SORT 49 (1) January-June 2025, 89-116 DOI: 10.57645/20.8080.02.33

A gentle introduction to deep neural networks for operations researchers

Pau Amaré[1] and Jordi Castro[2,3,4]

Abstract

Computing a neural network is, in essence, a large unconstrained optimization problem, though this fact is often obscured by machine learning jargon. We present a gentle introduction to deep learning for operations researchers, describing in a didactic manner the underlying optimization problem and providing examples developed from scratch. These examples are solved using both standard modeling languages and modern machine learning frameworks. We conclude by discussing applications of neural networks in operations research, illustrated through a concrete example involving the knapsack problem.

MSC: 90C30, 90C90.

Keywords: Operations research, nonlinear optimization, deep learning, neural networks, optimization modeling languages, machine learning frameworks.

1. Introduction

Neural networks (NNs) are one of the main tools in machine learning and data science, and they form the core computational model behind modern large language models (LLMs), such as ChatGPT. As we will show, computing a NN reduces to solving a very large unconstrained optimization problem. However, this fact is not always apparent from the notation and jargon used in machine learning. In this work, we aim to fill this gap by providing a gentle introduction to (deep) NNs for operations researchers.

[1] School of Mathematics and Statistics, Universitat Politècnica de Catalunya, Barcelona, Catalonia.

[2] Department of Statistics and Operations Research, Universitat Politècnica de Catalunya, Barcelona, Catalonia.

[3] Institute of Mathematics of the Universitat Politècnica de Catalunya (IMTech), Barcelona, Catalonia.

[4] Corresponding author
Received: January 2026
Accepted: March 2026

A NN can be defined as a function $y = f(x;\theta)$ that depends on a set of parameters $\theta \in \mathbb{R}^n$. Its purpose is to model the nonlinear relationship between the input and output vectors x and y. If n is large enough, any continuous function can be approximated by a NN; this result, known as the *universal approximation theorem*, was proven in Cybenko (1989); Hornik, Stinchcombe and White (1989). For instance, the number of parameters in modern NNs underlying LLMs is on the order of billions, and there are estimates that the latest version of ChatGPT ranges from one to a few trillion parameters. Therefore, to find the best values of θ, a huge unconstrained optimization problem must be solved.

There are excellent monographs on deep learning and NNs, focusing both on methodology (Goodfellow, Bengio and Courville, 2016) and software (Chollet, 2017), as well as surveys and monographs on optimization methods for NNs (Bottou, Curtis and Nocedal, 2018; Sra, Nowozin and Wright, 2011). Unlike those references, this manuscript is intended for an audience seeking a quick, gentle, and practical introduction to NNs that combines theory with examples developed from scratch and solved using both standard tools in operations research (e.g., the AMPL modeling language (Fourer, Gay and Kernighan, 2003)) and state-of-the-art machine learning frameworks (e.g., Google's TensorFlow (Abadi et al., 2016)).

This work focuses on two of the simplest types of NNs: feedforward and convolutional neural networks (CNNs). For more advanced NN functions (such as encoder–decoder models and transformers, which underlie LLMs), we refer the reader to Goodfellow et al. (2016); Sutskever, Vinyals and Le (2014); Vaswani et al. (2017).

Although NNs rely on optimization methods to compute their best parameters, once trained they can be applied to approximate, accelerate, or guide traditional operations research (OR) algorithms. The survey Bengio, Lodi and Prouvost (2021) reviews recent applications of machine learning techniques to combinatorial optimization problems. A short discussion of these topics is provided in the last part of the paper, together with an illustrative example involving the knapsack problem.

The remainder of the paper is structured as follows. Section 2 describes the structure of a feedforward NN. Section 3 presents two illustrative applications of feedforward NNs with different model complexity. Section 4 introduces CNNs, which are particularly well suited for image data. Section 5 presents an application of CNNs. Finally, Section 6 discusses applications of NNs in OR.

2. Feedforward neural networks

Mathematically, a NN can be defined as a function

$$f : \mathbb{R}^d \to \mathbb{R}^k$$

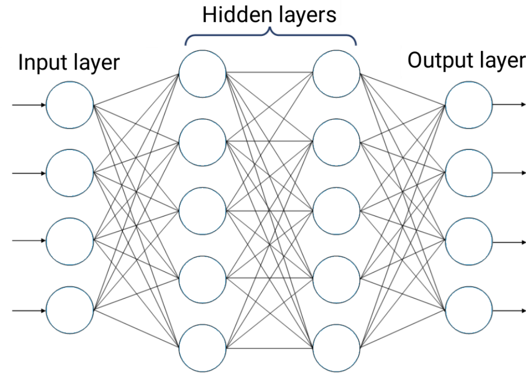

Figure 1. *Example of a 4-layer feedforward NN.*

that models the nonlinear relationship between an input vector $x \in \mathbb{R}^d$ and an output vector $y \in \mathbb{R}^k$. In other words,

$$y = f(x;\theta), \tag{1}$$

where $\theta \in \mathbb{R}^n$ denotes the set of parameters that determine the specific form of the function. A set of points $\{(x^i \in \mathbb{R}^d, y^i \in \mathbb{R}^k), i = 1,\dots,p\}$, called the *training set* in machine learning jargon, is used to compute the best θ through an optimization procedure that minimizes the discrepancies between $f(x^i;\theta)$ and y^i. Another set of points, called the *testing set*, is used to evaluate the accuracy of the computed parameter values θ.

The NN is composed of several elements, which are described in the following subsections.

2.1. Nodes and layers

A NN is composed of a number of *nodes* distributed across several layers. The number of layers, l, determines the depth of the network. We denote by d_j the number of nodes in layer j, for $j = 1,\dots,l$. When the number of layers l is large, the network is said to be deep, and we then speak of *deep learning*.

The first layer is called the *input layer*. This layer simply receives the input and passes it to the next layer (possibly reshaping it when the input is not already a vector). It consists of $d_1 = d$ nodes, matching the dimension of the input vector x or of its reshaped version. The layers from the second to the penultimate are called *hidden layers* (or intermediate layers). They analyze and process the data, and then send it to the next layer. The last layer is the *output layer*, which provides the final result of the network's computation. It consists of $d_l = k$ nodes, corresponding to the size of the network's output vector.

Nodes within the same layer are not connected to each other, but nodes in one layer may be connected to all the nodes of the next layer. This particular structure is known as a *feedforward* NN. Figure 1 shows an example of a 4-layer feedforward NN.

2.2. Parameters and mathematical formulation

Connections between nodes of different layers are represented by arcs joining pairs of nodes in consecutive layers. Each arc connecting node u in layer $j-1$ with node v in layer j, for $j \geq 2$, has an associated *weight* $w_{uv} \in \mathbb{R}$. These arcs are analogous to the synapses linking neurons in the nervous system. In addition, each node v also has a parameter $b_v \in \mathbb{R}$, called the *bias*. Nodes in the first layer usually do not have a bias.

Any node v in layer $j \geq 2$ receives the sum of the output values of all nodes from the previous layer $j-1$, each multiplied by the corresponding weight, plus the bias term b_v. Denoting by $x_h^{(j-1)}$, $h = 1, \dots, d_{j-1}$, the output values of the nodes in layer $j-1$, the result $z_v^{(j)}$ of this linear combination is

$$z_v^{(j)} = \sum_{h=1}^{d_{j-1}} w_{u_h v} x_h^{(j-1)} + b_v, \quad j = 2, \dots, l,$$

where (u_h, v) denotes the arc connecting the h-th node of layer $j-1$ with node v in layer j. We can compute the vector $z^{(j)}$ of these linear combinations for all nodes v in an arbitrary layer j as

$$z^{(j)} = W^{(j)} x^{(j-1)} + b^{(j)}, \quad j = 2, \dots, l, \tag{2}$$

where $W^{(j)} \in \mathbb{R}^{d_j \times d_{j-1}}$ are the weights between layers $j-1$ and j, $b^{(j)} \in \mathbb{R}^{d_j}$ are the biases of layer j, and $x^{(j-1)}$ is the output vector of the nodes in layer $j-1$, for $j = 2, \dots, l$. The vector $x^{(0)}$ is equal to the input x, the point at which the NN is evaluated. The output of the first layer, $x^{(1)}$, is equal to $x^{(0)}$ if layer 1 is the identity function; otherwise, $x^{(1)}$ is a reshaped version of $x^{(0)}$. In any case, layer 1 does not introduce any parameters into the NN. Hence, the parameters of the NN are $\theta = (W^{(j)}, b^{(j)}, j = 2, \dots, l) \in \mathbb{R}^n$, with $n = \sum_{j=2}^{l} (d_j d_{j-1} + d_j)$.

