Cambridge University Press, 2009.

[27] OpenAI. *GPT-4 Technical Report*. OpenAI Technical Report. 2023.

[28] S. Pramanik. 'Tensor Decompositions: A Powerful Tool for Dimensionality Reduction and Data Analysis in Big Data'. In: (2025).

[29] A. Radford et al. 'Language models are unsupervised multitask learners'. In: *OpenAI Blog* 1.8 (2019).

[30] A. Radford et al. 'Improving language understanding by generative pre-training'. In: *OpenAI Blog* 1.8 (2018).

[31] C. Raffel et al. 'Exploring the limits of transfer learning with a unified text-to-text transformer'. In: *Journal of Machine Learning Research* 21.140 (2020), pp. 1–67.

[32] S.J. Reddi, S. Kale, and S. Kumar. 'On the Convergence of Adam and Beyond'. In: *CoRR* abs/1904.09237 (2019).

[33] J. Riebesell and S. Bringuier. *Collection of standalone TikZ images*. Aug. 9, 2020. DOI: 10.5281/zenodo.7486911. URL: https://github.com/janosh/tikz.

[34] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. 3rd. Prentice Hall, 2010.

[35] T. Salimans et al. *Improved Techniques for Training GANs*. 2016. URL: https://arxiv.org/abs/1606.03498.

[36] J. Schmidhuber. *Annotated History of Modern AI and Deep Learning*. 2022. URL: https://arxiv.org/abs/2212.11279.

[37] R.M. Schmidt. 'Recurrent Neural Networks (RNNs): A gentle Introduction and Overview'. In: *CoRR* abs/1912.05911 (2019).

[38] A. Sherstinsky. 'Fundamentals of Recurrent Neural Network (RNN) and Long Short-Term Memory (LSTM) network'. In: *Physica D: Nonlinear Phenomena* 404 (2020).

[39] N. Srivastava et al. 'Dropout: A Simple Way to Prevent Neural Networks from Overfitting'. In: *Journal of Machine Learning Research* 15.56 (2014), pp. 1929–1958.

[40] I. Sutskever, O. Vinyals, and Q.V. Le. 'Sequence to sequence learning with neural networks'. In: *Advances in Neural Information Processing Systems*. Vol. 27. 2014, pp. 3104–3112.

[41] I. Sutskever et al. 'On the importance of initialization and momentum in deep learning'. In: *Proceedings of the 30th International Conference on International Conference on Machine Learning*. ICML'13. 2013.

[42] M. Uniyal. 'Convolutional Neural Network (CNN) in Machine Learning'. In: *Applied Roots* (2024).

[43] A. Vaswani et al. 'Attention is all you need'. In: *Advances in Neural Information Processing Systems*. Vol. 30. 2017, pp. 5998–6008.

[44] Wikipedia. 'Artificial Intelligence'. In: (2020).

[45] Wikipedia. *Tensor rank decomposition*. 2025.

[46] M. Zaheer et al. 'Adaptive methods for nonconvex optimization'. In: *Proceedings of the 32nd International Conference on Neural Information Processing Systems*. NIPS'18. Curran Associates Inc., 2018, pp. 9815–9825.

[47] M.D. Zeiler. *ADADELTA: An Adaptive Learning Rate Method*. 2012.

[48] A. Zhang et al. 'Dive into Deep Learning'. In: *CoRR* abs/2106.11342 (2021).

[49] X. Zhao et al. 'A review of convolutional neural networks in computer vision'. In: *Artif. Intell. Rev.* 57.4 (Mar. 2024).