The values $z^{(j)}$ must be nonlinearly transformed before being sent to the nodes of the next layer $j+1$. Otherwise, the overall NN would be just a composition of linear functions, hence linear, and could not model nonlinear relationships between x and y. For this purpose, a nonlinear function, called the *activation function*,

$$g^{(j)} : \mathbb{R} \to \mathbb{R},$$

is applied to all components of $z^{(j)}$, so that the output values $x^{(j)}$ at the nodes of layer $j \geq 2$ are

$$x^{(j)} = g^{(j)}(z^{(j)}) = g^{(j)}\left(W^{(j)} x^{(j-1)} + b^{(j)}\right) = \varphi^{(j)}(x^{(j-1)}), \tag{3}$$

that is, each layer $j \geq 2$ of the NN is a function $\varphi^{(j)} : \mathbb{R}^{d_{j-1}} \to \mathbb{R}^{d_j}$, defined as

$$x^{(j)} = \varphi^{(j)}(x^{(j-1)}) = g^{(j)}\left(W^{(j)} x^{(j-1)} + b^{(j)}\right) \quad j = 2, \dots, l. \tag{4}$$

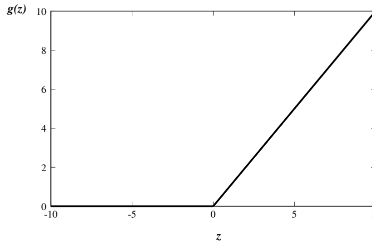

Figure 2. *ReLU.*

Therefore, the NN can be viewed as the composition of these functions:

$$f(x;\theta) = \varphi^{(l)}\left(\varphi^{(l-1)}(\dots\varphi^{(2)}(\varphi^{(1)}(x))\dots)\right), \tag{5}$$

where $\varphi^{(1)}$ is either the identity function or a function that reshapes the input (e.g., flattening a matrix or tensor into a vector). The next subsection provides an overview of several activation functions frequently employed in NNs.

2.3. Activation functions

As stated above, *activation functions* introduce nonlinearity at each level of the NN. The activation function of layer j is denoted by $g^{(j)}$. Each layer may use a different activation function, although it is common to employ the same function for all hidden layers, one for the input layer, and another for the output layer. Among the most commonly used activation functions are the following:

Identity. This activation function is

$$g(z) = I(z) = z, \qquad z \in \mathbb{R}, \tag{6}$$

and it is commonly used in layers with no trainable parameters.

ReLU. The *rectified linear unit* (know as ReLU) function is defined as:

$$g(z) = \mathrm{ReLU}(z) = \max(0,z) = \begin{cases} 0, & \text{if } z \leq 0, \\ z, & \text{if } z > 0. \end{cases} \tag{7}$$

This function, shown in Figure 2, is not differentiable at $z = 0$. To avoid numerical issues with the optimization methods used to compute the optimal parameters θ of the NN, the derivative at 0 is defined as $g'(0) = 0$, that is

$$g'(z) = \begin{cases} 0, & \text{if } z \leq 0, \\ 1, & \text{if } z > 0. \end{cases} \tag{8}$$

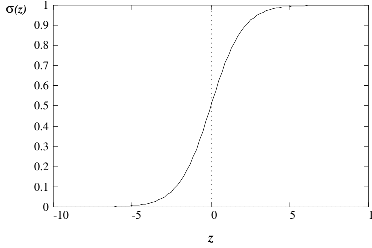

Figure 3. *Sigmoid.*

ReLU is one of the most widely used activation functions in NNs, especially in hidden layers. Since the function is zero for negative values, only a few nodes (or neurons) are activated, which makes computations more efficient. Moreover, the derivative never vanishes for positive z, improving the efficiency of the optimization methods used to compute the network parameters (which, as described later, rely on the gradient of the objective function).

Sigmoid.

The *sigmoid* or logistic function, denoted by σ, is defined as

$$\sigma(z) = \frac{1}{1+e^{-z}}, \qquad z \in \mathbb{R}. \tag{9}$$

This function, shown in Figure 3, has range $(0,1)$, is monotonically increasing, and changes very sharply in the interval $(-2,2)$, where its derivative is large. An appealing property of this function is that its derivative can be expressed directly in terms of $\sigma(z)$:

$$\sigma'(z) = \frac{e^{-z}}{(1+e^{-z})^2} = \frac{1}{1+e^{-z}}\left(1 - \frac{1}{1+e^{-z}}\right) = \sigma(z)(1-\sigma(z)),$$

which made it a very attractive activation function in early NNs implementations, as it simplifies computations.

Since this function returns values between 0 and 1, it is used in the output layer of models that predict probabilities, as well as in hidden layers. Its main drawback is that for large $|z|$, the derivative $\sigma'(z)$ approaches 0, slowing down the optimization algorithms used to compute the network parameters, which rely on the gradient of the objective function.

Softmax

The *softmax* function generalizes the sigmoid function to a k-dimensional vector and returns a probability distribution; therefore it depends on more than one variable. It is used in the output layer l when classifying among k classes, as it predicts

the probability associated with each class.

Unlike previous activation functions, softmax is a vector-valued function

$$g^{(l)} : \mathbb{R}^k \to [0,1]^k,$$

which maps the vector of values $z = (z_1, \dots, z_k)$ at the nodes of the last layer. Each component $g_i^{(l)}$ is defined as

$$g_i^{(l)}(z) = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}}, \qquad i = 1, \dots, k. \tag{10}$$

Clearly, $g_i^{(l)}(z) \in [0,1]$, for all $i = 1, \dots, k$, and $\sum_{i=1}^{k} g_i^{(l)}(z) = 1$, thus providing a valid probability distribution.

2.4. Optimization problem and objective function

Given a NN with a certain number of layers and nodes, suitable activation functions, and corresponding weights and biases, together with a training set $\{(x^i \in \mathbb{R}^d, y^i \in \mathbb{R}^k), i = 1, \dots, p\}$, for a given input vector x^i the network returns a response $f(x^i;\theta)$, which may be a scalar or a vector depending on whether the output layer has one or several nodes. This output vector depends on the network parameters θ. The goal is for $f(x^i;\theta)$ to be as close as possible to the true values y^i in the training set. When the discrepancies between $f(x^i;\theta)$ and y^i are small or close to zero, the network makes a good prediction. The function that measures these discrepancies is the objective function of the optimization problem used to compute the best parameter values θ; in machine learning jargon, this function is called the *loss function*. The optimization problem can thus be formulated as

$$\min_{\theta \in \mathbb{R}^n} F(\theta) = \frac{1}{p} \sum_{i=1}^{p} F_i(\theta), \tag{11}$$

where $F_i : \mathbb{R}^n \to \mathbb{R}$ measures the particular discrepancy between $f(x^i;\theta)$ and y^i at point i, and $F : \mathbb{R}^n \to \mathbb{R}$ computes the average overall discrepancy for all points in the training set.

Two of the most commonly used functions $F_i(\theta)$ to compute the discrepancy are:

Mean squared error. The *mean squared error* (MSE) is simply the square of the Euclidean norm of $f(x^i;\theta) - y^i$:

$$F_i(\theta) = MSE(f(x^i;\theta), y^i) = \|f(x^i;\theta) - y^i\|^2 = (f(x^i;\theta) - y^i)^\top (f(x^i;\theta) - y^i).$$

Cross-entropy. *Cross-entropy* is used when the last layer of the network has k nodes and the network output $f(x^i;\theta) \in \mathbb{R}^k$ for input x^i represents a probability distri-

bution, that is, $f(x^i;\theta)_j \geq 0$ for $j = 1,\ldots,k$ and $\sum_{j=1}^{k} f(x^i;\theta)_j = 1$. The values $f(x^i;\theta)_j$ give the probability that point i belongs to class j, which is the case when the activation function of the last layer is softmax. Cross-entropy measures the difference between the predicted distribution $f(x^i;\theta)$ and the true distribution y^i, and is defined as:

$$F_i(\theta) = H(f(x^i;\theta), y^i) = -\sum_{j=1}^{k} y^i_j \log(f(x^i;\theta)_j). \tag{12}$$

2.5. Optimization algorithms

Although both MSE and cross-entropy are convex functions with respect to $f(x^i;\theta)$, since $f(x;\theta)$, as defined in (5), is a composition of l nonlinear and nonconvex functions due to the activation functions, the objective function in (11) becomes a highly nonlinear, nonconvex function of θ. Therefore, optimization methods for solving the unconstrained problem (11) are only expected to find a local minimum rather than a global solution. Most current methods used for the optimization of (11) in state-of-the-art machine learning frameworks rely on variants of the first-order *steepest descent* method, commonly referred to as the *gradient* method, originally proposed by Cauchy (1847). Second-order methods (i.e., methods that compute or approximate second derivatives), such as Newton or limited-memory quasi-Newton methods, have also been tested, but only on small NNs (Bottou et al., 2018). Here, we focus exclusively on the gradient method.

Unconstrained optimization methods for solving (11) are iterative algorithms that generate a sequence of points $\{\theta^t\}_{t\geq 0}$ according to

$$\theta^{t+1} = \theta^t + \alpha^t \Delta\theta^t, \tag{13}$$

where $\Delta\theta^t \in \mathbb{R}^n$ is the search direction and $\alpha^t \in \mathbb{R}$ is the step length at iteration t. The step length is referred to as the *learning rate* in machine learning jargon.

Convergence to the solution θ^* is guaranteed if the objective function, the search direction, and the step length satisfy certain conditions. Among these conditions are the following: the objective function must be smooth and bounded below; the search direction must be a descent direction; and the step length must satisfy sufficient decrease conditions, known as the Armijo-Wolfe conditions (Luenberger and Ye, 2008; Nocedal and Wright, 2006). In machine learning, these conditions are only loosely enforced when solving (11) due to the very large scale of the problem, where n can be on the order of billions. In particular, the step length (learning rate) is not checked to satisfy the Armijo-Wolfe conditions and is instead adjusted heuristically.

Algorithm *Stochastic/mini-batch gradient for NNs*
Set starting point θ
Set mini-batch size $p' > 0$ for set $\mathcal{P}'$
Set T number of epochs
for $t = 1, \ldots, T$
  for $j = 1, \ldots, \left\lceil \frac{p}{p'} \right\rceil$
    Select subset of points $\mathcal{P}'$
    Compute $\nabla F_{\mathcal{P}'}(\theta)$
    Compute step length (learning rate) α
    $\theta := \theta - \alpha \nabla F_{\mathcal{P}'}(\theta)$
  end_for
end_for
Return: θ
End_algorithm

Figure 4. *The stochastic gradient method.*

The gradient method takes as its search direction the negative gradient of the objective (or loss) function in (11) at the current point:

$$\Delta\theta^t = -\nabla F(\theta^t) = \frac{1}{p}\sum_{i=1}^{p} \nabla F_i(\theta^t). \tag{14}$$

Computing $\nabla F(\theta^t)$ therefore requires the gradient $\nabla_\theta f(x;\theta)$ of the NN function (5) for all p points in the dataset. The challenging computation of $\nabla_\theta f(x;\theta)$ is carried out in machine learning frameworks using an algorithm known as *backpropagation*. Backpropagation is often mistaken for the optimization algorithm in NNs, whereas it is in fact the procedure used to compute the gradient.

In practice, instead of computing the p gradients $\nabla F_i(\theta^t)$, $i = 1, \ldots, p$, for the complete set of points $\mathcal{P} = \{1, \ldots, p\}$, it is common to consider only a subset $\mathcal{P}' \subseteq \mathcal{P}$ of $p' < p$ points, so that an estimate of the negative gradient is used as the search direction:

$$\Delta\theta^t = -\nabla F_{\mathcal{P}'}(\theta^t) = -\frac{1}{|\mathcal{P}'|}\sum_{i \in \mathcal{P}'} \nabla F_i(\theta^t). \tag{15}$$

In machine learning, the set $\mathcal{P}'$ is called the *mini-batch* set. When $p' = 1$ this procedure is known as the *stochastic gradient*; if $p > p' > 1$ it is called the *mini-batch gradient*; and when $p' = p$ (i.e., the classical gradient method) it is referred to as the *full-batch gradient*. The stochastic/mini-batch gradient algorithm is outlined in Figure 4. The procedure

has two main loops: the outer loop, called *epochs* in machine learning, corresponds to the iterations of the (full-batch) gradient method, while at each iteration of the inner loop the algorithm selects a subset $\mathcal{P}'$ of p' points to compute the search direction. Each epoch thus consists of $\left\lceil \frac{p}{p'} \right\rceil$ iterations, where the final mini-batch may contain fewer than p' points. The number of epochs is usually predetermined (parameter T in the algorithm of Figure 4), although early stopping may be used when the iterates stabilize.

More efficient variants of the standard algorithm in Figure 4 can be obtained either by modifying the search direction or by adapting the computation of the step length. Among the former are the *gradient with momentum* (Polyak, 1964) and the *Nesterov accelerated method* (Nesterov, 1983), which combine the current and previous gradients to determine the search direction. Among the latter, which compute adaptive step lengths, we find *AdaGrad*, *RMSProp*, and *Adam*; the last of these is one of the most widely used methods in modern machine learning frameworks (see Goodfellow et al. (2016) for details).

3. Two illustrative applications of feedforward neural networks

The next two subsections present two classic applications of feedforward NNs: the *XOR (exclusive or)* function and the *fashion-MNIST* problem. The XOR model is solved from scratch using both AMPL and Google's TensorFlow, whereas the fashion-MNIST problem, due to its greater complexity, is solved only with TensorFlow.

3.1. Neural network for XOR

The XOR problem was used in several of the seminal papers on NNs to illustrate their capabilities (Minsky and Papert, 1969; Rumelhart, Hinton and Williams, 1986). The XOR (exclusive or) is a binary operation that, given two binary inputs, returns 1 ("true") if and only if exactly one input is 1, and 0 ("false") otherwise. It is defined as XOR : $\{0,1\}^2 \rightarrow \{0,1\}$, with truth table shown below:

x_1	x_2	XOR
0	0	0
0	1	1
1	0	1
1	1	0

The training set for this problem contains the four points of the domain, so $p = 4$, and the dataset is

$$\{(x^i = (x^i_1, x^i_2), y^i), i = 1, \ldots, 4\} = \{((0,0),0), ((0,1),1), ((1,0),1), ((1,1),0)\}.$$

We consider a NN with $l = 3$ layers; an input layer, one hidden layer, and an output layer. Figure 5 shows the structure of the network. The number of nodes of the first

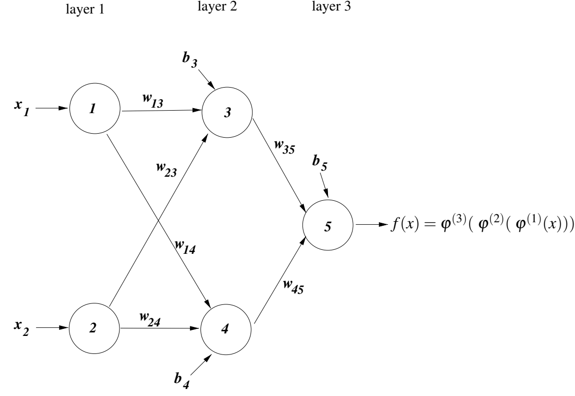

Figure 5. *XOR NN.*

(input) and third (output) layers are $d_1 = 2$ and $d_3 = 1$, matching the input and output dimensions of the XOR function. The hidden (second) layer has $d_2 = 2$ nodes. For the activation functions, we choose the identity for the first and third layers, and the sigmoid for the second layer.

The XOR NN is the composition of three functions $f(x) = \varphi^{(3)}(\ \varphi^{(2)}(\ \varphi^{(1)}(x)))$, where $x \in \mathbb{R}^2$. From (5), we have that $\varphi^{(1)} : \mathbb{R}^2 \to \mathbb{R}^2$ is the identity function, and, since $x^{(0)} = x$, it follows that

$$x^{(1)} = \varphi^{(1)}(x^{(0)}) = x^{(0)} = x, \tag{16}$$

that is, the first layer simply passes the input vector to the second layer. $f(x) = \varphi^{(3)}(\ \varphi^{(2)}(\ \varphi^{(1)}(x)))$, In the second layer, we have $\varphi^{(2)} : \mathbb{R}^2 \to \mathbb{R}^2$. Since $g^{(2)}(z) = \sigma(z)$, and the matrix of weights and vector of biases are

$$W^{(2)} = \begin{pmatrix} w_{13} & w_{23} \\ w_{14} & w_{24} \end{pmatrix}, \qquad b^{(2)} = \begin{pmatrix} b_3 \\ b_4 \end{pmatrix},$$

we obtain

$$\begin{aligned} x^{(2)} &= \varphi^{(2)}(x^{(1)}) = g^{(2)}(W^{(2)}x^{(1)} + b^{(2)}) = \begin{pmatrix} g^{(2)}(w_{13}x_1 + w_{23}x_2 + b_3) \\ g^{(2)}(w_{14}x_1 + w_{24}x_2 + b_4) \end{pmatrix} = \\ &= \begin{pmatrix} 1/(1+e^{-(w_{13}x_1+w_{23}x_2+b_3)}) \\ 1/(1+e^{-(w_{14}x_1+w_{24}x_2+b_4)}) \end{pmatrix}. \end{aligned} \tag{17}$$

Finally, in the third (output) layer we have $\varphi^{(3)} : \mathbb{R}^2 \to \mathbb{R}$, and, as a modeling decision, we choose the node to have no *bias*, i.e., $b_5 = 0$. Since $g^{(3)}(z) = I(z)$, and

$$W^{(3)} = \begin{pmatrix} w_{35} & w_{45} \end{pmatrix}, \qquad b^{(3)} = (b_5) = (0),$$

we obtain

$$f(x) = x^{(3)} = \varphi^{(3)}(x^{(2)}) = g^{(3)}(W^{(3)}x^{(2)} + b^{(3)}) = w_{35}x_1^{(2)} + w_{45}x_2^{(2)}. \quad (18)$$

Combining (16), (17), and (18), the final expression of the XOR NN is:

$$f(x) = \varphi^{(3)}(\ \varphi^{(2)}(\ \varphi^{(1)}(x))) = \frac{w_{35}}{1+e^{-(w_{13}x_1+w_{23}x_2+b_3)}} + \frac{w_{45}}{1+e^{-(w_{14}x_1+w_{24}x_2+b_4)}}. \quad (19)$$

Taking the MSE as the objective (loss) function, the resulting optimization problem is:

$$\min \frac{1}{4}\sum_{i=1}^{4}(f(x^i)-y^i)^2 = \frac{1}{4}\sum_{i=1}^{4}\left(\left(\sum_{k=3}^{4}\frac{w_{k5}}{1+e^{-(\sum_{j=1}^{2}w_{jk}x_j^i+b_k)}}\right)-y^i\right)^2 \quad (20)$$

```
1  param d >= 1, integer; #dimension input points
2  param p >= 1, integer; #number points
3  param y {i in 1..p};    #output vector
4  param x {1..p,1..d};    #input points
5  param nodes;            #number of nodes/neurons
6  set LINKS within (1..nodes cross 1..nodes); # set of links between nodes
7  param init{LINKS};      # initial values of weights
8  # variables
9  var w{(i,j) in LINKS} := init[i,j]; # weights
10 var b{1..nodes}; # biases
11 # Objective function
12 minimize MSE: 1/p*sum{i in 1..p} ( (sum{k in {3,4}} (w[k,5]*(1/(1+exp(-(
       b[k]+sum{j in 1..2} w[j,k]*x[i,j]))))) - y[i])^2 ;
```

Figure 6. *AMPL model for the XOR NN.*

Figure 6 shows the AMPL model for the XOR NN. The network topology is defined by the number of nodes (line 5) and the set of links (line 7). The optimization variables (the weights and biases) are declared in lines 9 and 10. The objective function in line 12 corresponds exactly to (20). Initial values for the weights are provided in the data file shown in Figure 7. Different initial values may lead to different solutions, as the objective function is nonlinear and nonconvex.

Solving the problem with MINOS, one of AMPL's available solvers (which implements, for unconstrained optimization problems, a quasi-Newton method based on the BFGS formula), we obtain an objective (loss) function value of $6.4 \cdot 10^{-13}$ in only 33 iterations, with the following optimal weights and biases:

$$W^{(2)} = \begin{pmatrix} 14.19 & 11.56 \\ 0.96 & 0.96 \end{pmatrix}, \qquad W^{(3)} = \begin{pmatrix} 5.85 & -6.71 \end{pmatrix}, \qquad b^{(2)} = \begin{pmatrix} 0.29 \\ 0 \end{pmatrix}.$$

```
1  param d := 2;
2  param p  := 4;
3  param nodes := 5;
4  set LINKS := (1,3) (1,4) (2,3) (2,4) (3,5) (4,5) ;
5  param init :=
6           1 3  1
7           1 4  0.1
8           2 3  0.1
9           2 4  0.1
10          3 5  0.1
11          4 5  -1.0 ;
12 param y :=
13    1      0
14    2      1
15    3      1
16    4      0 ;
17 param x :   1     2  :=
18    1        0     0
19    2        0     1
20    3        1     0
21    4        1     1 ;
```

Figure 7. *AMPL data for the XOR NN.*

Evaluating the function $f(x)$ at the four input points we obtain:

$$f(x) = \begin{pmatrix} -10^{-6} \\ 1.00 \\ 1.00 \\ 10^{-6} \end{pmatrix} \approx \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \end{pmatrix} = y,$$

showing the function is an excellent approximation to XOR. Having predicted and input values that are too similar (a situation known as *overfitting*) is not always desirable, since the results for new different testing points may be incorrect. This issue is not relevant for the XOR problem, since we only have the four training points.

Although modeling languages such as AMPL greatly simplify the formulation and solution of optimization problems (in particular, the symbolic computation of derivatives), implementing a deep NN (i.e., one with many layers) can be challenging, as the objective function is a composition of multiple functions, each potentially using a different activation function. To address this complexity, several machine learning frameworks have been developed, specifically designed for modeling deep NNs. One of the most widely used is Google's TensorFlow (Abadi et al., 2016).

Figure 8 shows the implementation of the XOR problem using the Python interface to TensorFlow, called *Keras* (Chollet, 2017). Line 2 of the code imports the TensorFlow package. Lines 4–7 define the training set and force the testing set to be identical to the training set. Lines 9–11 describe the structure of the network by defining the second and

```
import numpy as np
import tensorflow as tf
# XOR data
X_train = np.array([ [ 0, 0], [0 ,1] ,[1 ,0], [1 ,1]]);
y_train= np.array([0,1,1,0]);
X_test= X_train
y_test= y_train
# create NN
network = tf.keras.models.Sequential([
  tf.keras.layers.Dense(2,activation='sigmoid',input_shape=(2,)),
  tf.keras.layers.Dense(1, activation='linear', use_bias= False)
])
# information about NN structure
network.summary()
# create optimization problem
network.compile(optimizer='sgd',
              loss='mean_squared_error',
              metrics=['accuracy'])
#solve optimization problem
network.fit(X_train, y_train, epochs=100, batch_size=4)
#retrieve optimal solution
network.get_weights()
# test NN
results= network.evaluate(X_test, y_test)
print('test loss, test accuracy:', results)
```

Figure 8. *Python code using TensorFlow for the XOR NN.*

third layers; the first layer is omitted, as it simply passes the input data to the second layer. The second layer, defined in line 10, contains two nodes with the sigmoid activation function and receives two values from the first layer. The output layer, defined in line 11, consists of a single node with an identity activation function (*linear* activation in Keras/TensorFlow notation) and no bias term. Line 14 produces a summary of the network structure, including the number of parameters to be optimized. This summary, shown in Figure 9, indicates that there are six parameters in the second layer (four weights in $W^{(2)}$ and two biases in $b^{(2)}$) and two parameters in the output layer (the two weights in $W^{(3)}$, since $b^{(3)} = 0$). In this case, the summary does not provide information about the first layer because it is simply the identity function. Lines 6–8 define the optimization problem, setting the MSE as the objective function and using *sgd* (stochastic gradient descent) as the optimization method. In addition to the objective function value, another metric, the *accuracy* (defined as the fraction of correctly classified points in the testing set), is specified in line 8 to monitor the optimization process. The optimization itself is performed in line 20, considering $p' = 4$ as the size of the mini-batch set $\mathcal{P}'$, and $T = 100$ epochs. After the optimization, the optimal parameters are retrieved in line 22, and the accuracy and objective function value for the test data are reported in lines 24–25.

```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 2)                 6

 dense_1 (Dense)             (None, 1)                 2

=================================================================
Total params: 8
Trainable params: 8
Non-trainable params: 0
_________________________________________________________________
```

Figure 9. *Summary of the structure of the XOR NN.*

With the TensorFlow code of Figure 8, we obtain a solution with an objective function value of 0.41, an accuracy of 0.5, and the following optimal parameters:

$$W^{(2)} = \begin{pmatrix} 0.01 & 0.15 \\ 0.05 & 0.14 \end{pmatrix}, \qquad W^{(3)} = \begin{pmatrix} 0.01 & -0.20 \end{pmatrix}, \qquad b^{(2)} = \begin{pmatrix} 0.62 \\ -0.70 \end{pmatrix}.$$

From the accuracy value, it is evident that this solution is far from optimal. Indeed, evaluating the NN function at the four input points yields

$$f(x) = \begin{pmatrix} 0.0108 \\ 0.1486 \\ 0.0543 \\ 0.1417 \end{pmatrix} \neq \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \end{pmatrix} = y.$$

This result indicates that the quasi-Newton method used by MINOS clearly outperforms stochastic gradient descent as an optimization algorithm for this small NN. To obtain a better solution with TensorFlow, the size of the network must be increased. In particular, by replacing the sigmoid activation function of the second layer with a ReLU, increasing the number of nodes in the second layer to 15 (resulting in an optimization problem with 62 parameters), and raising the number of epochs to 500, we obtain an accuracy of 1.0, although $f(x)$ still exhibits significant discrepancies with y. In spite of these limitations, stochastic gradient descent and its variants are currently essential for the optimization of very large NNs.

3.2. Neural network for the fashion-MNIST problem

The fashion-MNIST problem was introduced in Xiao, Rasul, Vollgraf (2017). The goal of this problem is to classify a set of images of garments into $k = 10$ categories: “T-shirt/top”, “Trouser”, “Pullover”, “Dress”, “Coat”, “Sandal”, “Shirt”, “Sneaker”, “Bag”, and “Ankle boot”. The dataset consists of 70,000 images (60,000 for training and 10,000 for testing), where each image is represented by a 28×28 matrix of pixels, and each pixel is a number in [0,255] corresponding to a grayscale intensity. In other words, the training set consists of 60,000 matrices of size 28×28, and a vector $y \in \mathbb{R}^{60,000}$, where

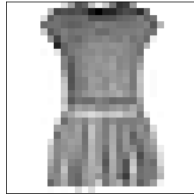

Figure 10. *Example of a garment of category "Dress".*

each component $y^i \in \{1,\ldots,10\}$ denotes the garment category. Figure 10 illustrates one of the images, corresponding to a garment of category "Dress".

For this multiclass classification problem, we consider a NN that, given a 28×28 matrix corresponding to an image, returns a probability distribution $f(x) \in \mathbb{R}^{10}$, where the predicted garment category is selected as $\arg\max_{i=1,\ldots,10} f(x)_i$. Figure 11 shows the Python code for this NN. Lines 4–10 read the dataset, declare the category names, and scale the pixel values from the range [0,255] to [0,1]. Lines 12–16 describe the structure of the network, which contains three layers. The input layer (line 13) simply reshapes the input matrix into a vector of $28 \cdot 28 = 784$ components, so this first layer has 784 nodes; this operation is called *flatten* in the code. The hidden layer (line 14) contains 128 nodes and uses a ReLU activation function. The output layer (line 15) has 10 nodes and applies a softmax activation function. Lines 20–22 formulate the optimization problem, where *adam* (a modification of stochastic gradient descent) is used as the optimization algorithm, and cross-entropy is the objective function. More precisely, line 21 sets *sparse categorical cross-entropy*, which means that for an input image i with output value $y^i = j$, where $j \in \{1,\ldots,k\}$, this value is transformed into $y^i = e_j$, with e_j being the j-th vector of the k-dimensional identity matrix. As in the XOR example, besides the objective function value, the accuracy is also used as a metric to monitor the optimization process (line 22). The optimization problem is solved in line 24 considering 10 epochs. Finally, the accuracy is computed and reported in lines 26–27.

Figure 12 shows the output produced by line 18 of the code in Figure 11. It can be observed that the input layer has no parameters, as it simply recasts the input matrix as a vector. The second layer has 100,480 parameters to optimize, corresponding to $784 \times 128 = 100,352$ weights plus 128 biases. The last layer contains 1,290 parameters, $128 \times 10 = 1,280$ weights plus 10 biases. Overall, the optimization problem involves 101,770 variables.

Once the optimization problem is solved, we obtain an optimal solution with an objective function value of 0.2391 and an accuracy of 0.9098. The accuracy for the testing data, reported by line 27, is 0.8797; that is, 87.97% of the test images are correctly classified. Figure 13 shows the predictions made by the NN on a subset of 25 test images. All 25 images are correctly classified.

```
import numpy as np
import tensorflow as tf
#load fashion-MNIST data
fashion_mnist = tf.keras.datasets.fashion_mnist
(x_train, y_train),(x_test, y_test) = fashion_mnist.load_data()
#name labels
class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',
               'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']
#scale pixels 0-255 to 0-1
x_train, x_test = x_train / 255.0, x_test / 255.0
#create NN
model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)),
  tf.keras.layers.Dense(128, activation='relu'),
  tf.keras.layers.Dense(10, activation='softmax')
])
#information about NN structure
model.summary()
#create optimization problem
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
#solve optimization problem
model.fit(x_train, y_train, epochs=10, validation_data=(x_test,y_test))
#test NN
results= model.evaluate(x_test, y_test)
print('test loss, test acc:', results)
```

Figure 11. *Python code using TensorFlow for the fashion-MNIST NN.*

4. Convolutional neural networks

CNNs (LeCun et al., 1998) are an extension of feedforward NNs, particularly suitable for image data (e.g., pictures, videos, and similar types of structured grids). CNNs introduce new types of layers in addition to the fully connected ones described in previous sections. Two of the most important types are the *convolutional* and *pooling* layers, which are outlined in the following two subsections.

4.1. Convolutional layers

Convolutional layers are applied to images, which can be represented as three-dimensional arrays of size $t_1 \times t_2 \times t_3$ (or higher); such structures are referred to as *tensors*. Commonly, the last dimension has $t_3 = 3$, meaning that the image is composed of three $t_1 \times t_2$ matrices of pixels. Each of these three matrices is referred to as a *channel*, corresponding to the RGB (red, green, blue) components of the image. The entry (i, j) in each channel gives the amount of red, green, or blue in pixel (i, j) of the image.

```
Layer (type)                 Output Shape              Param #
=================================================================
flatten (Flatten)            (None, 784)               0

dense_2 (Dense)              (None, 128)               100480

dense_3 (Dense)              (None, 10)                1290

=================================================================
Total params: 101,770
Trainable params: 101,770
Non-trainable params: 0
_________________________________________________________________
```

Figure 12. *Summary of the structure of the fashion-MNIST NN.*

Figure 13. *Predictions for some images.*

Convolutional layers consist of a set of matrices (called *kernels* or *filters*) of dimension $h_1 \times h_2 \times h_3$, where $h_1 = h_2 = h$ for a chosen h (typically 3, 5, or 7), and $h_3 = t_3$ (in the case of RGB images, $h_3 = t_3 = 3$). These kernels scan the tensor that defines the image in order to detect patterns. For the case of RGB images, the scan is performed over subtensors of dimension $h \times h \times 3$, moving from the first corner of the tensor (top left of first channel), to the last one (bottom right of last channel), and computing a linear combination of the subtensor with the kernel at each position. The kernel values are the parameters to be optimized in the NN.

Figure 14 illustrates the convolutional operation for the case of an image with only one 4×4 channel, considering a 3×3 kernel K. For instance, for an image with the following particular channel

$$A = \begin{pmatrix} 0.5 & 0.7 & 0.2 & 1 \\ 0.3 & 0.2 & 0.3 & 0.7 \\ 0.1 & 0.8 & 0.1 & 0.9 \\ 0 & 0.4 & 0.3 & 1 \end{pmatrix}, \tag{21}$$

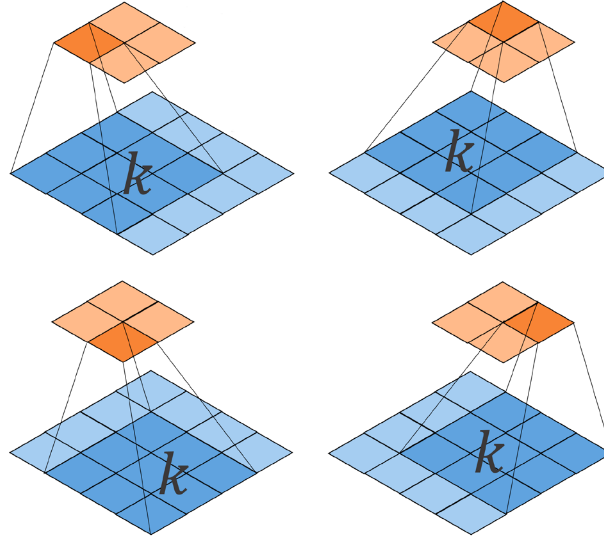

Figure 14. *Example of convolutional layer*

and a kernel 3×3,

$$K = \begin{pmatrix} k_{11} & k_{12} & k_{13} \\ k_{21} & k_{22} & k_{23} \\ k_{31} & k_{32} & k_{33} \end{pmatrix},$$

the convolutional operation would provide a matrix R of size 2×2 with the following four entries:

- For top left 3×3 submatrix of A: $R_{11} = 0.5k_{11} + 0.7k_{12} + 0.2k_{13} + 0.3k_{21} + 0.2k_{22} + 0.3k_{23} + 0.1k_{31} + 0.8k_{32} + 0.1k_{33}$.
- For top right 3×3 submatrix of A: $R_{12} = 0.7k_{11} + 0.2k_{12} + 1k_{13} + 0.2k_{21} + 0.3k_{22} + 0.7k_{23} + 0.8k_{31} + 0.1k_{32} + 0.9k_{33}$.
- For the bottom left 3×3 submatrix of A: $R_{21} = 0.3k_{11} + 0.2k_{12} + 0.3k_{13} + 0.1k_{21} + 0.8k_{22} + 0.1k_{23} + 0k_{31} + 0.4k_{32} + 0.3k_{33}$.
- For the bottom right 3×3 submatrix of A: $R_{22} = 0.2k_{11} + 0.3k_{12} + 0.7k_{13} + 0.8k_{21} + 0.1k_{22} + 0.9k_{23} + 0.4k_{31} + 0.3k_{32} + 1k_{33}$.

The resulting matrix after the convolution operation with the kernel has dimension $(t_1 - h + 1) \times (t_2 - h + 1)$. If layer j of the NN is a convolutional layer with d_j nodes, then the number of resulting matrices in this layer is d_j. Since for each of these d_j nodes, kernels of size $h \times h \times t_3$ are required, plus one bias, the total number of NN parameters to optimize for layer j is $d_j(t_3h^2 + 1)$. In state-of-the-art NN frameworks, the kernel is also allowed to move more than one pixel (horizontally and vertically) across the channel matrices during the convolution operation. This pixel step size is called the *stride*. To simplify the exposition, we assume a stride of one in the examples above and below.

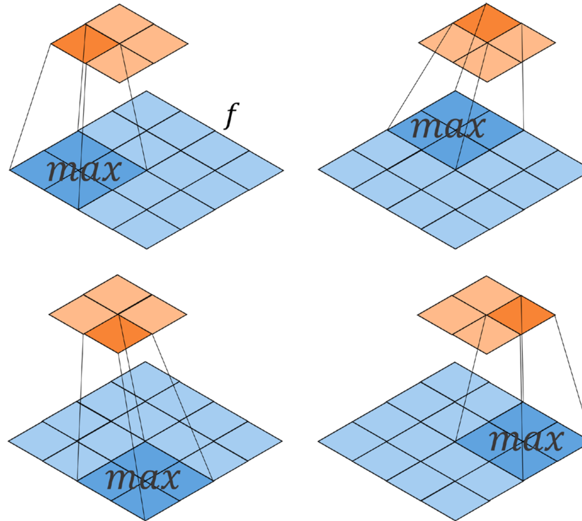

Figure 15. *Example of max pooling layer*

After the convolution operation is performed, the activation function of the layer is applied to each entry of the resulting matrix.

4.2. Pooling layers

Pooling layers are used to reduce the dimensionality of the data, allowing faster computations, freeing memory, and helping to prevent overfitting. These layers do not add new parameters to the NN; they simply apply a specific function to submatrices of the input matrices. Commonly used functions include the max function, which returns the maximum value of each submatrix, and the average function, which returns the mean of its values. For example, if layer j of the NN is a pooling layer that receives matrices of size $t \times t$ from layer $j-1$, and the size of the pooling operation is h (assuming h divides t), then the matrices generated by layer j will have dimension $t/h \times t/h$. Pooling layers are typically placed after convolutional layers to reduce their output dimensions. Figure 15 illustrates a max pooling operation of size 2 on a 4×4 matrix.

As an example, for matrix (21), applying a max pooling operation of size 2 yields the resulting matrix

$$R = \begin{pmatrix} R_{11} & R_{12} \\ R_{21} & R_{22} \end{pmatrix}, \text{ where } \begin{array}{ll} R_{11} = \max\{0.5, 0.7, 0.3, 0.2\}, & R_{12} = \max\{0.2, 1, 0.3, 0.7\}, \\ R_{21} = \max\{0.1, 0.8, 0, 0.4\}, & R_{22} = \max\{0.1, 0.9, 0.3, 1\}. \end{array}$$

Although the max function is not differentiable, NNs are still optimized using stochastic gradient methods, since it is piecewise linear and differentiable almost everywhere.

```
import tensorflow as tf
from tensorflow.keras import datasets, layers, models
import numpy as np
#load data
(train_images, train_labels), (test_images, test_labels) = datasets.
    cifar10.load_data()
#name labels
class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer',
               'dog', 'frog', 'horse', 'ship', 'truck']
# scale pixels from [0,255] to [0,1]
train_images, test_images = train_images / 255.0, test_images / 255.0
#create NN
m = models.Sequential()
m.add(layers.Conv2D(32,(3,3),activation='relu',input_shape=(32,32,3)))
m.add(layers.MaxPooling2D((2, 2)))
m.add(layers.Conv2D(64, (3, 3), activation='relu'))
m.add(layers.MaxPooling2D((2, 2)))
m.add(layers.Conv2D(64, (3, 3), activation='relu'))
m.add(layers.Flatten())
m.add(layers.Dense(64, activation='relu'))
m.add(layers.Dense(10,activation='softmax'))
#information about NN structure
m.summary()
#create optimization problem
m.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
#solve optimization problem
m.fit(train_images, train_labels, epochs=10,
                    validation_data=(test_images, test_labels))
#test NN
results = m.evaluate(test_images,  test_labels, verbose=2)
print('test loss, test acc:', results)
```

Figure 16. *Python code using TensorFlow for the CIFAR-10 CNN.*

5. An illustrative application of convolutional neural networks

To illustrate the use of CNNs we consider the CIFAR-10 dataset (Krizhevsky, 2009). CIFAR-10 contains 60,000 color images, 50,000 for training and 10,000 for testing. Each image consists of three 32×32 pixels matrices, each matrix corresponding to one of the RGB channels. As in the fashion-MNIST problem, the goal is to classify the color images into $k = 10$ categories: "airplane", "automobile", "bird", "cat", "deer", "dog", "frog", "horse", "ship", and "truck".

The CNN for the CIFAR-10 problem is a composition of nine functions, that is, $f(x) = \varphi^{(9)}(\varphi^{(8)}(\ldots\varphi^{(2)}(\varphi^{(1)}(x))\ldots))$, where each function (layer) is as follows:

- The first (input) layer simply takes the three RGB channels of an image and passes them to the second layer. Hence, $\varphi^{(1)} : \mathbb{R}^{32\times32\times3} \to \mathbb{R}^{32\times32\times3}$.
- The second layer is a convolutional layer with 32 kernels (and thus 32 nodes) of

size $3 \times 3 \times 3$, followed by a ReLU activation function. The 32 output matrices have dimensions 30×30. Hence, $\varphi^{(2)} : \mathbb{R}^{32\times32\times3} \rightarrow \mathbb{R}^{30\times30\times32}$. The NN parameters to be optimized include the kernel weights and the node biases, which amount to $32 \cdot 3 \cdot 3 \cdot 3 + 32 = 896$.

- The third layer is a max pooling layer of size 2×2; thus, the dimensions of the matrices generated by the second layer are halved. Therefore, $\varphi^{(3)} : \mathbb{R}^{30\times30\times32} \rightarrow \mathbb{R}^{15\times15\times32}$.
- The fourth layer is another convolutional layer with 64 kernels (64 nodes) of size $3 \times 3 \times 32$, with a ReLU activation function. The 64 output matrices have dimensions 13×13. Hence, $\varphi^{(4)} : \mathbb{R}^{15\times15\times32} \rightarrow \mathbb{R}^{13\times13\times64}$. The NN parameters to be optimized include the kernel weights and the 64 node biases, which amount to $64 \cdot 3 \cdot 3 \cdot 32 + 64 = 18{,}496$.
- The fifth layer is another max pooling layer of size 2×2. Therefore, it halves the dimensions of the matrices generated by the fourth layer, and $\varphi^{(5)} : \mathbb{R}^{13\times13\times64} \rightarrow \mathbb{R}^{6\times6\times64}$.
- The sixth layer is the third convolutional layer, with 64 kernels (and thus 64 nodes) of size $3 \times 3 \times 64$, followed by a ReLU activation function. The 64 matrices generated by this layer have dimensions 4×4. Therefore, $\varphi^{(6)} : \mathbb{R}^{6\times6\times64} \rightarrow \mathbb{R}^{4\times4\times64}$. The NN parameters to be optimized are the kernel weights and the 64 node biases, that is $64 \cdot 3 \cdot 3 \cdot 64 + 64 = 36{,}928$.
- The seventh layer simply recasts the 64 4×4 matrices produced by the previous layer into a vector (flatten operation) of size $4 \cdot 4 \cdot 64 = 1{,}024$. Hence, $\varphi^{(7)} : \mathbb{R}^{4\times4\times64} \rightarrow \mathbb{R}^{1{,}024}$.
- The last two layers (eighth and ninth) are standard fully connected layers of a feedforward NN. The first of them has 64 nodes and a ReLU activation function. Therefore, it adds $1{,}024 \cdot 64 = 65{,}536$ weights and 64 biases (a total of 65600 parameters) to the optimization problem. The last layer has 10 nodes and a softmax activation function, thus adding $64 \cdot 10 + 10 = 650$ parameters to the NN. Hence, $\varphi^{(8)} : \mathbb{R}^{1{,}024} \rightarrow \mathbb{R}^{64}$ and $\varphi^{(9)} : \mathbb{R}^{64} \rightarrow \mathbb{R}^{10}$.

From the previous description, the total number of parameters in the optimization problem is $896 + 18{,}496 + 36{,}928 + 65{,}600 + 650 = 122{,}570$. Figure 16 shows the Python code for solving the CIFAR-10 problem using TensorFlow. The structure of the CNN is defined in lines 12–20, one line per layer, except for line 13, which defines the first convolutional layer and implicitly the input layer as well. The rest of the code follows the same structure as in the fashion-MNIST example. The summary of the CNN produced by line 22 is shown in Figure 17; it can be observed that the dimensions and number of parameters to be optimized in the different layers are exactly as described above.

Once the optimization problem is solved using the cross-entropy objective function, we obtain an objective function (loss) value of 0.5891 and accuracies of 0.7889 and 0.6976 for the training and testing sets, respectively. Figure 18 shows the predictions made by the CNN for a subset of images. Although some errors remain, most images are correctly classified.

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 30, 30, 32)	896
max_pooling2d (MaxPooling2D)	(None, 15, 15, 32)	0
conv2d_1 (Conv2D)	(None, 13, 13, 64)	18,496
max_pooling2d_1 (MaxPooling2D)	(None, 6, 6, 64)	0
conv2d_2 (Conv2D)	(None, 4, 4, 64)	36,928
flatten (Flatten)	(None, 1024)	0
dense (Dense)	(None, 64)	65,600
dense_1 (Dense)	(None, 10)	650

Total params: 122,570 (478.79 KB)
Trainable params: 122,570 (478.79 KB)
Non-trainable params: 0 (0.00 B)

Figure 17. *Summary of the CNN.*

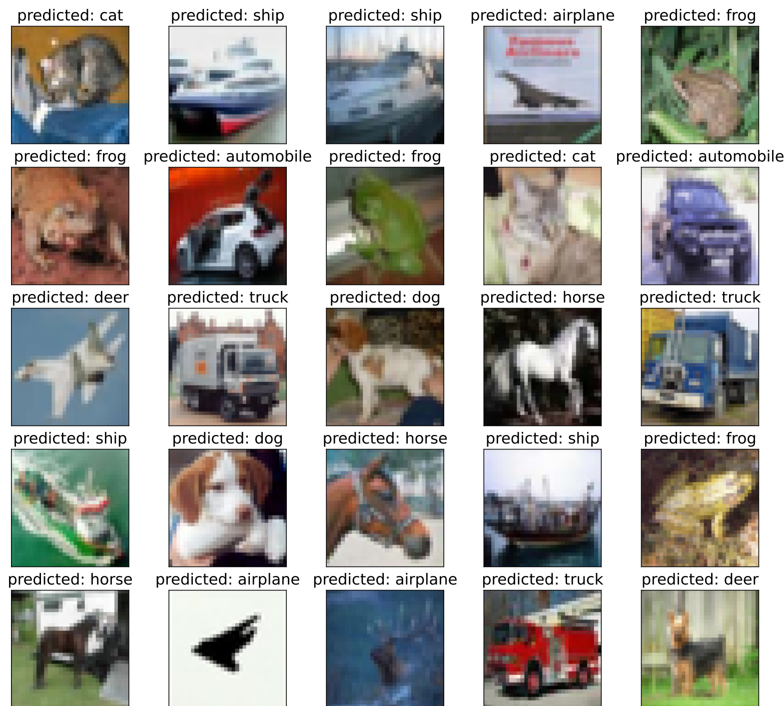

Figure 18. *Predictions for some images.*

6. Applications of neural networks in operations research

NNs have several applications in OR algorithms, for instance:

- *Predicting good solutions or heuristics.* NNs can predict near-optimal decisions (e.g., routing choices or resource allocations) instead of solving optimization models from scratch.
- *Learning objective or constraint structures.* In complex systems where the true objective or constraints are unknown or hard to model (for example, demand forecasting in supply chains), NNs can learn these relationships directly from data.
- *Combining with classical solvers (hybrid approaches).* For example, a NN can help generate feasible starting points or parameter settings for mixed-integer programming solvers, cutting planes, or metaheuristics.

Most applications of NNs in OR focus on heuristics for combinatorial optimization problems, including vehicle routing, knapsack, and assignment problems. These NN-based heuristics may follow several strategies, two of the most common being:

- *Supervised learning* (or *data-driven* approaches): a NN is trained on pairs of problem instances and their corresponding (optimal or heuristic) solutions, and later used to predict near-optimal solutions for new instances.
- *Reinforcement learning:* the NN acts as an agent that iteratively learns to make good decisions by interacting with an environment.

A discussion of these NN-based heuristics can be found in the recent survey Bengio et al. (2021).

For illustration purposes, the next subsection shows how NNs can be used to construct a simple data-driven heuristic for the knapsack problem. It is worth remarking that our purpose is not to promote the use of NNs for OR problems. Indeed, NNs have several limitations in this context: (1) they usually provide only approximate solutions rather than optimal ones; (2) they require large training datasets; and (3) the interpretability and feasibility of their solutions can be problematic for strict OR models with hard constraints. In addition, for most problems, classical optimization solvers (such as CPLEX) remain far superior.

6.1. Supervised learning for the 0–1 knapsack problem

Consider a dataset of N independent instances of the 0–1 knapsack problem. Each instance j contains n items with given values $v_i^j > 0$, weights $w_i^j > 0$, and capacity $C^j > 0$, for $i = 1, \ldots, n$ and $j = 1, \ldots, N$. Therefore, the optimization problem for instance j is:

$$\begin{aligned} \max_{x^j} \quad & \sum_{i=1}^{n} v_i^j x_i^j \\ \text{s. to} \quad & \sum_{i=1}^{n} w_i^j x_i^j \leq C^j, \\ & x_i^j \in \{0,1\}, \quad i = 1,\ldots,n. \end{aligned} \tag{22}$$

For each instance j, an optimal binary solution $x^{*j} \in \{0,1\}^n$ can be obtained exactly using a standard solver (such as CPLEX). These optimal solutions are used as training labels in a supervised learning framework.

For every item i in instance j, we define a feature vector $\mathbf{z}_i^j \in \mathbb{R}^4$ as

$$\mathbf{z}_i^j = \left(v_i^j,\ w_i^j,\ v_i^j/w_i^j,\ w_i^j/C^j\right), \tag{23}$$

and associate it with the corresponding optimal label x_i^{*j}. The resulting training set is therefore

$$\left\{(\mathbf{z}_i^j, x_i^{*j})\ , i = 1,\ldots,n,\ j = 1,\ldots,N\right\}, \tag{24}$$

which contains Nn labeled samples.

A small feedforward NN $f : \mathbb{R}^4 \to (0,1)$ is trained to approximate the probability that item i belongs to the optimal knapsack solution. Given the feature vector $\mathbf{z}_i^j$, the network predicts $f(\mathbf{z}_i^j;\theta)$, which is interpreted as the probability of including item i in the solution. The model parameters θ are estimated by minimizing a suitable objective (loss function) over the entire dataset, as done in the examples of the previous sections.

Once trained, the network can be used to make fast approximate decisions for new knapsack instances. Given the predicted probabilities $f(\mathbf{z}_i^j)$ for a new instance j, a feasible binary solution is obtained by sorting items according to $f(\mathbf{z}_i^j)$ (or the ratio $f(\mathbf{z}_i^j)v_i^j/w_i^j$) and selecting them greedily as long as the capacity constraint $\sum_{i=1}^n w_i^j x_i^j \leq C^j$ is satisfied.

The previous heuristic has been implemented in Python. The full code is not included here for space reasons, but it is freely available from the corresponding author. AMPL and CPLEX are used to define and solve knapsack problems exactly in order to obtain a training set of $N = 500$ instances, each with $n = 10$ items (thus $500 \cdot 10 = 5{,}000$ data points), and a testing set of 50 additional instances with the same number of items. The NN considered has four layers: an input layer with four nodes, two hidden layers with 64 nodes and ReLU activation functions, and an output layer with one node and a sigmoid activation that returns the estimated probability of an item being in the optimal solution of its corresponding instance.

For the $50 \cdot 10 = 500$ items in the testing set, using the naive rule that selects an item if its predicted probability $f(\mathbf{z}_i^j)$ exceeds 0.5, the accuracy of the NN lies in the range 85%–90%; that is, 85%–90% of the items predicted as optimal by the NN are indeed optimal in the exact solution of the instance. Using the more sophisticated greedy heuristic described above (i.e., sorting items by $f(\mathbf{z}_i^j)$ and selecting them greedily as long as the capacity constraint is satisfied) leads to better results. For example, for

the instance with objective coefficients $v = [82, 70, 8, 67, 28, 99, 71, 25, 13, 54]$, weights $w = [14, 14, 18, 11, 9, 3, 3, 10, 6, 7]$, and capacity $C = 38$, the heuristic produces the solution $x = [1, 0, 0, 1, 0, 1, 1, 0, 0, 1]$, which coincides with the optimal solution returned by CPLEX.

This simple supervised learning approach illustrates how NNs can approximate classical optimization rules (e.g., selecting items by decreasing value-to-weight ratio) through data-driven training, while providing a bridge between exact combinatorial optimization and predictive modeling.

7. Conclusions

In a didactic manner, this work has illustrated the origin of the optimization problems underlying modern deep NNs. We have shown that, due to the large composition of functions involved, standard modeling languages used in OR and optimization (such as AMPL) can only be conveniently applied to shallow NNs, and that specialized tools (such as TensorFlow) are required to model and optimize more complex architectures. Intentionally, we have left aside more theoretical aspects, aiming instead to provide an accessible introduction to the field of deep NNs for operations researchers.

In addition, a simple NN-based heuristic for the knapsack problem has illustrated how neural models can be combined with classical OR tools to obtain fast, data-driven approximations to combinatorial optimization problems. Because NNs are trained through stochastic gradient–based optimization, these heuristics can be interpreted as an approach based on unconstrained optimization for generating approximate solutions to combinatorial optimization problems. This connection emphasizes once more the central role of optimization in the design, training, and practical use of modern neural network techniques.

8. Acknowledgments

Pau Amaré was supported by the UPC scholarship 9637. Jordi Castro was supported by the MCIN/AEI/10.13039/501100011033/FEDER,EU grant PID2022-139219OB-I00.

References

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y. and Zheng, X. (2016). TensorFlow: a system for large-scale machine learning. In: *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2016)*, 265–283. USENIX Association.

Bengio, Y., Lodi, A. and Prouvost, A. (2021). Machine learning for combinatorial optimization: a methodological tour d'horizon. *European Journal of Operational Research*, 290(2), 405–421.

Bottou, L., Curtis, F.E. and Nocedal, J. (2018). Optimization methods for large-scale machine learning. *SIAM Review*, 60(2), 223–311.

Cauchy, A. (1847). Méthode générale pour la résolution des systèmes d'équations simultanées. *Comptes Rendus Hebdomadaires des Séances de l'Académie des Sciences*, 25, 536–538.

Chollet, F. (2017). *Deep Learning with Python*. Manning Publications, Shelter Island, NY, USA.

Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2(4), 303–314.

Fourer, R., Gay, D. M. and Kernighan, B. W. (2003). *AMPL: A Modeling Language for Mathematical Programming (2nd ed.)*. Thomson Brooks/Cole, Pacific Grove, CA, USA.

Goodfellow, I., Bengio, Y. and Courville, A. (2016). *Deep Learning*. MIT Press, Cambridge, MA, USA.

Hornik, K., Stinchcombe M. and White H. (1989). Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5), 359–366.

Krizhevsky, A. (2009). Learning multiple layers of features from tiny images. Technical Report, Department of Computer Science, University of Toronto.

LeCun, Y., Bottou, L., Bengio, Y. and Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11), 2278–2324.

Luenberger, D.G. and Ye, Y. (2008). *Linear and Nonlinear Programming (3rd ed.)*. Springer, New York, USA.

Minsky, M. and Papert, S. (1969). *Perceptrons: An Introduction to Computational Geometry*. MIT Press, Cambridge, MA, USA.

Nesterov, Y. (1983). A method of solving a convex programming problem with convergence rate $O(1/k^2)$. *Soviet Mathematics Doklady*, 27(2), 372–376.

Nocedal, J. and Wright, S.J. (2006). *Numerical Optimization, 2nd Ed.*. Springer, New York, USA.

Polyak, B. T. (1964). Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 4(5), 1–17.

Rumelhart, D.E., Hinton, G.E. and Williams, R.J. (1986). Learning representations by back-propagating errors. *Nature*, 323, 533–536.

Sra, S., Nowozin, S. and Wright, S.J. (eds.) (2011). *Optimization for Machine Learning*. MIT Press, Cambridge, MA, USA.

Sutskever, I., Vinyals, O. and Le, Q.V. (2014). Sequence to sequence learning with neural networks. In: *Advances in Neural Information Processing Systems (NeurIPS)*, 27.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017). Attention is all you need. In: *Advances in Neural Information Processing Systems (NeurIPS)*, 30.

Xiao, H., Rasul, K. and Vollgraf, R. (2017). Fashion-MNIST: A novel image dataset for benchmarking machine learning algorithms. *arXiv preprint* arXiv:1708.07747.